LIN
Specification Package

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.

All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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
LIN 2.1	2006-11-24	Clarifications, configuration modified, transport layer enhanced and diagnostics added.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

TABLE OF CONTENTS

Specification Package

1.1 LIN 10
1.1.1 Scope 10
1.1.2 Features and possibilities 10
1.1.3 Work flow concept 11
1.1.4 Node concept 12
1.1.5 Concept of operation 12
1.1.5.1 Master and slave 12
1.1.5.2 Frames 13
1.1.5.3 Data transport 13
1.1.5.4 Schedule table 14
1.1.6 Document overview 14
1.1.7 History and background 14
1.1.7.1 Compatibility with LIN 1.3 15
1.1.7.2 Compatibility with LIN 2.0 16
1.1.7.3 Changes between LIN 1.3 and LIN 2.0 17
1.1.7.4 Changes between LIN 2.0 and LIN 2.1 17
1.1.8 References 18
1.2 LIN Glossary 19

Protocol Specification

2.1 Introduction 24
2.2 Signal Management 25
2.2.1 Signal types 25
2.2.2 Signal consistency 25
2.2.3 Signal packing 25
2.2.4 Signal reception and transmission 26
2.3 Frame Transfer 28
2.3.1 Frame structure 28
2.3.1.1 Break field 29
2.3.1.2 Sync byte field 29
2.3.1.3 Protected identifier field 30
2.3.1.4 Data 30
2.3.1.5 Checksum 31
2.3.2 Frame length 31
2.3.3 Frame types 32
2.3.3.1 Unconditional frame 32
2.3.3.2 Event triggered frame 32

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.3.3.3 Sporadic frame 35
2.3.3.4 Diagnostic frames 36
2.3.3.5 Reserved frames 37
2.4 Schedule tables 38
2.4.1 Time definitions 38
2.4.2 frame Slot 38
2.4.3 Schedule table handling 39
2.5 Task Behavior Model 40
2.5.1 Master task state machine 40
2.5.2 Slave task state machine 40
2.5.2.1 Break/sync field sequence detector 40
2.5.2.2 Frame processor 41
2.6 Network Management 44
2.6.1 slave communication state diagram 44
2.6.2 Wake up 45
2.6.3 Go to sleep 46
2.7 Status Management 47
2.7.1 Concept 47
2.7.2 Event triggered frames 47
2.7.3 Reporting to the cluster 47
2.7.4 Reporting within own node 48
2.8 Appendices 49
2.8.1 Table of numerical properties 49
2.8.2 Table of valid frame identifiers 50
2.8.3 Example of checksum calculation 52
2.8.4 Syntax and mathematical symbols used in this standard 53

Transport Layer Specification

3.1 Introduction 55
3.2 Transport layer 56
3.2.1 PDU structure 56
3.2.1.1 Overview 57
3.2.1.2 NAD 57
3.2.1.3 PCI 57
3.2.1.4 LEN 58
3.2.1.5 SID 58
3.2.1.6 D1 to D6 58
3.2.2 Communication 58
3.2.2.1 Single Frame Transmission 59
3.2.2.2 Multiple Frame Transmission 59
3.2.3 Error Handling 59

Registered copy for cbarbaud@cmaisonneuve.qc.ca

3.2.4 Defined requests 60
3.2.5 timing constraints....60

Node configuration and Identification Specification

4.1 Introduction 64
4.2 Node configuration and identification....65
4.2.1 LIN product identification 65
4.2.1.1 Wildcards....65
4.2.2 Slave Node model 66
4.2.2.1 Initial NAD....67
4.2.3 PDU structure 68
4.2.3.1 Overview....69
4.2.3.2 NAD 69
4.2.3.3 PCI....69
4.2.3.4 SID....70
4.2.3.5 RSID 70
4.2.3.6 D1 to D5 70
4.2.4 Node configuration and identification 71
4.2.5 Node configuration services 71
4.2.5.1 Assign NAD 71
4.2.5.2 Conditional change NAD 72
4.2.5.3 Data dump 73
4.2.5.4 Save Configuration 73
4.2.5.5 Assign frame ID range....74
4.2.6 Identification 75
4.2.6.1 Read by identifier....75

Diagnostic specification

5.1 Introduction 78
5.1.1 using the transport layer 78
5.1.2 LIN master 79
5.1.3 slave nodes 79
5.2 Diagnostic classes 80
5.2.1 Diagnostic Class I....80
5.2.1.1 Transport protocol 80
5.2.1.2 Diagnostic services....80
5.2.2 Diagnostic Class II....80
5.2.2.1 Transport protocol 80
5.2.2.2 Diagnostic services....81
5.2.3 Diagnostic Class III....81
5.2.3.1 Addressing....81

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.2.3.2 Transport protocol 81
5.2.3.3 Diagnostic services....82
5.2.4 Summary of slave node classes....82
5.2.5 Master node requirements....83
5.2.5.1 Transport protocol 83
5.2.5.2 Fault management, sensor reading, I/O control 83
5.2.6 User defined diagnostics 84
5.3 Requirements for Signal based Diagnostics....85
5.4 Transport Protocol handling in LIN-master 87
5.4.1 Diagnostic master request schedule 87
5.4.2 Diagnostic slave response schedule 88
5.4.3 Diagnostic schedule execution 89
5.4.3.1 Diagnostics Interleaved Mode 90
5.4.3.2 Diagnostics Only Mode....92
5.4.4 Transmission handler requirements 94
5.4.4.1 Master node transmission handler 95
5.5 Slave node transmission handler....100
5.6 Slave diagnostic timing requirements 104

Physical Layer Specification

6.1 Introduction 107
6.2 Physical Layer Compatibility....108
6.3 Bit rate Tolerance 109
6.4 Timing Requirements....111
6.4.1 Bit Timing Requirements 111
6.4.2 Synchronization Procedure 111
6.4.3 Bit Sample Timing 111
6.5 Line Driver/Receiver 114
6.5.1 General Configuration 114
6.5.2 Definition of Supply Voltages for the Physical Interface 114
6.5.3 Signal Specification 116
6.5.4 Electrical DC parameters....117
6.5.4.1 Electrical AC Parameters 119
6.5.5 Line Characteristics 121
6.5.6 Performance in non-operation supply voltage range....122
6.5.7 Performance during fault modes 122
6.5.7.1 Loss of supply voltage connection or ground connection....122
6.5.7.2 Bus wiring short to battery or ground....122
6.5.8 ESD/EMI compliance....123

Application Program Interface Specification

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.1 Introduction 125
7.1.0.1 LIN cluster generation 125
7.1.1 Concept of operation 125
7.1.1.1 LIN core API 125
7.1.1.2 LIN node configuration and identification API 126
7.1.1.3 LIN transport layer API 126
7.2 Core API 127
7.2.1 Driver and cluster management 127
7.2.1.1 l_sys_init 127
7.2.2 Signal interaction 127
7.2.2.1 Signal types 128
7.2.2.2 Scalar signal read 128
7.2.2.3 Scalar signal write 128
7.2.2.4 Byte array read 129
7.2.2.5 Byte array write 129
7.2.3 Notification 130
7.2.3.1 l_flg_tst 130
7.2.3.2 l_flg_clr 131
7.2.4 Schedule management 131
7.2.4.1 l_sch_tick 131
7.2.4.2 l_sch_set 132
7.2.5 Interface management 133
7.2.5.1 l_ifc_init 133
7.2.5.2 l_ifc_goto_sleep 134
7.2.5.3 l_ifc_wake_up 134
7.2.5.4 l_ifc_ioctl 135
7.2.5.5 l_ifc_rx 135
7.2.5.6 l_ifc_tx 136
7.2.5.7 l_ifc_aux 137
7.2.5.8 l_ifc_read_status 137
7.2.6 User provided call-outs 140
7.2.6.1 l_sys_irq_disable 140
7.2.6.2 l_sys_irq_restore 141
7.3 Node configuration and identification 142
7.3.1 Node configuration 142
7.3.1.1 ld_is_ready 142
7.3.1.2 ld_check_response 143
7.3.1.3 ld_assign_frame_id_range 143
7.3.1.4 ld_assign_NAD 144
7.3.1.5 ld_save_configuration 144
7.3.1.6 ld_read_configuration 145
7.3.1.7 ld_set_configuration 145

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.3.2 ld_conditional_change_NAD 146
7.3.3 Identification 147
7.3.3.1 ld_read_by_id 147
7.3.3.2 ld_read_by_id_callout 148
7.4 Transport layer 149
7.4.1 Raw and Cooked API 149
7.4.2 Initialization 149
7.4.3 Raw API 150
7.4.3.1 ld_put_raw 150
7.4.3.2 ld_get_raw 150
7.4.3.3 ld_raw_tx_status 151
7.4.3.4 ld_raw_rx_status 151
7.4.4 Cooked API 152
7.4.4.1 ld_send_message 152
7.4.4.2 ld_receive_message 153
7.4.4.3 ld_tx_status 154
7.4.4.4 ld_rx_status 154
7.5 Examples 156
7.5.1 Master node example 156
7.5.2 Slave node example 158

Node Capability Language Specification

8.1 Introduction 162
8.1.1 Plug and play workflow 162
8.1.1.1 LIN cluster Generation 162
8.1.1.2 LIN cluster design 163
8.1.1.3 Debugging 163
8.2 Node capability file definition 164
8.2.1 Global definition 164
8.2.1.1 Node capability language version number definition 164
8.2.2 Node definition 164
8.2.3 General definition 164
8.2.3.1 LIN protocol version number definition 165
8.2.3.2 LIN Product Identification 165
8.2.3.3 Bit rate 165
8.2.3.4 Sends wake up signal 165
8.2.4 Diagnostic definition 165
8.2.5 Frame definition 166
8.2.5.1 Frame properties 167
8.2.5.2 Signal definition 167
8.2.5.3 Signal encoding type definition 168

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.2.6 Status management 169
8.2.7 Free text definition 169
8.3 Overview of Syntax 170
8.4 Example file 171

Configuration Language Specification

9.1 Introduction 173
9.2 LIN description file definition 174
9.2.1 Global definition 174
9.2.1.1 LIN protocol version number definition 174
9.2.1.2 LIN language version number definition 174
9.2.1.3 LIN speed definition 174
9.2.1.4 Channel postfix name definition 175
9.2.2 Node definition 175
9.2.2.1 Participating nodes 175
9.2.2.2 Node attributes 175
9.2.2.3 Node composition definition 177
9.2.3 Signal definition 178
9.2.3.1 Standard signals 178
9.2.3.2 Diagnostic signals 179
9.2.3.3 Signal groups 179
9.2.4 Frame definition 180
9.2.4.1 Unconditional frames 180
9.2.4.2 Sporadic frames 181
9.2.4.3 Event triggered frames 181
9.2.4.4 Diagnostic frames 182
9.2.5 Schedule table definition 183
9.2.6 Additional information 185
9.2.6.1 Signal encoding type definition 185
9.2.6.2 Signal representation definition 187
9.3 Overview of Syntax 188
9.4 Examples 189
9.4.1 LIN description file 189

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.1 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.

1.1.1 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.

1.1.2 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
- predictable behavior
- reconfigurability
- transport layer and diagnostic support

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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN provides a cost efficient bus communication where the bandwidth and versatility of CAN are not required. The specification of the line driver/receiver is based on the ISO 9141 standard [1] with some enhancements regarding the EMI behavior.

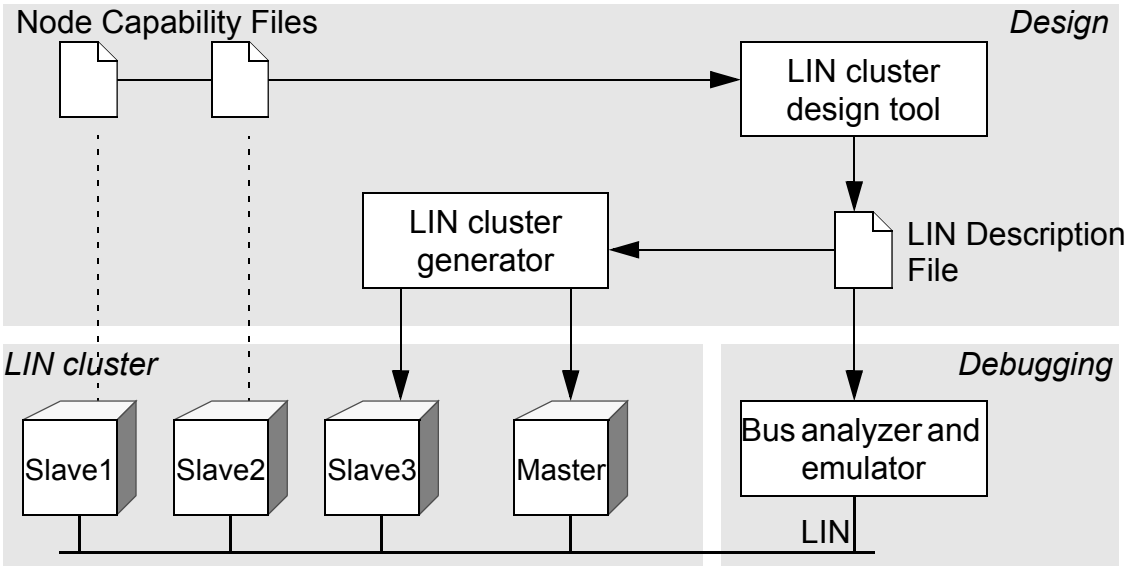
1.1.3 WORK FLOW CONCEPT

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

An example of the intended workflow is depicted below:



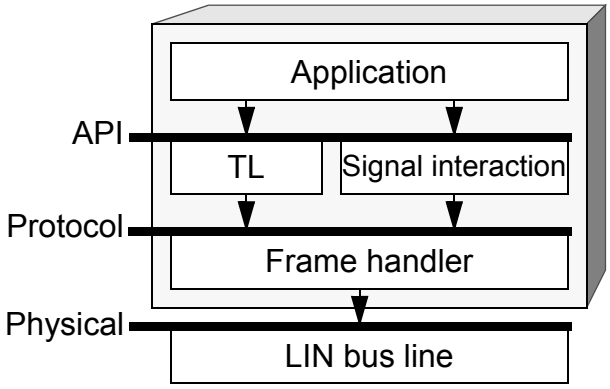
The slave nodes are connected to the master node forming a LIN cluster. The corresponding node capability files are parsed by the LIN cluster design tool to generate a LIN description file (LDF) in the LIN cluster design process. The LDF is parsed by the LIN cluster generator to automatically generate LIN related functions in the desired nodes (the Master node and Slave3 node 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.1.4 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:

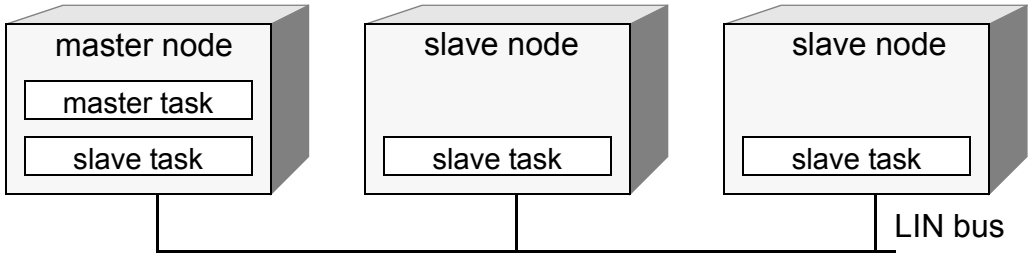
A node in a cluster interfaces to the physical bus wire using a frame transceiver. The frames are not accessed directly by the application; a signal based interaction layer is added in between. As a complement, a transport layer interface exists between the application and the frame handler, as depicted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



1.1.5 CONCEPT OF OPERATION

1.1.5.1 Master and slave

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



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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

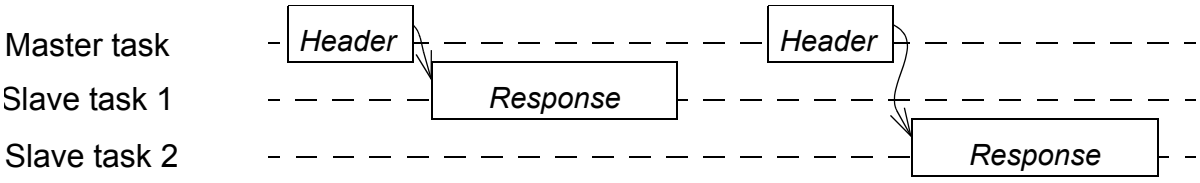
Both the master task and the slave task are parts of the Frame handler, see Section 1.1.4.

1.1.5.2 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 field and sync field followed by a frame identifier. The frame identifier uniquely defines the purpose of the frame. The slave task appointed for providing the response associated with the frame 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 frame 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 frame identifier (similar to CAN).
- Multicast: Any number of nodes can simultaneously receive and act upon a single frame.

1.1.5.3 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 frame identifier.

Diagnostic messages

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.1.5.4 Schedule table

The master task (in the master node) transmits headers based on a schedule table. The schedule table specifies the frames 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.

1.1.6 DOCUMENT OVERVIEW

The LIN Specification Package consists of the following specifications:

- The **Protocol Specification** describes the data link layer of LIN.
- The **Transport Layer Specification** describes how to transport data that can be up to 4095 bytes. Normally the transport layer is used for node configuration, identification and diagnostics.
- The **Node configuration and Identification Specification** defines how to configure a slave node and how to identify a slave node.
- The **Diagnostic specification** describes types of diagnostic services a slave node will support. All diagnostic services are using the transport layer.
- The **Physical Layer Specification** describes the physical layer, including bit rate, bit rate tolerances, etc.
- The **Application Program Interface Specification** describes the interface between the network and the application program, including the node configuration, identification and transport layer interfaces.
- The **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 nodes, as well as an input to development and analysis tools.
- The **Node Capability Language Specification** describes a format used to describe properties of slave nodes. A node capability file may be used with a LIN cluster design tool to automatically create LIN description files.

1.1.7 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The 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 1.1.7.1.

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

LIN 2.0 was 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.

Practical experience with LIN 2.0 has led to some findings in the specification. At the start of the work to update to LIN 2.1 it was decided that backwards compatibility is a major goal, see following sections. The LIN 2.1 contains mostly clarifications of the functionality. Some restrictions have been introduced to make different implementations more inline with each other. Some functionality has been deleted and some has been added, see Section 1.1.7.4.

1.1.7.1 Compatibility with LIN 1.3

LIN 2.1 is a superset of LIN 1.3.

A LIN 2.1 master node can handle clusters consisting of both LIN 1.3 slaves and/or LIN 2.1 slaves. The master will then avoid requesting the new LIN 2.1 features from a LIN 1.3 slave:

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

LIN 2.1 slave nodes can not operate with a LIN 1.3 node (e.g. the LIN1.3 master does not support the enhanced checksum).

