LIN
Specification Package

Revision 2.0

This specification is provided on an "AS IS" basis only and cannot be the basis for any claims.

LIN is a registered Trademark ®.

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 REVISION HISTORY

Issue	Date	Remark
LIN 1.0	1999-07-01	Initial Version of the LIN Specification
LIN 1.1	2000-03-06	
LIN 1.2	2000-11-17	
LIN 1.3	2002-12-13	
LIN 2.0	2003-09-16	Major Revision Step

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

2 LIN

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.

2.1 SCOPE

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 promotes the interoperability of network nodes from the viewpoint of hardware and software, and a predictable EMC behavior.

2.2 FEATURES AND POSSIBILITIES

The LIN is a serial communications protocol which efficiently supports the control of mechatronics nodes in distributed automotive applications.

The main properties of the LIN bus are:

- single master with multiple slaves concept
- low cost silicon implementation based on common UART/SCI interface hardware, an equivalent in software, or as pure state machine.
- self synchronization without a quartz or ceramics resonator in the slave nodes
- deterministic signal transmission with signal propagation time computable in advance
- low cost single-wire implementation
- speed up to 20 kbit/s.
- signal based application interaction

The intention of this specification is to achieve compatibility with any two LIN implementations with respect to the scope of the standard, i.e. from the application interface, API, all the way down to the physical layer.

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 [1] with some enhancements regarding the EMI behavior.

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

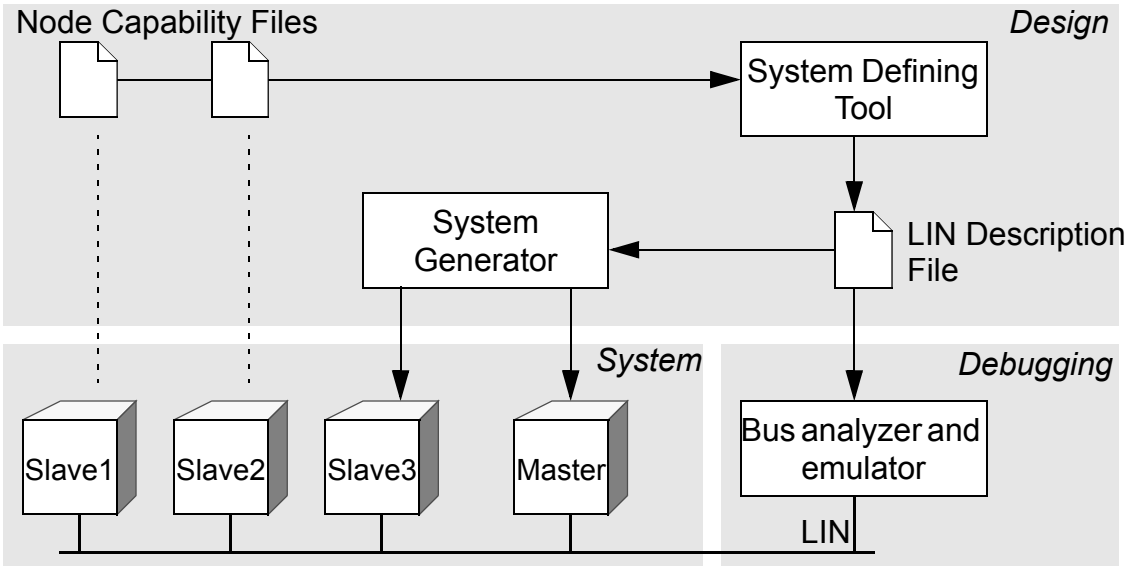
2.3 WORK FLOW CONCEPT

The LIN workflow concept allows for the implementation of a seamless chain of design and development tools and it enhances the speed of development and the reliability of the LIN cluster.

The **LIN Configuration Language** allows for safe sub-contracting of nodes without jeopardizing the LIN system functionality by e.g. message incompatibility or network overload. It is also a powerful tool for debugging of a LIN cluster, including emulation of non-finished nodes.

The **LIN Node Capability Language**, which is a new feature in LIN 2.0, provides a standardized syntax for specification of off-the-shelves slave nodes. This will simplify procurement of standard nodes as well as provide possibilities for tools that automate cluster generation. Thus, true Plug-and-Play with nodes in a cluster will become a reality.

An example of the intended workflow is depicted below:



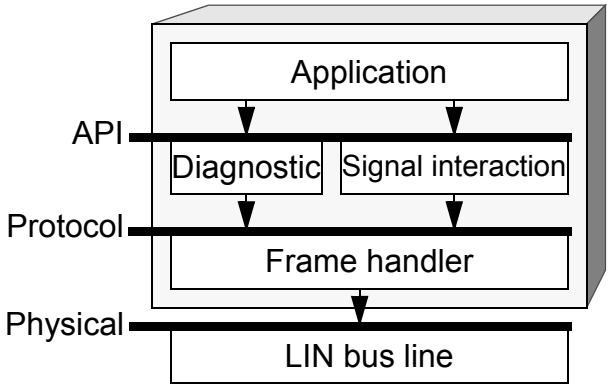
The slave nodes are connected to the master forming a LIN cluster. The corresponding node capability files are parsed by the system defining tool to generate a LIN description file (LDF) in the system definition process. The LDF is parsed by the System Generator to automatically generate LIN related functions in the desired nodes (the Master and Slave3 in the example shown in the picture above). The LDF is also used by a LIN bus analyzer/emulator tool to allow for cluster debugging.

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

2.4 NODE CONCEPT

The workflow described above generates the complete LIN cluster interaction module and the developer only has to supply the application performing the logic function of a node. Although much of the LIN specifications assumes a software implementation of most functions, alternative realizations are promoted. In the latter case, the LIN documentation structure shall be seen as a description model only:

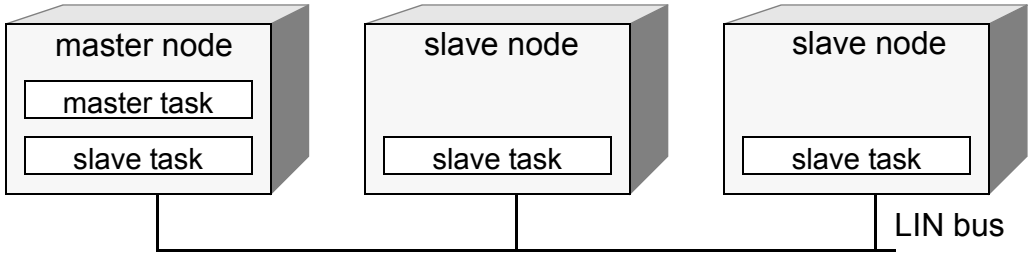
A node in a LIN cluster interfaces to the physical bus wire using a frame transceiver. The frames are not accessed directly by the application; a signal based interaction layer is added in between. As a complement, a diagnostic interface exist between the application and the frame handler, as depicted below.



2.5 CONCEPT OF OPERATION

2.5.1 Master and slave

A LIN cluster consists of one master task and several slave tasks. A master node[1] contains the master task as well as a slave task. All other nodes contain a slave task only. A sample LIN cluster with one master node and two slave nodes is depicted below:



Note 1: A node may participate in more than one cluster. The term node relates to a single bus interface of a node if the node has multiple LIN bus interfaces.

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

The master task decides when and which frame shall be transferred on the bus. The slave tasks provide the data transported by each frame.

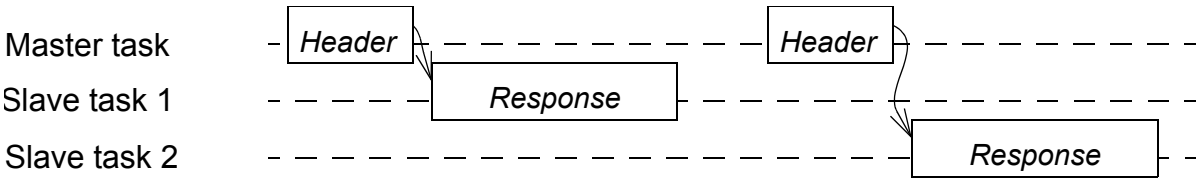
Both the master task and the slave task are parts of the Frame handler, see Section 2.4.

2.5.2 Frames

A frame consists of a header (provided by the master task) and a response (provided by a slave task).

The header consists of a break and sync pattern followed by an identifier. The identifier uniquely defines the purpose of the frame. The slave task appointed for providing the response associated with the identifier transmits it, as depicted below. The response consists of a data field and a checksum field.

The slave tasks interested in the data associated with the identifier receives the response, verifies the checksum and uses the data transported.



This results in the following desired features:

- System flexibility: Nodes can be added to the LIN cluster without requiring hardware or software changes in other slave nodes.
- Message routing: The content of a message is defined by the identifier[2].
- Multicast: Any number of nodes can simultaneously receive and act upon a single frame.

2.5.3 Data transport

Two types of data may be transported in a frame; signals or diagnostic messages.

Signals

Signals are scalar values or byte arrays that are packed into the data field of a frame. A signal is always present at the same position of the data field for all frames with the same identifier.

Note 2: This is similar to CAN

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

Diagnostic messages

Diagnostic messages are transported in frames with two reserved identifiers. The interpretation of the data field depends on the data field itself as well as the state of the communicating nodes.

2.5.4 Schedule table

The master task (in the master node) transmits frame headers based on a schedule table. The schedule table specifies the identifiers for each header and the interval between the start of a frame and the start of the following frame. The master application may use different schedule tables and select among them.

2.6 DOCUMENT OVERVIEW

The LIN Specification Package consists of the following specifications:

- The **LIN Physical Layer Specification** describes the physical layer, including bit rate, clock tolerances, etc.
- The **LIN Protocol Specification** describes the data link layer of LIN.
- The **LIN Diagnostic and Configuration Specification** describes the service that can be layered on top of the data link layer to provide for diagnostic messages and node configuration.
- The **LIN API Specification** describes the interface between the network and the application program, including the diagnostic module.
- The **LIN Configuration Language Specification** describes the format of the LIN description 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 Node Capability Language Specification** describes a format used to describe off-the-shelf slave nodes that can be used with a Plug-and-Play tool to automatically create LIN description files.

2.7 HISTORY AND BACKGROUND

LIN revision 1.0 was released in July 1999 and it was heavily influenced by the VLITE bus used by some automotive companies. The LIN standard was updated twice in year 2000, resulting in LIN 1.2 in November 2000. In November 2002 the LIN Consortium released the LIN 1.3 standard. Changes where mainly made in the physical layer and they where targeted at improving compatibility between nodes.

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

The new LIN 2.0 represents an evolutionary growth from its predecessor, LIN 1.3. Nodes designed for LIN 2.0 and LIN 1.3 will communicate with each other with a few exceptions, as described in Section 2.7.1.

At the same time, the LIN 2.0 specification is completely reworked and areas where problems have been found are clarified and, when needed, reworked.

LIN 2.0 is an adjustment of the LIN specification to reflect the latest trends identified; especially the use of off-the-shelves slave nodes. Three years of experience with LIN and inputs from the SAE J2602 Task Force have contributed to this major revision. LIN 2.0 also incorporates new features, mainly standardized support for configuration/ diagnostics and specified node capability files, both targeted at simplifying use of off-the-shelves slave nodes.

2.7.1 Compatibility with LIN 1.3

LIN 2.0 is a superset of LIN 1.3 and it is the recommended version for all new developments.

A LIN 2.0 master node can handle clusters consisting of both LIN 1.3 slaves and/or LIN 2.0 slaves. The master will then avoid requesting the new LIN 2.0 features from a LIN 1.3 slave:

- Enhanced checksum,
- Reconfiguration and diagnostics,
- Automatic baudrate detection,
- Response_error status monitoring.

A LIN 2.0 slave nodes can not operate with a LIN 1.3 master node (it needs to be configured).

2.7.2 Changes between LIN 1.3 and LIN 2.0

The items listed below are changed between LIN 1.3 and LIN 2.0. Renamings and clarifications are not listed in this section.

- Byte array signals are supported, thus allowing signals sizes up to eight bytes.
- Signal groups are deleted (replaced by byte arrays).
- Automatic bit rate detection is incorporated in the specification.
- Enhanced checksum (including the protected identifier) as an improvement to the LIN 1.3 classic checksum.
- Sporadic frames are defined.
- Network management timing is defined in seconds, not in bit times.
- Status management is simplified and reporting to the network and the application is standardized.
- Mandatory node configuration commands are added, together with some optional commands.
- Diagnostics is added.
- A LIN Product Identification for each node is standardized.
- The API is made mandatory for micro controller based nodes programmed in C.
- The API is changed to reflect the changes; byte array, go-to-sleep, wake up and status reading.
- A diagnostics API is added.
- A node capability language specification is added.
- The configuration language specification is updated to reflect the changes made; node attributes, node composition, byte arrays, sporadic frames and configuration are added.

2.8 REFERENCES

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

3 LIN GLOSSARY

The following terms are used in one or more of the **LIN 2.0 Specification Package** documents. Each term is briefly described in the glossary and a reference to the main document and section is also given, abbreviated as:

PHY	**LIN Physical Layer Specification**
PROT	**LIN Protocol Specification**
DIAG	**LIN Diagnostic and Configuration Specification**
CLS	**LIN Configuration Language Specification**
API	**LIN API Specification**
NCL	**LIN Node Capability Language Specification**

active mode	The nodes of the cluster communicate with each other as a cluster. [PROT 5]
bus interface	The logic (transceiver, UART, etc.) of a **node** that is connected to the physical bus wire in a **cluster**.
byte field	Each byte on the LIN bus is sent in a byte field; the byte field includes the start bit and stop bit transmitted. [PROT 2.1]
checksum model	Two checksum models are defined; **classic checksum** and **enhanced checksum**, enhanced includes the **protected identifier** in the checksum, classic does not. [PROT 2.1.5]
classic checksum	The checksum used in earlier LIN versions and for **diagnostic frames**: It is summed over the **data bytes** only. [PROT 2.1.5]
cluster	A cluster is the LIN bus wire plus all the **nodes**.
data	The **response** of a LIN **frame** carries one to eight bytes of data, collectively called data. [PROT 2.1.4]
data byte	One of the bytes in the **data**. [PROT 2.1.4]
diagnostic frame	The **master request frame** and **slave response frame** are called diagnostic frames. [PROT 2.3.4]
enhanced checksum	A new checksum with slightly better performance: It includes the **protected identifier** in the sum, not only **data bytes.** The enhanced checksum is used for communication with LIN 2.0 **slave nodes**. [PROT 2.1.5]

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

event triggered frame	An event triggered frame is used as a “placeholder” to allow multiple slave nodes to provide its **response**. This is useful when the signals involved are changed infrequently. [PROT 2.3.2]
frame	All information is sent packed as frames; a frame consist of the **header** and a **response**. [PROT 2]
frame slot	The time period reserved for the transfer of a specific **frame** on the LIN bus. Corresponds to one entry in the **schedule table**. [PROT 2.2]
go-to-sleep-command	A special diagnostic frame issued to force slave nodes to sleep mode. [PROT 5.2] [API 2.5.4]
header	A header is the first part of a **frame**; it is always sent by the **master task**. [PROT 2.1]
identifier	The identity of a **frame** in the range 0 to 63. [PROT 2.1.3]
LIN Description File	The LDF file is created in the **system definition** and parsed in the **system generation** or by debugging tools. [CLS] [NCL 1.1]
LIN Product Identification	A unique number for each LIN node. [DIAG 2.4]
master node	The master node not only contains a **slave task**, but also the **master task** that is responsible for sending all frame **headers** on the bus, i.e. it controls the timing and **schedule table** for the bus.
master request frame	The master request frame has **identifier** 60 and is used for **diagnostic frames** issued by the **master node**. [PROT 2.3.4] [DIAG]
master task	The master task is responsible for sending all frame **headers** on the bus, i.e. it controls the timing and **schedule table** for the bus. [PROT 4.1]
message identifier	Each **frame** in a **slave node** has a unique 16 bit message number. During node configuration this number is associated with a **protected identifier**, which is then used in the normal communication with the **node**. [DIAG 2.5.1]
NAD	Node Address for Diagnostic. Diagnostic frames are broadcasted and the NAD specifies the addressed **slave node**. The NAD is both the physical address and the logical address. [DIAG 2.3.2]

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

node	Loosely speaking, a node is an ECU (electronic control unit). However, a single ECU may be connected to multiple LIN **clusters**; in the latter case the term node should be replaced with **bus interface**.
Node Capability File	A NCF file describes a **slave node** as seen from the LIN bus. It is used in the **system definition**. [NCL 1.1]
protected identifier	The **identifier** (6 bit) together with its two parity bits. [PROT 2.1.3]
publish	A **signal** (or an **unconditional frame**) have exactly one publisher; the **node** that is the source of the information, compare with **subscribe**. [PROT 2.1.4] [PROT 4.2]
request	The master node puts request on the slave nodes in node configuration and in the diagnostic transport layer. [DIAG 2.3.1] [DIAG 3.3.1]
reserved frame	Reserved frames have an **identifier** that shall not be used: 63 (0x3f). [PROT 2.3.6]
response	(1) A LIN **frame** consists of a **header** and a response [PROT 2.1]. Also called a Frame response. (2) The reply message for an ISO request is a response [DIAG 2.3.1] [DIAG 3.3.1]. Also called a Diagnostic response.
schedule table	The schedule table determines the traffic on the LIN bus. [PROT 3] [CLS 2.5] [API 2.4]
slave node	A **node** that contains a **slave task** only, i.e. it does not contain a **master task**.
slave response frame	The slave response frame has **identifier** 61 and is used for **diagnostic frames** issued by one of the **slave nodes**. [PROT 2.3.4] [DIAG]
slave task	The slave task is responsible for listening to all frame **headers** on the bus and react accordingly, i.e. either **publish** a **frame response** or **subscribe** to it (or ignore it). [PROT 4.2]
sleep mode	No communication occurs in the cluster. [PROT 5]
signal	A signal is a value transported in the LIN **cluster** using a **signal-carrying frame**. [PROT 1]

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

signal-carrying frame	A frame that carries signals shall have an **identifier** in the range 0 to 59 (0x3b). **Unconditional frames**, **sporadic frames** and **event triggered frames** are signal-carrying frames [PROT 2.1.3].
sporadic frame	A sporadic frame is a **signal-carrying frame** similar to **unconditional frames**, but only transferred in its **frame slot** if one of its **signals** is updated by the **publisher**. [PROT 2.3.3]
subscribe	Subscribe is the opposite of **publish**, i.e. to receive a **signal** (or a **signal-carrying frame**). [PROT 2.1.4] [PROT 4.2]
system definition	The process of creating the **LIN Description File**. [NCL 1.1.2]
system generation	The process of targeting one (or multiple) of the **nodes** in the **cluster** to the **LIN Description File**. [NCL 1.1.1]
unconditional frame	A **signal-carrying frame** that is always sent in its allocated **frame slot**. [PROT 2.3.1]
user-defined frame	A frame with **identifier** 62. Its purpose or usage is not part of the LIN specification. [PROT 2.3.5]

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

TABLE OF CONTENTS

Specification Package

Protocol Specification

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

Diagnostic and Configuration Specification

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

Physical Layer Specification

Application Program Interface Specification

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

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

Node Capability Language Specification

Configuration Language Specification

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

LIN
Protocol Specification

Revision 2.0

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 SIGNAL MANAGEMENT

A signal is transported in the data field of a frame. Several signals can be packed into one frame as long as they do not overlap each other.

Each signal has exactly one producer, i.e. it is always written by the same node in the cluster. Zero, one or multiple nodes may subscribe to the signal.

1.1 SIGNAL TYPES

A signal is either a scalar value or a byte array.

A scalar signal is between 1 and 16 bits long. A one bit scalar signal is called a boolean signal. Scalar signals in the size of 2 to 16 bits are treated as unsigned integers. Any interpretation outside of this, i.e. offsetting and scaling is out of scope.

A byte array is an array of between one and eight bytes. Interpretation of the byte array is out of scope for the LIN specification. Especially, this applies to the byte endianness when representing entities larger than a byte with a byte array.