The LIN 2.1 physical layer is backwards compatible with the LIN1.3 physical layer. But not the other way around. The LIN 2.1 physical layer sets harder requirement, i.e. a node using the LIN 2.1 physical layer can operate in a LIN 1.3 cluster.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.1.7.2 Compatibility with LIN 2.0

A LIN 2.1 master node may handle a LIN 2.0 slave node if the master node also contains all functionality of a LIN 2.0 master node, e.g. obsolete functions like Assign frame Id.

A LIN 2.1 slave node can be used in a cluster with a LIN 2.0 master node if the LIN 2.1 slave node is pre-configured, i.e. the LIN 2.1 slave node has a valid configuration after reset.

A LIN 2.0 slave node shall not use NAD 0x7E since it is reserved as functional address for diagnostics in LIN2.1. The LIN 2.1 slave node will consider NAD 0x7E as a functional NAD and a LIN 2.0 slave node as a NAD.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.1.7.3 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 slave 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.

1.1.7.4 Changes between LIN 2.0 and LIN 2.1

The major work has been to extend descriptions for better understanding.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Following functional changes have been done:

- Message identifiers for slave node frames are removed
- Assign frame ID configuration service is removed
- Assign frame ID range configuration service is added
- Save configuration service is added.
- Status reporting to application is enhanced
- Event-triggered frame collision handling modified
- The IDs 2 to 31 of the service Read by Identifier are reserved.
- Implementation of Diagnostic Classes 1 to 3 and respective Diagnostic Services.
- Transport layer enhanced with timings.
- A node operating on more than one cluster has been clarified.
- Packing a signal in more than one frame has been clarified.
- NAD 0x7E (functional NAD) is reserved as functional address for diagnostics
- Node capability language specification is extended with new parameters.
- The configuration language specification is updated to reflect the changes made; node attributes, node composition, event triggered frames and configuration are added.

1.1.8 REFERENCES

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

[2] LIN Slave Node Position Detection - Implementation Note, version 1.0

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

[4] “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.

[5] “Road vehicles - Diagnostic systems — Part 1: Diagnostic services”, *International Standard ISO 14229-1.6*, Issue 6, 2001-02-22

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1.2 LIN GLOSSARY

The following terms are used in one or more of the **LIN 2.1 Specification Package** documents. Each term is briefly described in the glossary and a reference (if applicable) to the section where the term is defined/described. Words written in bold have own entries in the glossary.

break field	A break field consist of a dominant part, the break, and a recessive part, the break delimiter.
bus interface	The logic (transceiver, UART, etc.) of a **node** that is connected to the physical bus wire in a **cluster**.
bus sleep mode	No communication occurs in the cluster. Nodes switch output level to the recessive state. Section 2.6.3.
byte field	Each field (except the **break field**) on the bus is sent in a byte field; the byte field includes the start bit and stop bit. Section 2.3.1.
checksum model	Two checksum models are defined; **classic checksum** and **enhanced checksum**. The enhanced checksum includes the **protected identifier** in the checksum calculation, classic checksum does not. Section 2.3.1.5.
classic checksum	The **checksum model** used in the LIN specification versions up to version 1.3 for all frames. In LIN 2.x it is used only for the **diagnostic frames**. The classic checksum considers the **data bytes** only. Section 2.3.1.5.
cluster design	The process of designing the information in the **LIN Description File**. Section 1.1.3.
cluster generation	The process of targeting one (or multiple) of the **nodes** in the **cluster** to the **LIN Description File**. Section 1.1.3.
checksum error	The checksum of the frame is not correct. The cause may be that the frame was corrupted on the bus or that the wrong **checksum model** was used in the frame.
cluster	A cluster is defined as the LIN bus wire plus all the **nodes**.
data	The **response** of a **frame** carries one to eight **data bytes**, collectively called data. Section 2.3.1.4.
data byte	One of the bytes in the **data**. Section 2.3.1.4.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

diagnostic frame	The collective name for the **master request frame** and the **slave response frame**. Section 2.3.3.4**.**
DTC	Diagnostic Trouble Code. **Diagnostic specification**.
enhanced checksum	The **checksum model** used in the LIN specification versions starting from LIN 2.0 for all frames, except the **diagnostic frames**. The enhanced checksum includes the **PID** and the **data bytes**. Section 2.3.1.5.
event triggered frame	An event triggered frame is used to allow multiple slave nodes to provide their **response** to the same **header**. This is useful when the signals in **slave nodes** involved are changed sporadically. Section 2.3.3.2.
frame	All information is transmitted packed as frames; a frame consist of the **header** and a **response**. Section 1.1.5.2.
frame identifier	The identity of a **frame** is in the range from 0 to 63 (six-bit value). Section 2.3.1.3.
frame slot	The time period reserved for the transfer of a specific **frame** on the bus. Corresponds to one entry in the **schedule table**. Section 2.4.2.
go to sleep command	A special **master request frame** issued to force slave nodes to bus sleep mode. Section 2.6.3.
header	A header is the first part of a **frame**; it is always sent by the **master task**. Section 2.3.1.
LIN Description File	The LDF file is created in the **LIN cluster design** and parsed in the **LIN cluster generation** or by debugging tools. **Configuration Language Specification**.
LIN Product Identification	A number containing the supplier and function identification in a LIN slave node. Section 4.2.1.
master node	The master node not only contains a **slave task**, but also the **master task** that is responsible for sending all **headers** on the bus, i.e. it controls the timing and **schedule table** for the bus.
master request frame	The master request frame has **frame identifier** 60 (0x3C) and is used for **diagnostic frames** issued by the **master node**. Section 2.3.3.4.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

master task	The master task is responsible for sending all **headers** on the bus, i.e. it controls the timing and **schedule table** for the bus. Section 2.5.1.
NAD	Node Address for slave nodes. Diagnostic frames are broadcasted and the NAD specifies the addressed, respectively responding **slave node**. The NAD is the address of a logical node. Section 4.2.3.2.
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**. A physical slave node may be composed of one or more logical nodes.
Node Capability File	A NCF describes a **slave node** as seen from the LIN bus. It is used in the **cluster design**. **Node Capability Language Specification**.
Operational state	The slave node may transmit/receive frames in this state. Section 2.6.1.
PID	**Protected identifier**. Section 2.3.1.3.
protected identifier	An eight-bit value containing the **frame identifier** together with its two parity bits. Section 2.3.1.3.
publish	A **signal** or an **unconditional frame** has exactly one publisher; i.e. the **node** that is the source of the information. Compare with **subscribe**.
request	The master node puts request on the slave nodes in node configuration and in the diagnostics. Section 4.2.5 and Section 5.2.
reserved frame	Reserved frames have **frame identifiers** that shall not be used: 62 (0x3E) and 63 (0x3F). Section 2.3.3.5.
response	(1) A **frame** consists of a **header** and a response. Section 2.3.1. (2) The reply frame for a node configuration or a diagnostic request is a response. Section 4.2.5.
service	A service is the composite name for the **request/response** combination.
schedule table	The schedule table specifies the traffic on the LIN bus. Section 2.4.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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 **frame identifier** 61 (0x3D) and is used for **diagnostic frames** issued by one of the **slave nodes**. Section 2.3.3.4.
slave task	The slave task is responsible for listening to all **headers** on the bus and react accordingly, i.e. either **publish** a **frame response** or **subscribe** to it (or ignore it).
signal	A signal is a value or byte array transported in the **cluster** using a **signal carrying frame**. Section 2.2.
signal carrying frame	A frame that carries signals shall have an **frame identifier** in the range 0 (zero) to 59 (0x3B). **Unconditional frames**, **sporadic frames** and **event triggered frames** are signal carrying frames. Section 2.3.3.
SNPD	Slave Node Position Detection [2]. Defines a recommended practice to position and separate identical slave nodes.
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**. Section 2.3.3.3.
subscribe	A **signal** or an **unconditional frame** may have none, one or more subscribers. See also **publish**.
UDS	Unified Diagnostic Service (ISO 14229-1 [5]). **Diagnostic specification**.
unconditional frame	A **signal carrying frame** that is always sent in its allocated **frame slot**. Section 2.3.3.1.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Protocol Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.1 INTRODUCTION

The protocol specification describes the behavior on the bus (e.g. frame transportation) and in the nodes (e.g. status management).

The scope is covering one LIN bus and its LIN nodes. A node (normally a master node) that is connected to more than one LIN bus must be handled by higher layers (e.g. the application).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.2 SIGNAL MANAGEMENT

A signal is transported in the data field of a frame.

2.2.1 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.

A byte array is an array of between one and eight bytes.

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

All signals have initial values. The initial value for a published signal is valid until the node writes a new value to this signal. The initial value for a subscribed signal is valid until a new updated value is received from another node.

2.2.2 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. This also applies to byte arrays. However, no consistency is guaranteed between any signals.

2.2.3 SIGNAL PACKING

A signal is transmitted with the LSB first and the MSB last. There is no restriction on packing scalar signals over byte boundaries. Each byte in a byte array shall map to a single frame byte starting with the lowest numbered data byte, see section 2.3.1.4.

Several signals can be packed into one frame as long as they do not overlap each other.

Note that 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.

The same signal can be packed into multiple frames as long as the publisher of the signal is the same. If a node is receiving one signal packed into multiple frames the latest received signal value is valid. Handling the same signal packed into frames on different LIN clusters is out of the scope.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.2.4 SIGNAL RECEPTION AND TRANSMISSION

The point in time when a signal is transmitted/received needs to be defined to help design tools and testing tools to analyze timing of signals. This means that all implementations will behave in a predictable way.

The definitions below do not contain factors such as bit rate tolerance, jitter, buffer copy execution time, etc. These factors must be taken into account to get a more detailed analysis. The intention for the definitions below is to have a basis for such analysis.

The timing is different for a master node and a slave node. The reason is that the master node controls the schedule and knows which frame will be sent. A slave node gets this information first when the header is transmitted on the bus.

The time base and time base tick are defined in section 2.4.

A signal is considered received and available to the application as follows (see also Figure 2.1):

- Master node - at next time base tick after the maximum frame length. The master node updates its received signals periodically at the time base start (i.e. at task level).
- Slave node - when the checksum for the received frame is validated. The slave node updates its received signals directly after the frame is finished (i.e. at interrupt level).

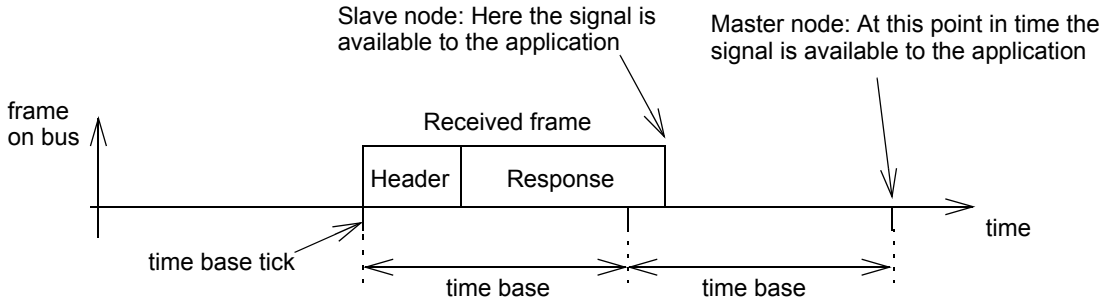


Figure 2.1: Timing of signal reception

Registered copy for cbarbaud@cmaisonneuve.qc.ca

A signal is considered transmitted (latest point in time when the application may write to the signal) as follows (see also Figure 2.2):

- Master node - before the frame transmission is initiated.
- Slave node - when the ID for the frame is received.

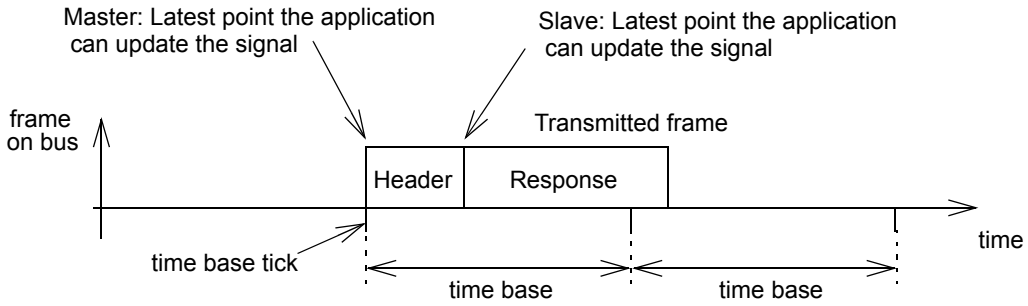


Figure 2.2: Timing of signal transmission

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.3 FRAME TRANSFER

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

2.3.1 FRAME STRUCTURE

The structure of a frame is shown in Figure 2.3. The frame is constructed of a number of fields, one break field followed by four to eleven byte fields, labeled as in the figure.

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 spaces.

The header starts at the falling edge of the break field and ends after the end of the stop bit of the protected identifier (PID) field. The response starts at the end of stop bit of the PID field and ends at the after the stop bit of the checksum field.

The inter-byte space is the time between the end of the stop bit of the preceding field and the start bit of the following byte. The response space is the inter-byte space between the PID field and the first data field in the data. Both of them must be non-negative.

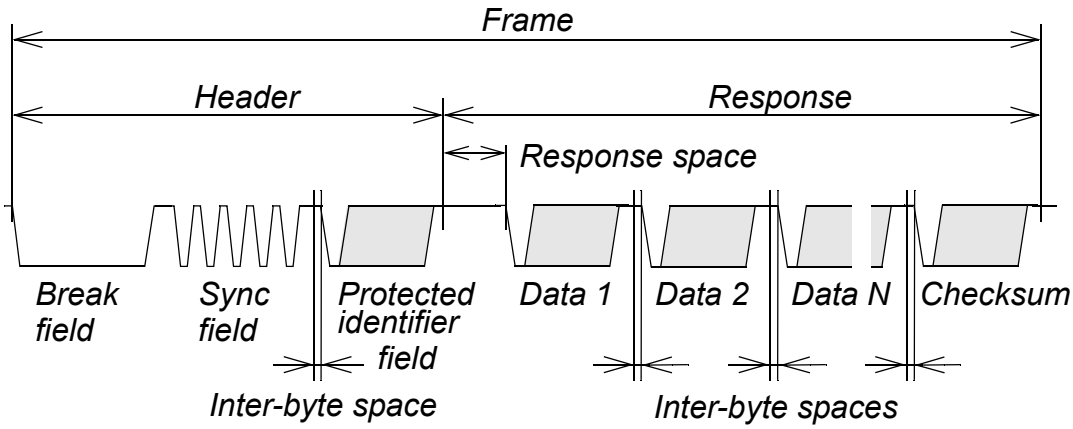


Figure 2.3: The structure of a frame.

Each byte field, except the break field, is transmitted as the byte field shown in Figure 2.4. 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).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

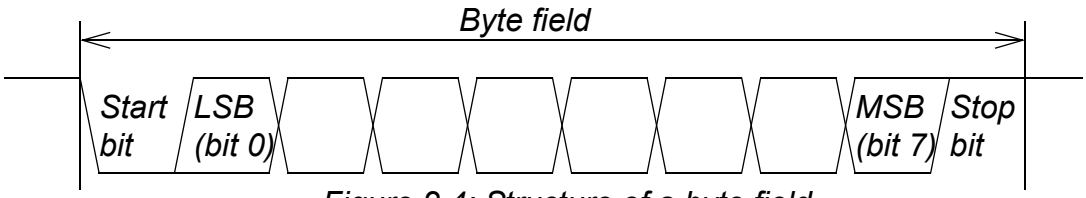


Figure 2.4: Structure of a byte field.

2.3.1.1 Break field

The break field is used to signal the beginning of a new frame. It is the only field that does not comply with Figure 2.4. A break field is always generated by the master task (in the master node) and it shall be at least 13 nominal bit times of dominant value, followed by a break delimiter, as shown in Figure 2.5. The break delimiter shall be at least one nominal bit time long.

A slave node shall use a break detection threshold of 11 dominant local slave bit times. Slave nodes with a bit rate tolerance better than $F_{TOL_RES_SLAVE}$, see section 6.3, (typically a crystal or ceramic resonator) may use a 9.5 dominant nominal bit times break detection threshold.

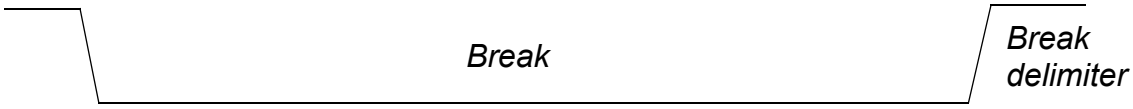


Figure 2.5: The break field

2.3.1.2 Sync byte field

Sync is a byte field with the data value 0x55, as shown in Figure 2.6.

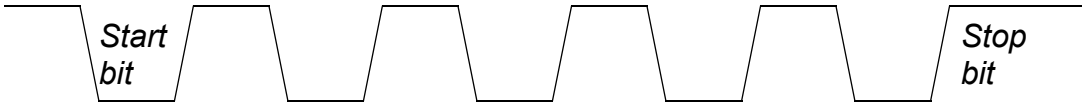


Figure 2.6: The sync byte field.

A slave task shall always be able to detect the break/sync field sequence, even if it expects a byte field (assuming the byte fields are separated from each other). A desired, but not required, feature is to detect the break/sync field sequence even if the break is partially superimposed with a data byte. When a break/sync field sequence happens, the transfer in progress shall be aborted and processing of the new frame shall commence.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.3.1.3 Protected identifier field

A protected identifier field consists of two sub-fields; the frame identifier and the parity. Bits 0 to 5 are the frame identifier and bits 6 and 7 are the parity.

Frame identifier

Six bits are reserved for the frame identifier, values in the range 0 to 63 can be used. The frame identifiers are split in three categories:

- Values 0 to 59 (0x3B) are used for signal carrying frames,
- 60 (0x3C) and 61 (0x3D) are used to carry diagnostic and configuration data,
- 62 (0x3E) and 63 (0x3F) are reserved for future protocol enhancements.

Parity

The parity is calculated on the frame 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 2.7.

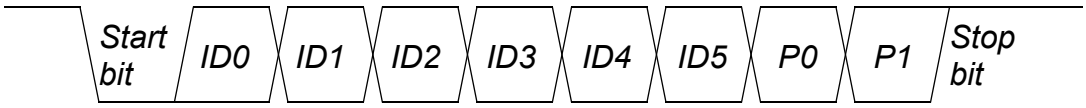


Figure 2.7: Mapping of frame identifier and parity to the protected identifier byte field.

2.3.1.4 Data

A frame carries between one and eight bytes of data. The number of data contained in a frame with a specific frame identifier shall be agreed by the publisher and all subscribers. A data byte is transmitted as part of a byte field, see Figure 2.4.

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 2.8.

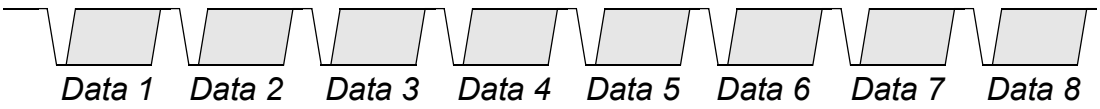


Figure 2.8: Numbering of the data bytes in a frame with eight data bytes.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.3.1.5 Checksum

The last field of a frame is the checksum. The checksum contains the inverted eight bit sum with carry 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 the master request frame, slave response frame and communication with LIN 1.x slaves.

Eight bit sum with carry is equivalent to sum all values and subtract 255 every time the sum is greater or equal to 256. See section 2.8.3 for examples how to calculate the checksum.

Checksum calculation over the data bytes and the protected identifier byte is called enhanced checksum and it is used for communication with LIN 2.x slaves.

The checksum is transmitted in a byte field, see Figure 2.4.

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.x slave nodes and enhanced in communication with LIN 2.x slave nodes.

Frame identifiers 60 (0x3C) to 61 (0x3D) shall always use classic checksum.

2.3.2 FRAME LENGTH

The nominal value for transmission of a frame exactly matches the number of bits sent (no response space and no inter-byte spaces). The nominal break field is 14 nominal bits long (break is 13 nominal bits and break delimiter is 1 nominal bit). 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 section 6.3.

The break field is 14 nominal bits or longer, see section 2.3.1.1. This means that $T_{Header_Maximum}$ puts a requirement on the maximum length of the break field.

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

$$T_{Header_Maximum} = 1.4 * T_{Header_Nominal} \quad (6)$$

$$T_{Response_Maximum} = 1.4 * T_{Response_Nominal} \quad (7)$$

$$T_{Frame_Maximum} = T_{Header_Maximum} + T_{Response_Maximum} \quad (8)$$

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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

Tools and tests shall check the $T_{Frame_Maximum}$. Nodes shall not check this time. The receiving node of the frame shall accept the frame up to the next frame slot (i.e. next break field), even if it is longer then $T_{Frame_Maximum}$.

2.3.3 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 recessive (ones).

2.3.3.1 Unconditional frame

Unconditional frames carry signals and their frame identifiers are in the range 0 (zero) 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 (the 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 2.9 shows a sequence of three unconditional frames. A transfer is always initiated by the master. It has a single publisher and one or multiple subscribers.

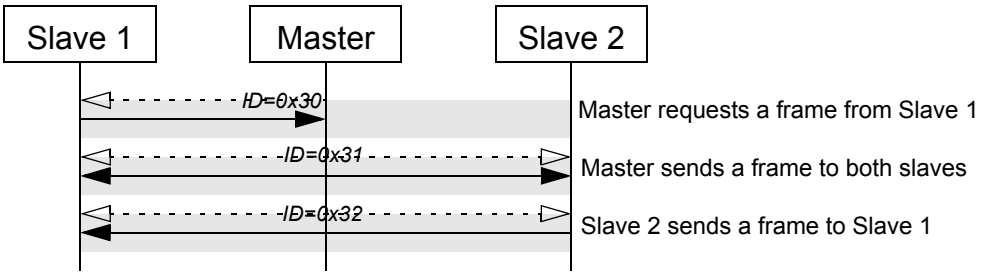


Figure 2.9: Three unconditional frame transfers.

2.3.3.2 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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.

If the unconditional frame associated with an event triggered frame is scheduled as an unconditional frame the response shall always be transmitted (i.e. behave as a scheduled unconditional frame).

Uncoditional frames associated with the event triggered frame

Event triggered frames carry the response of one or more unconditional frames.

The unconditional frames associated with an event triggered frame shall:

- Have equal length.
- Use the same checksum model (i.e. mixing LIN 1.x and LIN 2.x frames is not allowed).
- Reserve the first data field to its protected identifier (even if the associated unconditional frame is scheduled as a unconditional frame in the same or another schedule table).
- Be published by different slave nodes.
- Shall not be included directly in the same schedule table as the event triggered frame is scheduled.

Transmission of the event triggered frame

The header of an event triggered frame is transmitted when a frame slot allocated to the event triggered frame is processed. The publisher of an associated unconditional frame shall only transmit the response if at least one of the signals carried in its unconditional frame is updated. If the response is successfully transmitted, the signal is no longer considered to be updated.

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

If more than one slave node responds to the header in the same frame slot, a collision will occur.

Collision resolving

The master node has to resolve the collision in a collision resolving schedule table. Each event triggered frame has an associated collision resolving schedule table. The switch to the collision resolving schedule is made automatically by the driver in the master node (i.e. not by the application). The collision resolving schedule will be activated at the start of the subsequent frame slot after the collision.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

At least all the associated unconditional frames shall be listed in this collision resolving schedule table. The collision resolving schedule may contain other unconditional frames than the associated frames. These other unconditional frames may be of different length.

After the collision schedule table has been processed once, the driver in the master node shall switch back to the previous schedule table. It shall continue with the schedule entry subsequent to the schedule entry where the collision occurred (or first schedule entry in case the collision occurred in the last entry).

If one of the colliding slave nodes withdraws without corrupting the transfer, the master node will not detect this. A slave node that has withdrawn its response must therefore retry transmitting its response until successful, otherwise the response will be lost.

In case the master node application switches the schedule table before the collision is resolved, the collision resolving is lost. The new schedule table is activated as described in section 2.4.3. Note that the colliding slave nodes will still have their responses pending for transmission.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Example 1

A schedule table contains only one event-triggered frame (ID=0x10). The event-triggered frame is associated with two unconditional frames from slave 1 (ID=0x11) and slave 2 (ID=0x12). The collision resolving schedule table contains the two unconditional frames. See Figure 2.10 for the behavior on the bus.

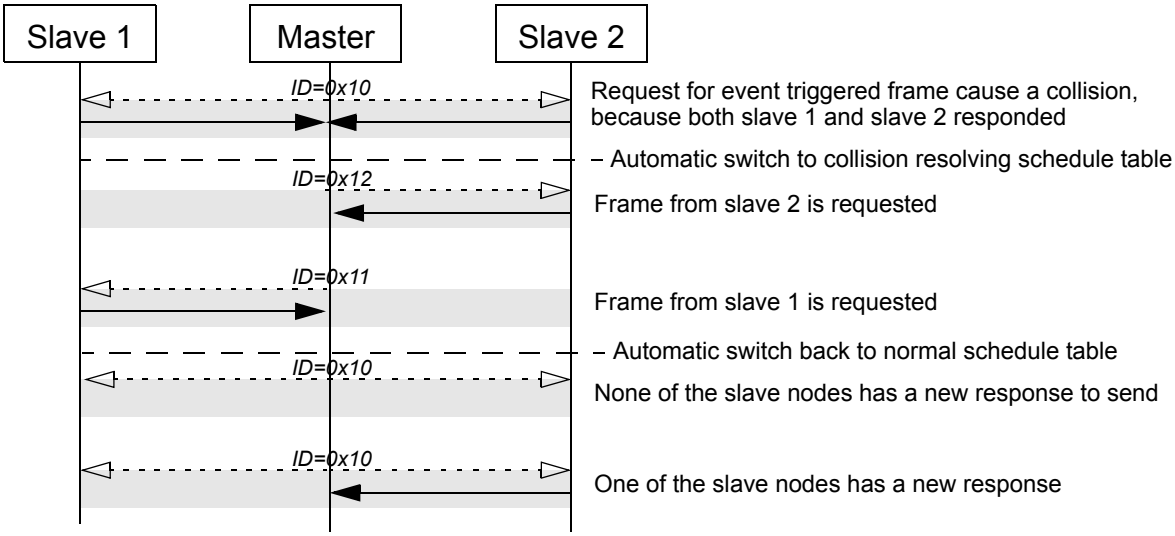


Figure 2.10: Event-triggered frame example.

Example 2

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 press a knob each, the system will not lose any of the pushes, but it will take some additional time.

2.3.3.3 Sporadic frame

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

A sporadic frame is a group of unconditional frames that share the same frame slot. When the sporadic frame is due for transmission the unconditional frames are checked if they have any updated signals. If no signals are updated, no frame will be transmitted and the frame slot will be empty. If one signal (or more signals packed in the same frame) has been updated, the corresponding frame will be transmitted. If more than one signal (packed in different frames) has been updated the highest prior-

Registered copy for cbarbaud@cmaisonneuve.qc.ca

itized (see below) frame will be transmitted. The candidate frames not transmitted will not be lost. They will be candidates to be transmitted every time the sporadic frame is due, as long as they have not been transmitted.

If the unconditional frame was successfully transmitted, the unconditional frame shall no longer be pending for transmission until a signal is updated in the unconditional frame again.

Normally multiple sporadic frames are associated with the same frame slot, the most prioritized of the pending unconditional frames shall be transmitted in the frame slot. If none of the unconditional frames is pending for transmission the frame slot shall be silent. How the sporadic frames are prioritized is described in Configuration Language Specification, section 9.2.4.2.

The master node is the only publisher of the uncoditional frames in a sporadic frame. Therefore only the master task knows when an unconditional frame is pending for transmission.

Example

A sporadic frame is the only frame in the active schedule table. The sporadic frame has a number of associated unconditional frames, where one has ID 0x22. Normally sporadic frame slots are empty. In the second slot, see Figure 2.11, at least one signal of the associated frame with ID 0x22 is updated.

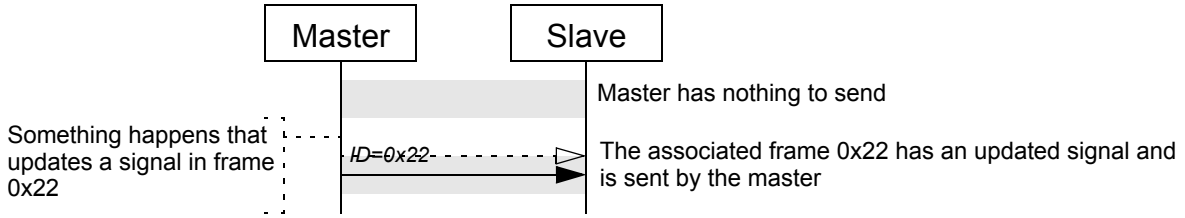


Figure 2.11: Sporadic frame example

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

2.3.3.4 Diagnostic frames

Diagnostic frames always carry transport layer, see Transport Layer Specification, data and they always contain eight data bytes. The frame identifier is either 60 (0x3C), called master request frame, or 61 (0x3D), called slave response frame. The interpretation of the data is given in Node configuration and Identification Specification and Diagnostic specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Before transmitting a master request frame, the master task queries its diagnostic module if it shall be transmitted or if the bus shall be silent. A slave response frame header shall be sent unconditionally.

The slave tasks publish and subscribe to the response according to their diagnostic modules.

2.3.3.5 Reserved frames

Reserved frames shall not be used in a LIN 2.x cluster. Their frame identifier are 62 (0x3E) and 63 (0x3F).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.4 SCHEDULE TABLES

A key property of the LIN protocol is the use of schedule tables. Schedule tables make 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 node to assure that all frames relevant in a mode of operation are given enough time to be transferred.

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 and efficient implementation of the LIN protocol.

2.4.1 TIME DEFINITIONS

The minimum time unit that is used in a LIN cluster is the time base (T_{base}). The time base is implemented in the master node and is used to control the timing of the schedule table. This means that the timing for the frames in a schedule table is based upon the time base. Usually a time base is 5 or 10 ms.

The starting point of the time base is defined as the time base tick. A frame slot always start at a time base tick.

The jitter, see Figure 2.12, specifies the differences between the maximum and minimum delay from time base tick to the header sending start point (falling edge of break field).

The inter-frame space, see Figure 2.12, is the time from the end of the frame until start of the next frame. The inter-frame space must be non-negative.

2.4.2 FRAME SLOT

The T_{Frame_Slot} **(9)** is the time that is controlling the schedule table timing. It is the time from when a schedule table entry is due (a frame transmission will be initiated) until the subsequent schedule entry is due. It is defined as a integer multiple of the time base. The integer multiple is normally different for each frame slot.

$$T_{Frame_Slot} = T_{base} * n \quad (9)$$

A frame slot **(9)** must have a duration **(10)** long enough to allow for the jitter introduced by the master task and the $T_{Frame_Maximum}$ defined in equation **(8)**.

$$T_{Frame_Slot} > \text{jitter} + T_{Frame_Maximum} \quad (10)$$

Registered copy for cbarbaud@cmaisonneuve.qc.ca

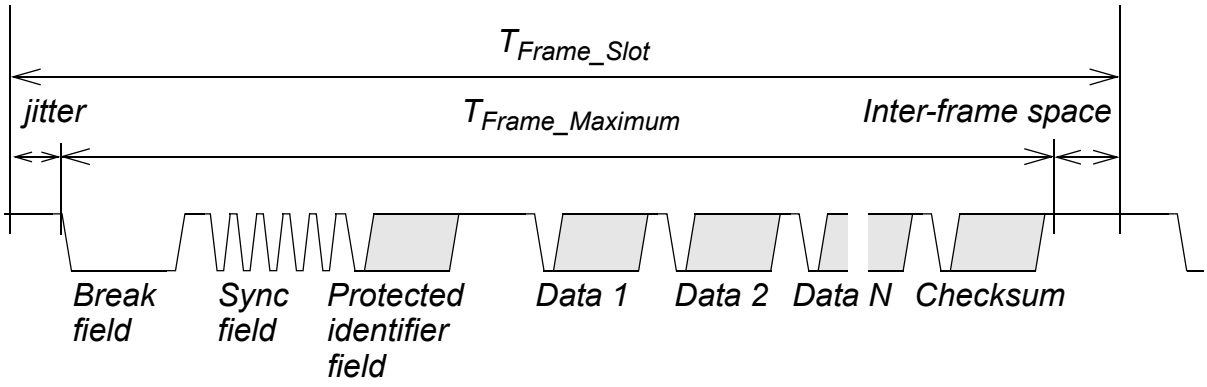


Figure 2.12: Frame slot

2.4.3 SCHEDULE TABLE HANDLING

The active schedule table shall be processed until another requested schedule table is selected. When the end of the current schedule is reached, the schedule is started again at the beginning of the schedule. The actual switch to the new schedule is made at start of a frame slot. This means that a schedule table switch request will not interrupt any ongoing transmission on the bus.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.5 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.

2.5.1 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 2.13.

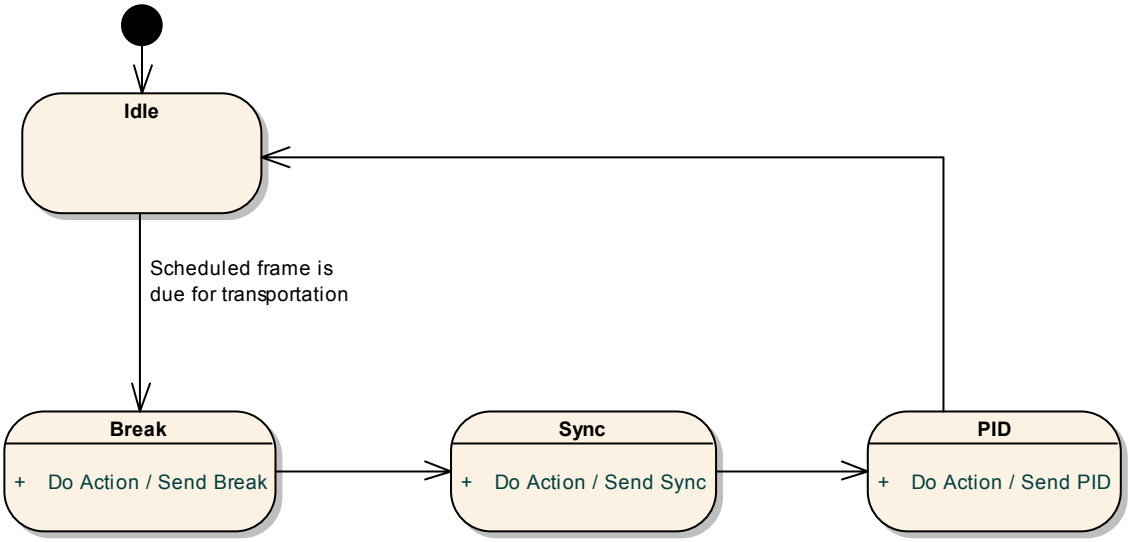


Figure 2.13: Complete state machine for the master task.

2.5.2 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/sync field sequence detector
- Frame processor

2.5.2.1 Break/sync field sequence 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 remainder of the frame, as specified in section 6.3. For this purpose every frame starts with a sequence starting with break field followed by a sync byte field. This sequence is

Registered copy for cbarbaud@cmaisonneuve.qc.ca

unique in the whole LIN communication and provides enough information for any slave task to detect the beginning of a new frame and to be synchronized at the start of the identifier field.

2.5.2.2 Frame processor

The frame processing consists of two states: Idle and Active. Active contains five sub-states. As soon as a break/sync field sequence is received (from any state or sub-state) the Active state is entered in the PID sub-state. This implies that processing of one frame will be aborted by the detection of a new break/sync field sequence. The frame processor state machine is depicted in Figure 2.14.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

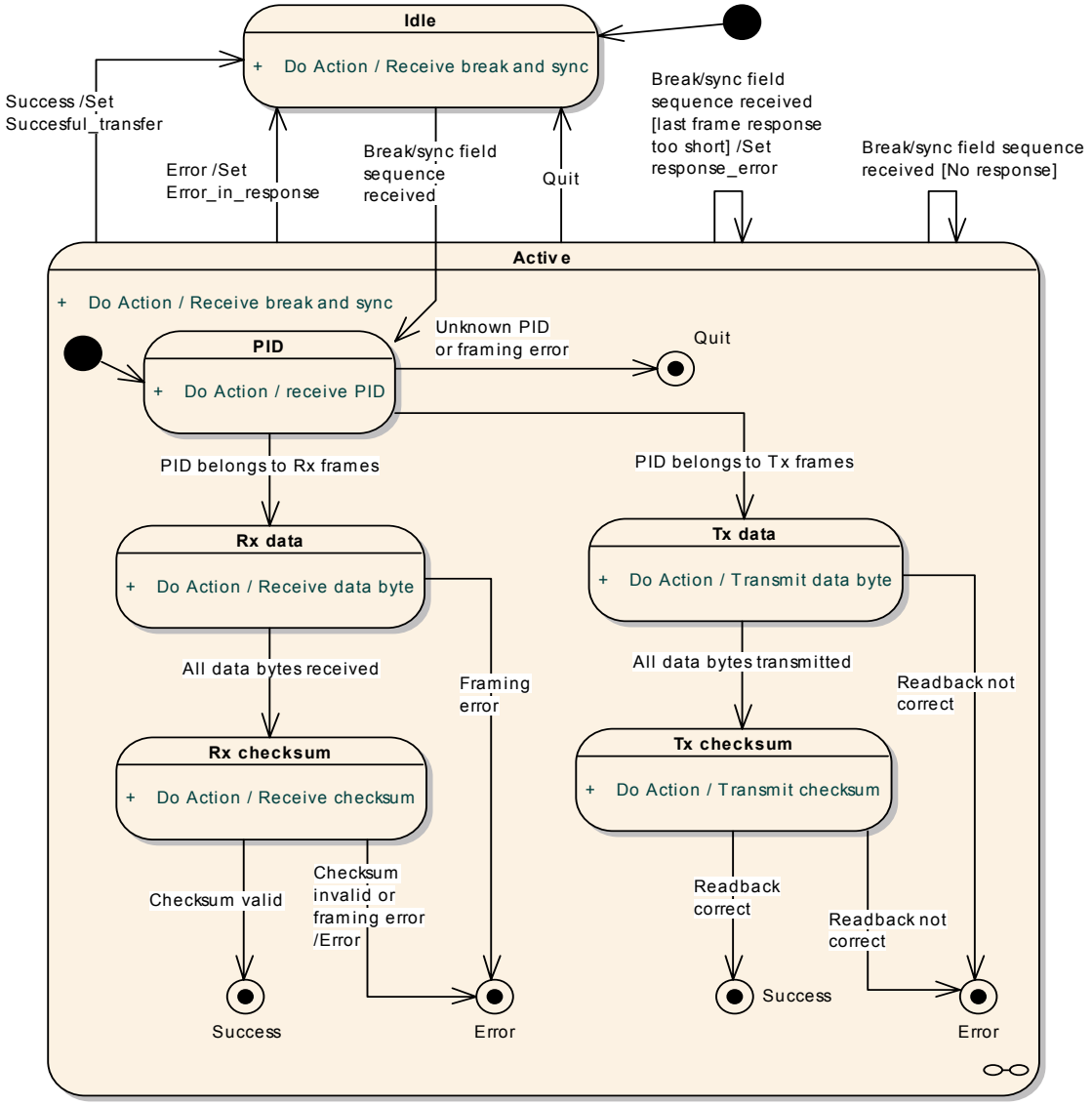


Figure 2.14: Frame processor state machine.