1.2 SIGNAL CONSISTENCY

Scalar signal writing or reading must be atomic operations, i.e. it should never be possible for an application to receive a signal value that is partly updated. However, no consistency is given between signals or between the individual bytes in a byte array.

1.3 SIGNAL PACKING

A signal is transmitted with the LSB first and the MSB last. The only additional rule for scalar signal packing within a frame is that maximum one byte boundary may be crossed by a scalar signal[1]. Each byte in a byte array shall map to a single frame byte starting with the lowest numbered data byte (Section 2.1.4).

Note 1: Signal packing/unpacking is implemented more efficient in software based nodes if signals are byte aligned and/or if they do not cross byte boundaries.

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

2 FRAME TRANSFER

The entities that are transferred on the LIN bus are frames.

The time it takes to send a frame is the sum of the time to send each byte plus the response space and the inter-byte space. The inter-byte space is the period between the end of the stop bit of the preceding byte and the start bit of the following byte. Both of them must be non-negative.

The inter-frame space is the time from the end of the frame until start of the next frame. The inter-frame space must also be non-negative.

2.1 FRAME STRUCTURE

The structure of a frame is shown in **Figure 2.1**. The frame is constructed of a break followed by four to eleven byte fields, labeled as in the figure.

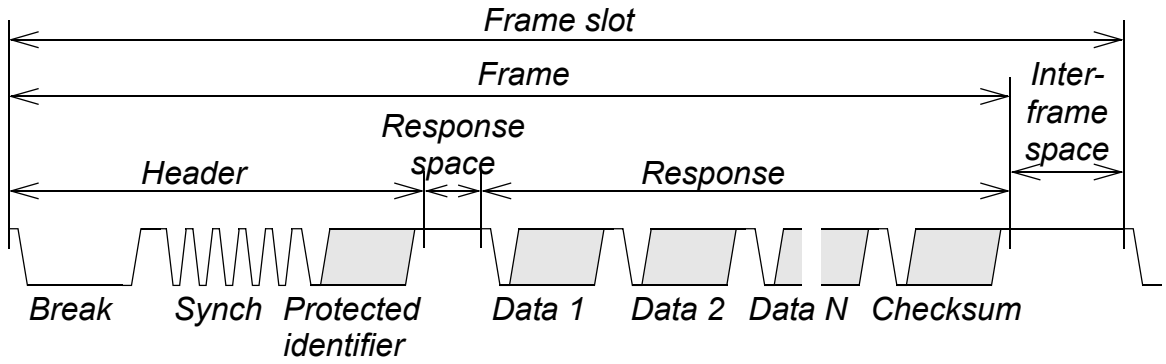


Figure 2.1: The structure of a LIN frame.

Each byte field[2] is transmitted as a serial byte, as shown in **Figure 2.2**. The LSB of the data is sent first and the MSB last. The start bit is encoded as a bit with value zero (dominant) and the stop bit is encoded as a bit with value one (recessive).

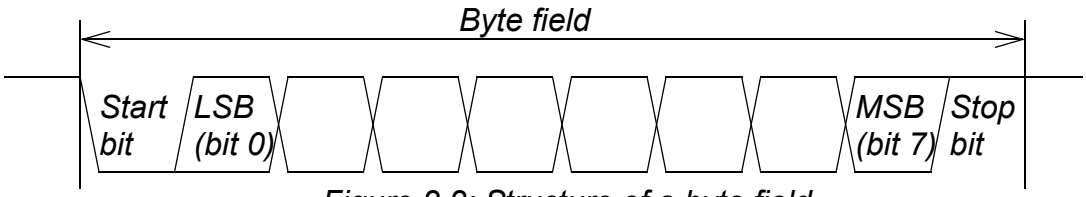


Figure 2.2: Structure of a byte field.

Note 2: Except the break byte field, see Section 2.1.1.

2.1.1 Break

The break symbol is used to signal the beginning of a new frame. It is the only field that does not comply with **Figure 2.2**: A break is always generated by the master task (in the master node) and it shall be at least 13 bits of dominant value, including the start bit, followed by a break delimiter, as shown in **Figure 2.3**. The break delimiter shall be at least one nominal bit time long.

A slave node shall use a break detection threshold of 11 nominal bit times[3].

Figure 2.3: The break field.

2.1.2 Synch byte

Synch is a byte field with the data value 0x55, as shown in **Figure 2.4**.

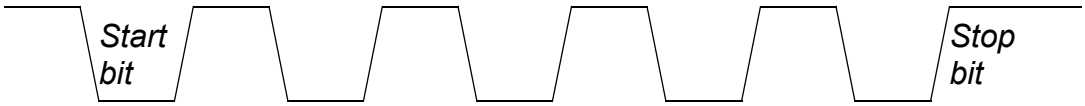


Figure 2.4: The synch byte field.

A slave task shall always be able to detect the break/synch symbol sequence, even if it expects a byte field (assuming the byte fields are separated from each other[4]). If this happens, detection of the break/synch sequence shall abort the transfer in progress[5] and processing of the new frame shall commence.

2.1.3 Protected identifier

A protected identifier consists of two sub-fields; the identifier and the identifier parity. Bit 0 to 5 is the identifier and bit 6 and 7 is the parity.

Note 3: Slave nodes with a clock tolerance better than F_{TOL_SYNCH}, see **LIN Physical Layer Table 1.2** (typically a crystal or ceramic resonator) may use a 9.5 bit break detection threshold.
Note 4: A desired, but not required, feature is to detect the break/sync sequence even if the break is partially superimposed with a data byte.
Note 5: Response_error and error in response shall be set assuming the frame is processed by the node, see Section 5.

Identifier

Six bits are reserved for the identifier (ID), values in the range 0 to 63 can be used. The identifiers are split in four categories:

- Values 0 to 59 (0x3b) are used for signal-carrying frames,
- 60 (0x3c) and 61 (0x3d) are used to carry diagnostic data,
- 62 (0x3e) is reserved for user-defined extensions,
- 63 (0x3f) is reserved for future protocol enhancements.

Parity

The parity is calculated on the identifier bits as shown in equations **(1)** and **(2)**:

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \quad (1)$$

$$P1 = \neg (ID1 \oplus ID3 \oplus ID4 \oplus ID5) \quad (2)$$

Mapping

The mapping of the bits (ID0 to ID5 and P0 and P1) is shown in **Figure 2.5**.

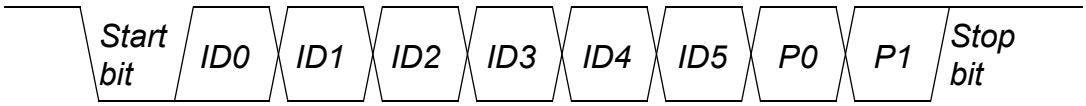


Figure 2.5: Mapping of identifier and parity to the protected identifier byte field.

2.1.4 Data

A frame carries between one and eight bytes of data. The number of bytes contained in a frame with a specific identifier shall be agreed by the publisher and all subscribers. A data byte is transmitted in a byte field, see **Figure 2.2**.

For data entities longer than one byte, the entity LSB is contained in the byte sent first and the entity MSB in the byte sent last (little-endian). The data fields are labeled data 1, data 2,... up to maximum data 8, see **Figure 2.6**.

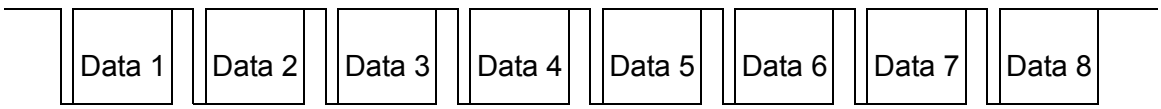


Figure 2.6: Numbering of the data bytes in a frame with eight data bytes.

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

2.1.5 Checksum

The last field of a frame is the checksum. The checksum contains the inverted eight bit sum with carry[6] over all data bytes **or** all data bytes and the protected identifier. Checksum calculation over the data bytes only is called classic checksum and it is used for communication with LIN 1.3 slaves.

Checksum calculation over the data bytes and the protected identifier byte is called enhanced checksum and it is used for communication with LIN 2.0 slaves.

The checksum is transmitted in a byte field, see **Figure 2.2**.

Use of classic or enhanced checksum is managed by the master node and it is determined per frame identifier; classic in communication with LIN 1.3 slave nodes and enhanced in communication with LIN 2.0 slave nodes.

Identifiers 60 (0x3c) to 63 (0x3f) shall always use classic checksum.

2.2 FRAME SLOTS

Each scheduled frame allocates a slot on the bus. The duration of a slot must be long enough to carry the frame even in the worst case.

The nominal value for transmission of a frame exactly matches the number of bits sent, i.e. no response space, no byte spaces and no inter-frame space. Therefore:

$$T_{Header\_Nominal} = 34 * T_{Bit} \quad (3)$$

$$T_{Response\_Nominal} = 10 * (N_{Data} + 1) * T_{Bit} \quad (4)$$

$$T_{Frame\_Nominal} = T_{Header\_Nominal} + T_{Response\_Nominal} \quad (5)$$

where T_{Bit} is the nominal time required to transmit a bit, as defined in **LIN Physical Layer**.

The maximum space between the bytes is an additional 40% duration compared to the nominal transmission time. The additional duration is split between the frame header (the master task) and the frame response (a slave task). This yields:

$$T_{Header\_Maximum} = 1.4 * T_{Header\_Nominal} \quad (6)$$

$$T_{Response\_Maximum} = 1.4 * T_{Response\_Nominal} \quad (7)$$

$$T_{Frame\_Maximum} = T_{Header\_Maximum} + T_{Response\_Maximum} \quad (8)$$

Each frame slot shall be longer than or equal to $T_{Frame\_Maximum}$ for the frame specified.

Note 6: Eight bit sum with carry equivalent to sum all values and subtract 255 every time the sum is greater or equal to 256 (which is not the same as modulo-255 or modulo-256).

Notes

All subscribing nodes shall be able to receive a frame that has a zero overhead, i.e. that is $T_{Frame\_Nominal}$ long.

The $T_{Header\_Maximum}$ puts a requirement on the maximum length of the break symbol.

2.3 FRAME TYPES

The frame type refers to the pre-conditions that shall be valid to transmit the frame. Some of the frame types are only used for specific purposes, which will also be defined in the following subsections. Note that a node or a cluster does not have to support all frame types specified in this section.

All bits not used/defined in a frame shall be be recessive (ones).

2.3.1 Unconditional frame

Unconditional frames always carry signals and their identifiers are in the range 0 to 59 (0x3b).

The header of an unconditional frame is always transmitted when a frame slot allocated to the unconditional frame is processed (by the master task). The publisher of the unconditional frame (slave task) shall always provide the response to the header. All subscribers of the unconditional frame shall receive the frame and make it available to the application (assuming no errors were detected).

Figure 2.7 shows a sequence of three unconditional frames.

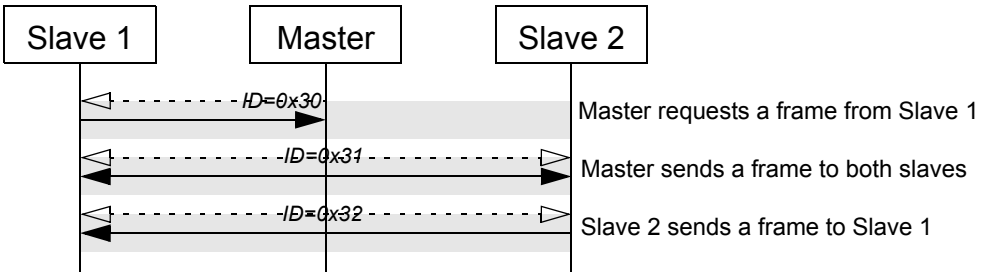


Figure 2.7: Three unconditional frame transfers. A transfer is always initiated by the master. It has a single publisher and one or multiple subscribers.

2.3.2 Event triggered frame

The purpose of an event triggered frame is to increase the responsitivity of the LIN cluster without assigning too much of the bus bandwidth to the polling of multiple slave nodes with seldom occurring events.

Event triggered frames carry the data field of one or more unconditional frames and the identifier of an event triggered frame shall be in the range 0 to 59 (0x3b). The first data byte of the carried unconditional frame shall be equal to its protected identifier. This implies that, at maximum, seven bytes of signals can be carried.

If more than one unconditional frame is associated with one event triggered frame (which is the normal case) they shall all be of equal length, use the same checksum model (i.e. mixing LIN 1.3 and LIN 2.0 frames is incorrect) and, furthermore, they shall all be published by different slave tasks.

The header of an event triggered frame is normally transmitted (the conditions are explained below) when a frame slot allocated to the event triggered frame is processed. The publisher of an associated unconditional frame **shall only** provide the response to the header **if** one of the signals carried by its frame is updated.

If none of the slave tasks respond to the header, the rest of the frame slot is silent and the header is ignored.

If more than one slave task responds to the header in the same frame slot, a collision will occur. The master has to resolve the collision by requesting all associated unconditional frames before requesting the event-triggered frame again.

If one of the colliding slave nodes withdraws without corrupting the transfer, the master will not detect this. A slave must therefore retry sending its response until successful, or the response would be lost.

All subscribers of the event triggered frame shall receive the frame and use its data (if the checksum is validated) as if the associated unconditional frame was received.

Figure 2.8 is an example of an event triggered frame sequence.

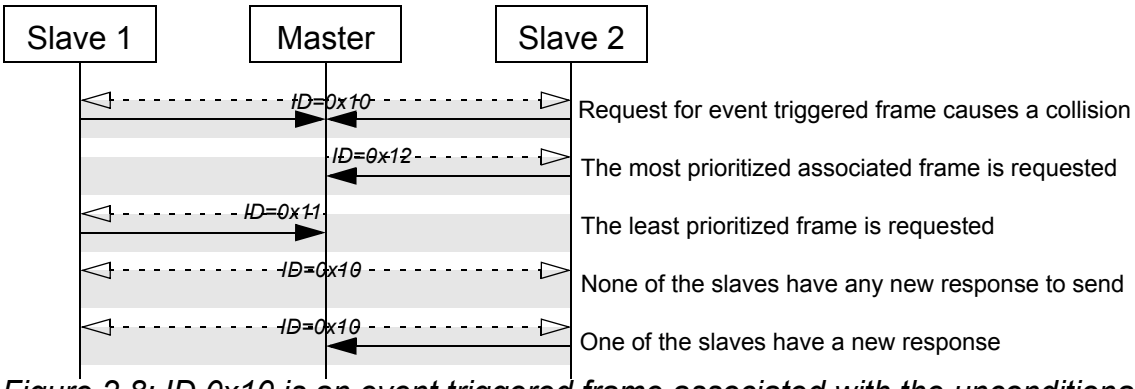


Figure 2.8: ID 0x10 is an event triggered frame associated with the unconditional frames 0x11 and 0x12. Between each of the five frame slots in the figure, other frames may be transferred, defined by the schedule table.

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

Example

A typical use for the event triggered frame is to monitor the door knobs in a four door central locking system. By using an event triggered frame to poll all four doors the system shows good response times, while still minimizing the bus load. In the rare occasion that multiple passengers presses a knob each, the system will not loose any of the pushes, but it will take some additional time.

Note

If the enhanced checksum is used for an event triggered frame, it is the protected identifier in the transferred header that shall be used in the checksum calculation.

2.3.3 Sporadic frame

The purpose of sporadic frames is to blend some dynamic behavior into the deterministic and real-time focused schedule table without loosing the determinism in the rest of the schedule table.

Sporadic frames always carry signals and their identifiers are in the range 0 to 59 (0x3b).

The header of a sporadic frame shall only be sent in its associated frame slot when the master task knows that a signal carried in the frame has been updated. The publisher of the sporadic frame shall always provide the response to the header. All subscribers of the sporadic frame shall receive the frame and use its data (if the checksum is validated).

If multiple sporadic frames are associated with the same frame slot (the normal case), the most prioritized[7] of the sporadic frames (which has an updated signal) shall be transferred in the frame slot. If none of the sporadic frames associated with a frame slot has an updated signal the frame slot shall be silent.

The requirement that the master task shall know that a carried signal has been updated makes the master node the normal publisher of sporadic frames. After a collision in an event triggered frame however, the master task is also aware of the associated unconditional frames.

Figure 2.9 is an example of an sporadic frame sequence.

Note 7: See **LIN Configuration Language Specification**, Section 2.4.3.

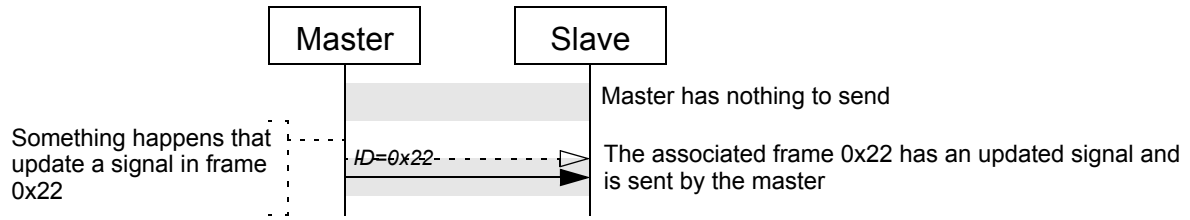


Figure 2.9: Normally sporadic frame slots are empty. In the second slot in the figure, one of the associated frames (0x22) is updated. Between the frame slots in the figure, other frames may be transferred, defined by the schedule table.

2.3.4 Diagnostic frames

Diagnostic frames always carry diagnostic or configuration data and they always contain eight data bytes. The identifier is either 60 (0x3c), called master request frame, or 61 (0x3d), called slave response frame. The interpretation of the data is given in **LIN Diagnostic and Configuration Specification**.

Before generating the header of a diagnostic frame, the master task queries its diagnostic module if it shall be sent or if the bus shall be silent. The slave tasks publish and subscribe to the response according to their diagnostic modules.

2.3.5 User-defined frames

User-defined frames carry any kind of information. Their identifier is 62 (0x3e). The header of a user-defined frame is always transmitted when a frame slot allocated to the frame is processed.

2.3.6 Reserved frames

Reserved frames shall not be used in a LIN 2.0 cluster. Their identifier is 63 (0x3f).

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

3 SCHEDULES

A key property of the LIN protocol is the use of schedule tables. Schedule table makes it possible to assure that the bus will never be overloaded. They are also the key component to guarantee the periodicy of signals.

Deterministic behavior is made possible by the fact that all transfers in a LIN cluster are initiated by the master task. It is the responsibility of the master to assure that all frames relevant in a mode of operation are given enough time to be transferred.

3.1 SLOT ALLOCATION

This section identifies all requirements that a schedule table shall adhere. The rationale for most of the requirements are to provide a conflict-free standard or to provide for a simple an efficient implementation of the LIN protocol.

An unconditional frame associated with a sporadic frame or an event triggered frame may not be allocated in the same schedule table as the sporadic frame or the event triggered frame.

A frame slot must have a duration long enough to allow for the jitter introduced by the master task and the $T_{Frame\_Maximum}$ defined in equation **(8)**. As noted just after the equation, $T_{Frame\_Maximum}$ may be reduced if the publisher supports it.

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

4 TASK BEHAVIOR MODEL

This chapter defines a behavior model for a LIN node. The behavior model is based on the master task/slave task concept. It is not necessary to implement a master node with three independent state machines or a slave node with two independent state machines, they may very well be merged into one block per node.

4.1 MASTER TASK STATE MACHINE

The master task is responsible for generating correct headers, i.e. deciding which frame shall be sent and for maintaining the correct timing between frames, all according to the schedule table. The master task state machine is depicted in **Figure 4.1**.