Error and Success refers to the status management described in section 2.7.

The last frame response too short means that the last frame contained at least one field (correct data byte or even framing error) in the response. This is to distinguish between error in response and no response.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.6 NETWORK MANAGEMENT

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

2.6.1 SLAVE COMMUNICATION STATE DIAGRAM

The state diagram in Figure 2.15 shows the behavior model for communication of a slave node.

Initializing

This state is entered after reset and after wakeup. The slave node will make necessary initialization and then enter the Operational state. The initialization here refers to the LIN related initialization. A reset and wakeup may imply different initialization.

Operational

The LIN protocol behavior (transmitting and receiving frames) specified in this document only applies to the Operational state.

Bus sleep mode

The level on the bus is set to recessive. Only the wake up signal may be transmitted on the cluster.

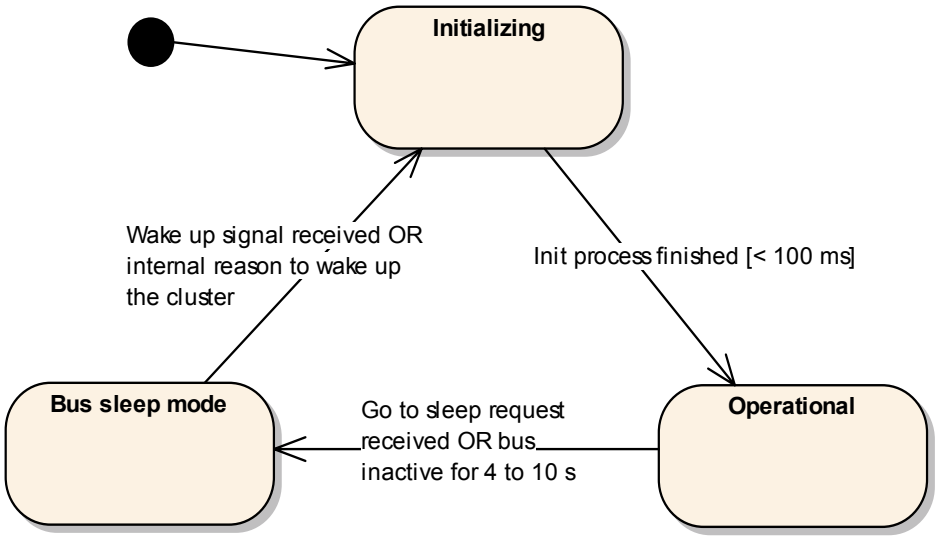


Figure 2.15: Slave node communication state diagram

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.6.2 WAKE UP

Any node in a sleeping LIN cluster may request a wake up, by transmitting a wake up signal. The wake up signal is issued by forcing the bus to the dominant state for 250 μs to 5 ms. The master node may issue a break field, e.g. by issuing an ordinary header since the break will act as a wake up signal (in this case the master must be aware of that this frame may not be processed by the slave nodes since they may not yet awake and ready to listen to headers).

Every slave node (connected to power) shall detect the wake up signal (a dominant pulse longer than 150 μs) and be ready to listen to bus commands within 100 ms, measured from the ending edge of the dominant pulse, see Figure 2.16. A detection threshold of 150 μs combined with a 250 μs pulse generation gives a detection margin that is enough for uncalibrated slave nodes. If the node that transmitted the wake up signal is a slave node, it will be ready to receive or transmit frames immediately. The master node shall also wake up and, when the slave nodes are ready, start transmitting headers to find out the cause (using signals) of the wake up.

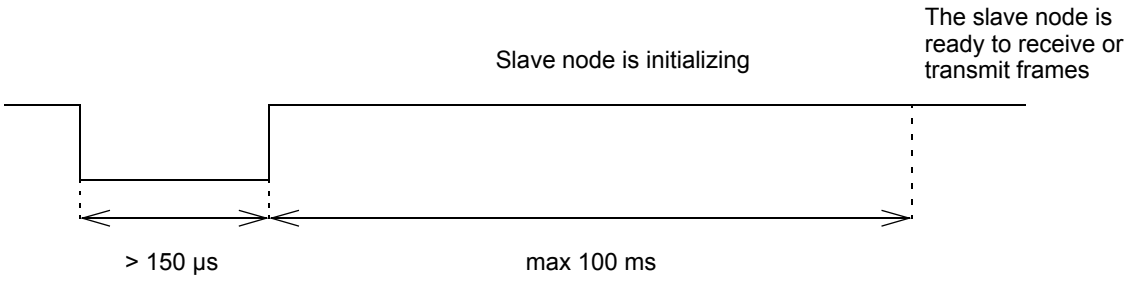


Figure 2.16: Wake up signal reception in slave nodes

If the master node does not transmit a break field (i.e. starts to transmit a frame) or if the node issuing the wake up signal does not receive a wake up signal (from another node) within 150 ms to 250 ms from the wake up signal, the node issuing the wake up signal shall transmit a new wake up signal, see Figure 2.17. In case the slave node transmits a wake up signal in the same time as the master node transmits a break field, the slave shall receive and recognize this break field.

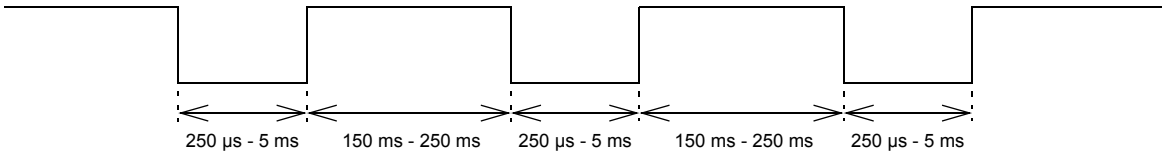


Figure 2.17: One block of wake up signals

Registered copy for cbarbaud@cmaisonneuve.qc.ca

After three (failing) requests the node shall wait minimum 1.5 seconds before issuing a fourth wake up signal. The reason for this longer duration is to allow the cluster to communicate in case the waking slave node has problems, e.g. if the slave node has problems with reading the bus it will probably retransmit the wake up signal infinitely.

There is no restriction of how many times a slave may transmit the wake up signal. However, it is recommended that a slave node transmits not more than one block of three wake up signals for each wake up condition. The Figure 2.18 shows how wake up signals are transmitted over a longer time

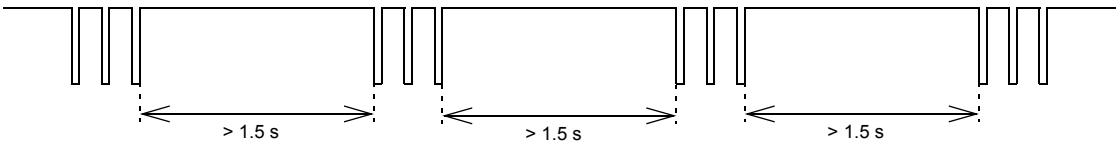


Figure 2.18: Wake up signals over long time

2.6.3 GO TO SLEEP

The master sets the cluster into bus sleep mode by transmitting a go to sleep command. The request will not necessarily enforce the slave nodes into a low-power mode. The slave node application may still be active after the go to sleep command has been received. This behavior is application specific.

The go to sleep command is a master request frame with the first data field set to 0 and rest set to 0xFF, see Table 2.1. The slave nodes shall ignore the data fields 2 to 8 and interpret only the first data field.

data1	data2	data3	data4	data5	data6	data7	data8
0	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF

Table 2.1: Go to sleep command

The normal way for setting the cluster to sleep is that the master node transmits the go to sleep command.

In case of bus inactivity a slave node must be able to receive/transmit frames for 4 s.

The slave node shall automatically enter bus sleep mode earliest 4 s and latest 10 s of bus inactivity. Bus inactivity is defined as no transitions between recessive and dominant bit values[1]. Bus activity is the inverse.

Note 1: LIN transceivers normally have filters to remove short spikes on the bus. The transition here refers to the signal after this filter.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.7 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.

2.7.1 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.

2.7.2 EVENT TRIGGERED FRAMES

Event triggered frames, section 2.3.3.2, are defined to allow collisions. Therefore, a bus error, e.g. framing error, shall not affect the response_error (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.

2.7.3 REPORTING TO THE CLUSTER

The master node will monitor the status on the cluster by checking the behavior of a specific signal published by all slave nodes.

Each slave node shall publish a one bit scalar signal, named response_error, to the master node in one of its transmitted unconditional frames. In case the unconditional frame is associated with an event triggered frame the frame should additionally be scheduled as unconditional. The response_error signal shall be set whenever a frame (except for event triggered frame responses) that is transmitted or received by the slave node contains an error in the frame response. The response_error signal shall be cleared when the unconditional frame containing the response_error signal is successfully transmitted.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Based on this single bit the master node can make the conclusions as presented in Table 2.2:

response_error	Interpretation
false	the slave node is operating correctly
true	the slave node has intermittent problems
The slave node did not answer	the slave node, bus or master node has serious problems

Table 2.2: Interpretation of the response_error

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.

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.

2.7.4 REPORTING WITHIN OWN NODE

This section applies to software based nodes, however ASIC based state machine implementations are recommended to use the same concepts. See section 7.2.5.8 for further information of this reporting.

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. It shall not be set in case there is no response.

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.

The reporting within the own node is standardized in the Application Program Interface 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.8 APPENDICES

2.8.1 TABLE OF NUMERICAL PROPERTIES

Property	Min	Max	Unit	Reference
Scalar signal size	1	16	bit	section 2.2.1
Byte array size	1	8	byte	section 2.2.1
Break field length (dominant + delimiter)	14		T_{bit}	section 2.3.1.1
Break detect threshold	11	11	T_{bit}	section 2.3.1.1
Wake up signal duration	0.25	5	ms	section 2.6.2
Slave initialize time		100	ms	section 2.6.2
Silence period between wake up signals	150	250	ms	section 2.6.2
Silence period after three wake up signals	1.5		s	section 2.6.2

Table 2.3: Defined numerical properties

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.8.2 TABLE OF VALID FRAME IDENTIFIERS

ID[0..5] Dec	ID[0..5] Hex	P0 = ID0⊕ID1⊕ID2⊕ID4	P1 = ¬ ID1⊕ID3⊕ID4⊕ID5	PID-Field P1 P0 5 4 3 2 1 0	PID-Field Dec	PID-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
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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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

a. Frame identifier 60 (0x3C) is reserved for the Master Request frame (see section 2.3.3.4).

b. Frame identifier 61 (0x3D) is reserved for the Slave Response frame (see section 2.3.3.4).

c. Frame identifier 62 (0x3E) and 63 (0x3F) are reserved for a future LIN extended format (see section 2.3.3.5).

Table 2.4: Valid frame identifiers

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.8.3 EXAMPLE OF CHECKSUM CALCULATION

Below is the checksum calculation of four bytes shown. If the frame have four data bytes or the protected identifier and three data bytes; the calculation is the same. Data = 0x4A, 0x55, 0x93, 0xE5

Action	hex	Carry	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

Table 2.5: Example of checksum calculation

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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

2.8.4 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 2.19. 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.

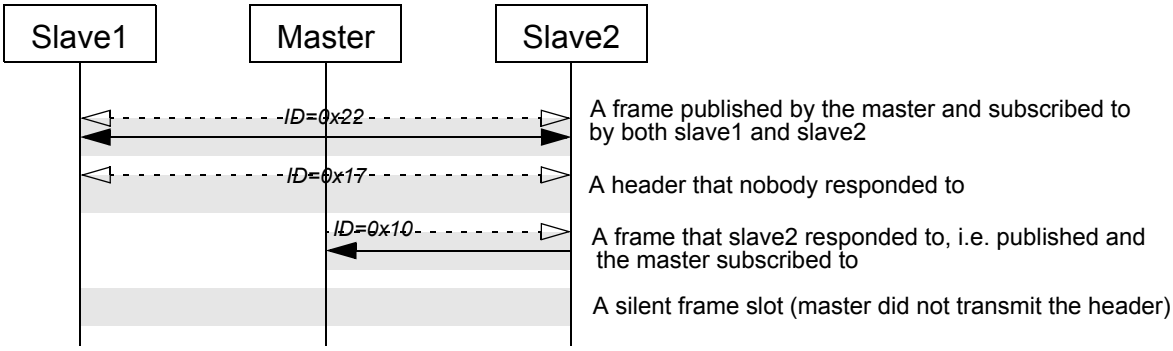


Figure 2.19: Frame sequence example.

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 or b is true.
$\neg a$	Negate. True if a is false.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Transport Layer Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

3.1 INTRODUCTION

The transport layer defines transportation of data that is contained in one or more frames.

The transport layer messages are transported by diagnostic frames as specified in the Protocol Specification. A standardized API for the transport layer is specified in the Application Program Interface Specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

3.2 TRANSPORT LAYER

Use of the transport layer is targeting systems where diagnostics are performed on the back-bone bus (e.g. CAN) 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 15765-2 transport layer [3] and the PDUs carrying the messages are very similar, as defined in Section 3.2.1. A typical system configuration is shown in Figure 3.1.

The goals of the transport layer are:

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

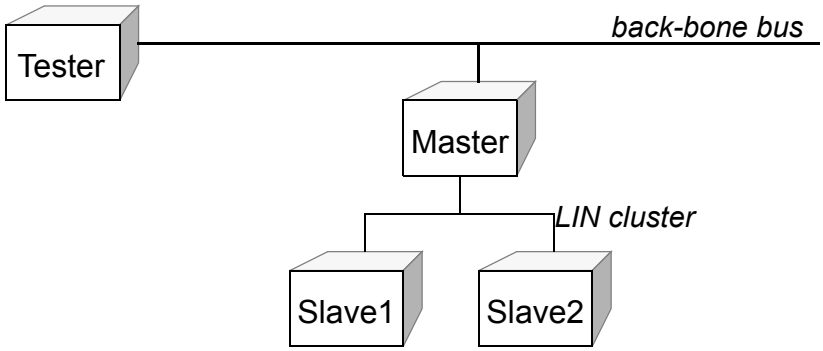


Figure 3.1: Typical system setup for a LIN cluster using the transport layer.

3.2.1 PDU STRUCTURE

The units that are transported in a transport layer 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 (tester, master node) are called requests and messages issued by the server (master node, slave node) are called responses.

The first byte in the pay-load is used as an node address (NAD). The transport layer frames have fixed frame IDs, since the diagnostic frames are used. This means that the addressing of a node (or function) is made using the NAD. In ISO 15765-2 transport layer [3] terms this means that the extended or mixed addressing are used.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Flow control [3] 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 on the back-bone side.

3.2.1.1 Overview

To simplify conversion between ISO transport layer frames [3] and LIN transport layer frames a very similar structure is defined, which support the PDU types shown in Figure 3.2. The left byte (NAD) is sent first and the right byte (D4, D5 or D6) is sent last.

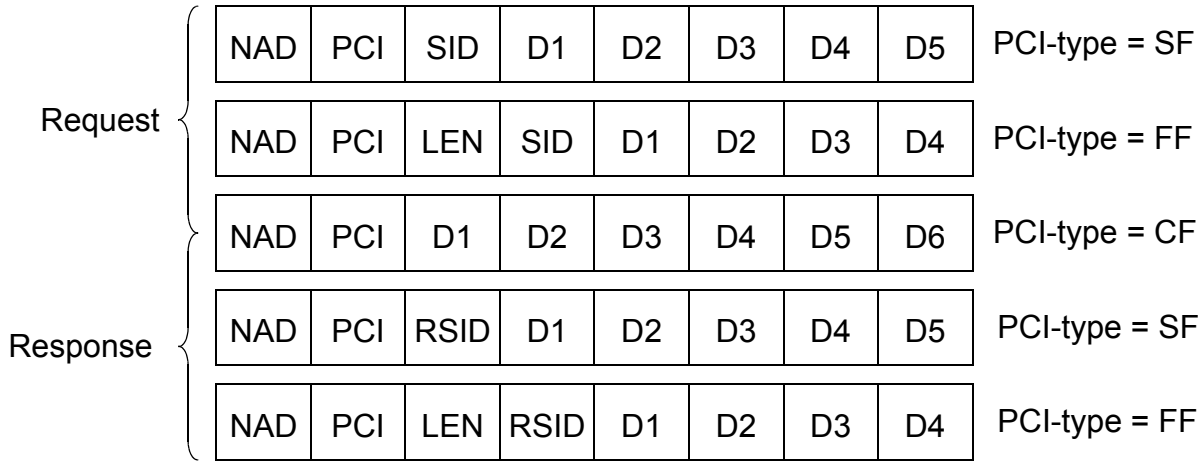


Figure 3.2: PDUs supported by the LIN transport layer.

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.

3.2.1.2 NAD

The NAD is defined in Section 4.2.3.2.

3.2.1.3 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.

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			

Table 3.1: Structure of the PCI byte.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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 Consecutive 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,...

3.2.1.4 LEN

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

3.2.1.5 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.

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

3.2.1.6 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).

3.2.2 COMMUNICATION

It is required that a transport layer message is exclusive on one bus. This means that only one message can be active at one time.

If a node is receives a message with a NAD equal to the node's own NAD or the broadcast NAD and no other message is active the message shall be received and processed.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

If a functional addressed message is received and no other message is active the message shall be received and processed.

Functional addressed messages shall be ignored by slave nodes while receiving a message.

3.2.2.1 Single Frame Transmission

Transmission of messages up to six bytes (including SID) shall be performed via transmission of a single frame PDU (SF).

Functional addressed messages can only be SF.

3.2.2.2 Multiple Frame Transmission

Transmission of messages with more than six bytes (including SID) up to a maximum of 4095 bytes is performed via segmentation and transmission of multiple PDUs. A segmented transmission starts with a First Frame PDU (FF) and continues with multiple Consecutive Frame PDUs (CF).

3.2.3 ERROR HANDLING

A Single Frame PDU (SF) with a length value greater than six (6) bytes shall be ignored by the receiver.

First Frame PDU (FF) with a length value less than seven (7) bytes shall be ignored by the receiver.

A First Frame PDU (FF) with a length value greater than the maximum available receive buffer size of the slave node shall be ignored by the receiver and the receiver shall not start the reception of a segmented message. This implies of course that the receiving node is receiving the complete message (the target node) and not a fragmented (in case of gateway).

PDUs with unexpected PCI types from any node shall be ignored except Single Frame (SF) and First Frame PDUs (FF).

After reception of a Single Frame (SF) or First Frame (FF) PDU, with a NAD that is not equal to the functional NAD, during an ongoing message transmission the current reception shall be aborted. Reception of the new message shall be started on the receiver side if the NAD equals the node's own NAD or broadcast NAD.

After reception of a Consecutive Frame (CF) with an unexpected sequence number (SN) the reception of the message transfer shall be aborted by the receiver.

The message reception shall be aborted by the receiver after occurrence of an N_Cr_LIN timeout.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The message transmission shall be aborted by the transmitter after occurrence of an N_As_LIN timeout, see Section 3.2.5.

3.2.4 DEFINED REQUESTS

The LIN transport layer uses the same diagnostic messages as the ISO 15765-3 diagnostics standard [4]. 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 [4].

3.2.5 TIMING CONSTRAINTS

The timing constraints for the transport layer (based on ISO-15765-2 [3]) is shown in Table 3.2. The properties shall be within a defined range. Since LIN is slower than CAN, the values have to be adjusted accordingly. These properties are part of the transport layer and will not have any constraint on the node configuration.

Performance requirement values are the binding communication requirements to be met by each communication peer in order to be compliant with the specification. Since the LIN schedules vary with the specific use-case a certain application may define specific performance requirements within the ranges defined in Table 3.2.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Timeout values are defined to be higher than the values for the performance requirements to ensure a working system and to overcome communication conditions where the performance requirement can absolutely not be met (e.g. high bus load). Specified timeout values in Table 3.2 or values given by the node (i.e. the NCF) shall be treated as the upper limit for any given implementation.

Timing	Description	Data Link Layer service		Timeout	Performance
Parameter		Start	End	(ms)	requirement (ms)
N_As	Time for transmission of the LIN frame (MRF or SRF) on the transmitter side	When the transport layer requests a diagnostic frame to be transmitted	When the diagnostic frame has been confirmed as transmitted	1000	N/A
N_Cs	Time until transmission of the next Consecutive Frame (CF)	When the last diagnostic frame in the same message has been confirmed as transmitted.	When the transport layer requests the CF to be transmitted	N/A	(N_Cs + N_As) < (0.9 * N_Cr timeout)
N_Cr	Time until reception of the next Consecutive Frame (CF)	When the previous diagnostic frame in the message has been indicated as received.	When the next diagnostic frame in the message has been indicated as received.	1000	-

Table 3.2: Transport layer timing parameters

Note: The N_Cs parameter does not require a timeout monitoring in the transmitting node since N_As ensures the correct timeout behavior. However N_Cs must be considered in the system design (scheduling and transmitter software design) so that a timeout on the receiver's side (N_Cr) can be avoided.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Figure 3.3 and Figure 3.4 illustrates the parameters in the time domain. The intention of the figures is to show the transport layer timing parameters, not to require a certain implementation. The behavior for the master node and the slave node in the lower layers are generalized.

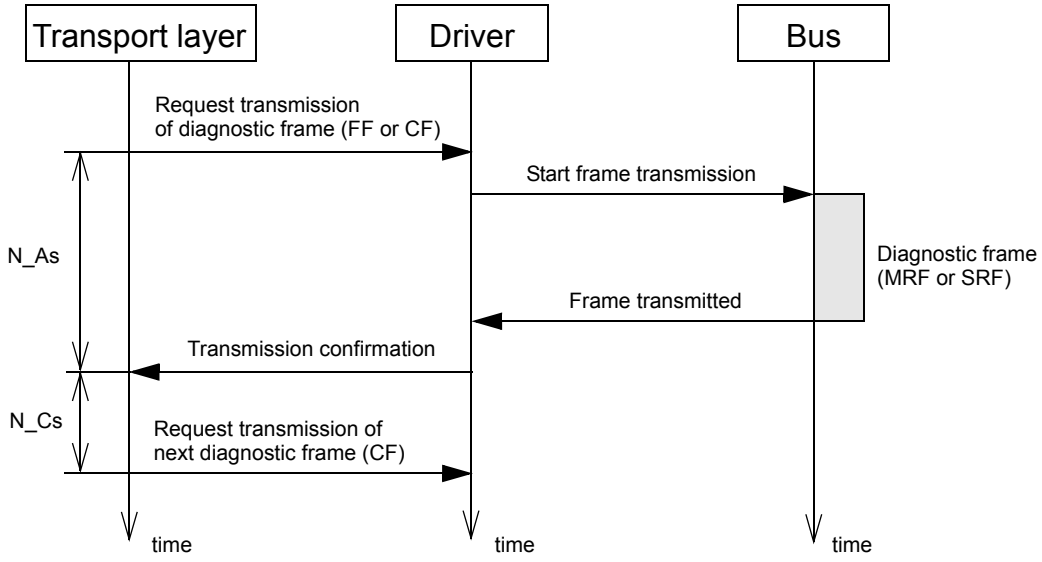


Figure 3.3: Transport layer timing on transmitter side

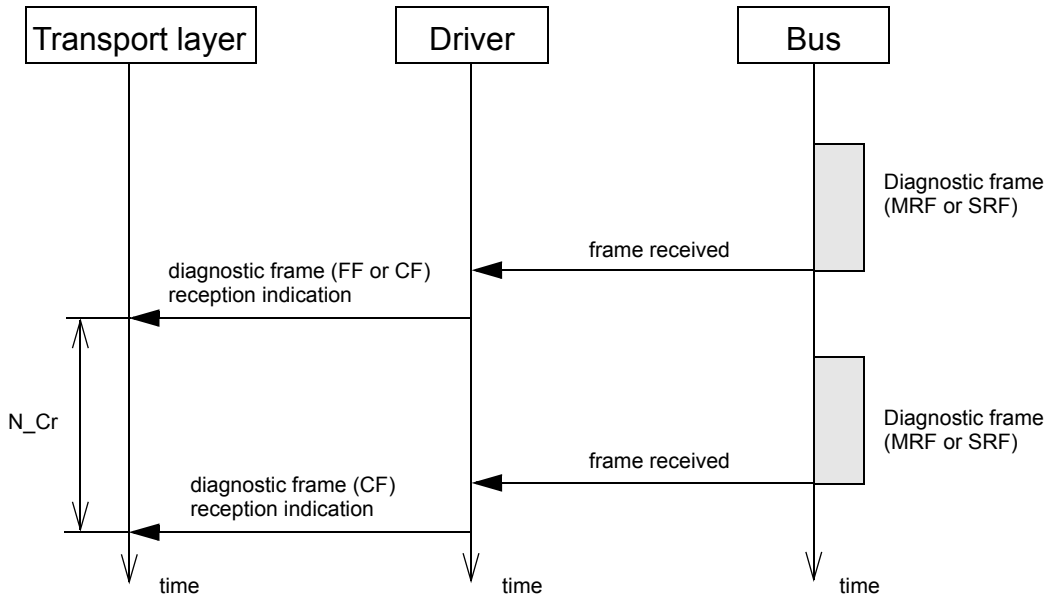


Figure 3.4: Transport layer timing on receiver side

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Node configuration and Identification Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.1 INTRODUCTION

The node configuration and identification services define how a slave node is configured, and identifying a slave node using the identification service.

The node configuration and identification services are transported by the transport layer as specified in the Transport Layer Specification. A standardized API for the node configuration and identification is specified in the Application Program Interface Specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.2 NODE CONFIGURATION AND IDENTIFICATION

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

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

4.2.1 LIN PRODUCT IDENTIFICATION

Each slave node shall have a LIN product identification, as outlined in Table 4.1.

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

Table 4.1: LIN product identification

The supplier ID is a 16 bit value, with the most significant bit equal to zero. Most significant bit set to one is reserved for future extended numbering systems. The supplier ID is assigned to each supplier by the LIN Consortium. The supplier ID shall represent the supplier of the fully operational slave node.

The function ID is a 16 bit value 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.

The variant ID is an 8 bit value. It shall be changed whenever the product is changed but with unaltered function. The variant ID is a property of the slave node and not the LIN cluster.

A slave node may have a serial number to identify a specific instance of a slave node product. The serial number is 4 bytes, as outlined in Table 4.2.

D1	D2	D3	D4
LSB	...	...	MSB

Table 4.2: Serial number

4.2.1.1 Wildcards

To be able to leave some information unspecified the wildcard values in Table 4.3 may be used in node configuration requests. All slave nodes shall understand the wildcards in requests.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Property	Wildcard value
NAD	0x7F
Supplier ID	0x7FFF
Function ID	0xFFFF

Table 4.3: Wildcards usable in all requests

4.2.2 SLAVE NODE MODEL

The memory of a slave node can be described as in Figure 4.1.

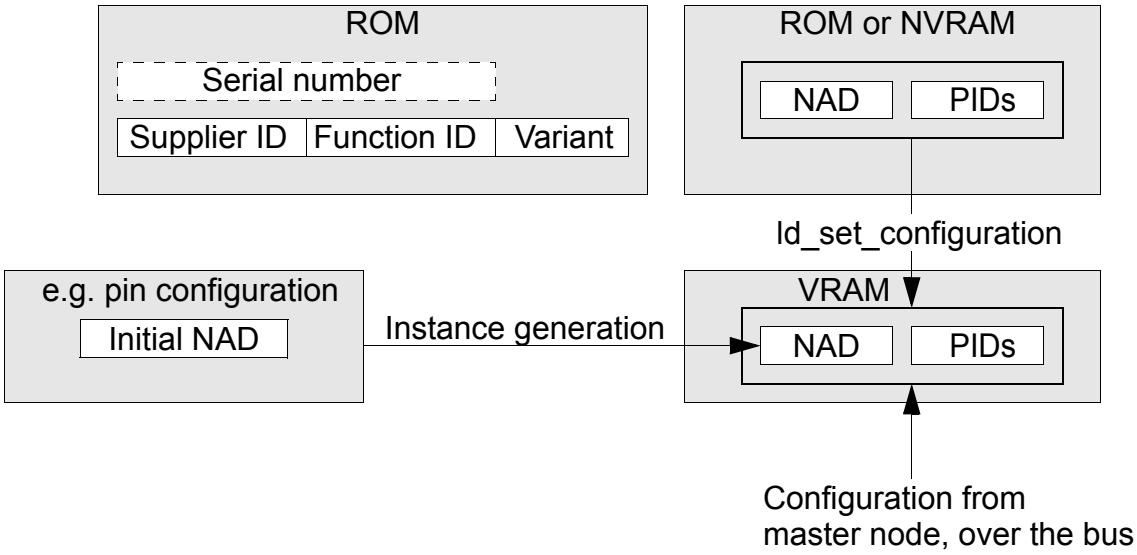


Figure 4.1: Slave node memory model.

VRAM (Volatile RAM) is considered a memory that is not valid after reset. The NVRAM (Non-Volatile RAM) is memory that is maintained after reset and can be modified with internal processes (e.g. the application). ROM (Read Only Memory) is considered as constant memory that cannot be modified with internal processes (e.g. application).

A slave node has a fixed LIN Product Identification, see Section 4.2.1. The serial number is optional.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Three slave node variants are defined:

- Unconfigured slave node - After reset the slave node does not contain a valid configuration. The slave node must be configured by the master after reset, because the configuration is stored in VRAM.
- Preconfigured slave node - This slave node has a valid configuration after reset (after l_ifc_init is called). The configuration is normally stored in ROM. But reconfigured data will be lost after reset.
- Full configured slave node - The slave node stores the configuration in NVRAM, so it will still be active after reset.

Note that all variants of the slave nodes above must understand at least the mandatory configuration services.

When a slave node enters operational state (see Protocol Specification section Section 2.6.1) it shall fulfill the following requirements for node configuration.

- It shall have a unique supplier ID, function ID and configured NAD combination.
- If the slave node does not contain a configuration, all frames (except the master request frame and slave response frame) in the slave node are marked as invalid.
- Understand and be able to process all supported configuration requests.

4.2.2.1 Initial NAD

Each slave node has an initial NAD list, defined in the NCF, see Section 8.2.4. For slave nodes that have no instance generation of the initial NAD the list contains only one entry. The instance generation will set the initial NAD based on the initial NAD list. The instance generation of the initial NAD is not part of this specification (e.g. the slave node position detection may be used for this).

The configuration, using ld_set configuration or Assign NAD, will set the NAD to the configured NAD. If the initial NAD is already equal to the configured NAD then no action is taken.

Figure 4.2 shows the relationship between the initial NAD list, the initial NAD and the configured NAD.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

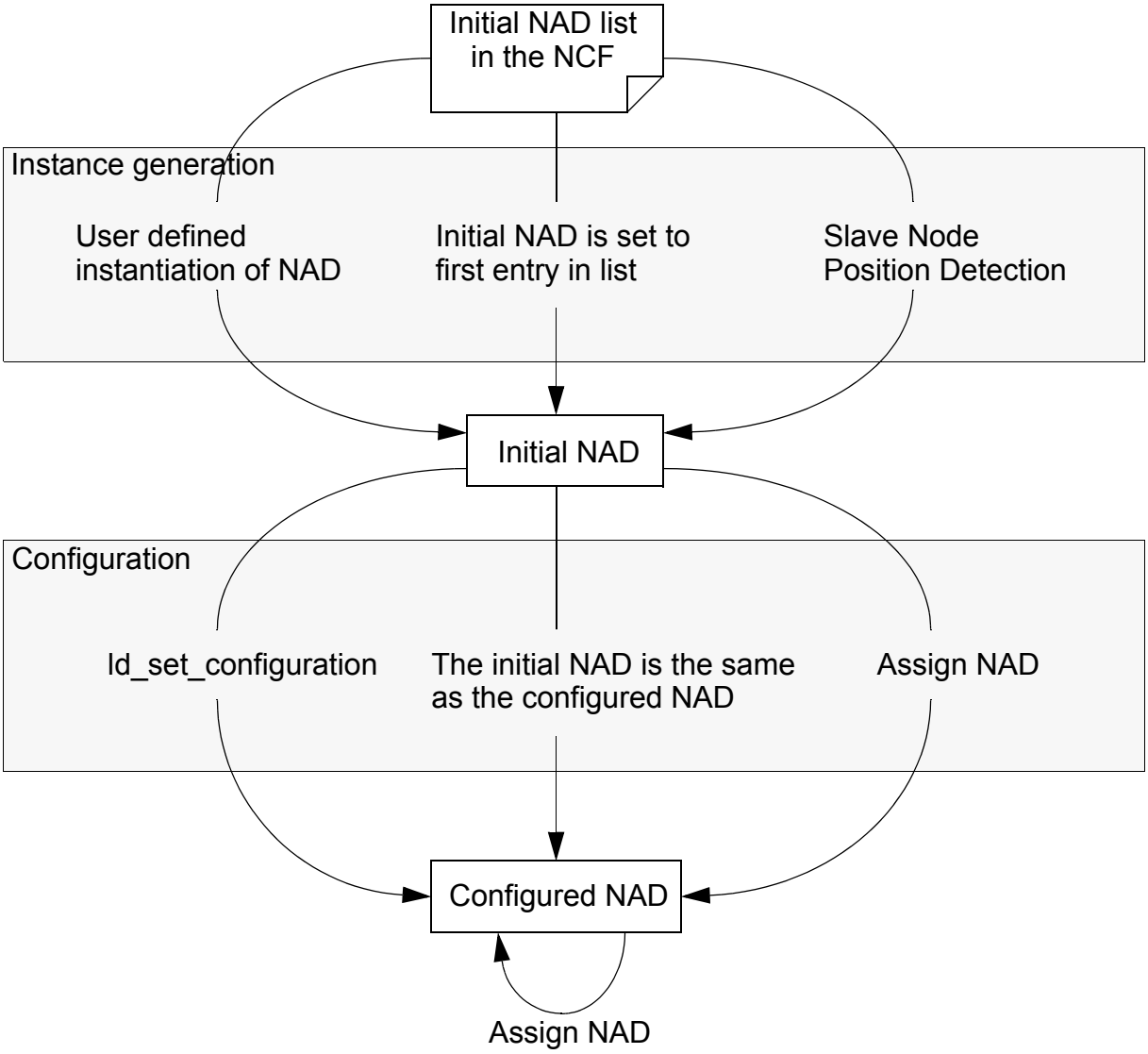


Figure 4.2: NAD instantiation and configuration process

4.2.3 PDU STRUCTURE

The node configuration and identification services are transported using the transport layer, see Transport Layer Specification. Only the single frame (SF) is used for all requests and responses here.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.2.3.1 Overview

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.

4.2.3.2 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 (0x7F), while 0 (zero) and 128 (0x80) to 255 (0xFF) are reserved for other purposes:

NAD value	Description
0	Reserved for go to sleep command, see Section 2.6.3
1 - 125 (0x7D)	Slave node addresses (NAD)
126 (0x7E)	Functional node address (functional NAD), only used for diagnostics (using the transport layer)
127 (0x7F)	Slave node address broadcast (broadcast NAD)
128 (0x80) - 255 (0xFF)	Free usage. 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. See user defined diagnostics Section 5.2.6.

Table 4.4: NAD values

Note that there is a one-to-many mapping between a physical slave node and a logical slave node and it is addressed using the NAD. This means that one physical slave node may be composed of several logical slave nodes.

Note: It is recommended not to use functional addressing during configuration since the behavior is different for LIN 2.1 and LIN 2.0 slave nodes.

4.2.3.3 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 4.5.

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

Table 4.5: Structure of the PCI byte for configuration PDUs.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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).

4.2.3.4 SID

The Service Identifier (SID) specifies the request that shall be performed by the slave node addressed. This SID numbering is consistent with ISO 15765-3 [4] and places node configuration (0xB0 to 0xB7) in an area “Defined by vehicle manufacturer”.

The following Table 4.6 shows what SIDs are used:

SID	Service	type	Reference
0 - 0xAF	reserved	reserved	See ISO15765-3 [4]
0xB0	Assign NAD	Optional	Section 4.2.5.1
0xB1	Assign frame identifier	Obsolete	See LIN2.0 specification
0xB2	Read by Identifier	Mandatory	Section 4.2.6.1
0xB3	Conditional Change NAD	Optional	Section 4.2.5.2
0xB4	Data Dump	Optional	Section 4.2.5.3
0xB5	Assign NAD via SNPD	Reserved by SNPD	See Node Position Detection specification [2]
0xB6	Save Configuration	Optional	Section 4.2.5.4
0xB7	Assign frame identifier range	Mandatory	Section 4.2.5.5
0xB8 - 0xFF	reserved	reserved	See ISO15765-3

Table 4.6: Node configuration and identification services

4.2.3.5 RSID

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

If the service is supported in the slave node the response is mandatory (even if the request is broadcast). The support of a specific service will be listed in the NCF, see Node Capability Language Specification.