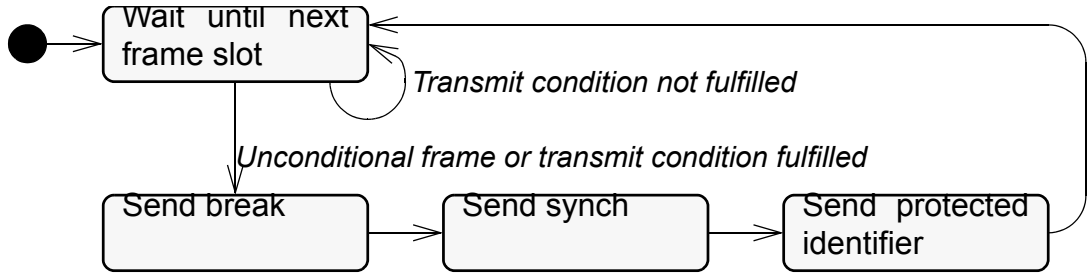


Figure 4.1: Complete state machine for the master task.

The depicted state machine does not describe how selection of identifiers for the identifier field shall be chosen.

Note

Monitoring of errors in the master task state machine is not required. Errors that might occur, e.g. a dominant bit detected when a recessive bit is sent will cause the slaves to ignore the header.

4.2 SLAVE TASK STATE MACHINE

The slave task is responsible for transmitting the frame response when it is the publisher and for receiving the frame response when it is a subscriber. The slave task is modelled with two state machines:

- Break and synch detector
- Frame processor

4.2.1 Break and synch detector

A slave task is required to be synchronized at the beginning of the protected identifier field of a frame, i.e. it must be able to receive the protected identifier field correctly. It must stay synchronized within the required bit-rate tolerance throughout the remain-

der of the frame, as specified in Section 1 in **LIN Physical Layer Specification**. For this purpose every frame starts with a sequence starting with break field followed by a synch byte field. This sequence is unique in the whole LIN communication and provides enough information for any slave task to detect the beginning of a new frame and be synchronized at the start of the identifier field.

4.2.2 Frame processor

Frame processing consists of two states, Dormant and Active. Active contains five sub-states. As soon as BreakAndSynch is signalled the Active state is entered in the Receive Identifier sub-state. This implies that processing of one frame will be aborted by the detection of a new break and synch sequence. The frame processor state machine is depicted in **Figure 4.2**.

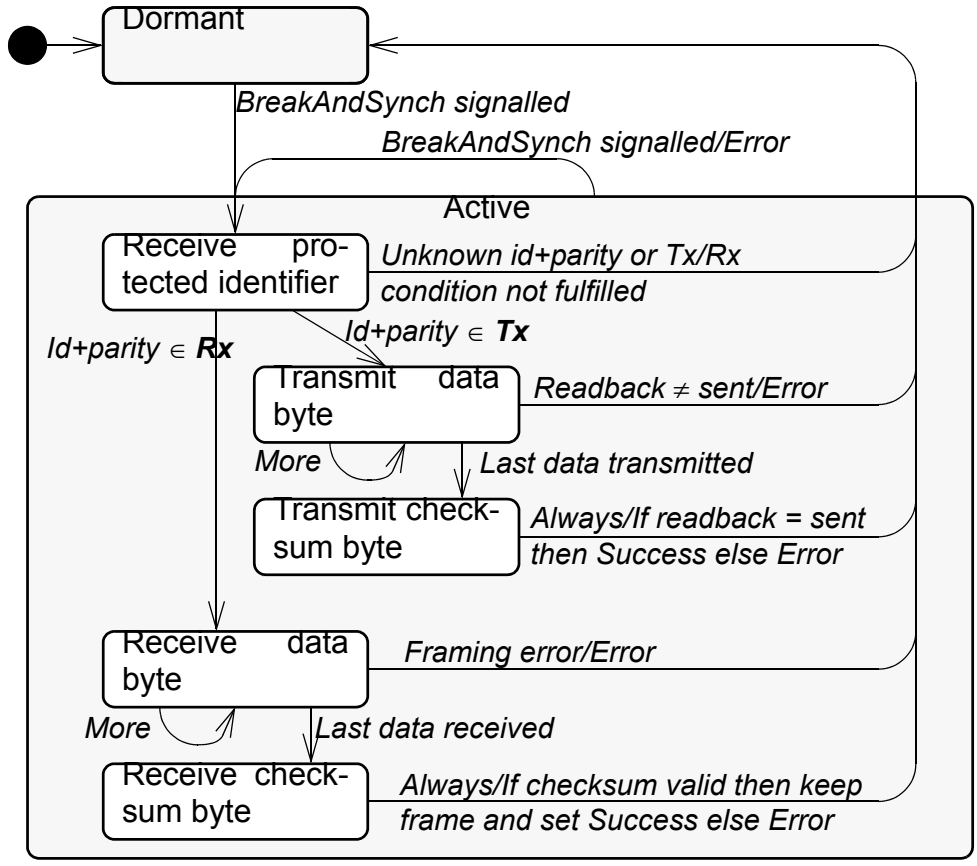


Figure 4.2: Frame processor state machine.

Error and Success refers to the status management described in Section 6.

A mismatch between readback and sent data shall be detected not later than after completion of the byte field containing the mismatch. When a mismatch is detected, the transmission shall be aborted.

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

5 NETWORK MANAGEMENT

Network management in a LIN cluster refers to cluster wake up and goto sleep only. Other network management features, e.g. configuration detection and limp home management are left to the application.

5.1 WAKE UP

Any node in a sleeping LIN cluster may request a wake up[8]. The wake-up request is issued by forcing the bus to the dominant state for 250 μs to 5 ms.

Every slave node (connected to power) shall detect the wake-up request (a dominant pulse longer than 150 μs[9]) and be ready to listen to bus commands within 100 ms, measured from the ending edge of the dominant pulse. The master shall also wake up and, when the slave nodes are ready[10], start sending frame headers to find out the cause of the wake up.

If the master does not issue frame headers within 150 ms from the wake up request, the node issuing the request may try issuing a new wake up request. After three (failing) requests the node shall wait minimum 1.5 seconds before issuing a fourth wake up request.

5.2 GOTO SLEEP

All slave nodes in an active cluster can be forced into sleep mode[11] by sending a diagnostic master request frame (frame identifier = 0x3c) with the first data byte equal to zero[12]. This special use of a diagnostic frame is called a go-to-sleep-command.

Slave nodes shall also automatically enter a sleep mode if the LIN bus is inactive[13] for more than 4 seconds.

Note 8: The master may issue a break symbol, e.g. by issuing an ordinary frame header since the break will act as a wake up pulse.
Note 9: A detection threshold of 150 μs combined with a 250 μs pulse generation gives a detection margin that is enough for uncalibrated slave nodes.
Note 10: This may take 100 ms (from the wake up) unless the master has additional information, e.g. only one slave in the cluster may be the cause of the wake up. Obviously, this slave is immediately ready to communicate.
Note 11: Sleep covers the cluster only, the application in the node may still be active.
Note 12: The first data byte is normally interpreted as the node address, NAD, and the address zero is not allowed.
Note 13: Inactive is defined as: No transitions between recessive and dominant bit values occur.

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

5.3 POWER MANAGEMENT

The state diagram in **Figure 5.1** shows the behavior model for power management of a LIN node. The LIN protocol behavior specified in this document only applies to the Operational state.

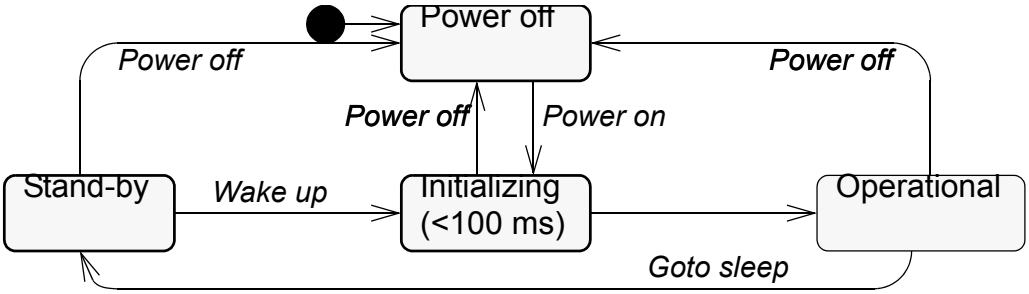


Figure 5.1: LIN node power management.

6 STATUS MANAGEMENT

The purpose of status management is to detect errors during operation. The purpose of detecting errors is twofold;

- to provide means to easily replace faulty units and
- to provide for nodes to enter a limp home mode when problems occur.

In addition to the status management function mandated in this chapter, a node may provide further detailed error information, although this is not standardized by the specification.

6.1 CONCEPT

Central cluster status management is made in the master node. The master node monitors status reports from each node and filters/integrates the reports to conclude if one or more nodes are faulty.

Each node application may also monitor its interaction with the LIN bus. This can be used to enter a limp home mode, if applicable.

6.2 EVENT TRIGGERED FRAMES

Event triggered frames, Section 2.3.2, are defined to allow collisions. Therefore, a bus error, e.g. framing error, shall not affect the status bits (it is neither a successful transfer, nor an error in response). Of course, if an error in the associated unconditional frame occurs, this shall be counted as an error.

6.3 REPORTING TO THE NETWORK

Reporting to the network is intended for processing by the master node and is used to monitor the cluster. Only slave nodes are required to report their status to the network.

Each slave shall send one status bit signal, Response_Error, to the master node in one of its transmitted frames. Response_Error shall be set whenever a frame received by the node or a frame transmitted by the node contains an error in the response field. Response_Error shall be cleared after transmission.

Based on this single bit the master node can conclude the following:

Response_Error = False ⇨ the node is operating correctly

Response_Error = True ⇨ the node has intermittent problems

the node did not answer ⇨ the node (or bus or master) has serious problems.

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

Since the master node also receives information from the fact that a frame is not sent, it comes that the Response_Error status bit can not be placed in an event triggered frame. Apart from this restriction, any frame transmitted by the node can be used to carry the Response_Error status bit.

It is the responsibility of the master node application to integrate and filter the individual status reports as well as to do a synthesis of the reports from different slave nodes[14].

Note

The response_error is enough to perform a conformance test of the frame transceiver (the protocol engine) independent of the application and the signal interaction layer.

A slave node may provide more status information, if desired, but the single response_error bit shall always be present.

6.4 REPORTING WITHIN OWN NODE

This section applies to software based nodes, however ASIC based state machine implementations are recommended to use the same concepts.

The node provides two status bits for status management within the own node; error_in_response and successful_transfer. The own node application also receives the protected identifier of the last frame recognized by the node.

Error_in_response is set whenever a frame received by the node or a frame transmitted by the node contains an error in the response field, i.e. by the same condition as will set the Response_Error signal.

Successful_transfer shall be set when a frame has been successfully transferred by the node, i.e. a frame has either been received or transmitted.

Both status bits are cleared after reading.

The two status bits can be interpreted according to **Table 6.1**.

Table 6.1: Node internal error interpretation.

error in response	successful transfer	Interpretation
0	0	No communication
1	1	Intermittent communication (some successful transfers and some failed)
0	1	Full communication
1	0	Erroneous communication (only failed transfers)

Note 14: For example, if multiple nodes do not answer, the master may conclude that he is the faulty node, not all the slaves.

It is the responsibility of the node application to integrate and filter the individual status reports.

Note

The reporting within the own node is standardized in the **LIN API Specification** and can be used to automatically generate applications that perform an automatic conformance test of the complete LIN driver module, including the signal interaction layer.

7 APPENDICES

7.1 TABLE OF NUMERICAL PROPERTIES

Table 7.1: Defined numerical properties

Property	Min	Max	Unit	Reference	Remark
Scalar signal size	1	16	bit	Section 1.1	
Byte array size	1	8	byte	Section 1.1	
Break length	13		T_{bit}	Section 2.1.1	
Break detect threshold	11	11	T_{bit}	Section 2.1.1	See also footnote 3
Wake up request duration	0.25	5	ms	Section 5.1	
Slave initialize time		100	ms	Section 5.1	
Silence period between wake-up requests	150		ms	Section 5.1	
Silence period after three wake-ups	1.5		s	Section 5.1	

7.2 TABLE OF VALID IDENTIFIERS

ID[0..5] Dec	ID[0..5] Hex	P0 = ID0⊕ID1⊕ID2⊕ID4	P1 = ¬ ID1⊕ID3⊕ID4⊕ID5	ID-Field 7 6 5 4 3 2 1 0	ID-Field Dec	ID-Field Hex
0	0x00	0	1	1 0 0 0 0 0 0 0	128	0x80
1	0x01	1	1	1 1 0 0 0 0 0 1	193	0xC1
2	0x02	1	0	0 1 0 0 0 0 1 0	66	0x42
3	0x03	0	0	0 0 0 0 0 0 1 1	3	0x03
4	0x04	1	1	1 1 0 0 0 1 0 0	196	0xC4
5	0x05	0	1	1 0 0 0 0 1 0 1	133	0x85
6	0x06	0	0	0 0 0 0 0 1 1 0	6	0x06
7	0x07	1	0	0 1 0 0 0 1 1 1	71	0x47
8	0x08	0	0	0 0 0 0 1 0 0 0	8	0x08
9	0x09	1	0	0 1 0 0 1 0 0 1	73	0x49
10	0x0A	1	1	1 1 0 0 1 0 1 0	202	0xCA
11	0x0B	0	1	1 0 0 0 1 0 1 1	139	0x8B
12	0x0C	1	0	0 1 0 0 1 1 0 0	76	0x4C
13	0x0D	0	0	0 0 0 0 1 1 0 1	13	0x0D
14	0x0E	0	1	1 0 0 0 1 1 1 0	142	0x8E
15	0x0F	1	1	1 1 0 0 1 1 1 1	207	0xCF
16	0x10	1	0	0 1 0 1 0 0 0 0	80	0x50
17	0x11	0	0	0 0 0 1 0 0 0 1	17	0x11
18	0x12	0	1	1 0 0 1 0 0 1 0	146	0x92
19	0x13	1	1	1 1 0 1 0 0 1 1	211	0xD3
20	0x14	0	0	0 0 0 1 0 1 0 0	20	0x14
21	0x15	1	0	0 1 0 1 0 1 0 1	85	0x55

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

ID[0..5] Dec	ID[0..5] Hex	P0 = ID0⊕ID1⊕ID2⊕ID4	P1 = ¬ ID1⊕ID3⊕ID4⊕ID5	ID-Field 7 6 5 4 3 2 1 0	ID-Field Dec	ID-Field Hex
22	0x16	1	1	1 1 0 1 0 1 1 0	214	0xD6
23	0x17	0	1	1 0 0 1 0 1 1 1	151	0x97
24	0x18	1	1	1 1 0 1 1 0 0 0	216	0xD8
25	0x19	0	1	1 0 0 1 1 0 0 1	153	0x99
26	0x1A	0	0	0 0 0 1 1 0 1 0	26	0x1A
27	0x1B	1	0	0 1 0 1 1 0 1 1	91	0x5B
28	0x1C	0	1	1 0 0 1 1 1 0 0	156	0x9C
29	0x1D	1	1	1 1 0 1 1 1 0 1	221	0xDD
30	0x1E	1	0	0 1 0 1 1 1 1 0	94	0x5E
31	0x1F	0	0	0 0 0 1 1 1 1 1	31	0x1F
32	0x20	0	0	0 0 1 0 0 0 0 0	32	0x20
33	0x21	1	0	0 1 1 0 0 0 0 1	97	0x61
34	0x22	1	1	1 1 1 0 0 0 1 0	226	0xE2
35	0x23	0	1	1 0 1 0 0 0 1 1	163	0xA3
36	0x24	1	0	0 1 1 0 0 1 0 0	100	0x64
37	0x25	0	0	0 0 1 0 0 1 0 1	37	0x25
38	0x26	0	1	1 0 1 0 0 1 1 0	166	0xA6
39	0x27	1	1	1 1 1 0 0 1 1 1	231	0xE7
40	0x28	0	1	1 0 1 0 1 0 0 0	168	0xA8
41	0x29	1	1	1 1 1 0 1 0 0 1	233	0xE9
42	0x2A	1	0	0 1 1 0 1 0 1 0	106	0x6A
43	0x2B	0	0	0 0 1 0 1 0 1 1	43	0x2B
44	0x2C	1	1	1 1 1 0 1 1 0 0	236	0xEC
45	0x2D	0	1	1 0 1 0 1 1 0 1	173	0xAD
46	0x2E	0	0	0 0 1 0 1 1 1 0	46	0x2E
47	0x2F	1	0	0 1 1 0 1 1 1 1	111	0x6F
48	0x30	1	1	1 1 1 1 0 0 0 0	240	0xF0
49	0x31	0	1	1 0 1 1 0 0 0 1	177	0xB1
50	0x32	0	0	0 0 1 1 0 0 1 0	50	0x32
51	0x33	1	0	0 1 1 1 0 0 1 1	115	0x73
52	0x34	0	1	1 0 1 1 0 1 0 0	180	0xB4
53	0x35	1	1	1 1 1 1 0 1 0 1	245	0xF5
54	0x36	1	0	0 1 1 1 0 1 1 0	118	0x76
55	0x37	0	0	0 0 1 1 0 1 1 1	55	0x37
56	0x38	1	0	0 1 1 1 1 0 0 0	120	0x78
57	0x39	0	0	0 0 1 1 1 0 0 1	57	0x39
58	0x3A	0	1	1 0 1 1 1 0 1 0	186	0xBA
59	0x3B	1	1	1 1 1 1 1 0 1 1	251	0xFB
60[a]	*0x3C*	*0*	*0*	*0 0 1 1 1 1 0 0*	*60*	*0x3C*
61[b]	*0x3D*	*1*	*0*	*0 1 1 1 1 1 0 1*	*125*	*0x7D*

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

ID[0..5] Dec	ID[0..5] Hex	P0 = ID0⊕ID1⊕ID2⊕ID4	P1 = ¬ ID1⊕ID3⊕ID4⊕ID5	ID-Field 7 6 5 4 3 2 1 0	ID-Field Dec	ID-Field Hex
62[c]	0x3E	1	1	1 1 1 1 1 1 1 0	254	0xFE
63[d]	0x3F	0	1	1 0 1 1 1 1 1 1	191	0xBF

a. Identifier 60 (0x3C) is reserved for the Master Request command frame (see Section 2.3.4).
b. Identifier 61 (0x3D) is reserved for the Slave Response command frame (see Section 2.3.4).
c. Identifier 62 (0x3E) is reserved for the user-defined extended frame (see Section 2.3.5).
d. Identifier 63 (0x3F) is reserved for a future LIN extended format (see Section 2.3.6).