A slave shall process the configuration request immediately and be able to respond in the next schedule slave response frame (t_sep_min and t_resp_min, see Section 5.6, are not used for node configuration).

4.2.3.6 D1 to D5

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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

4.2.4 NODE CONFIGURATION AND IDENTIFICATION

All requests are carried in master request frames, and all responses are carried in slave response frames. All requests and responses are using single frames only.

A slave node shall remember the response for a request until a new request with a NAD or broadcast NAD (i.e. any NAD except a functional NAD) from the master node. For example, if the master node transmits an unconditional frame between the request and the response the slave node shall not forget the response.

A master node may choose not to ask for the response from the slave node, e.g. after transmitting a broadcast (0x7F) request.

All services shall support the use of the wildcards, as defined in Section 4.2.1.1.

4.2.5 NODE CONFIGURATION SERVICES

4.2.5.1 Assign NAD

Assign NAD is used to resolve conflicting NADs in LIN clusters built using off-the-shelves slave nodes or reused slave nodes. This request uses the initial NAD (or the NAD wildcard); this is to avoid the risk of losing the address of a slave node. The NAD used for the response shall be the same as in the request, i.e. the initial NAD.

It shall be structured as shown in Table 4.7.

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 4.7: Assign NAD request

A response shall only be sent if the NAD, the Supplier ID and the Function ID match. If successful, the message in Table 4.8 shall be sent as response.

NAD	PCI	RSID	Unused				
Initial NAD	0x01	0xF0	0xFF	0xFF	0xFF	0xFF	0xFF

Table 4.8: Positive assign NAD response

Note that the response is using the initial NAD and not the new NAD.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.2.5.2 Conditional change NAD

The conditional change NAD is used to detect unknown slave nodes in a cluster and to separate their NADs. Potential reasons for unknown slave nodes to appear in a cluster are, e.g. incorrect assembly when manufacturing the cluster or incorrect slave node replacement during service. This service will be used a number of times until the NADs in the slave nodes are separated. The outcome will be a conflict free cluster where the master node can identify the slave nodes.

The behavior in the slave node when the request is received:

1. Get the identifier specified in Table 4.20 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.

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

Table 4.9: Conditional change NAD request

A response from the slave node shall be structured as shown in Table 4.10.

NAD	PCI	RSID	Unused				
NAD	0x01	0xF3	0xFF	0xFF	0xFF	0xFF	0xFF

Table 4.10: Positive Conditional change NAD response

The Conditional Change NAD is addressed with the current NAD, i.e. it does not use the initial NAD as opposed to the Assign NAD request.

Example

Two slave nodes in a cluster are designed so that the lower byte (LSB) of the function ID decides the NAD. If the first bit in the function ID is set then NAD will be set to 1, if the second bit is set the NAD will be set to 2.

The master node will then transmit two conditional change NAD services with the following parameters:

NAD	PCI	SID	Id	Byte	Mask	Invert	New NAD
0x7F	0x06	0xB3	0x01	0x03	0xFE	0xFF	0x01
0x7F	0x06	0xB3	0x01	0x03	0xFD	0xFF	0x02

Table 4.11: Conditional change NAD example

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.2.5.3 Data dump

This service is reserved for initial configuration of a slave node by the slave node supplier and the format of this message is supplier specific. This service shall only be used by supplier diagnostics and not in a running cluster, e.g at the car OEM. The data dump request shall be structured as shown in Table 4.12.

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB4	User defined	User defined	User defined	User defined	User defined

Table 4.12: Data dump request

A response from the slave node shall be structured as shown in Table 4.13.

NAD	PCI	RSID	D1	D2	D3	D4	D5
NAD	0x06	0xF4	User defined	User defined	User defined	User defined	User defined

Table 4.13: Data dump response

4.2.5.4 Save Configuration

This service tells the slave node(s) that the slave application shall save the current configuration. The save configuration request shall be structured as shown in Table 4.14. This service is used to notify a slave node to store its configuration. A configuration in the slave node may be valid even without the master node using this request (i.e. the slave node does not have to wait for this request to have a valid configuration).

NAD	PCI	SID	Unused				
NAD	0x01	0xB6	0xFF	0xFF	0xFF	0xFF	0xFF

Table 4.14: Save configuration request

The software based (using the API) slave node will set a flag in the status register, see Section 7.2.5.8, after receiving the request. It will not store the configuration automatically. The application has to read out the configuration, see Section 7.3.1.6, and save the configuration in a non-volatile space.

After reception of the service and the NAD is correct the slave node shall respond (not wait until the configuration is saved). The positive response from the slave shall be structured as shown in Table 4.15.

NAD	PCI	RSID	Unused				
NAD	0x01	0xF6	0xFF	0xFF	0xFF	0xFF	0xFF

Table 4.15: Save configuration positive response

Registered copy for cbarbaud@cmaisonneuve.qc.ca

4.2.5.5 Assign frame ID range

Assign frame ID range is used to set or disable PIDs up to four frames. The request shall be structured as shown in Table 4.16.

It is important to notice that the request provides the protected identifier, i.e. the frame identifier and its parity. Furthermore, frames with frame identifiers 60 (0x3C) to 63 (0x3F) can not be changed (diagnostic frames and reserved frames).

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB7	start index	PID (index)	PID (index+1)	PID (index+2)	PID (index+3)

Table 4.16: Assign frame PID range request

A response shall only be sent if the NAD match. If successful, the message in Table 4.17 shall be transmitted as a response.

NAD	PCI	RSID	unused				
NAD	0x01	0xF7	0xFF	0xFF	0xFF	0xFF	0xFF

Table 4.17: Positive assign frame PID range response

The start index specifies which is the first frame to assign a PID. The order of the list is specified in the node attributes section in the NCF and LDF of the slave node, see Section 8.2.5 respective Section 9.2.2.2. The first frame in the list has index 0 (zero).

The PIDs are an array of four PID values that will be used in the configuration request. Valid PID values here are the PID values for signal carrying frames, the unassign value 0 (zero) and the do not care value 0xFF. The unassign value is used to invalidate this frame for transportation on the bus. The do not care is used to keep the previous assigned value of this frame.

In case the slave cannot fulfill all of the assignments set PID or unassign or do not care, the slave shall reject the request. The do not care can always be fulfilled.

The slave node will not validate the given PIDs (i.e. validating the parity flags), the slave node relies on that the master sets the correct PIDs.

Example 1

A slave node has five frames {power_status, IO_1, IO_2, IO_3, IO_4}. The master node application will setup a assign frame id request with the parameters: start index set to 1 and PID (index 1..4) set to {0x80, 0xC1, 0x42, 0x0}. When the slave node receives the request it will set the PIDs to {IO_1=0x80, IO_2=0xC1, IO_3=0x42, IO_4=unassigned}. Note that the power_status frame will not be affected. The slave will respond with a positive response if requested.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Example 2

A slave node has only two frames {status_frame, response_frame}. To assign PIDs to these two frames the master application will setup the following request: start index set to 0 and PID (index 0..3) set to {0xC4, 0x85, 0xFF, 0xFF}. Since the slave node has only two frames the last two must be set to do not care, otherwise the request will fail.

4.2.6 IDENTIFICATION

4.2.6.1 Read by identifier

It is possible to read the supplier identity and other properties from a slave node using the request in Table 4.18.

NAD	PCI	SID	D1	D2	D3	D4	D5
NAD	0x06	0xB2	Identifier	Supplier ID LSB	Supplier ID MSB	Function ID LSB	Function ID MSB

Table 4.18: Read by identifier request

The identifiers defined are listed in Table 4.19.

Identifier	Interpretation	Length of response
0	LIN Product Identification	5 + RSID
1	Serial number	4 + RSID
2 - 31	Reserved	-
32 - 63	User defined	User defined
64 - 255	Reserved	-

Table 4.19: Supported identifiers using read by identifier request.

Support of identifier 0 (zero) is the only mandatory identifier, i.e. the serial number is optional.

If the slave successfully processed the request it will respond according to Table 4.20. 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
32-63	NAD	0x05	0xF2	user defined	user defined	user defined	user defined	user defined

Table 4.20: Possible positive read by identifier response.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

If the slave is not supporting this request or could not process the request it will respond according to Table 4.21.

NAD	PCI	RSID	D1	D2	Unused		
NAD	0x03	0x7F	Requested SID (= 0xB2)	Error code (= 0x12)	0xFF	0xFF	0xFF

Table 4.21: Negative response

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Diagnostic specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.1 INTRODUCTION

The LIN diagnostics defines methods to implement diagnostic data transfer between a master node, respectively a diagnostic tester, and the slave nodes.

Three different classes of diagnostic nodes are defined. Class I is using normal signalling and class II and class III uses the transport layer, see Transport Layer Specification.

5.1.1 USING THE TRANSPORT LAYER

Two communication cases exist using the transport layer; the tester wants to transmit a diagnostic request to a slave node OR the slave node wants to transmit a diagnostic response to the tester. Below, Figure 5.1 and Figure 5.2 shows the message flow in these two cases.

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



Figure 5.1: Gatewaying of CAN messages to LIN.

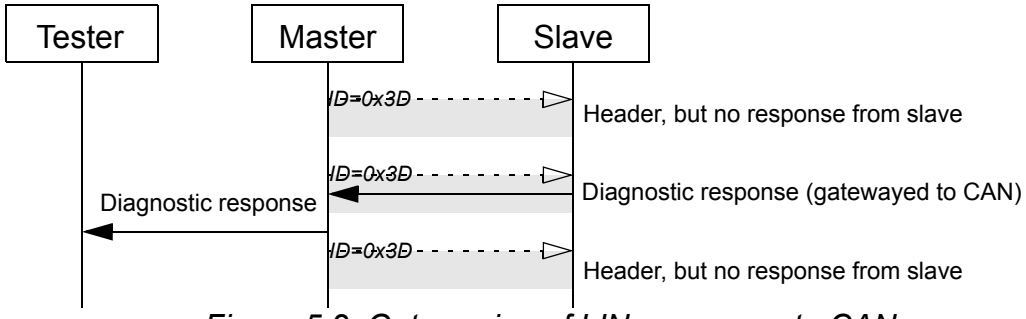


Figure 5.2: Gatewaying of LIN messages to CAN.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.1.2 LIN MASTER

Since the master node usually is a high performance ECU, normally supports the ISO 14229-1 [5] protocol or its customer specific recommended practice. The master node and the diagnostic tester are connected via a back-bone bus (e.g. CAN). The master node shall receive all diagnostic requests addressed to the slave nodes from the back-bone bus, and gateway them to the correct LIN cluster(s). Responses from the slave nodes shall be gatewayed back to the back-bone bus through the master node.

All diagnostic requests and responses (services) addressed to the slave nodes can be routed in the network layer (i.e. no application layer routing), if the Diagnostic and Transport Layer Protocol of tester back-bone-bus master node fulfills the respective needs. In this case, the master node must implement the LIN transport protocol, see Transport Layer Specification, as well as the transport protocols used on the back-bone busses (e.g. ISO15765-2 on CAN). See Section 7.4 for details about the implementation of gatewaying in the LIN master.

5.1.3 SLAVE NODES

Slave nodes are typically electronic devices that are not involved in a complex data communication. Also, their need of distributing diagnostic data is low. However, most slaves must transmit simple diagnostic information such as error indications in signal carrying frames, see Section 2.7.3.

Due to different reasons, e.g. for architectural reasons, diagnostic and transport protocol shall be realized as defined in this chapter. LIN diagnostics defines 3 classes of diagnostic slave nodes detailing the diagnostic communication and performance.

The diagnostic data is transported by the LIN protocol as specified in the Protocol Specification. A standardized API for the C programming language is specified in the Application Program Interface Specification.

Although diagnostics and node configuration services use the same frame IDs, i.e. 0x3C (master request frame) and 0x3D (slave response frame), different services are used for configuration and diagnostics. Node configuration can be performed by the master node independently while diagnostic services are always routed on request from an external or internal test tool. Both use-cases use the same node address (NAD) and transport protocol with the exception that configuration is always performed via Single Frames. Only slave nodes have a NAD, see Section 4.2.3.2. The NAD is also used as the source address in a diagnostic slave response frame.

Note that there is a one-to-many mapping between a physical node and a logical node and it is addressed using the NAD. See Section 9.2.2.3.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.2 DIAGNOSTIC CLASSES

Diagnostics in slave nodes are divided into three different diagnostic classes. A diagnostic class is assigned to each slave node according to its level of diagnostic functionality and complexity.

5.2.1 DIAGNOSTIC CLASS I

Smart and simple devices like intelligent sensors and actuators, requiring none or very low amount of diagnostic functionality. Actuator control, sensor reading and fault memory handling is done by the master node, using signal carrying frames. Therefore, specific diagnostic support for these tasks is not required. Fault indication is always signal based.

5.2.1.1 Transport protocol

Diagnostic frames usage is limited to node configuration. Therefore single frame (SF) transport protocol support is sufficient.

5.2.1.2 Diagnostic services

Only the node configuration services as defined in Table 4.6 are supported. The slave does not support any other diagnostic services.

Node Identification is limited to the mandatory read by identifier service, see Section 4.2.6.1.

5.2.2 DIAGNOSTIC CLASS II

A class II slave node is similar to a class I slave node, but is provides node identification support. The extended node identification is normally required by vehicle manufacturers. Testers or master nodes use ISO 14229-1 [5] diagnostic services to request the extended node identification information. Actuator control, sensor reading and fault memory handling is done by the master node, using signal carrying frames. Therefore, specific diagnostic support for these tasks is not required. Fault indication is always signal based.

5.2.2.1 Transport protocol

Full transport layer implementation, see Transport Layer Specification, is required to support multi-frame transmissions.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.2.2.2 Diagnostic services

Slave-nodes must support a set of ISO 14229-1 [5] diagnostic services:

- Node identification (SID 0x22) defined by the user e.g. reading HW and SW version, HW part number, diagnostic version
- Reading data parameter (SID 0x22) if applicable. Data parameter means: every data that can be read from the ECU, e.g. oil temperature, vehicle speed
- Writing parameters (SID 0x2E) if applicable

Diagnostic class II slave nodes may also be operated as diagnostic class I slave node in clusters if the master node does not support the diagnostic class II (i.e. does not implement the full LIN transport protocol). According to Table 5.1, a Class II slave node fulfills all Class I requirements.

If a slave node complies with diagnostic class I and diagnostic class II, it shall not require execution of diagnostic class II services to be able to operate normally. All minimum required sensor/actuator I/O and fault handling shall be transmitted via signal carrying frames.

5.2.3 DIAGNOSTIC CLASS III

Diagnostic class III slave nodes are devices with enhanced application functions, typically doing their own local information processing (e.g. function controllers, local sensor/actuator loops). These slave nodes execute tasks beyond the basic sensor/actuator functionality, and therefore require extended diagnostic support. Direct actuator control and raw sensor data is often not exchanged with the master node, and therefore not included in signal carrying frames. ISO 14229-1 [5] diagnostic services for I/O control, sensor value reading and parameter configuration (beyond node configuration) are required.

Class III slave nodes have internal fault memory, along with associated reading and clearing services. Optionally, reprogramming (flash/eeprom reprogramming) of the slave node is possible. This requires an implementation of a boot-loader and necessary diagnostic services to unlock the device, initiate downloads and transfer data, etc.

5.2.3.1 Addressing

Class III slave nodes require a unique NAD in the cluster.

5.2.3.2 Transport protocol

Full transport layer, see Transport Layer Specification, is required to support multi-frame transmissions.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.2.3.3 Diagnostic services

Slave nodes shall support all services as of Class II. Additionally, the services according Table 5.1 may be supported depending on the features which are implemented by the slave node.

Only class III slave nodes can reprogram the application via the LIN bus. Flash reprogramming via LIN bus is not defined in this specification. In case this functionality is supported the relevant services must be implemented in the slave node.

5.2.4 SUMMARY OF SLAVE NODE CLASSES

Table 5.1 shows a list of diagnostic services supported by the different diagnostic classes.

All supported configuration and diagnostic services of a slave node are listed in the node capability file, see Node Capability Language Specification.

+ = it is mandatory for the class
empty cell = not applicable, not supported

Slave Diagnostic Class	I	II	III	UDS service index [Hex]
Diagnostic Transport Protocol Requirements				
Single frame transport only	+			
Full transport protocol (multi-segment)		+	+	
Required Configuration Services				
Assign frame identifier range	+	+	+	0xB7
Read by identifier (0 = product id)	+	+	+	0xB2 0x00
Read by identifier (all others)	optional	optional	+	0xB2 0xXX
Assign NAD	optional	optional	optional	0xB0
Conditional change NAD	optional	optional	optional	0xB3
Positive response on supported configuration services	+	+	+	service + 0x40
Required UDS Services				
Read data by identifier:				0x22
- hardware and software version		+	+	0x22
- hardware part number (OEM specific)		+	+	0x22
- diagnostic version		+	+	0x22
Read by identifier (parameters)		+	+	0x22
Write by identifier (parameters)		if applicable	if applicable	0x2E

Table 5.1: Slave node diagnostic properties

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Slave Diagnostic Class	I	II	III	UDS service index [Hex]
Session control			+	0x10
Read by identifier (sensor and actuator data)			+	0x22
I/O control by identifier			+	0x2F
Read and clear DTC (fault memory)			+	0x19, 0x14
Routine control			if applicable	0x31
Other diagnostic services			if applicable	...
Flash Reprogramming Services				
Flash programming services			optional	0xXX

Table 5.1: Slave node diagnostic properties

5.2.5 MASTER NODE REQUIREMENTS

5.2.5.1 Transport protocol

If only Diagnostic Class I slaves are on the LIN cluster, implementation of minimum required LIN configuration support is sufficient. The master does not need to implement the full LIN diagnostic transport protocol.

For Diagnostic Class II and III slave nodes on the LIN-bus, the master node shall implement the full LIN transport layer

5.2.5.2 Fault management, sensor reading, I/O control

Diagnostic Class I and Diagnostic Class II slave nodes provide signal based fault information, and sensor and I/O access via signal carrying frames. The master node is responsible to handle the slave nodes signal based faults and handle the associated DTCs. It serves UDS requests directly to the tester, and acts as a diagnostic application layer gateway. UDS services provide access to the sensor/actuator signals on the LIN bus.

Diagnostic Class III slave nodes appear as independent diagnostic entities. The master node does not implement diagnostic services for the diagnostic capabilities of its Diagnostic Class III slave nodes.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.2.6 USER DEFINED DIAGNOSTICS

In addition to above classes it is also 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 4.2.3.2. The characteristics of a solution based on the user defined diagnostic are:

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

Since the user defined transportation is not standardized, this is not the preferred solution.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.3 REQUIREMENTS FOR SIGNAL BASED DIAGNOSTICS

Signal based diagnostics are implemented by slave nodes (diagnostic class I and II), which do not implement a fault memory and the diagnostic protocol to directly access this fault memory from an external test tool.

There are two types of failure transmission via signal carrying frames:

1. Failure information periodically transmitted and encoded into an existing signal (e.g. upper values of signal range used to indicate specific failure conditions)
2. Failure information not periodically transmitted for components which do not generate a signal that is periodically transmitted (e.g. slave node internal failure)

Since failure type 1 is use-case specific and defined by OEMs it is not standardized here.

Additional signal based failure transmission shall be implemented for type 2 failures (i.e. if a slave node is capable of locally detecting faults which are not transmitted via the associated signal in signal carrying frames already).

A failure status signal shall be assigned for each failure that would result in a separate DTC in the master node.

Each slave node shall transmit the failure status information that is monitored by the slave node to the master node via signal carrying frames. The status information shall contain the current failure status of the slave nodes's components. The signal shall support the states as defined in Table 5.2.

Description
no test result available, default, initialization value
test result: failed
test result: passed

Table 5.2: Signal based fault states

This information is used to indicate a failure of one of the components to the master node's application, which can then store the associated DTC.

There should be one signal per replaceable component to simplify maintenance and repair.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

If a slave node implements more than one independent function, a status signal can be assigned to each function. In this case only the failing function could be disabled by the application.

The fault state signals are set in the NCF, status management clausal, see Section 8.2.6.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.4 TRANSPORT PROTOCOL HANDLING IN LIN-MASTER

The LIN master is responsible for handling the scheduling according to the currently active diagnostic transmissions. This section defines the requirements for schedules and schedule handling which has to be implemented to enable diagnostic communication with any slave node. The master node acts as a network layer router between the back-bone bus and the LIN cluster, implying that the transport protocols on the back-bone bus and on the LIN cluster are handled by the master.

Within the next chapters several communication sequence diagrams are used. Refer to Figure 5.3 for the explanation of communication diagrams. For clearness of the communication sequence charts, the size of the interleaved normal communication schedules as well as the size of the diagnostic schedules does not represent the correct delays in the schedules and is only a schematic graphical representation.

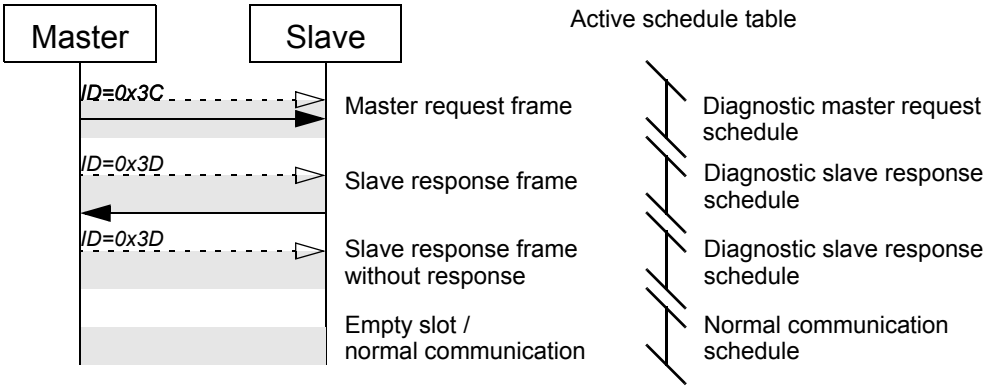


Figure 5.3: Legend of communication sequence charts

The transport layer defines a special functional NAD (0x7E) that is used to broadcast diagnostic requests. The broadcast NAD (0x7F) is normally not used in diagnostics, since there will be collisions if the master requests responses. If used anyway, the behavior will be the same as if a request with the slave node's own NAD was received (i.e. same behavior as for the configuration).

5.4.1 DIAGNOSTIC MASTER REQUEST SCHEDULE

The master node shall support a diagnostic master request schedule table that contains a single master request frame.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The diagnostic master request schedule table shall be executed whenever a master request frame shall be transmitted (see Figure 5.4). Note that by insertion of an execution of the diagnostic master request schedule table the overall timing of the normal communication schedule is affected.

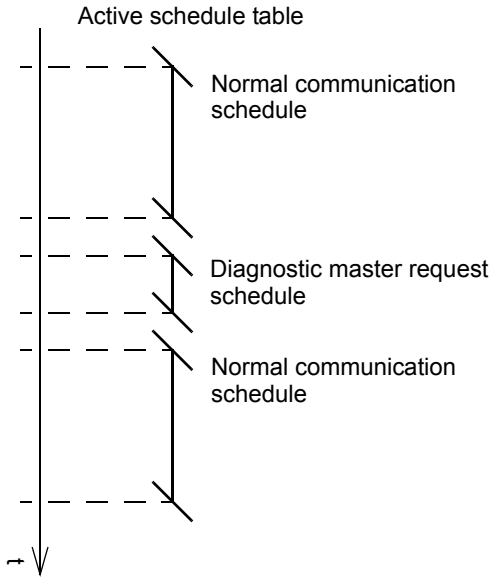


Figure 5.4: Interleaving of a diagnostic master request schedule table

5.4.2 DIAGNOSTIC SLAVE RESPONSE SCHEDULE

The master node shall support a diagnostic slave response schedule table that contains a single slave response frame.

The diagnostic slave response schedule table shall be inserted between the executions of the normal communication schedules whenever a slave response frame shall be transmitted (see Figure 5.5). Note that by insertion of an additional execution of the diagnostic slave response schedule table the overall timing of the normal communication is affected.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

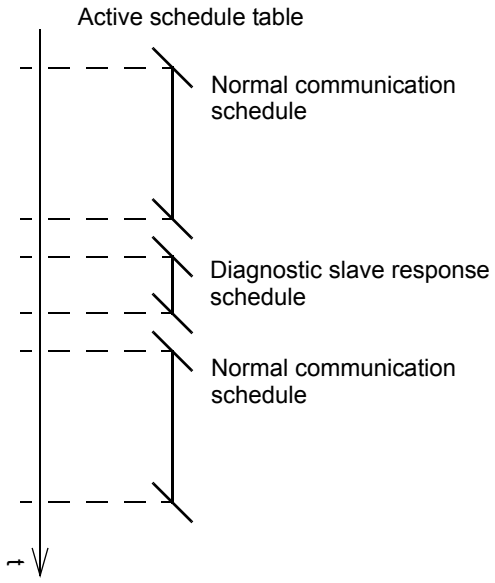


Figure 5.5: Interleaving of a diagnostic slave response schedule table

5.4.3 DIAGNOSTIC SCHEDULE EXECUTION

When no diagnostic communication is active, the master node shall not execute diagnostic schedules tables (see Figure 5.6). This shall be the default behavior of the master node.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

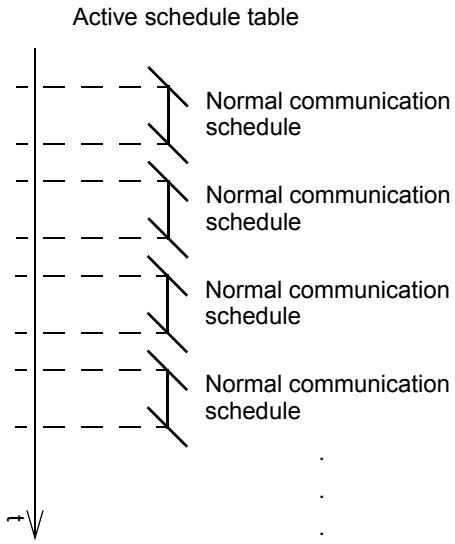


Figure 5.6: No diagnostic communication

A master node supports the following different scheduling modes:

1. Interleaved Diagnostics Mode (mandatory)
2. Diagnostics-Only-Mode (optional)

The two modes are defined in greater detail in Section 5.4.3.1 and Section 5.4.3.2.

The master node shall support to operate each of its connected LIN clusters in the one or the other mode upon request from an external diagnostic test tool.

5.4.3.1 Diagnostics Interleaved Mode

When diagnostic schedules need to be executed, the master node shall finish the currently running normal communication schedule and then switch to the required diagnostic schedule to perform the transmission (see Figure 5.4 and Figure 5.5). After execution of a diagnostic schedule the master node shall execute a normal communication schedule before executing the next diagnostic schedule. This is the Diagnostics Interleaved Mode (see Figure 5.7) and shall be the default mode of the master node.

When using the Diagnostics Interleaved Mode it shall be ensured (via normal communication schedule design) that the time between two subsequent diagnostic schedules fulfills the OEM specific diagnostic requirements.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

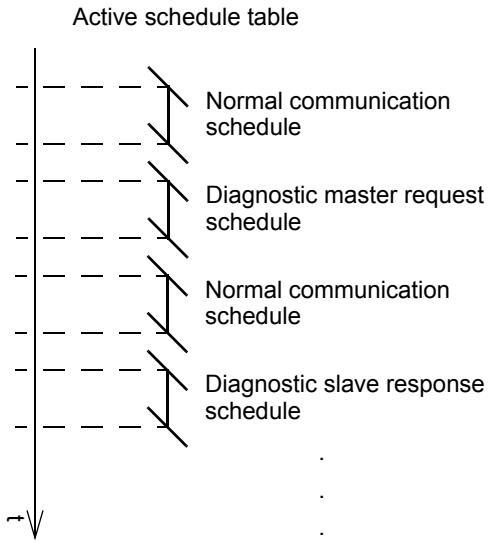


Figure 5.7: Normal diagnostic communication mode

The number of executions of the diagnostic master request schedule table depends on the amount of data that needs to be transmitted and shall be determined by the master node considering the LIN transport protocol (e.g. Two executions of the schedule for transmitting 10 user data bytes using the LIN transport protocol).

The subsequent interleaved execution of the diagnostic slave response schedule table depends on the amount of data to be transferred and shall therefore be performed by the master node until the transmission has been successfully finished or a transport protocol timeout occurs.

If a diagnostic transmission from a slave node to the master node has been started, the master node shall keep executing the diagnostic slave response schedule even when one or several slave response frame headers have not been answered (see Figure 5.8) until:

- A P2max / P2*max timeout occurs, see Section 5.6.
- A transport protocol timeout occurs, see Section 3.2.5

Registered copy for cbarbaud@cmaisonneuve.qc.ca

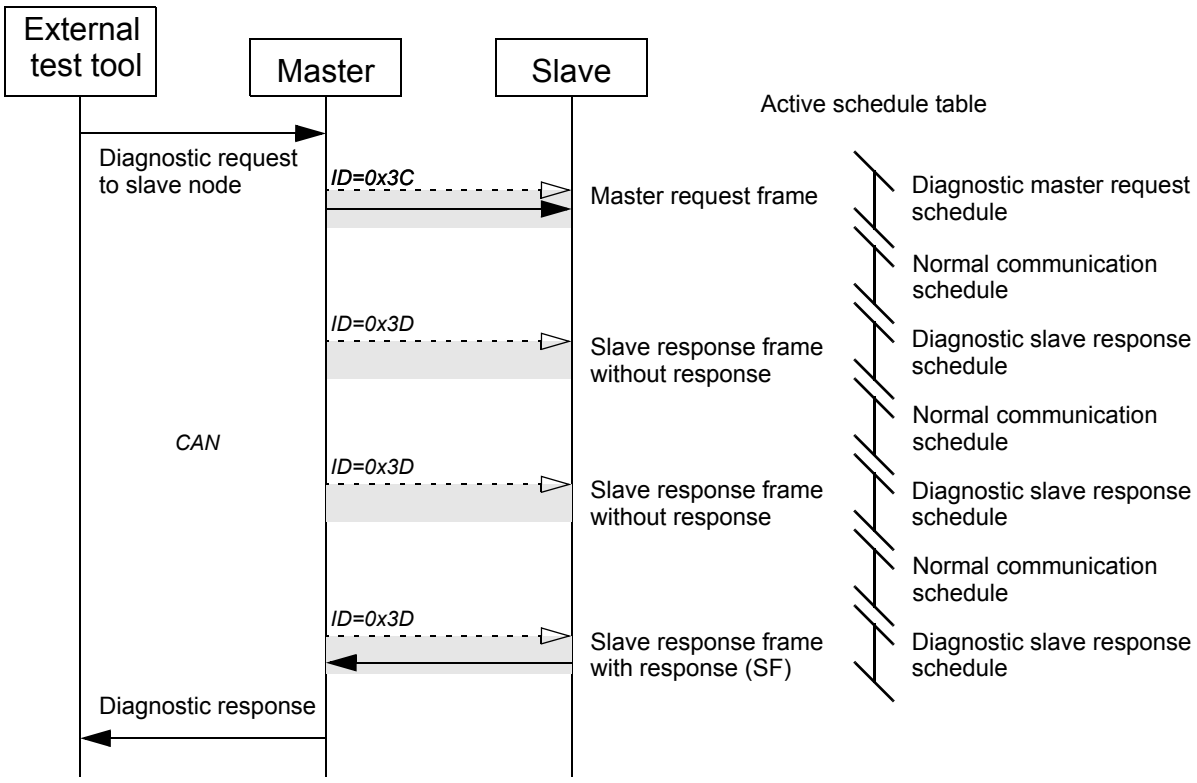


Figure 5.8: Continued execution of diagnostic slave response schedule table until response is received

5.4.3.2 Diagnostics Only Mode

The master node may optionally implement a Diagnostics Only Mode in which only the diagnostic schedules and no normal communication schedules are executed. The basic principles to use master request frame schedule tables and slave response frame schedule tables are the same as for the diagnostics interleaved mode except that no normal communication schedules are interleaved between the diagnostic schedule tables.

This is to allow for optimized diagnostic data transmission (e.g. when reading slave-node Identifications or during flash reprogramming, see Figure 5.9 for the different use cases).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

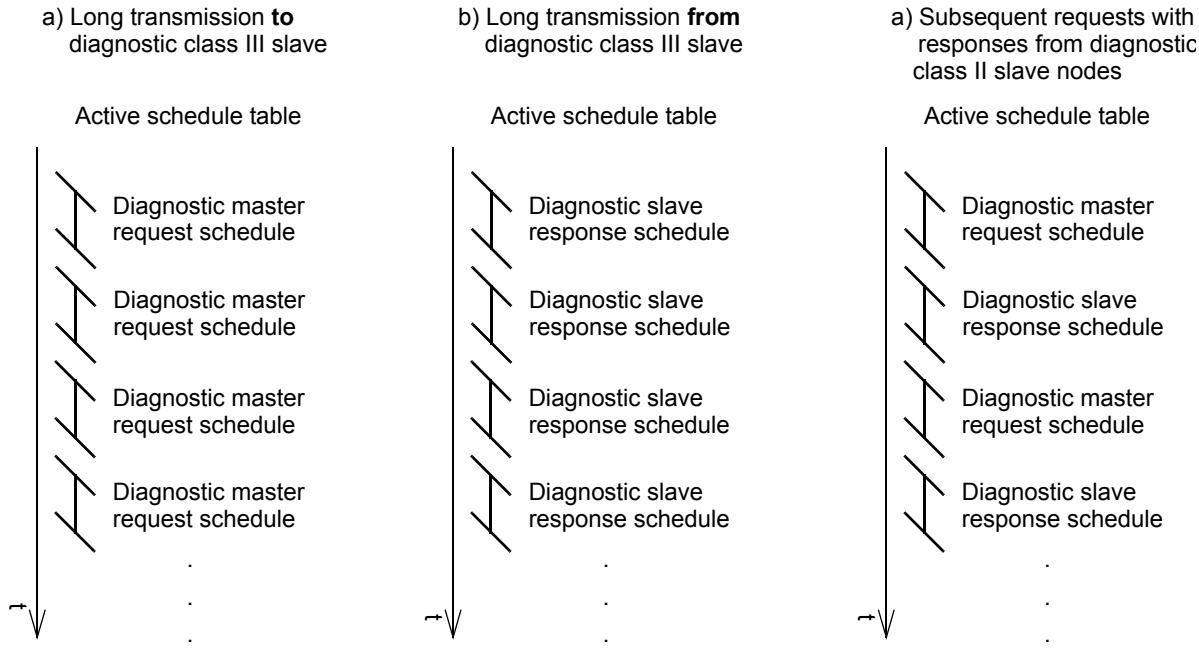


Figure 5.9: Use cases for diagnostic only mode

The diagnostics only mode shall be enabled and disabled via diagnostic service request from external test tool (e.g. service "Communication Control" in UDS to disable normal communication on the LIN cluster will lead to the activation of the Diagnostics Only Mode). When operating in the diagnostics only mode without any active transmission the master node shall execute diagnostic slave response schedule tables (see Figure 5.10).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

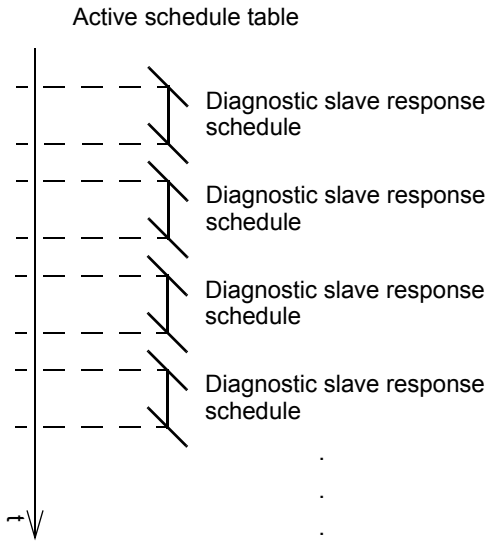


Figure 5.10: Default schedule in the diagnostics only mode

5.4.4 TRANSMISSION HANDLER REQUIREMENTS

The master node shall implement a transmission handler as shown in Figure 5.11.

One transmission handler per cluster connected to the master node may be implemented that operate independently from each other. The transmission handler shall be capable to operate in either interleaved diagnostics mode or diagnostics only mode.

Note: This implies that at least one active master to slave node physical transmission plus one functional transmission can be handled per cluster.

Broadcasting to all LIN clusters of a master node shall always be possible regardless of the currently active connections.

Due to the communication restrictions on a cluster the following communication scenarios are not supported:

- Responses from slave nodes to a functional request. The external test tool must ensure that functional requests do not require a response (e.g. TesterPresent request with SuppressPositiveResponseMsgIndicationFlag set

Registered copy for cbarbaud@cmaisonneuve.qc.ca

to 1) as otherwise an interleaved functional request will destroy the next slave response frame as multiple slave nodes will transmit a response.

- Asynchronous responses/transmissions from slaves without any prior request (in Diagnostics Only Mode this limitation may be circumvent via use-case specific implementations)

5.4.4.1 Master node transmission handler

A transmission handler shall be implemented by the master node according to Figure 5.11. Depending on whether the master node is operating in the diagnostics interleaved mode or in the diagnostics only mode the corresponding actions to be performed in the individual state slightly differ.