7.3 EXAMPLE OF CHECKSUM CALCULATION

Below is the checksum calculation of four bytes shown. If the bytes are four data bytes or the protected identifier and three data bytes is not important; the calculation is the same.

```
Data = 0x4A, 0x55, 0x93, 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 sum is 0x19. Inversion yields the final result: checksum = 0xE6.

The receiving node can easily check the consistency of the received frame by using the same addition mechanism. When the received checksum (0xE6) is added to the intermediate result (0x19) the sum shall be 0xFF.

7.4 SYNTAX AND MATHEMATICAL SYMBOLS USED IN THIS STANDARD

Sequence diagrams

To visualize the implications of the standard, sequence diagrams are used when appropriate. The syntax used in these diagrams are exemplified in **Figure 7.1**.

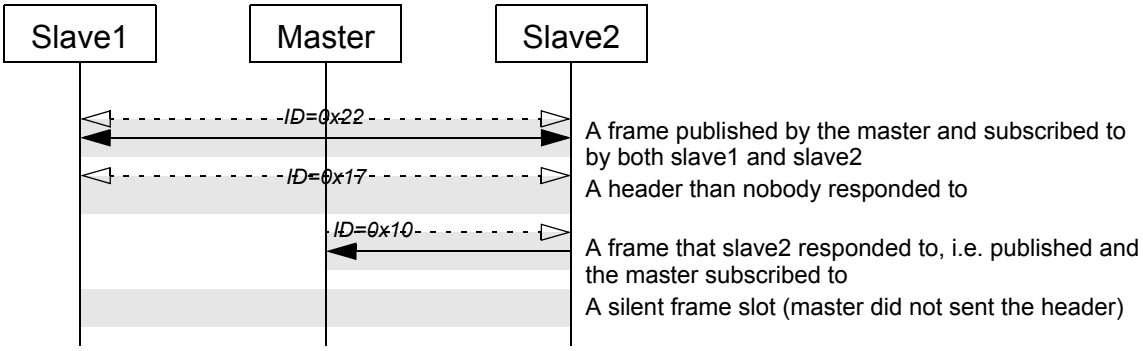


Figure 7.1: Frame sequence example. The shaded areas represent the frame slots (with gaps added to clarify the drawing). Dotted/hollow arrows represent the headers and solid arrows represent responses.

Mathematical symbols

The following mathematical symbols and notations are used in this standard:

$f \in \mathbf{S}$	Belongs to. True if f is in the set **S**.
$a \oplus b$	Exclusive or. True if exactly one of a and b is true.
$\neg a$	Negate. True if a is false.

LIN
Diagnostic and Configuration Specification

Revision 2.0

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 diagnostic and configuration standard defines how a node shall be configured (which is mandatory for all nodes to implement) and three alternative methods to implement diagnostic data gathering (all of them are optional to implement).

The diagnostic and configuration data is transported by the LIN protocol as specified in the **LIN Protocol Specification**. A standardized API for the C programming language is specified in the **LIN API Specification**.

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

2 NODE CONFIGURATION

LIN node configuration is used to set up LIN slave nodes in a cluster. It is a tool to avoid conflicts between slave nodes within a cluster built out of off-the-shelf nodes.

Configuration is done by having a large address space, consisting of a message identifier per frame, a LIN Product Identification per slave node and an initial NAD per slave node. Using these numbers it is possible to map unique frame identifiers to all frames transported in the cluster.

It is mandatory for a LIN node to support node configuration.

2.1 NODE MODEL

The memory of a slave node can be described as in **Figure 2.1**.

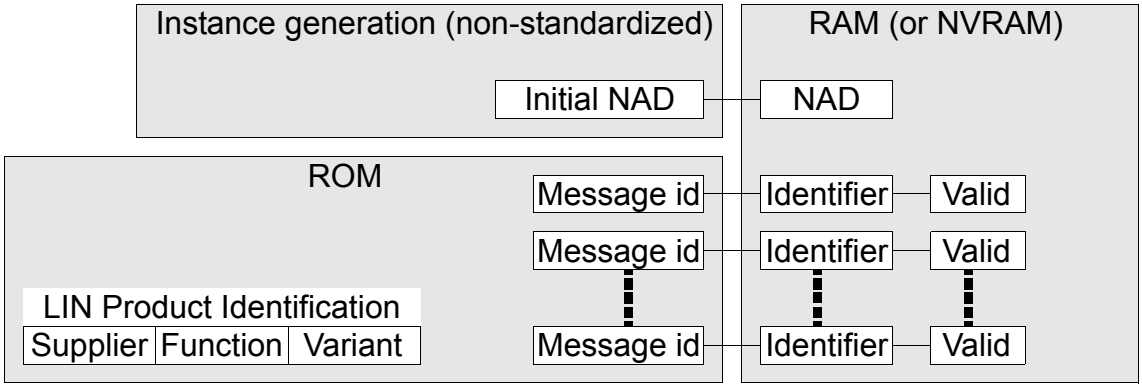


Figure 2.1: Slave node memory model.

A slave node has a fixed LIN Product Identification, see Section 2.4, and message identifiers for all frames.

After reset[1], a slave node shall be in the following state:

- It shall have a NAD set equal to a dynamically determined instance value (the initial NAD) starting with the first instance being 1, the second 2, etc. The method for determinating the instance number is not part of the LIN standard[2].
- It shall have all protected identifiers marked as invalid.

Frames with identifiers 0x3c and above have fixed (and valid) identifiers and do not have any message identifiers.

Note 1: If the configuration is saved in NVRAM at power off, the power on is not considered a reset.
Note 2: One possibility is to use jumpers that configure the instance.

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

2.2 WILDCARDS

To be able to leave some information unspecified the wildcard values in **Table 2.1** may be used in node configuration requests:

Table 2.1: Wildcards usable in all requests

Property	Wildcard value
NAD	127
Supplier ID	0x7FFF
Function ID	0xFFFF
Message ID	0xFFFF

Implementation of Supplier ID wildcard, Function ID wildcard and Message ID wildcard is optional.

2.3 PDU STRUCTURE

The information in this section is a subset of the information provided in Section 3.3.1. The reason for this is that this subset is mandatory for a LIN node while the superset in Section 3.3.1 is optional.

The units that are transported in a LIN diagnostic frame are called PDU (Packet Data Unit). A PDU used for node configuration is be a complete message.

Messages issued by the client (ISO: tester, LIN: master) are called requests and messages issued by the server (ISO: master, LIN: slave) are called responses.

Flow control (as defined in ISO [1]) is not used in LIN clusters. If the back-bone bus test equipment needs flow control PDUs, these must be generated by the master node.

2.3.1 Overview

To simplify conversion between ISO diagnostic frames, [1], and LIN diagnostic frames a very similar structure is defined, which support the PDU types shown in **Figure 2.2**.

Request	NAD	PCI	SID	D1	D2	D3	D4	D5	PCI-type = SF
Response	NAD	PCI	RSID	D1	D2	D3	D4	D5	PCI-type = SF

Figure 2.2: PDUs supported by LIN configuration. The left byte (NAD) is sent first and the right byte (D5) is sent last.

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

Requests are always sent in master request frames and responses are always sent in slave response frames. The meaning of each byte in the PDUs is defined in the following sections.

2.3.2 NAD

NAD is the address of the slave node being addressed in a request, i.e. only slave nodes have an address. NAD is also used to indicate the source of a response.

NAD values are in the range 1 to 127, while 0 and 128 to 255 are reserved for other purposes:

0	Reserved for go-to-sleep-command, see **LIN Protocol Specification**
1 - 126 (0x7e)	Diagnostic slave node addresses
127 (0x7f)	Reserved for broadcast
128 (0x80) - 255 (0xff)	Free usage, the frame is not to be interpreted as a diagnostic frame[3]. Also, see Section 3.2.

Note that there is a one-to-one mapping between a physical node and a logical node and it is addressed using the NAD.

2.3.3 PCI

The PCI (Protocol Control Information) contains the transport layer flow control information. For node configuration, one interpretation of the PCI byte exist, as defined in **Table 2.2**.

Table 2.2: Structure of the PCI byte for configuration PDUs.

Type	PCI type				Additional information			
	B7	B6	B5	B4	B3	B2	B1	B0
SF	0	0	0	0	Length			

The PCI type Single Frame (SF) indicates that the transported message fits into the single PDU, i.e. it contains at maximum five data bytes. The length shall then be set to the number of used data bytes plus one (for the SID or RSID).

2.3.4 SID

The Service Identifier (SID) specifies the request that shall be performed by the slave node addressed. 0xb0 to 0xb4 are used for configuration. This SID numbering is consistent with ISO 15765-3 and places node configuration in an area “Defined by vehicle manufacturer”.

Note 3: Diagnostic frames with the first byte in the range 128 (0x80) to 255 (0xff) are allocated for free usage since the LIN 1.2 standard.

2.3.5 RSID

The Response Service Identifier (RSID) specifies the contents of the response. The RSID for a positive response is always SID + 0x40.

2.3.6 D1 to D5

The interpretation of the data bytes (up to five in a node configuration PDU) depends on the SID or RSID.

If a PDU is not completely filled the unused bytes shall be filled with ones, i.e. their value shall be 255 (0xff). This is necessary since a diagnostic frame is always eight bytes long.

2.4 LIN PRODUCT IDENTIFICATION

Each LIN part shall have a unique number, as outlined in **Table 2.3**.

Table 2.3: LIN product identification

D1	D2	D3	D4	D5
Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB	Variant ID

The supplier ID is assigned to each supplier by the LIN Consortium. It is a 16 bit value, with the most significant bit equal to zero[4].

The function ID is assigned by each supplier. If two products differ in function, i.e. LIN communication or physical world interaction, their function ID shall differ. For absolutely equal function, however, the function ID shall remain unchanged.

Finally, the variant ID shall be changed whenever the product is changed but with unaltered function.

Incorporation of the LIN product identification into the LIN node is mandatory.

2.5 MANDATORY REQUESTS

Requests listed in this section shall be supported by all LIN slave nodes.

2.5.1 Assign frame identifier

Assign frame id is used to set a valid protected identifier to a frame specified by its message identifier. It shall be structured as shown in **Table 2.4**.

It is important to notice that the request provides the protected identifier, i.e. the identifier and its parity. Furthermore, frames with identifier 60 (0x3c) and up can not be changed (diagnostic frames, user-defined frames and reserved frames).

Note 4: Most significant bit set to one is reserved for future extended numbering systems.

Table 2.4: Assign frame id request

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb1	Supplier ID LSB	Supplier ID MSB	Message ID LSB	Message ID MSB	Protected ID

A response shall only be sent if the NAD and the Supplier ID match. If successful, the message in **Table 2.5** shall be sent as a response. Implementation of a response is optional.

Table 2.5: Positive assign frame id response

NAD	PCI	RSID	Unused				
NAD	0x01	0xf1	0xff	0xff	0xff	0xff	0xff

2.5.2 Read by identifier

It is possible to read the supplier identity and other properties from a slave node using the request in **Table 2.6**.

A response shall only be sent if the NAD, the Supplier ID and the Function ID match. A positive response shall be as shown in **Table 2.8**. If the request fails, i.e. sub-function not supported, the negative response in **Table 2.9** shall be sent.

Table 2.6: Read by identifier request

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb2	Identifier	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB

The identifiers defined are listed in **Table 2.7**.

Table 2.7: Identifiers that may be read using read by identifier request.

Identifier	Interpretation	Length of response
0	LIN Product Identification	5 + RSID
1	Serial number	4 + RSID
2 - 15	Reserved	Negative response; Sub-function not supported.
16 - 31	Message ID 1..16	3 bytes + RSID
32 - 63	User defined	User defined
64 - 255	Reserved	Reserved

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

Table 2.8: Possible positive read by identifier response. Each row represents one possible response.

Id	NAD	PCI	RSID	D1	D2	D3	D4	D5
0	NAD	0x06	0xf2	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB	Variant
1	NAD	0x05	0xf2	Serial 0, LSB	Serial 1	Serial 2	Serial 3, MSB	0xff
	Reserved							
16	NAD	0x04	0xf2	Message ID 1 LSB	Message ID 1 MSB	Protected ID (or FF)	0xff	0xff
17	NAD	0x04	0xf2	Message ID 2 LSB	Message ID 2 MSB	Protected ID (or FF)	0xff	0xff
18	NAD	0x04	0xf2	Message ID 3 LSB	Message ID 3 MSB	Protected ID (or FF)	0xff	0xff
19	NAD	0x04	0xf2	Message ID 4 LSB	Message ID 4 MSB	Protected ID (or FF)	0xff	0xff
20	NAD	0x04	0xf2	Message ID 5 LSB	Message ID 5 MSB	Protected ID (or FF)	0xff	0xff
21	NAD	0x04	0xf2	Message ID 6 LSB	Message ID 6 MSB	Protected ID (or FF)	0xff	0xff
22	NAD	0x04	0xf2	Message ID 7 LSB	Message ID 7 MSB	Protected ID (or FF)	0xff	0xff
23	NAD	0x04	0xf2	Message ID 8 LSB	Message ID 8 MSB	Protected ID (or FF)	0xff	0xff
24	NAD	0x04	0xf2	Message ID 9 LSB	Message ID 9 MSB	Protected ID (or FF)	0xff	0xff
25	NAD	0x04	0xf2	Message ID 10 LSB	Message ID 10 MSB	Protected ID (or FF)	0xff	0xff
26	NAD	0x04	0xf2	Message ID 11 LSB	Message ID 11 MSB	Protected ID (or FF)	0xff	0xff
27	NAD	0x04	0xf2	Message ID 12 LSB	Message ID 12 MSB	Protected ID (or FF)	0xff	0xff
28	NAD	0x04	0xf2	Message ID 13 LSB	Message ID 13 MSB	Protected ID (or FF)	0xff	0xff
29	NAD	0x04	0xf2	Message ID 14 LSB	Message ID 14 MSB	Protected ID (or FF)	0xff	0xff
30	NAD	0x04	0xf2	Message ID 15 LSB	Message ID 15 MSB	Protected ID (or FF)	0xff	0xff
31	NAD	0x04	0xf2	Message ID 16 LSB	Message ID 16 MSB	Protected ID (or FF)	0xff	0xff

Table 2.9: Negative response.

NAD	PCI	RSID	D1	D2	Unused		
NAD	0x03	0x7f	Requested SID (= 0xb2)	Error code (= 0x12)	0xff	0xff	0xff

Notes

Support of Identifier 0 (LIN Product Identification) is the only mandatory identifier, i.e. the serial number and message IDs are optional.

Identifiers 32 - 63 are user defined and therefore not listed in **Table 2.8**.

2.6 OPTIONAL REQUESTS

Requests listed in this section are optional and may be implemented in a LIN slave node. The requests are optional because their cost impact is not always motivated. Still, they are specified in order to encourage a uniform solution to the problem they intend to solve. That is, solving the same task as the following requests in a different way is recommended.

2.6.1 Assign NAD

Assign NAD is used to resolve conflicting node addresses. It shall be structured as shown in **Table 2.10**.

A response shall only be sent if the NAD, the Supplier ID and the Function ID match. If successful, the message in **Table 2.11** shall be sent as response. Implementation of the response is optional.

Table 2.10: Assign NAD request

NAD	PCI	SID	D1	D2	D3	D4	D5
Initial NAD	0x06	0xb0	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB	New NAD

Table 2.11: Positive assign NAD response

NAD	PCI	RSID	Unused				
Initial NAD	0x01	0xf0	0xff	0xff	0xff	0xff	0xff

Note

This service always uses the initial NAD; this is to avoid the risk of losing the address of a node. The NAD used for the response shall be the same as in the request, i.e. the initial NAD.

2.6.2 Conditional change NAD

The conditional change NAD is used to detect and separate unknown slave nodes in a cluster. Potential reasons for unknown nodes to appear in a cluster are, e.g. incorrect assembly when manufacturing the cluster or incorrect node replacement during service. The conditional change NAD is intended to detect the node and allow the master node to report a diagnostic message describing the problem.

The behaviour of the request is:

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. Get the identifier specified in **Table 2.8** and selected by Id.
2. Extract the data byte selected by Byte (Byte = 1 corresponds to the first byte, D1).
3. Do a bitwise XOR with Invert.
4. Do a bitwise AND with Mask.
5. If the final result is zero then change the NAD to New NAD.

Table 2.12: Conditional change NAD request

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb3	Id	Byte	Mask	Invert	New NAD

Table 2.13: Optional positive Conditional change NAD response

NAD	PCI	RSID	Unused				
NAD	0x01	0xf3	0xff	0xff	0xff	0xff	0xff

Note

Note that Conditional Change NAD is addressed with the present NAD, i.e. it does not always use the initial NAD as opposed to the Assign NAD request.

2.6.3 Data dump

Note

The SID = 0xb4 is reserved for initial configuration of a node by the node supplier and the format of this message is supplier specific. Hence, this SID shall not be used in operational LIN clusters.

Table 2.14: Data dump request

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xb4	User defined	User defined	User defined	User defined	User defined

Table 2.15: Data dump response

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xf4	User defined	User defined	User defined	User defined	User defined

3 DIAGNOSTICS

Three methods for gathering diagnostic information are specified below. The methods may coexist in the same cluster or even in the same node. Implementation of any of the methods listed herein is optional.

3.1 SIGNAL BASED DIAGNOSTICS

The simplest form of diagnostic information gathering is to use ordinary signals in normal, unconditional frames. The characteristics of this solution are:

- Very low overhead in slave nodes.
- Standardized in concept (since it uses normal signals/frames).
- Inflexible since the data contents is fixed in the frame structure.

3.2 USER DEFINED DIAGNOSTICS

It is possible to use the free range of diagnostic frames. The free range of diagnostic frames must all have the first data byte in the range 128 (0x80) to 255 (0xff), see Section 2.3.2. The characteristics of a solution based on the free range diagnostics are:

- Non-standardized and, hence, non-portable.
- Reasonable in overhead since the design is made specifically to fit the needs.

Since the user defined diagnostics is not standardized, the signal based diagnostics is the preferred solution.

3.3 DIAGNOSTICS TRANSPORT LAYER

Use of the LIN diagnostic transport layer is targeting systems where ISO diagnostics are performed on the (CAN-based) back-bone bus and where the system builder wants to use the same diagnostic capabilities on the LIN sub-bus clusters. The messages are in fact identical to the ISO messages and the PDUs carrying the messages are very similar, as defined in Section 3.3.1. A typical system configuration is shown in **Figure 3.1**.

The goals of the LIN diagnostic transport layer are:

- Low load on master.
- Providing full (or a subset thereof) ISO diagnostics directly on the LIN slaves.
- Targeting clusters built with powerful nodes (not the mainstream low-cost LIN).

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

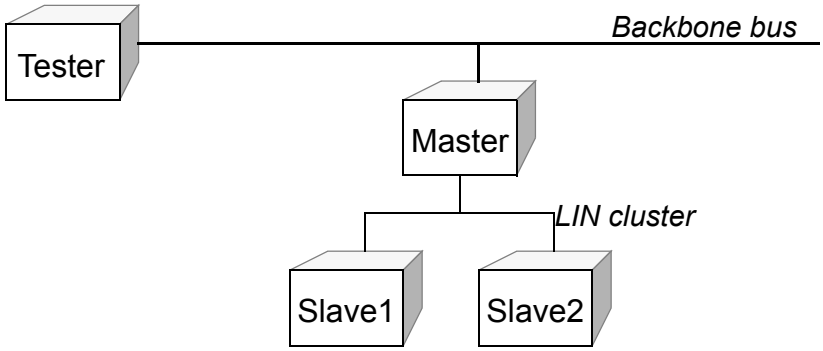


Figure 3.1: Typical system setup for a LIN cluster using the transport layer.

3.3.1 PDU structure

The information in this section is a superset of the information provided in Section 2.3.

The units that are transported in a LIN diagnostic frame are called PDU (Packet Data Unit). A PDU can be a complete message or a part of a message; in the latter case, multiple concatenated PDUs form the complete message.

Messages issued by the client (ISO: tester, LIN: master) are called requests and messages issued by the server (ISO: master, LIN: slave) are called responses.

Flow control (as defined in ISO [1]) is not used in LIN clusters. If the back-bone bus test equipment needs flow control PDUs, these must be generated by the master node.

Overview

To simplify conversion between ISO diagnostic frames, [1], and LIN diagnostic frames a very similar structure is defined, which support the PDU types shown in **Figure 3.2**.

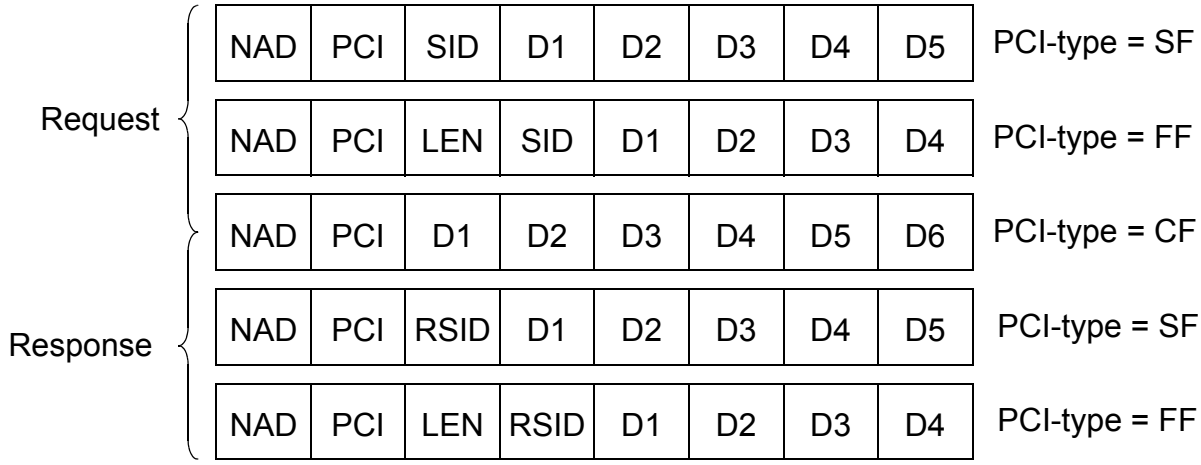


Figure 3.2: PDUs supported by the LIN diagnostic transport layer. The left byte (NAD) is sent first and the right byte (D4, D5 or D6) is sent last.

Requests are always sent in master request frames and responses are always sent in slave response frames. The meaning of each byte in the PDUs is defined in the following sections.

NAD

The NAD is defined in Section 2.3.2.

PCI

The PCI (Protocol Control Information) contains the transport layer flow control information. Three interpretations of the PCI byte exist, as defined in **Table 3.1**.

Table 3.1: Structure of the PCI byte.

Type	PCI type				Additional information			
	B7	B6	B5	B4	B3	B2	B1	B0
SF	0	0	0	0	Length			
FF	0	0	0	1	Length/256			
CF	0	0	1	0	Frame counter			

The PCI type Single Frame (SF) indicates that the transported message fits into the single PDU, i.e. it contains at maximum five data bytes. The length shall then be set to the number of used data bytes plus one (for the SID or RSID).

The PCI type First Frame (FF) is used to indicate the start of a multi PDU message; the following frames are of CF type, see below. The total number of data bytes in the message plus one (for the SID or RSID) shall be transmitted as Length: The four most significant bits of Length is transmitted in the PCI byte (the eight least significant bits are sent in LEN, see below).

A multi-PDU message is continued with a number of Continuation Frames (CF). The first CF frame of a message is numbered 1, the second 2 and so on. If more than 15 CF PDUs are needed to transport the complete message, the frame counter wraps around and continues with 0, 1,...

LEN

A LEN byte is only used in FF frames; it contains the eight least significant bits of the message Length. Thus, the maximum length of a message is 4095 (0xfff) bytes.

SID

The Service Identifier (SID) specifies the request that shall be performed by the slave node addressed. 0 to 0xaf and 0xb8 to 0xfe are used for diagnostics while 0xb0 to 0xb7 are used for node configuration. This SID numbering is consistent with ISO 15765-3 and places node configuration in an area “Defined by vehicle manufacturer”.

RSID

The Response Service Identifier (RSID) specifies the contents of the response.

D1 to D6

The interpretation of the data bytes (up to six in a single PDU) depends on the SID or RSID. In multi-PDU messages, all the bytes in all PDUs of the message shall be concatenated into a complete message, before being parsed.

If a PDU is not completely filled (applies to CF and SF PDUs only) the unused bytes shall be filled with ones, i.e. their value shall be 255 (0xff).

3.3.2 Defined requests

The LIN transport layer uses the same diagnostic messages as the ISO diagnostics standard, [2]. From this follows that SID and RSID shall also be according to the ISO standard. A node may implement a sub-set of the services defined in the ISO standard.

3.3.3 ISO timing constraints

The timing used in ISO [1] [2] is based on several properties, e.g. P2, ST and T1. The properties shall be within a defined range. Since LIN is slower than CAN, the values have to be adjusted accordingly.

The values to use for these properties are not part of the LIN standard. They are controlled by selecting a schedule table that has the desired period between the diagnostic frames. This way, the values are under full control of the cluster developer and may be set according to the relevant trade-offs.

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

3.3.4 Sequence diagrams

Two communication cases exist; the tester wants to send a diagnostic request to a slave node and the slave node wants to send a diagnostic response to the tester. Below, **Figure 3.3** and **Figure 3.4** shows the message flow in these two cases.

It is important that the unit orchestrating the communication (the tester or the master) avoids requesting multiple slaves to respond simultaneously (as this would cause bus collisions).

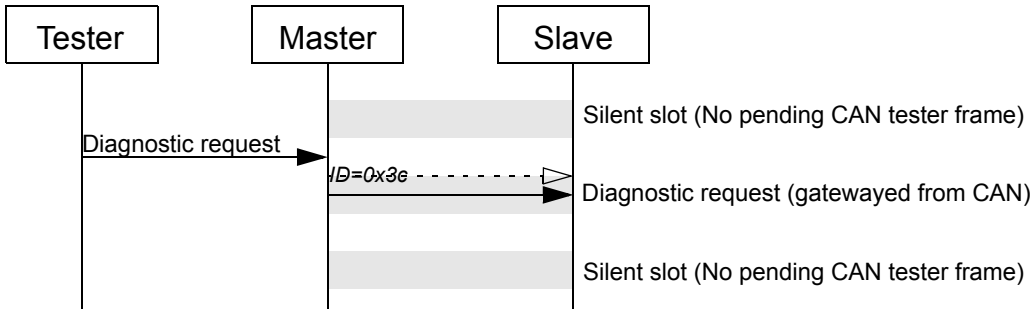


Figure 3.3: Gatewaying of CAN messages to LIN.

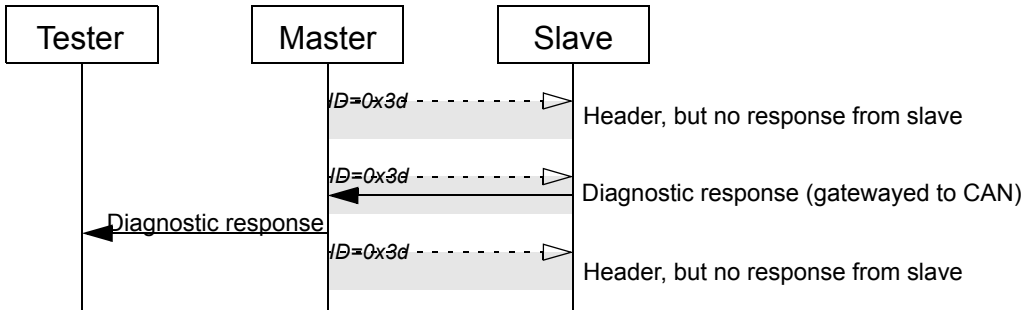


Figure 3.4: Gatewaying of LIN messages to CAN.

4 REFERENCES

[1] “Road vehicles - Diagnostics on Controller Area Network (CAN) - Part 2: Network layer services“, *International Standard ISO 15765-2.4*, Issue 4, 2002-06-21

[2] “Road vehicles - Diagnostics on controller area network (CAN) - Part 3: Implementation of diagnostic services”, *International Standard ISO 15765-3.5*, Issue 5, 2002-12-12.

LIN
Physical Layer Specification

Revision 2.0

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 OSCILLATOR TOLERANCE

On-chip clock generators can achieve a frequency tolerance of better than ± 14% 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 allow for accurate bite rate measurement and generation for the remainder of the message frame, taking into account effects of anything, which affects oscillator frequency, such as temperature and voltage drift during operation.

no.	clock tolerance	Name	$\Delta F / F_{Nom.}$
1.1.1	master node (deviation from nominal clock rate. The nominal clock rate **F_{Nom}** is defined in the LIN Description File).	$F_{TOL\_RES\_MASTER}$	< ±0.5%
1.1.2	slave node without making use of synchronization (deviation from nominal clock rate) Note: For communication between any two nodes their bit rate must not differ by more than ±2%.	$F_{TOL\_RES\_SLAVE}$	< ±1.5%
1.1.3	deviation of slave node clock from the nominal clock rate before synchronization; relevant for nodes making use of synchronization and direct SYNCH BREAK detection.	$F_{TOL\_UNSYNCH}$	< ±14%

Table 1.1: Oscillator Tolerances relative to nominal Clock

no.	clock tolerance	Name	$\Delta F / F_{Master}$
1.2.1	deviation of slave node clock relative to the master node clock after synchronization; relevant for nodes making use of synchronization; any slave node must stay within this tolerance for all fields of a frame which follow the SYNCH FIELD. Note: For communication between any two nodes their bit rate must not differ by more than ±2%.	F_{TOL_SYNCH}	< ±2%

Table 1.2: Slave Oscillator Tolerance relative to Master Node

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

2 BIT TIMING REQUIREMENTS AND SYNCHRONIZATION PROCEDURE

2.1 BIT TIMING REQUIREMENTS

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

2.2 SYNCHRONIZATION PROCEDURE

The SYNCH FIELD consists of the data ’0x55’ inside a byte field. 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}.

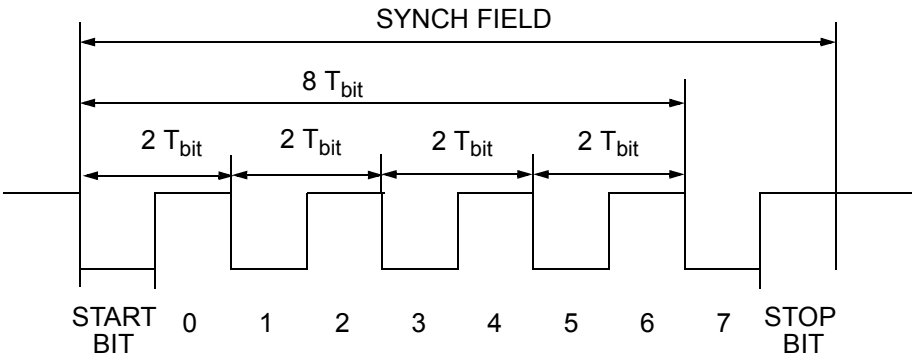


Figure 2.1: SYNCHRONIZATION FIELD

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.

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

3 LINE DRIVER/RECEIVER

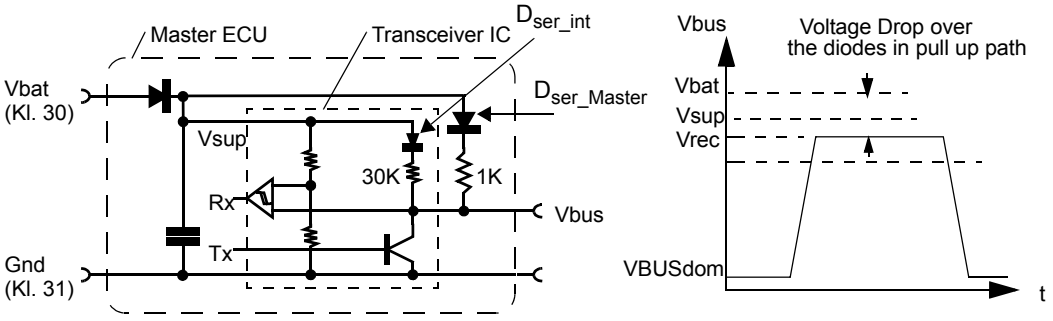
3.1 GENERAL CONFIGURATION

The bus line driver/receiver is an enhanced implementation of the ISO 9141 standard [1]. 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 3.1**). The diode is mandatory to prevent an uncontrolled power-supply 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.

3.2 DEFINITION OF SUPPLY VOLTAGES FOR THE PHYSICAL INTERFACE

V_{BAT} denotes the supply voltage at the connector of the control unit. Electronic components within the unit may see an internal supply V_{SUP} being different from V_{BAT} (see **Figure 3.1**). 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 3.1: Illustration of the Difference between External Supply Voltage V_{BAT} and the Internal Supply Voltage V_{SUP}

3.3 SIGNAL SPECIFICATION

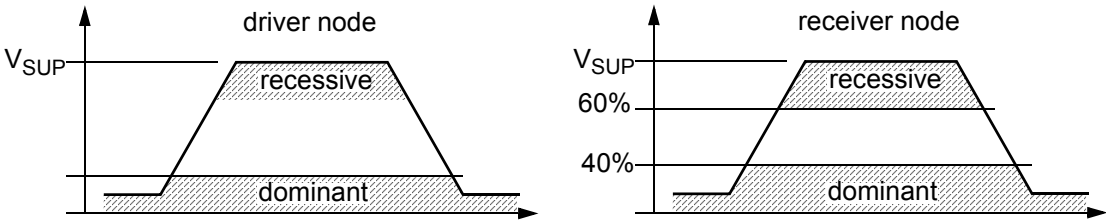


Figure 3.2: Voltage Levels on the Bus Line

For a correct transmission and reception of a bit it has to made sure that the signal is available with the correct voltage level (dominant or recessive) at the bit sampling time of the receiver. Ground shifts as well as drops in the supply voltage have to be taken into consideration as well as symmetry failures in the propagation delay. **Figure 3.3** shows the timing parameters that impact the behaviour of the LIN Bus.
The minimum and maximum values of the different parameters are listed in the following tables.

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

Timing diagram:

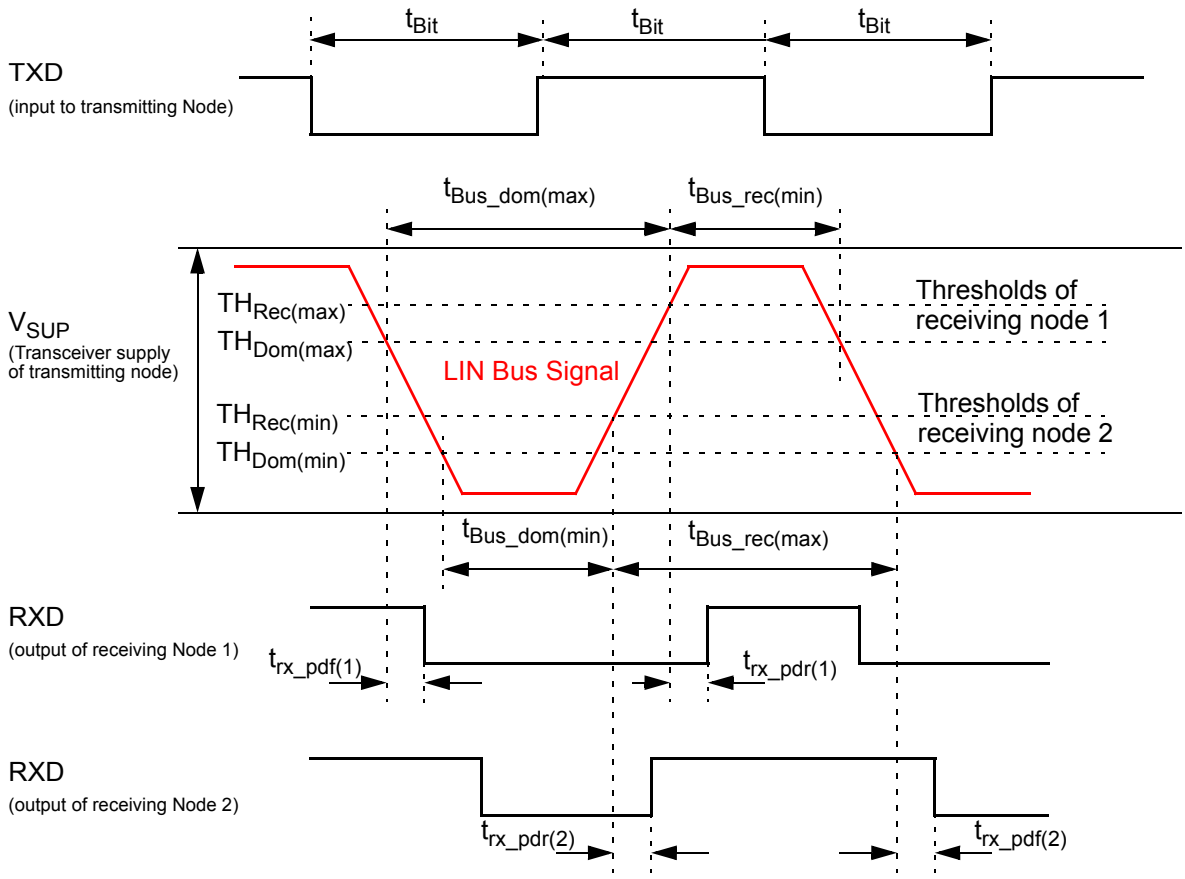


Figure 3.3: Definition of bus timing parameters

3.4 ELECTRICAL DC PARAMETERS

The electrical DC parameters of the LIN Physical Layer and the termination resistors are listed in **Table 3.1** and **Table 3.2**, 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.

no.	parameter	min.	typ.	max.	unit	comment / condition
3.1.1	V_{BAT} [a]	8		18	V	operating voltage range
3.1.2	V_{SUP} [b]	7.0		18	V	supply voltage range
3.1.3	$V_{SUP\_NON\_OP}$	-0.3		40	V	voltage range within which the device is not destroyed
3.1.4	I_{BUS_LIM} [c]	40		200	mA	Current Limitation for Driver dominant state driver on $V_{BUS} = V_{BAT\_max}$ [d]
3.1.5	$I_{BUS\_PAS\_dom}$	-1			mA	Input Leakage Current at the Receiver incl. Pull-Up Resistor as specified in **Table 3.2** driver off V_{BUS} = 0V V_{BAT}= 12V
3.1.6	$I_{BUS\_PAS\_rec}$			20	µA	driver off $8V<V_{BAT}<18V$ $8V<V_{BUS}<18V$ $V_{BUS} \geq V_{BAT}$
3.1.7	$I_{BUS\_NO\_GND}$	-1		1	mA	Control unit disconnected from ground $GND_{Device} = V_{SUP}$ $0V<V_{BUS}<18V$ V_{BAT} = 12V Loss of local ground must not affect communication in the residual network.

Table 3.1: Electrical DC Parameters of the LIN Physical Layer

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

no.	parameter	min.	typ.	max.	unit	comment / condition
3.1.8	I_{BUS}			100	µA	V_{BAT} disconnected V_{SUP_Device} = GND $0<V_{BUS}<18V$ Node has to sustain the current that can flow under this condition. Bus must remain operational under this condition.
3.1.9	V_{BUSdom}			0.4	V_{SUP}	receiver dominant state
3.1.10	V_{BUSrec}	0.6			V_{SUP}	receiver recessive state
3.1.11	V_{BUS_CNT}	0.475	0.5	0.525	V_{SUP}	$V_{BUS\_CNT} =(V_{th\_dom}+ V_{th\_rec})/2$ [e]
3.1.12	V_{HYS}			0.175	V_{SUP}	$V_{HYS} = V_{th\_rec} - V_{th\_dom}$
3.1.13	$V_{SerDiode}$	0.4	0.7	1.0	V	Voltage Drop at the serial Diodes D_{ser_Master} and D_{ser_int} in pull up path (**Figure 3.1**).
3.1.14	V_{Shift_BAT}	0		10%	V_{BAT}	V_{BAT}-Shift
3.1.15	V_{Shift_GND}	0		10%	V_{BAT}	GND-Shift

Table 3.1: 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 Section 3.2).
b. V_{SUP} denotes the supply voltage at the transceiver inside the control unit and may be different from the external supply V_{BAT} for control units (see Section 3.2).
c. I_{BUS}: Current flowing into the node.
d. A transceiver must be capable to sink at least 40mA. The maximum current flowing into the node must not exceed 200mA under DC conditions to avoid possible damage.
e. V_{th_dom}: receiver threshold of the recessive to dominant LIN bus edge.
V_{th_rec}: receiver threshold of the dominant to recessive LIN bus edge.

no.	parameter	min.	typ.	max.	unit	comment
3.2.1	R_{master}	900	1000	1100	Ω	The serial diode is mandatory (**Figure 3.1**).
3.2.2	R_{slave}	20	30	60	KΩ	The serial diode is mandatory.

Table 3.2: Parameters of the Pull-Up Resistors

Note:

All parameters are defined for the ambient temperature range from -40°C to 125°C.

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

3.5 ELECTRICAL AC PARAMETERS

The electrical AC parameters of the LIN Physical Layer are listed in **Table 3.3**, **Table 3.4**, and **Table 3.5**, with the parameters being defined in **Figure 3.3**. The electrical AC-Characteristics of the bus can be strongly affected by the line characteristics as shown in Section 3.3. The time constant τ (and thus the overall capacitance) of the bus (Section 3.6) has to be selected carefully in order to allow for a correct signal implementation under worst case conditions.

The following table (**Table 3.3**) specifies the timing parameters for proper operation at 20.0 kBit/sec.

no.	parameter	min.	typ.	max.	unit	comment / condition
LIN Driver, Bus load conditions (C_{BUS} ; R_{BUS}): 1nF; 1kΩ / 6,8nF;660Ω / 10nF;500Ω						
3.3.1	D1 (Duty Cycle 1)	0.396				$TH_{Rec(max)} = 0.744 \times V_{SUP}$; $TH_{Dom(max)} = 0.581 \times V_{SUP}$; V_{SUP} = 7.0V...18V; t_{Bit} = 50µs; D1 = $t_{Bus\_rec(min)}$ / (2 x t_{Bit})
3.3.2	D2 (Duty Cycle 2)			0.581		$TH_{Rec(min)} = 0.284 \times V_{SUP}$; $TH_{Dom(min)} = 0.422 \times V_{SUP}$; V_{SUP} = 7.6V...18V; t_{Bit} = 50µs; D2 = $t_{Bus\_rec(max)}$ / (2 x t_{Bit})

Table 3.3: Driver Electrical AC Parameters of the LIN Physical Layer (20kBit/s)

For improved EMC performance, exception is granted for speeds of 10.4 kBit/sec or below. For details see the following table (**Table 3.4**), which specifies the timing parameters for proper operation at 10.4 kBit/sec.

no.	parameter	min.	typ.	max.	unit	comment / condition
LIN Driver, Bus load conditions (C_{BUS} ; R_{BUS}): 1nF; 1kΩ / 6,8nF;660Ω / 10nF;500Ω						
3.4.1	D3 (Duty Cycle 3)	0.417				$TH_{Rec(max)} = 0.778 \times V_{SUP}$; $TH_{Dom(max)} = 0.616 \times V_{SUP}$; V_{SUP} = 7.0V...18V; t_{Bit} = 96µs; D3 = $t_{Bus\_rec(min)}$ / (2 x t_{Bit})
3.4.2	D4 (Duty Cycle 4)			0.590		$TH_{Rec(min)} = 0.251 \times V_{SUP}$; $TH_{Dom(min)} = 0.389 \times V_{SUP}$; V_{SUP} = 7.6V...18V; t_{Bit} = 96µs; D4 = $t_{Bus\_rec(max)}$ / (2 x t_{Bit})

Table 3.4: Driver Electrical AC Parameters of the LIN Physical Layer (10.4kBit/s)

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

Application specific implementations (ASICs) shall meet the parameters in Table 3.3 and/or Table 3.4. If both sets of parameters are implemented, the proper mode shall be selected based on the bus bit rate.

no.	parameter	min.	typ.	max.	unit	comment / condition
LIN Receiver, RXD load condition (C_{RXD}): 20pF; (if open drain behaviour: $R_{pull-up}$ = 2.4kΩ)						
3.5.1	t_{rx_pd}			6	µs	propagation delay of receiver
3.5.2	t_{rx_sym}	-2		2	µs	symmetry of receiver propagation delay rising edge w.r.t. falling edge

Table 3.5: Receiver Electrical AC Parameters of the LIN Physical Layer

The EMC behavior of the LIN bus depends on the signal shape represented by slew rate and other factors such as di/dt and d^2V/dt^2. The signal shape should be carefully selected in order to reduce emissions on the one hand and allow for bit rates up to 20 kBit/sec on the other.

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

3.6 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 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 **(3.6.1)** as

Equation 3.6.1: $$C_{BUS} = C_{MASTER} + n \cdot C_{SLAVE} + C'_{LINE} \cdot LEN_{BUS}$$

the RC time constant τ is calculated by **(3.6.2)** as

Equation 3.6.2: $$\tau = C_{BUS} \cdot R_{BUS}$$

with

Equation 3.6.3: $$R_{BUS} = R_{Master} \parallel R_{Slave1} \parallel R_{Slave2} \parallel ... \parallel R_{Slave\_n}$$

under consideration of the parameters given in **Table 3.6**.

			min	typ.	max	unit
3.6.1	total length of bus line	LEN_{BUS}			40	m
3.6.2	total capacitance of the bus including slave and master capacitances	C_{BUS}	1	4	10	nF
3.6.3	time constant of overall system	τ	1		5	µs
3.6.4	capacitance of master node	C_{MASTER}		220		pF
3.6.5	capacitance of slave node	C_{SLAVE}		220	250	pF
3.6.6	line capacitance	C'_{LINE}		100	150	pF/m

Table 3.6: Line Characteristics and Parameters.

C_{MASTER} and C_{SLAVE} are defining the total node capacitance at the connector of an ECU including the physical bus driver (Transceiver) and all other components applied to the LIN bus pin like capacitors or protection circuitry.

The number of nodes in a LIN cluster should not exceed 16.[1]

Note 1: The network impedance may prohibit a fault free communication under worst case conditions with more than 16 nodes. Every additional node lowers the network resistance by approximately 3% (30 kΩ || ~1 kΩ).

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

3.7 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.

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

LIN
Application Program Interface Specification

Revision 2.0

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

A LIN device driver may be implemented in hardware or in software. In the latter case, a multitude of languages exist to chose from. It is also possible to integrate the LIN device driver directly with the application.

This document defines a mandatory interface to a software LIN device driver implemented in the C programming language. Thus, hardware implementations are not standardized nor are implementations in other programming languages. If LIN device drivers appear in other languages, e.g. Ada, they are encouraged to use the concepts presented in this document, although the syntax will be different.

The API is split in two sections; the LIN core API and the LIN diagnostic API, since the diagnostic features are optional in LIN.

The behavior covered by the LIN core API is defined in the **LIN Protocol Specification** and the behavior of the LIN diagnostic API is covered in the **LIN diagnostic and Configuration Specification**.

1.1 CONCEPT OF OPERATION

1.1.1 System generation

The LDF file (see **LIN Configuration Language Specification**) is parsed by a tool and the API and driver module is generated. This is called system generation.

Since the LDF file only concerns LIN cluster aspects, more information may be needed in the system generation process, but this is not part of the LIN standard[1].

1.1.2 API

LIN core API

The LIN core API uses a signal based interaction between the application and the LIN core. This implies that the application does not have to bother with frames and transmission of frames. Tools exist to detect transfer of a specific frame if this is necessary, see Section 2.3. Of course, API calls to control the LIN core also exist.

Two versions exist of most of the API calls; static routines and dynamic routines. The static routines embed the name of the signal or interface in the routine name, while the dynamic routines provide this as a parameter. The choice between the two is a matter of taste.

Note 1: LIN development tool vendors are free to implement this to fit their own tool chains.

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

LIN node configuration API

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

LIN diagnostic transport layer API

The LIN diagnostic transport layer is also message based but its intended use is to work as a transport layer for messages to a diagnostic message parser outside of the LIN device driver, typically a ISO diagnostic module. Two exclusively alternative APIs exist, one raw that allows the application to control the contents of every frame sent and one “cooked” that performs the full transport layer function.

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

2 CORE API

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

The LIN core shall 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 static functions (no parameters are needed, since one function exist per signal, per interface, etc.).

2.1 DRIVER AND CLUSTER MANAGEMENT

2.1.1 l_sys_init

Prototype

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

Description

l_sys_init performs the initialization of the LIN core.

Returns

Zero if the initialization succeeded and
Non-zero if the initialization failed

Notes

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

2.2 SIGNAL INTERACTION

2.2.1 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

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

2.2.2 Scalar signal read

Dynamic prototype

```
l_bool l_bool_rd (l_signal_handle sss);
l_u8 l_u8_rd (l_signal_handle sss);
l_u16 l_u16_rd (l_signal_handle sss);
```

Static 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.

2.2.3 Scalar signal write

Dynamic prototype

```
void l_bool_wr (l_signal_handle sss, l_bool v);
void l_u8_wr (l_signal_handle sss, l_u8 v);
void l_u16_wr (l_signal_handle sss, l_u16 v);
```

Static 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.

2.2.4 Byte array read

Dynamic prototype

```
void l_bytes_rd (l_signal_handle sss,
                 l_u8            start,  /* first byte to read from    */
                 l_u8            count,  /* number of bytes to read    */
                 l_u8* const     data);  /* where data will be written */
```

Static implementation

```
void l_bytes_rd_sss (l_u8            start,
                     l_u8            count,
                     l_u8* const     data);
```

Where sss is the name of the signal, e.g. l_bytes_rd_EngineSpeed (..).

Description

Reads and returns the current value of the selected bytes in the signal specified by the name sss.

Assume that a byte array is 6 bytes long, numbered 0 to 5. Reading byte 2 and 3 from this array requires start to be 2 (skipping byte 0 and 1) and count to be 2 (reading byte 2 and 3). In this case byte 2 is written to data[0] and byte 3 is written to data[1].

The sum of start and count shall never be greater than the length of the byte array, although the device driver may choose not to enforce this in runtime.

2.2.5 Byte array write

Dynamic prototype

```
void l_bytes_wr (l_signal_handle   sss,
                 l_u8              start,  /* first byte to write to   */
                 l_u8              count,  /* number of bytes to write */
                 const l_u8* const data);  /* where data is read from  */
```

Static implementation

```
void l_bytes_wr_sss (l_u8              start,
                     l_u8              count,
                     const l_u8* const data);
```

Where sss is the name of the signal, e.g. l_bytes_wr_EngineSpeed (..).

Description

Sets the current value of the selected bytes in the signal specified by the name sss to the value specified.

Assume that a byte array is 7 bytes long, numbered 0 to 6. Writing byte 3 and 4 from this array requires start to be 3 (skipping byte 0, 1 and 2) and count to be 2 (writing byte 3 and 4). In this case byte 3 is read from data[0] and byte 4 is read from data[1].

The sum of start and count shall never be greater than the length of the byte array, although the device driver may choose not to enforce this in runtime.

2.3 NOTIFICATION

Flags are local objects in a node and they are used to synchronize the application program with the LIN core. The flags will be automatically set by the LIN core and can only be tested or cleared by the application program.

2.3.1 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.

Note

The flag is set when the associated object (signal or frame) is updated by the LIN module.

2.3.2 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.

2.4 SCHEDULE MANAGEMENT

2.4.1 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 schedule.

The l_sch_tick must be called individually for each interface within the node, 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 if the next call of l_sch_tick will not start transmission of a frame.

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.

2.4.2 l_sch_set

Dynamic prototype

```
void l_sch_set (l_ifc_handle      iii,
                l_schedule_handle schedule,
                l_u8              entry);
```

Static implementation

```
void l_sch_set_iii (l_schedule_handle schedule, l_u8 entry);
```

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

Description

Sets up the next schedule 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 entry defines the starting entry point in the new schedule table. The value should be in the range 0 to N if the schedule table has N entries, and if entry 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.

A possible use of the entry value is in combination with the l_sch_tick return value to temporarily interrupt one schedule with another schedule table, and still be able to switch back to the interrupted schedule table at the point where this was interrupted.

A predefined schedule table, L_NULL_SCHEDULE, shall exist and may be used to stop all transfers on the LIN cluster.

2.5 INTERFACE MANAGEMENT

2.5.1 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 initializes 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 the L_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 ().

2.5.2 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 cluster 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

2.5.3 l_ifc_disconnect

Dynamic prototype

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

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

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 cluster and thus disable the interaction with the other nodes in the cluster.

Returns

Zero if the "disconnect operation" was successful and
non-zero if the "disconnect operation" failed.

2.5.4 l_ifc_goto_sleep

Dynamic prototype

```
void l_ifc_goto_sleep (l_ifc_handle iii);
```

Static implementation

```
void l_ifc_goto_sleep_iii (void);
```

Where iii is the name of the interface, e.g. l_ifc_goto_sleep_MyLinIfc ().

Description

l_ifc_goto_sleep commands all slave nodes on the cluster connected to the interface to enter sleep mode by issuing the special go-to-sleep-mode-command, see also Section 2.5.10.

Notes

l_ifc_goto_sleep may only be used in the master node.

2.5.5 l_ifc_wake_up

Dynamic prototype

```
void l_ifc_wake_up (l_ifc_handle iii);
```

Static implementation

```
void l_ifc_wake_up_iii (void);
```

Where iii is the name of the interface, e.g. l_ifc_wake_up_MyLinIfc ().

Description

The call transmits a 0xf0 byte on the LIN bus, i.e. a dominant pulse of between 250 μs and 5 ms (depending on the configured bit rate). See also Section 5.1 in **LIN Protocol Specification**.

2.5.6 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.

2.5.7 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

The function shall be called when the interface iii has received one character of data.

It may, for example, be called from a user-defined interrupt handler triggered 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).

2.5.8 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

The function shall be called when the interface iii has transmitted one character of data.

It may, for example, be called from a user-defined interrupt handler triggered 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.

2.5.9 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 synchronize to the BREAK and SYNC characters sent by the master on the interface specified by iii.

It may, for example, be called from a user-defined interrupt handler raised upon a flank detection on a hardware pin connected to the interface iii.

Notes

l_ifc_aux may only be used in a slave node.

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

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

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

2.5.10 l_ifc_read_status

Dynamic prototype

```
l_u16 l_ifc_read_status (l_ifc_handle iii);
```

Static implementation

```
l_u16 l_ifc_read_status_iii(void);
```

Where iii is the name of the interface, e.g. l_ifc_read_status_MyLinIfc ().

Description

The call returns a 16 bit value, as shown in **Table 2.1**.

Table 2.1: Return value of l_ifc_read_status (bit 15 is MSB, bit 0 is LSB).

15	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Last frame protected identity								0	0	0	0	Goto sleep	Overrun	Successful transfer	Error in response

Error in response is set if one (or multiple) frames processed by the node had an error in the frame response section, e.g. checksum error, framing error, etc., since the previous call to l_ifc_read_status.[2]

Successful transfer is set if one (or multiple) frame responses have been processed without an error since the previous call to l_ifc_read_status.

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

Goto sleep is set if a a go-to-sleep-mode-command has been received since the previous call to l_ifc_read_status.

Last frame protected identity is the protected identity last detected on the bus **and** processed in the node. If overrun is set one or more values of last frame protected identity are lost; only the latest value is maintained.

Note 2: An error in the header results in the header not being recognized and thus, the frame is ignored.

Note

A LIN slave node is required to enter sleep mode after 4 seconds of silence on the bus. This can be implemented by the application monitoring the status bits; one second of continuous zero readings imply bus silence[3].

Example

The l_ifc_read_status is designed to allow reading at a much lower frequency than the frame slot frequency, e.g. once every 50 frame slots. In this case, the last frame protected identity has little use. Overrun is the used as a check that the traffic is running as it should, i.e. is shall always be set.

It is, however, also possible to call l_ifc_read_status every frame slot and get a much better error statistics; you can see the protected identifier of the failing transfers and by knowing the topology, it is possible to draw better conclusion of faulty nodes. This is maybe most useful in the master node, but is also possible in any slave node.

The provided information, especially in conjunction with the error_response signals described in **LIN Protocol Standard** (Section 6.3), provides for very detailed car OEM specific logging of faulty nodes or wiring.

Implementation notes

Calling l_ifc_read_status twice without any frame transfer in between shall return zero in the second call.

Successful transfer shall be set after processing and verifying the checksum of the frame.

Error in response shall be set when an error is detected in the frame response processing.

Last frame protected identity shall be set simultaneously with successful transfer or error in response.

Overrun shall be set if either of successful transfer or error in response is already set and the driver needs to set one of them.

An implementation for a master node may set goto sleep when the API call is issued, when the go-to-sleep-mode-command has been sent or not at all[4].

Note 3: This implies that the master must communicate with all slave nodes at least once a second; if nothing else is needed, it should at least poll the receive_error status bit.

Note 4: The reason for this flexibility is that the master node implementation shall not be forced to receive its own transmissions.

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

2.6 USER PROVIDED CALL-OUTS

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

2.6.1 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.

2.6.2 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 the provided l_irqmask previous.

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

3 NODE CONFIGURATION

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

A requirement for LIN node configuration to operate correctly is that the active schedule table contains the two diagnostic frames (master request frame and slave response frame) in sequence at least once. If the master does not care about the responses (not recommended) it is enough that the master request frame is contained in the schedule table.

Note
All calls in the LIN node configuration API are for the master node only. Slave nodes manage node configuration automatically.

Any need for initialization shall be automatically performed by the LIN core module call l_sys_init.

3.0.1 ld_is_ready

Dynamic prototype

```
l_bool ld_is_ready (l_ifc_handle iii);
```

Description
This routine returns true if the diagnostic module for the interface specified is ready for a new command. This also implies that the previous command has completed, e.g. the response is valid and may be analyzed.

Notes
The call is available in the master only.

You may never issue another node configuration API call unless the previous call has completed, i.e. ld_is_ready has returned true.

Implementation notes
ld_is_ready shall be cleared by any imperative call in the API (all calls except ld_is_ready or ld_check_response).

ld_is_ready shall be set when the following master request frame and slave response frame has been completed (checksum of slave response frame received). If the master request frame is not followed by a slave response frame or if the slave does not respond, ld_is_ready shall also be set when the next frame slot is being processed.

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

In the last case, the module shall also memorize that the request failed (in case of the application calling ld_check_response).

3.0.2 ld_check_response

Dynamic prototype

```
l_u8 ld_check_response (l_ifc_handle iii,
                        l_u8*        RSID,
                        l_u8*        error_code);
```

Description

This routine returns the result of the last node configuration call completed. RSID and error_code is also returned for a more detailed analysis. The result is interpreted as follows:

LD_SUCCESS	The call was successfully carried out.
LD_NEGATIVE	The call failed, more information can be found by parsing error_code.
LD_NO_RESPONSE	No response was received on the request.
LD_OVERWRITTEN	The slave response frame has been overwritten by another operation, i.e. the result is lost[5].

Notes

The call is available in the master only.

Implementation note

If the slave does not respond the routine may not respond LD_SUCCESS. However, the values of RSID and error_code may be return a successful reply. (This allows for the routine to read RSID and error_code directly from the response RAM buffer.)

3.0.3 ld_assign_NAD

Dynamic prototype

```
void ld_assign_NAD (l_ifc_handle iii,
                    l_u8         NAD,
                    l_u16        supplier_id,
                    l_u16        function_id,
                    l_u8         new_NAD);
```

Note 5: This can only occur if the cluster uses both node configuration and the diagnostic transport layer, see Section 4.

Description

This call assigns the NAD (node diagnostic address) of all slave nodes that matches the NAD, the supplier identity code and the function identity code. The new NAD of those nodes will be new_NAD.

Notes

The call is available in the master only.

LD_BROADCAST, LD_ANY_SUPPLIER and/or LD_ANY_FUNCTION may be used in this call (assuming all nodes in the cluster have unique supplier/function id).

The purpose of this call is to change conflicting NADs in LIN clusters built using off-the-shelves nodes or reused nodes.

3.0.4 ld_assign_frame_id

Dynamic prototype

```
void ld_assign_frame_id (l_ifc_handle iii,
                         l_u8         NAD,
                         l_u16        supplier_id,
                         l_u16        message_id,
                         l_u8         PID);
```

Description

This call assigns the protected identifier of a frame in the slave node with the address NAD and the specified supplier ID. The frame changed shall have the specified message ID and will after the call have PID as the the protected identifier.

Notes

The call is available in the master only.

LD_BROADCAST, LD_ANY_SUPPLIER and/or LD_ANY_MESSAGE may be used in this call (assuming all nodes in the cluster have unique supplier/function id).

3.0.5 ld_read_by_id

Dynamic prototype

```
void ld_read_by_id (l_ifc_handle iii,
                    l_u8         NAD,
                    l_u16        supplier_id,
                    l_u16        function_id,
                    l_u8         id,
                    l_u8* const  data);
```

Description

The call requests the node selected with the NAD to return the property associated with the Id parameter. When the next call to ld_is_ready returns true, the RAM area specified by data contains between one and five bytes data according to the request.

Notes

The call is available in the master only.

Table 3.1 shows the possible values for Id.

Table 3.1: Id that can be read using ld_read_by_id.

Id	Interpretation
0	LIN Product Identification
1	Serial number
2 - 15	Reserved
16 - 31	Message ID 1..16
32 - 63	User defined
64 - 255	Reserved

Implementation note

The result is returned in a big-endian style. It is up to little-endian CPUs to swap the bytes, not the LIN diagnostic driver. (The reason for using big-endian data is to simplify message routing to a CAN back-bone network.)

3.0.6 ld_conditional_change_NAD

Dynamic prototype

```
void ld_conditional_change_NAD (l_ifc_handle iii,
                                l_u8         NAD,
                                l_u8         id,
                                l_u8         byte,
                                l_u8         mask,
                                l_u8         invert,
                                l_u8         new_NAD);
```

Description

This call changes the NAD if the node properties fulfil the test specified by id, byte, mask and invert, see **LIN Diagnostic Specification**.

Id shall be in the range 0 to 31, see **Table 3.1**, and byte in the range 1 to 5 (specifying the byte to use in the id). Mask and Invert shall have values between 0 and 255.

Notes

The call is available in the master only.

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

4 DIAGNOSTIC TRANSPORT LAYER

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

Use of the LIN diagnostic transport layer API requires knowledge of the underlaying protocol. The relevant information can be found in the **LIN Diagnostic and Configuration Specification**.

LIN diagnostic transport layer is intended to transport ISO diagnostic requests/responds between a test equipment on a back-bone CAN network to LIN slave nodes via the master node.

Since ISO PDUs on CAN are quite similar to LIN diagnostic frames, a raw API is provided. The raw API is frame/PDU based and it is up to the application to manage the PCI information. The features are that this is simple when the source is CAN based ISO PDUs and that the API requires very little resources (RAM and CPU cycles).

An alternative API is message based; provide a pointer to a message buffer and the transfer commences and the LIN diagnostic driver will do the packing/unpacking, i.e. act as a transport layer. Typically, this is useful in slave nodes since they shall not gateway the messages but parse them.

Note

The behavior of the system is undefined in the pathologic case where the application tries to process a single frame using both the raw and the cooked API.

4.1 RAW API

Implementation note

The raw API is based on transferring PDUs and it is typically used to gateway PDUs between CAN and LIN. If the speed of the networks differ, a FIFO function on transmit as well as receive is useful. All implementations of the LIN diagnostic module are encouraged to incorporate such FIFOs and to make their size configurable at system generation time.

4.1.1 ld_put_raw

Dynamic prototype

```
void ld_put_raw (l_ifc_handle       iii,
                 const l_u8* const  data);
```

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

Description
The call queues the transmission of the eight bytes specified byte data.

Notes
The data is sent in the next suitable frame (master request frame for master nodes and slave response frame for slave nodes).

Implementation note
The data area must be copied in the call, it is not allowed to memorize the pointer only.

If no more queue resources are available, the data may be jettisoned.

4.1.2 ld_get_raw

Dynamic prototype
```
void ld_get_raw (l_ifc_handle iii,
                 l_u8* const  data);
```

Description
The call copies the oldest received diagnostic frame to the memory specified by data.

Notes
The data returned is received from suitable frames (master request frame for slave nodes and slave response frame for master nodes).

Implementation note
If the receive queue is empty no action shall be taken.

4.1.3 ld_raw_tx_status

Dynamic prototype
```
l_u8 ld_raw_tx_status (l_ifc_handle iii);
```

Description
The call returns the status of the raw frame transmission function:

LD_QUEUE_FULL	The transmit queue is full and can not accept further frames.
LD_QUEUE_EMPTY	The transmit queue is empty i.e. all frames put in the queue has been transmitted.
LD_TRANSFER_ERROR	LIN protocol errors occurred during the transfer; abort and redo the transfer.

Notes
How to perform an abortion of a transfer is not part of the standard.

4.1.4 ld_raw_rx_status

Dynamic prototype
```
l_u8 ld_raw_rx_status (l_ifc_handle iii);
```

Description
The call returns the status of the raw frame receive function:

LD_DATA_AVAILABLE	The receive queue contains data that can be read.
LD_TRANSFER_ERROR	LIN protocol errors occurred during the transfer; abort and redo the transfer.

Notes
How to perform an abortion of a transfer is not part of the standard.

4.2 COOKED API

Implementation note
Cooked processing of diagnostic messages manages one message at a time. Therefore, it is not needed to implement a message FIFO, nor to copy the messages between the application buffer and a buffer internal in the diagnostic module.

4.2.1 ld_send_message

Dynamic prototype
```
void ld_send_message (l_ifc_handle      iii,
                      l_u16             length,
                      l_u8              NAD,
                      const l_u8* const data);
```

Description
The call packs the information specified by data and length into one or multiple diagnostic frames. If the call is made in a master node the frames are sent to the node with the address NAD (slave nodes send them to the master).

Notes
SID (or RSID) shall be the first byte in the data area and it shall be included in the length. Length must be in the range 1 to 4095 bytes.

The parameter NAD is not used in slave nodes but included to make a common API.

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

The call shall return immediately, i.e. not suspend until the message has been sent, and the buffer may not be changed by the application as long as calls to ld_tx_status returns LD_IN_PROGRESS.

The data is sent as frame responses to the succeeding suitable frames headers (master request frame for master nodes and slave response frame for slave nodes).

The call is not legal if the previous transmission is still in progress.

4.2.2 ld_receive_message

Dynamic prototype

```
void ld_receive_message (l_ifc_handle iii,
                         l_u16*       length,
                         l_u8*        NAD,
                         l_u8* const  data);
```

Description

The call prepares the LIN diagnostic module to receive one message and store it in the buffer pointed to by data. At the call length shall specify the maximum length allowed. When the reception has completed, length is changed to the actual length, NAD to the NAD in the message (applies to master nodes only).

Notes

SID (or RSID) will be the first byte in the data area and it is included in the length. Length will be in the range 1 to 4095 bytes, but never more than the value originally set in the call.

The parameter NAD is not used in slave nodes but included to make a common API.

The call shall return immediately, i.e. not suspend until the message has been received, and the buffer may not be changed by the application as long as calls to ld_rx_status returns LD_IN_PROGRESS. If the call is made “too late”, i.e. after the message transmission has commenced, the call will wait for the next message.

The data is received from the succeeding suitable frames (master request frame for slave nodes and slave response frame for master nodes).

The call is not legal if the previous transmission is still in progress, i.e ld_rx_status returns LD_IN_PROGRESS.

4.2.3 ld_tx_status

Dynamic prototype

```
l_u8 ld_tx_status (l_ifc_handle iii);
```

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

Description

The call returns the status of the last made call to ld_send_message. The following values can be returned:

LD_IN_PROGRESS	The transmission is not yet completed.
LD_COMPLETED	The transmission has completed successfully (and you can issue a new ld_send_message call).
LD_FAILED	The transmission ended in an error. The data was only partially sent. (You can make a new call to ld_send_message.)

Note

To find out why a transmission has failed, check the status management functions in LIN core.

4.2.4 ld_rx_status

Dynamic prototype

```
l_u8 ld_rx_status (l_ifc_handle iii);
```

Description

The call returns the status of the last made call to ld_receive_message. The following values can be returned:

LD_IN_PROGRESS	The reception is not yet completed.
LD_COMPLETED	The reception has completed successfully and all information (length, NAD, data) is available. (You can also issue a new ld_receive_message call).
LD_FAILED	The reception ended in an error. The data was only partially received and should not be trusted. (You can make a new call to ld_receive_message.)

Note

To find out why a reception has failed, check the status management functions in LIN core.

5 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.

5.1 LIN CORE API USAGE

```
/*****************************************************************
*         File: hello.c
*       Author: Christian Bondesson
*  Description: Example code for using the LIN API in a LIN master node
*               NOTE! This example uses the static API
*/

#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... */
} /* 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... */
} /* 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 function to do the actual work */
```

```
  l_ifc_rx_MyLinIfc ();
} /* 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 function to do the actual work */
  l_ifc_tx_MyLinIfc ();
} /* lin_char_tx_handler */


/*****************************************************************
*   PROCEDURE : main
* DESCRIPTION : Main program... initialization part
*/
void main (void)
{
  /* Initialize the LIN interface */
  if (l_sys_init ())
  {
    /* The init of the LIN software failed */
  }
  else
  {
    l_ifc_init_MyLinIfc ();      /* Initialize the interface            */
    if (l_ifc_connect_MyLinIfc ())
    {
      /* Connection of the LIN interface failed */
    }
    else
    {
      /* Connected, now ready to send/receive set the normal
       * schedule to run from beginning for this specific interface */
      l_sch_set_MyLinIfc (MySchedule1, 0);
    }
  }
  start_main_application (); /* Ready with init, start actual applic */
} /* main */


/* 10 ms 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 reading 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();
} /* main_application_10ms */
```

5.2 LIN DESCRIPTION FILE

```
/**************************************************************************
*         File: hello.ldf
*       Author: Christian Bondesson
*  Description: The LIN description file for the example program
*/

LIN_description_file ;
LIN_protocol_version = "2.0";
LIN_language_version = "2.0";
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 {
  Dig2Bit {
        logical_value, 0, "off";
        logical_value, 1, "on";
        logical_value, 2, "error";
        logical_value, 3, "void";
    }
}

Signal_representations {
  Dig2Bit: InternalLightsRequest, InternalLightsSwitch;
}
```

LIN
Node Capability Language Specification

Revision 2.0

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 intention of a LIN Node capability language is to be able to describe the possibilities of a slave node in a standardized, machine readable syntax.

The availability of pre-made off-the-shelf slave nodes is expected to grow in the next years. If they are all accompanied by a node capability file, it will be possible to generate both the LIN configuration file, see **LIN Configuration Language Specification**, and initialization code[1] for the master node.

If the setup and configuration of any LIN cluster is fully automatic, a great step towards plug-and-play development with LIN will be taken. In other words, it will be just as easy to use distributed nodes in a LIN cluster as a single CPU node with the physical devices connected directly to the node.

1.1 PLUG AND PLAY WORKFLOW

Figure 1.1 shows the development of a LIN cluster split in three areas; design, debugging and the LIN physical system. This specification focuses on the design phase.

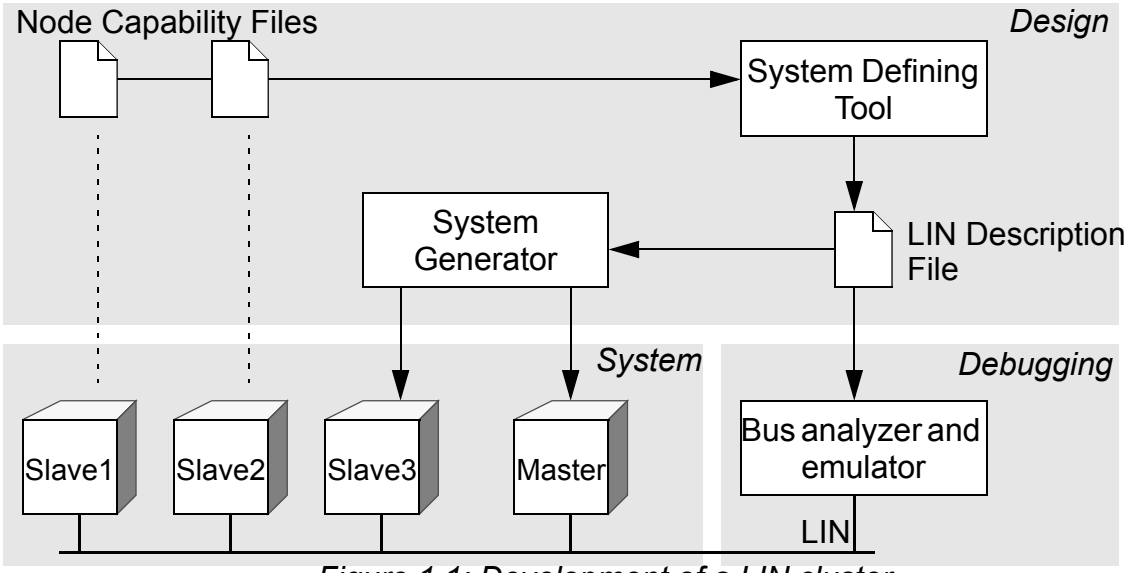


Figure 1.1: Development of a LIN cluster.

1.1.1 System Generation

The core description file of a LIN cluster is the LIN description file, LDF. Based on this file it is possible to generate communication drivers of all nodes in the cluster, a process named system generation. All signals and frames of the cluster are declared in this file.

Note 1: The code shall configure the cluster, e.g. reconfigure conflicting identifiers.

1.1.2 System Definition

The process of creating the LDF file is named system definition. When you design a completely new cluster, writing the LDF file (by hand or with computer aid) is an efficient way to define the communication of your cluster.

However, when you have existing slave nodes and want to create a cluster of them starting from scratch is not that convenient. This is especially true if the defined system contains node address conflicts or frame identifier conflicts.

By receiving a node capability file, NCF, with every existing slave node, the system definition step is automatic: Just add the NCF files to your project in the system definition tool and it produces the LDF file.

If you want to create new slave nodes as well, (Slave3 in **Figure 1.1**) the process becomes somewhat more complicated. The steps to perform depend of the system definition tool being used, which is not part of the LIN specification. A useful tool will allow for entering of additional information before generating the LDF file. (It is always possible to write a fictive NCF file for the non-existent slave node and thus, it will be included.)

It is worth noticing that the generated LDF file reflects the configured network; any conflicts originally between nodes or frames must have been resolved before activating the cluster traffic.

1.1.3 Debugging

Debugging and node emulation is based on the LDF file produced in the system definition. Thus, the monitoring will work just as in earlier versions of the LIN specification.

Emulation of the master adds the requirement that the cluster must be configured to be conflict free. Hence, the emulator tool must be able to read reconfiguration data produced by the system definition tool.

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

2 NODE CAPABILITY FILE DEFINITION

```
node_capability_file ;
<language_version>
[<node_definition>]
```

2.1 GLOBAL DEFINITION

Global definition data defines general properties of the file.

2.1.1 Node capability language version number definition

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

Shall be in the range of "0.01" to "99.99". This specification describes version 2.0.

2.2 NODE DEFINITION

```
<node_definition> ::=
node <node_name> {
  <general_definition>
  <diagnostic_definition>
  <frame_definition>
  <status_management>
  (<free_text_definition>)
}
<node_name> ::= identifier
```

If a node capability file contains more than one node, the node_name shall be unique within the file. The declared nodes shall be seen as classes (templates) for physical node instances.

The properties of a node_definition are defined in the following sections.

2.3 GENERAL DEFINITION

```
<general_definition> ::=
general {
  LIN_language_version = <protocol_version> ;
  supplier = <supplier_id> ;
  function = <function_id> ;
  variant = <variant_id> ;
  bitrate = <bitrate_definition> ;
  (volt_range = real_or_integer, real_or_integer ;)
  (temp_range = real_or_integer, real_or_integer ;)
  (conformance = char_string ;)
}
```

The general_definition declare the properties that specify the general compatibility with the cluster.

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

2.3.1 LIN protocol version number definition

```
<protocol_version> ::= char_string ;
```

This specifies the protocol used by the node and it shall be in the range of "0.01" to "99.99". At the time of publishing this is 2.0.

2.3.2 LIN Product Identification

```
<supplier_id> ::= integer
<function_id> ::= integer
<variant_id>  ::= integer
```

The supplier_id is assigned to each LIN consortium member as a 16 bit number. The function_id is a 16 bit number assigned to the product by the supplier to make it unique. Finally, variant_id is an 8 bit value specifying the variant (see **LIN Diagnostic and Configuration Specification**, Section 2.4).

2.3.3 Bit rate

```
<bitrate_definition> ::=
automatic (min <bitrate>) (max <bitrate>) |
select {<bitrate> [, <bitrate>]} |
<bitrate>
```

Three kinds of bitrate_definition are possible:

- automatic, the node can adopt to any legal bit rate used on the bus. If the words min and/or max is added any bit rate starting from/up to the provided bit rate can be used.
- select, the node can detect the bit rate if one of the listed bit rates are used, otherwise it will fail.
- fixed, only one bit rate can be used.

Manufacturers of standardized, off-the-shelf, nodes are encouraged to build automatic nodes since this gives the most flexibility to the cluster builder.

```
<bitrate> ::= integer
```

The fixed bit rates are specified as an integer in the range 1000 to 20000 (bit/s).

2.3.4 Non-network parameters

The optional specifications of volt_range and temp_range specifies the minimum and maximum values allowed for the node i Volt and Celcius, respectively. Finally, conformance specifies if the slave node has passed a conformance test procedure; it shall be either “LIN2.0” or “none”.

2.4 DIAGNOSTIC DEFINITION

```
<diagnostic_definition> ::=
diagnostic {
```

```
  NAD = integer (, <integer>) ;
  [ P2_min = integer ms; ]
  [ ST_min = integer ms; ]
  [ support_sid {<sid> ([, <sid>]) }; ]
  [ max_message_length = integer ; ]
}
```

The diagnostic_definition specifies the properties for diagnostic and configuration.

The NAD property defines the initial node address; it shall be according to **LIN Diagnostic and Configuration Specification**, Section 2.3.2. If two values are given, the slave will dynamically select one of the values within the range based on a physical property.

P2_min specifies the minimum time between a master request frame and the following slave response frame for the node to be able to prepare the response. Default 0 ms.

ST_min specifies the minimum time between two slave response frames in a multi-PDU response, i.e. it only applies to the diagnostic transport layer. Default 0 ms.

The max_message_length property applies to the diagnostic transport layer only; it defines the maximum length of a diagnostic message. Default: 4095.

support_sid lists all SID values that are supported by the node. Default: {0xb1, 0xb2}.

Implementation note

Future versions of this specification may add new properties to the diagnostic_definition. The intention is to follow the structure <property> = value or <property> { <list_of_entities> }.

2.5 FRAME DEFINITION

```
<frame_definition> ::=
frames {
  [ <single_frame> ]
}
```

All frames published or subscribed by the node shall be listed in this declaration[2].

```
<single_frame> ::=
<frame_kind> <frame_name> {
  <frame_properties>
  (<signal_definition>)
}
<frame_kind> ::= publish | subscribe
<frame_name> ::= identifier
```

Note 2: With the exception of diagnostic frames and user-defined frames.

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

Each frame published or subscribed is declared as defined above. frame_name is the symbolic[3] name of the frame. Of course, published frames shall start with frame_kind publish and subscribed with subscribe.

2.5.1 Frame properties

```
<frame_properties> ::=
message_ID = integer ;
length = integer ;
( min_period = integer ms ; )
( max_period = integer ms ; )
( event_triggered_message_ID = integer ; )
```

message_ID defines the message identifier of the frame (0 to 0xFFFF) and length the frame length (1 to 8).

The optional values for min_period and max_period are used to guide the tool in generation of the schedule table. Both values are specified in milliseconds.

The event_triggered__message_ID is also optional and it provides a secondary message ID for publishing of an event triggered frame.

Note

Several restrictions apply when a frame is also event triggered, see **LIN Protocol Specification**.

2.5.2 Signal definition

```
<signal_definition> ::=
signals {
  [<signal_name> {  <signal_properties> } ]
}
<signal_name> ::= identifier
```

All frames (except diagnostic frames) carry signals, which are declared in according to the signal_definition.

```
<signal_properties> ::=
  <init_value> ;
  size = integer ;
  offset = integer ;
  (<encoding> ;)
<init_value> ::=
  init_value = integer | init_value = { integer ([, integer ]) }
```

The init_value specifies the value used for the signal from power on until first set by the publishing application. The size is the number of bits reserved for the signal and the offset specifies the position of the signal in the frame (number of bits in offset from the first bit in the frame).

Note 3: A system definition tool can use the name directly for a physical frame, unless more than one instance of the node defined is used in the cluster.

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

For a byte array, both size and offset must be multiples of eight. Encoding does not apply for byte arrays.

Note

The only way to describe if a signal with size 8 or 16 is a byte array with one or two elements or a scalar signal is by analyzing the init_value, i.e. the curly paranthesis are very important to distinguish between arrays and scalar values.

2.5.3 Signal encoding type definition

The encoding is intended for providing representation and scaling properties of signals.

```
<encoding> ::=
encoding <encoding_name> {
  [<logical_value>  |
   <physical_range> |
   <bcd_value>      |
   <ascii_value>]
}
<encoding_name> ::= identifier
<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. If the raw value is within the range defined by the min and max value, the physical value shall be calculated as in **(1)**.

physical_value = scale * raw_value + offset. (1)

2.6 STATUS MANAGEMENT

```
<status_management> ::=
  status_management {
    error_response = <published_signal> ;
  }
<published_signal> ::= identifier
```

The status_management section specifies which published signal the master node shall monitor to know if the slave node is operating as expected.

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

Note

Status management is separated from the general_definition in order to always have declarations precede usage, i.e. the signal has to be declared before being specified as the error_response signal.

2.7 FREE TEXT DEFINITION

```
<free_text_definition> ::=
free_text {
    <anything_but_right_curly_paranthesis>
  }
```

The free_text_definition is used to bring up help text, limitations, etc., in the system definition tool, if desired. The only limitation of its contents is that the ´}´ character may not be used.

Note

Recommended information to provide in the free text definition is:

- Node purpose and physical world interaction, e.g. motor speed, power consumption etc.
- Node availability.
- Deviations from the LIN standard, i.e. un-implemented functionality.

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

3 OVERVIEW OF SYNTAX

The syntax is described using a modified BNF (Bachus-Naur Format), as summarized in **Table 3.1** below.

Table 3.1: BNF syntax used in this document.

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 multiple times
( )	The text between the parenthesis are optional, i.e. shall appear once or zero times
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.

4 EXAMPLE FILE

```
node_capability_file;
LIN_language_version = 2.0;

node step_motor {
  general {
    LIN_protocol_version = 2.0;
    supplier = 0x0011;
    function = 0x1234;
    variant = 1;
    bitrate = automatic max 10400;
    volt_range = 8.0, 15.0;
  }
  diagnostic {
    NAD = 1, 3;
    P2_min = 40 ms;
    support_sid {0xb0, 0xb1, 0xb2};
  }
  frames {
    publish node_status {
      message_ID = 0x1001;
      length = 2;
      min_period = 10 ms;
      max_period = 100 ms;
      signals {
        state {init_value = 0; size = 8; offset = 0;}
        error_bit {init_value = 0; size = 1; offset = 8;}
      }
    }
    subscribe control {
      message_ID = 0x1002;
      length = 1;
      max_period = 100 ms;
      signals {
        command {init_value = 0; size = 8; offset = 0;
                 encode position {physical_value 0, 199, 1.8, 0, deg;}; }
      }
    }
  }
  status_management { error_response = error_bit; }
  free_text {
    The step_motor command signal shall be in the range 0 to 199, or
    the command will be ignored.
  }
}
```


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

LIN
Configuration Language Specification

Revision 2.0

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 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 of one or multiple nodes if it/they are not available.

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 program interface has been defined, see **LIN API Specification**, in order to have a uniform way to access the LIN network from within different application programs. However, the functional behavior of the application program is not addressed by the LIN description file.

The syntax of a LIN description file is simple enough to be entered manually, but the development and use of computer based tools is encouraged. Node capability files, as described in **LIN Node Capability Language Specification**, provides one way to (almost) automatically generate LIN description files. The same specification also gives an example of a possible workflow in development of a LIN cluster.

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

2 LIN DESCRIPTION FILE DEFINITION

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

The overall syntax of a LIN description file shall be as above.

2.1 GLOBAL DEFINITION

Global definition data defines general properties of the LIN cluster.

2.1.1 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". At the time of publishing this is 2.0.

2.1.2 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". This specification describes version 2.0.

2.1.3 LIN speed definition

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

Shall be in the range of 1.00 to 20.00 kilobit/second.

2.2 NODE DEFINITION

The node definition sections identifies the name of all participating nodes as well as specifying time base and jitter for the master. The definitions in this section creates a node identifier set. All identifiers in this set shall be unique.

2.2.1 Participating nodes

```
<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 node identifier set.

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.)

2.2.2 Node attributes

Node attributes provides all necessary information on the behaviour of a single node.

```
<Node_attributes_def> ::=
Node_attributes {
  [<node_name> {
    LIN_protocol = <protocol_version> ;
    configured_NAD = <diag_address> ;
    (product_id = <supplier_id>, <function_id>, <variant> ; )
    (response_error = <signal_name> ; )
    (P2_min = <real_or_interger> ms ; )
    (ST_min = <real_or_interger> ms ; )
    (configurable_frames {
      [ <frame_name> = <message_id> ; ]
    } )
  }]
}
<node_name>   ::= identifier
<protocol_version> ::= 1.2 | 1.3 | 2.0
<supplier_id> ::= integer
<function_id> ::= integer
<variant>     ::= integer
<message_id>  ::= integer
```

All optional clauses are refer to LIN 2.0 only, i.e. they shall not be used for LIN 1.2/1.3.

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

All node_name identifiers shall exist within the node identifier set and refer to a slave node. supplier_id shall be in the range 0 .. 0x7FFE, function_id in the range 0 .. 0xFFFE, variant in the range 0 .. 255 and message_id in the range 0 ..0xFFFE.

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

All signal_name identifiers shall exist within the signal identifier set and refer to a one bit standard signal, see Section 2.3.1. The signal shall be published by the specified node. Refer to status management in **LIN Protocol Specification** for more information.

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

The diag_addr specifies the diagnostic address for the identified node in the range of 1 to 127 as further defined in **LIN Diagnostic and Configuration Specification**. It shall specify the unique NAD used for the node after resolving any cluster conflicts, i.e. it shall be unique within the cluster.

Configurable frames shall list all frames processed by the node and their associated message identity. This section applies to LIN 2.0 nodes only (not to LIN 1.3).

2.2.3 Node composition definition

The LDF file is describing the functionality of nodes from communication point of view and by default each such a “functional node” is a physical (real) node as well. It is possible, however, to express that physical nodes are composed of functional nodes. The purpose of this clause is to allow a single master node software to handle multiple node configurations without changes.

```
<Node_composition_def> ::=
composite {
  [ configuration <configuration_name> {
    [<composite_node> {
      <functional_node> ([ , <functional_node> ])
    } ]
  } ]
}
<configuration_name> ::= identifier
<composite_node> ::= identifier
<functional_node> ::= identifier
```

If this clause is used all composite nodes must be listed in Section 2.2.1 and Section 2.2.2 while signals and frames (Section 2.3.1 and Section 2.4.2) shall be published or subscribed by functional nodes only.

All composite_node identifiers and functional_node identifiers must be unique within the node identifier set.

A physical cluster is statically built according to one of the configuration_names.

2.3 SIGNAL DEFINITION

The signal definition sections identifies the name of all signals in the cluster and their properties. The definitions in this section creates a signal identifier set. All identifiers in this set shall be unique.

2.3.1 Standard signals

```
<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 signal identifier set.

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

The signal_size specifies the size of the signal. It shall be in the range 1 to 16 bits for scalar signals and 8, 16, 24, 32, 40, 48, 56 or 64 for byte array signals.

```
<init_value> ::= integer | { integer ([ , 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 until the application program has updated the signal. Of course, the vectorized representation is used for byte array signals.

Note

The only way to describe if a signal with size 8 or 16 is a byte array with one or two elements or a scalar signal is by analyzing the init_value, i.e. the curly parenthesis are very important to distinguish between arrays and scalar values.

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

The published_by identifier and the subscribed_by identifier shall all exist in the node identifier set.

2.3.2 Diagnostic signals

```
<Diagnostic_signal_def> ::=
Diagnostic_signals {
  MasterReqB0:8,0;
  MasterReqB1:8,0;
  MasterReqB2:8,0;
  MasterReqB3:8,0;
  MasterReqB4:8,0;
  MasterReqB5:8,0;
  MasterReqB6:8,0;
  MasterReqB7:8,0;
  SlaveRespB0:8,0;
```

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

```
    SlaveRespB1:8,0;
    SlaveRespB2:8,0;
    SlaveRespB3:8,0;
    SlaveRespB4:8,0;
    SlaveRespB5:8,0;
    SlaveRespB6:8,0;
    SlaveRespB7:8,0;
}
```

Diagnostic signals have a separate section in the LIN description file due to the fact that the publisher/subscriber information does not apply here.

2.3.3 Signal groups

The group definition was a feature of LIN 1.3. Use of signal groups is deprecated and the following syntactical definition does not affect a LIN 2.0 cluster.

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

2.4 FRAME DEFINITION

The frame definition sections identifies the name of all frames in the cluster as well as their properties. The definitions in this section creates a frame identifier set (their symbolic name) and an associated frame ID set (the LIN frame identifier). All members in these sets shall be unique.

2.4.1 Dynamic frame ids

```
<Dynamic_frame_def> ::=
dynamic_frames { <frame_id> [, <frame_id> ] }
```

The dynamic frame definition is used to specify the frame identifiers that are allowed to be non-unique, i.e. multiple frames can use the same frame identifier, see Section 2.4.2.

2.4.2 Unconditional frames

```
<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 frame identifier set.

`<frame_id> ::= integer`

The frame_id specifies the frame ID number in range 0 to 59 or 62. The ID shall be unique for all frames within the frames ID set unless the frame_id is listed in the dynamic frame definition, see Section 2.4.1.

`<published_by> ::= identifier`

The published_by identifier shall exist in the node identifier set.

`<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 based on the frame ID according to **Table 2.1**. The frame_size is optional to be backwards compatible but you are encouraged to provide the value, even if it matches the default value.

Table 2.1: Default frame lengths

ID range	Frame length
0 - 31 (0x1f)	2
32 (0x20) - 47 (0x2f)	4
48 (0x30) - 63 (0x3f)	8

`<signal_name> ::= identifier`

The signal_name identifier shall exist in the signal identifier set.

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

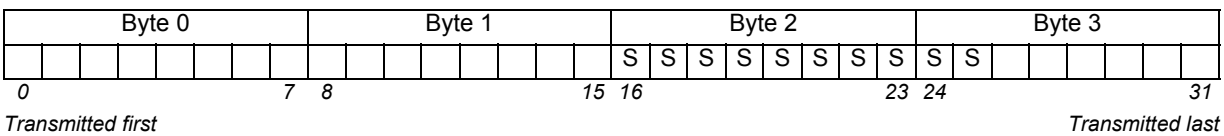
`<signal_offset> ::= integer`

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

Example

Table 2.2 below shows a ten bit signal packed in a frame with a four byte data field. The LSB of S is at offset 16 and the MSB is at offset 25. Note that the figure is drawn as the bytes are transmitted (LSB first).

Table 2.2: Packing of a signal.



The only rule for signal packing within a frame is that maximum one byte boundary may be crossed by a signal[1].

2.4.3 Sporadic frames

```
<Sporadic_frame_def> ::=
Sporadic_frames {
  [<sporadic_frm_name>:<frame_name>([,<frame_name>]);]
}
<sporadic_frm_name> ::= identifier
```

All sporadic_frm_name identifiers shall be unique within the frame identifier set.

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

All frame_name identifiers shall exist in the frame identifier set and refer to unconditional frames. In the case that more than one of the declared frames needs to be transferred, the one first listed shall be chosen.

All frame_name identifiers shall either be published by the master or be associated with an event_trig_frm_name, Furthermore, they shall not be included directly in the same schedule table as the sporadic_frm_name.

2.4.4 Event triggered frames

```
<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 frame identifier set.

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

The frame_id specifies the frame ID number in range 0 to 59. The ID shall be unique for all frames within the frames ID set.

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

All frame_name identifiers shall exist in the frame identifier set and refer to unconditional frames. In the case of a collision, the list of frame_name will be transferred instead of the event_trig_frm_name in the sequence they are listed in the declaration.

Note 1: Signal packing/unpacking is implemented more efficient in software based nodes if signals are byte aligned and/or if they do not cross byte boundaries.

For all frame_name identifiers declared with one event_trig_frm_name, the following shall apply:

- they shall use the same checksum model, i.e. the same LIN protocol standard (LIN 2.0 or LIN 1.3),
- they shall be published by different slave nodes in the cluster,
- they shall be of equal length, in the range 1 to 8 bytes,
- the first byte of the frame shall not carry any signals,
- they shall not be included directly in the same schedule table as the event_trig_frm_name.

Remark

The first byte of the frame carries the protected identifier of the associated frame and, hence, cannot be used for other purposes.

2.4.5 Diagnostic frames

```
<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 and shall be unique in the frame identifier set.

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

The MasterReq frame has a fixed identity, 60 (0x3c) and a fixed size (8 bytes), as specified in the **LIN Protocol Specification**. The MasterReq frame can only be sent by the master node.

The SlaveResp frame has a fixed identity, 61 (0x3d) and a fixed size (8 bytes) specified in the **LIN Protocol Specification**. The SlaveResp frame can only be sent by the selected slave node, see **LIN Diagnostic and Configuration Specification**. The selection of the slave node is based on the diagnostic addresses specified in Section 2.2.2.

2.5 SCHEDULE TABLE DEFINITION

The schedule table has changed slightly compared to earlier versions: The reason is to be able to describe and handle more complex, dynamically changing clusters without the need of the master application intervention.

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

All schedule_table_name identifiers shall be unique within the schedule table identifier set.

```
<command> ::=
  <frame_name> |
  MasterReq    |
  SlaveResp    |
  AssignFrameId { <node_name>, <frame_name> } |
  UnassignFrameId { <node_name>, <frame_name> } |
  AssignNAD { <old_NAD>, <new_NAD>, <supplier_id>, <function_id> } |
  FreeFormat { <D1>, <D2>, <D3>, <D4>, <D5>, <D6>, <D7>, <D8> }
```

The command specifies what will be done in the frame slot. Providing a frame name will transfer the specified frame.

MasterReq and SlaveResp are either defined as frames in Section 2.4.5 or, if this clause is left out, automatically defined. The contents of these frames is provided via the diagnostics and configuration API, see **LIN API Specification**.

AssignFrameId generates an Assign_frame_id master request frame with a contents based on the parameters: NAD, supplier_id and message_id are taken from the node attributes of the <node_name>, see Section 2.2.2 and the protected_id is taken from the frame definition for <frame_name>, see Section 2.4. All data in this frame is fixed and determined during the processing of the LDF file.

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

If a message ID of an event triggered frame shall be assigned a frame protected identifier (PID) the associated unconditional frame must first have been assigned a protected identifier.

UnassignFrameId generates an Assign_frame_id master request frame with a contents based on the parameters just like AssignFrameId but with the exception that the protected identifier is 0x40, i.e. identifier zero with both parity bits invalid. This is used to disable reception/transmission of a dynamically assigned frame identifiers (which is necessary to use the identifier for another frame, see also Section 2.4.1.)

AssignNAD generates an Assign_NAD master request frame with a contents directly taken from the parameters. All data in this frame is fixed and determined during the processing of the LDF file.

Finally, **FreeFormat** sends a fixed master request frame with the eight data bytes provided. This may for instance be used to issue user specific fixed frames.

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

The frame_name identifier shall exist in the frame identifier set. If the frame_name refers to an event triggered frame or a sporadic frame, the associated unconditional frames may not be used in the same schedule table.

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

The frame_time specifies the duration of the frame slot. It must be longer than the maximum allowed frame transfer time and it shall be exact multiple of the master node's time base value, as defined in Section 2.2.1. 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, see **LIN API Specification**.

Example

Figure 2.1 shows a time line that corresponds to the schedule table VL1_ST1. It is assumed that the time_base (see Section 2.2.1) is set to 5 ms.

```
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;
  }
}
```

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

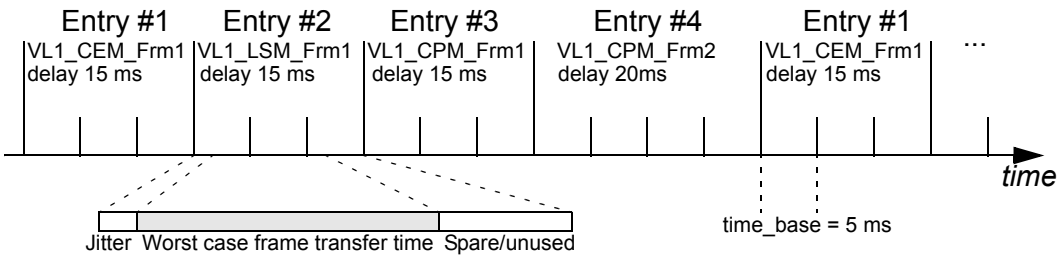


Figure 2.1: Time line for the VL1_ST1 schedule table.

The delay specified for every schedule entry shall be longer than the jitter and the worst-case frame transfer time.

2.6 ADDITIONAL INFORMATION

The following sub-sections provide additional information that does not change the behavior of the LIN cluster but provide hints for presentation of the traffic by bus snooping tools. All declarations are optional.

2.6.1 Signal encoding type definition

The signal encoding type is intended for providing representation and scaling properties of signals. Although this information may be used to generate automatically scaling API routines in LIN nodes, those API routines would require quite powerful nodes. The main purpose of the signal encoding type declarations is in bus traffic snooping tools, which can present the recorded traffic in an easily accessed way.

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

All signal_encoding_type_name identifier shall be unique within the signal encoding type identifier set.

```
<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. If the raw value is within the range defined by the min and max value, the physical value shall be calculated as in **(1)**.

physical_value = scale * raw_value + offset. (1)

Example

The V_battery signal is an eight bit representation that follows the graph in **Figure 2.2**, i.e. the resolution is high around 12 V and has three special values for out-of-range values.

```
Signal_encoding_types {
  power_state {
    logical_value, 0, "off";
    logical_value, 1, "on";
  }
  V_battery {
    logical_value, 0, "under voltage";
    physical_value, 1, 63, 0.0625, 7.0, "Volt";
    physical_value, 64, 191, 0.0104, 11.0, "Volt";
    physical_value, 192, 253, 0.0625, 13.0, "Volt";
    logical_value, 254, "over voltage";
    logical_value, 255, "invalid";
  }
}
```

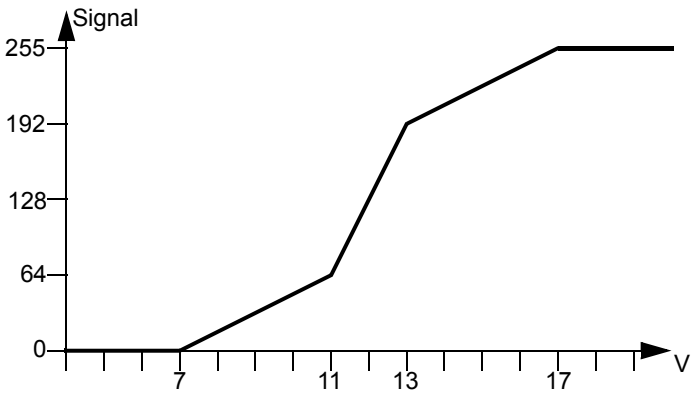


Figure 2.2: V_battery representation.

2.6.2 Signal representation definition

The signal representation declaration is used to associate signals with the corresponding signal encoding type.

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

The signal_encoding_type_name identifier shall exist in the signal encoding type identifier set.

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

The signal_name identifier shall exist in the signal identifier set. Each signal may only be associated with one signal_encoding_type_name and may not be nested in a signal_group_name.

3 OVERVIEW OF SYNTAX

The syntax is described using a modified BNF (Bachus-Naur Format), as summarized in **Table 3.1** below.

Table 3.1: BNF syntax used in this document.

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 multiple times
( )	The text between the parenthesis are optional, i.e. shall appear once or zero times
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.