The following states are defined for both modes:

- Idle:

  In this state the master node is neither receiving nor transmitting any transmission on the LIN cluster. It is consequently available for any incoming transmission from the back-bone bus (e.g. CAN).

- Tx functional active:

  In this state the master node is routing a functional addressed request from the back-bone bus to the LIN cluster. This can only be a single frame (SF) according to restrictions for the transport protocol on LIN (see Transport Layer Specification).

- Tx physical active:

  In this state the master node is currently routing data from the back-bone bus to one slave node in the LIN cluster. The master node is consequently occupied and cannot route any other physical transmission from the back-bone bus to the LIN cluster. Also no response from a slave node can be routed to the back-bone bus.

- Rx physical active:

  In this state the master node is routing a transmission from a slave node to the back-bone bus. It is possible to transmit functional addressed requests to the LIN cluster but cannot handle further physical transmissions to a slave node.

- Interleaved functional during Tx

Registered copy for cbarbaud@cmaisonneuve.qc.ca

This is the state in which the master node routes a functional addressed request from the back-bone bus to the LIN cluster while a transmission to a slave node is currently active. Functional addressed Single Frames (SF) can be transmitted, but shall be ignored by the active slave node while receiving a physically addressed transmission.

- Interleaved functional during Rx

  In this state the master node routes a functional addressed request from the back-bone bus to the cluster while a reception from a slave node is currently active. Functional addressed Single Frames (SF) can be transmitted, but shall be ignored by the active slave node while transmitting the physically addressed response.

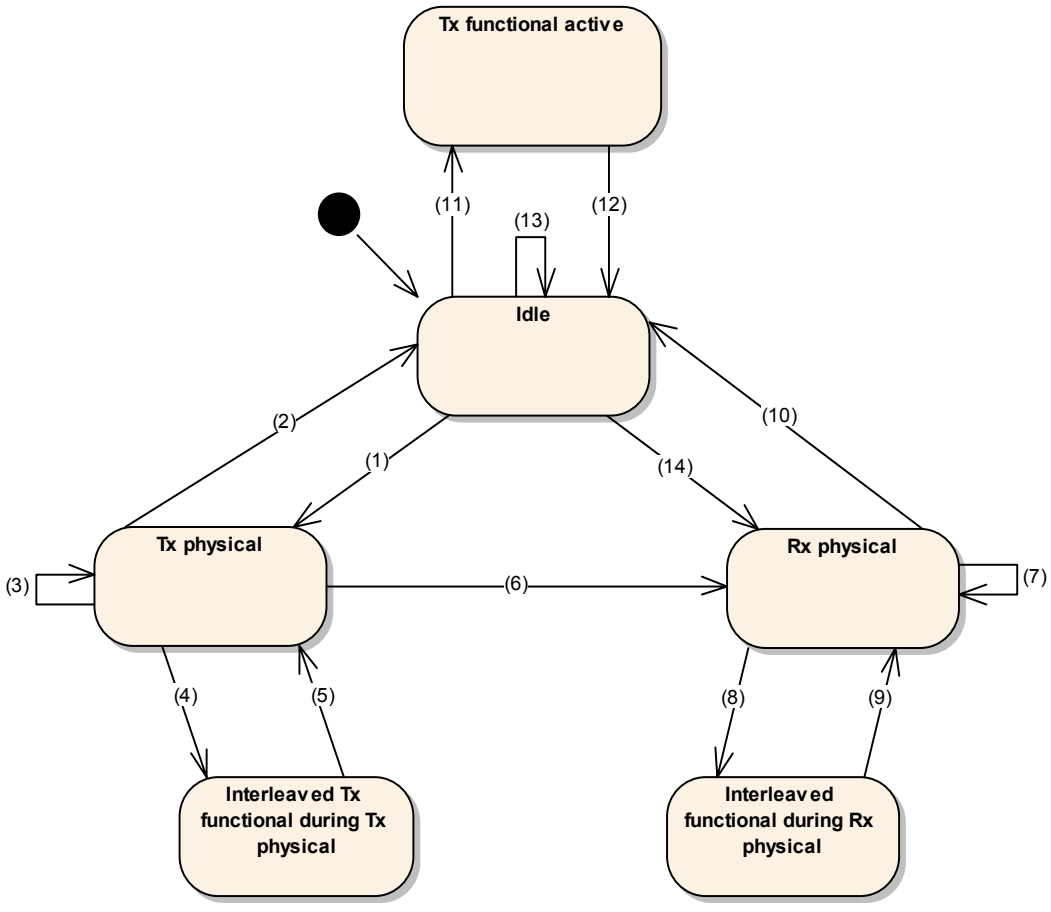


Figure 5.11: Master node transmission handler

Registered copy for cbarbaud@cmaisonneuve.qc.ca

1. Idle state → Tx physical active state
   - Trigger:
     Start of a physical transmission from the back-bone bus to a slave node.
   - Effect:
     Start executing diagnostic master request frame schedule tables and handle the transport protocol.
2. Tx physical active state → Idle state
   - Condition: Routing of the physical transmission from the back-bone bus to the cluster has finished or a transport protocol transmission error (e.g. timeout) on the back-bone bus occurred.
   - Action: Stop executing master request frame schedule tables.
3. Tx physical active state → Tx physical active state
   - Condition: Routing of the physical transmission from the back-bone bus to the slave node has not finished (i.e. data still needs to be routed).
   - Action: Continue executing master request schedule table and handle transport protocol on LIN (i.e. route transmission from the back-bone bus to the slave node).
4. Tx physical active state → Interleaved functional during Tx physical state
   - Condition: Functional addressed request from the back-bone bus has been received.
   - Action: Interrupt the physical transmission to the slave node and execute a single master request frame schedule to transmit the functional addressed request onto the cluster.
5. Interleaved functional during Tx physical state → Tx physical active state
   - Condition: Functional addressed request has been routed onto the cluster.
   - Action: Continue the interrupted physical transmission to the slave node.
6. Tx physical active state → Rx physical active state
   - Condition: The physical transmission to the slave node has been successfully completed (according to the Transport Layer Specification).
   - Action: Stop executing master request frame schedule tables, start executing slave response frame schedule tables and handle the incoming response from the previously addressed slave node.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7. Rx physical active state → Rx physical active state
   - Condition: The response from the slave node has not been started or has not finished yet (according to the LIN transport protocol).
   - Action: Transmit slave response frame schedule tables and handle the LIN transport protocol (i.e. route transmission from the slave node to the back-bone bus).
8. Rx physical active state → Interleaved functional during Rx physical state
   - Condition: Functional addressed request from the back-bone bus has been received.
   - Action: Interrupt the response from the slave node and execute a single master request frame schedule table to transmit the functional addressed request onto the cluster.
9. Interleaved functional during Rx physical state → Rx physical active state
   - Condition: Functional addressed request has been routed onto the cluster.
   - Action: Restart executing slave response frame schedules and continue the interrupted reception from the slave node.
10. Rx physical active state → Idle state
    - Condition: Reception from the slave node has been completed (according to the Transport Layer Specification) or a transport protocol error on the back-bone bus has occurred or the timeout P2max respective P2*max has elapsed (according to the negative response code 0x78 handling as defined in ISO 15765-3 [4])
    - Action: Stop executing slave response frame schedules.
11. Idle state → Tx functional active state
    - Condition: Functional addressed request from the back-bone bus has been received.
    - Action: Execute a single master request frame schedule table to transmit the functional addressed request onto the cluster.
12. Tx functional active state → Idle state
    - Condition: Functional addressed request has been routed onto the cluster.
    - Action: Stop executing master request frame schedules.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

13. Idle state → Idle state
    - Condition: No physical transmission from the back-bone bus to be routed to a slave node nor any response to be routed from a slave node to the back-bone bus.
    - Action:
      Diagnostics interleaved mode: Do not execute any master request frame schedule tables or slave response frame schedule tables.
      Diagnostics only mode: Execute slave response frame schedule tables
14. Idle state → Rx physical active state (for Diagnostics only mode only)
    - Condition: A slave node has initiated a transmission via one of the slave response frame schedule tables.
    - Action: Handle the incoming response from the responding slave node and start routing to the back-bone bus.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.5 SLAVE NODE TRANSMISSION HANDLER

Each slave node shall implement a transmission handler as defined in Figure 5.12. This is to allow for diagnostic communication without frame collisions on the cluster.

During diagnostics the broadcast NAD is normally not used. If this will happen the slave node will process requests with broadcast NAD (0x7F) in the same way as if it is the slave node's own NAD. Note the difference between the broadcast NAD (0x7F) and functional NAD (0x7E).

The following states are defined:

- Idle:

  In this state the slave node is neither receiving nor transmitting any messages in the cluster. It is consequently available for any incoming request from the master node. It shall not respond to slave response frames.
- Receive physical request:

  In this state the slave node is receiving a transmission from the master node. It is receiving and processing the transport layer frames as received from the master node. The slave node shall ignore any interleaved functional addressed transmission from the master node.
- Transmit physical response:

  In this state a slave node is currently still processing the previously received request, is ready to transmit a physical response or is actually transmitting the response to the previously received request. A slave node shall not receive nor process interleaved functional addressed (NAD 0x7E) transmissions from the master node. Physical transmissions shall be received and will make the slave node discard the current request data or response data. If the request is addressed to the slave node the request shall be received and processed.
- Receive functional request:

  In this state a slave node is receiving a functional transmission from the master node. The slave node shall not respond to any slave response frames.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

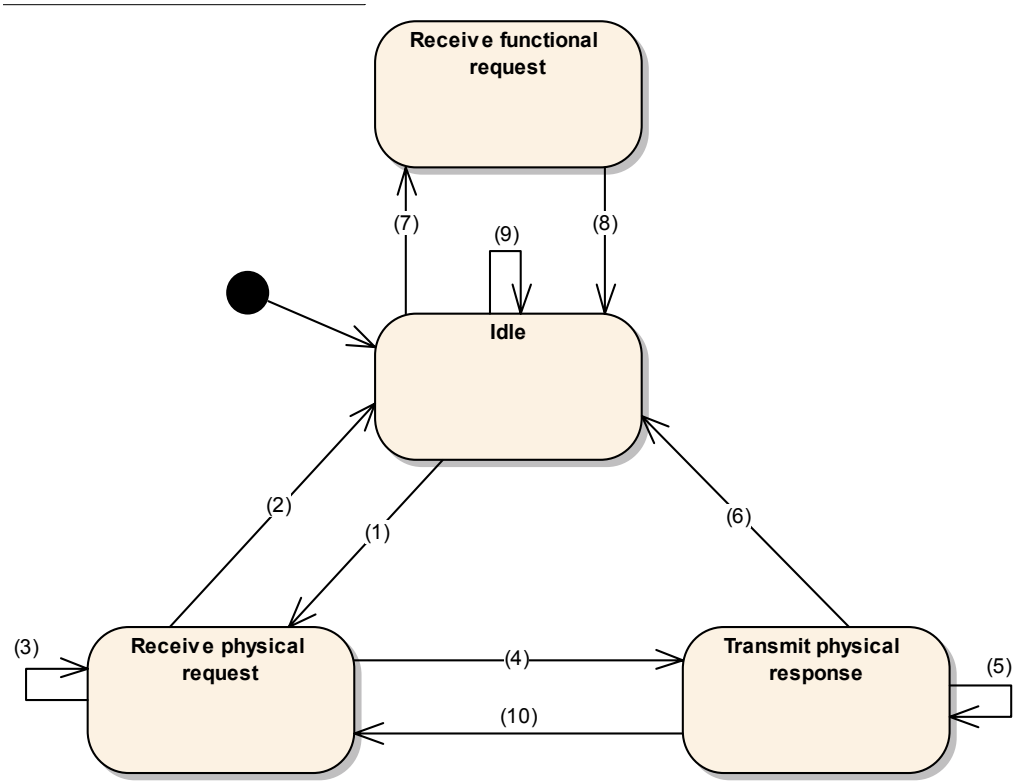


Figure 5.12: Slave node transmission handler

1. Idle state → Receive physical request state
   - Condition: A master request frame has been received with the NAD matching the slave node's own NAD.
   - Action: Start receiving and processing the physical request according to the transport layer requirements.
2. Receive physical request state → Idle state
   - Condition: A transport layer error has occurred or a master request frame with an NAD different from the slave node's own NAD has been received.
   - Action: Stop receiving and processing the physical request. Do not respond to slave response frames.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

3. Receive physical request state → Receive physical request state
   - Condition: The physical request has not been completely received yet and master request frames are received with the NAD set to the slave node's own NAD. A functional addressed master request frame shall be ignored
   - Action: Continue receiving and processing the physical request.
4. Receive physical request state → Transmit physical response state
   - Condition: The physical request has been completely received.
   - Action: Process the diagnostic request. If a new physical request with the NAD set to the slave node's own address is received while processing the previous request, the slave node shall discard the current request or response data and shall start receiving the new request.
5. Transmit physical response state → Transmit physical response state
   - Condition: The physical response has not been completely transmitted yet. A functional addressed request shall be ignored.
   - Action: Keep responding to slave response frames according to the transport layer requirements.
   - Note: A slave node will not process a functional addressed request while in the transmit physical response state. Therefore it must be ensured by the external test tool that functionally addressed requests that shall be processed by all slave nodes are only transmitted if no further responses from any slave node are expected. Otherwise there's no guarantee nor indication for the external test tool whether a slave node has processed the functional request.
6. Transmit physical response state → Idle state
   - Condition: The physical response has been completely transmitted, a LIN transport layer error occurred or a request with the NAD set to a different or the same value as the slave node's own NAD has been received.
   - Action: Discard the request and response data. Stop responding to slave response frames.
7. Idle state → Receive functional request state
   - Condition: A master request frame with the NAD parameter set to the functional NAD has been received.
   - Action: Receive and process the master request frame according to the transport layer. Do not respond to the slave response frame headers.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8. Receive functional request state → Idle state
   - Condition: The functional request was processed.
   - Action: Discard any response data. Stop responding to slave response frames.
9. Receive functional request state → Idle state
   - Condition: No request is received and no response is pending.
   - Action: Do not respond to any slave response frames.
10. Transmit physical response state → Receive physical request state
   - Condition: The previous request has been processed and a diagnostic master request frame with the NAD parameter set to the slave node's own node address has been received.
   - Action: Discard the response data. Start receiving and processing the physical request according to the LIN transport protocol requirements.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

5.6 SLAVE DIAGNOSTIC TIMING REQUIREMENTS

This chapter contains the requirements for the timing parameters that have to be taken into account when designing the LIN cluster. The monitoring of the timing parameters for the diagnostic classes I and II have to be implemented in the master node, the monitoring for diagnostic class III-nodes have to implemented by the slave node itself.

Parameter	Affected device	Description	min. value / performance requirement	max. value / timeout
P2	master node	Time between reception of the last frame of a diagnostic request on the LIN bus and the slave node being able to provide data for a response. The maximum value defines the time after which a slave node must received a slave response header before it discards its response. Each slave node defines this minimum value in the NCF, see Node Capability Language Specification.	50 ms	500 ms
STmin	master node	The minimum time the slave node needs to prepare the reception of the next frame of a diagnostic request or the transmission of the next frame of a diagnostic response. Each slave node defines this minimum value in the NCF, see Node Capability Language Specification.	0 ms	n/a
P2*	master node	Time between sending a NRC 0x78 and the LIN-slave being able to provide data for a response.	P2	2000 ms

Table 5.3: Diagnostic communication timings

Registered copy for cbarbaud@cmaisonneuve.qc.ca

A timing sequence chart of the diagnostic communication is shown in Figure 5.13. The tester tool is shown only as an example.

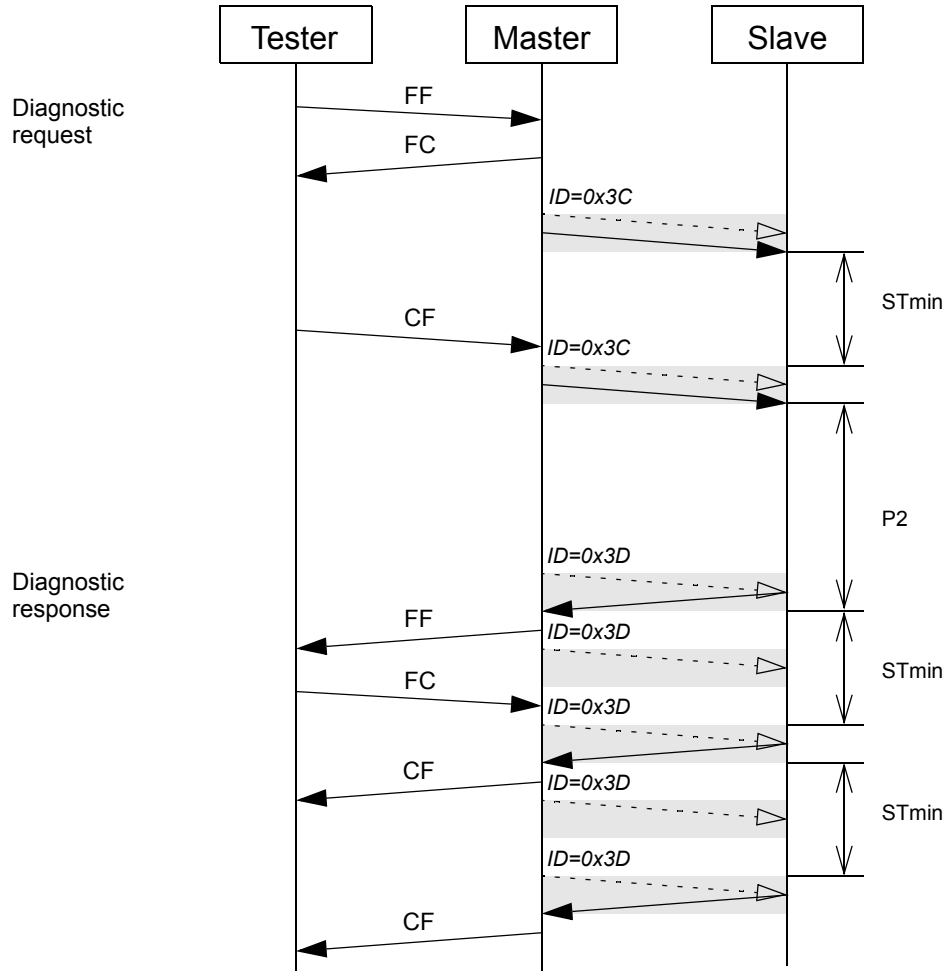


Figure 5.13: Timing sequence chart of the diagnostic communication from the tester to LIN via a back-bone bus.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Physical Layer Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.1 INTRODUCTION

In Revision 2.0 of the LIN Physical Layer Specification the receiver specification has been left unchanged. Just the LIN transmitter specification has been modified in order to provide higher transmission reliability compared to older LIN physical layer specification versions (e.g. Rev. 1.3).

The LIN physical layer of this revision is technically identical to Revision 2.0. Only ambiguous and incorrect specifications have been clarified and missing specifications have been added.

- A chapter about the physical layer compatibility has been added.
- The ambiguous term "clock tolerance" has been changed into "bit rate tolerance".
- The constraint for slave-to-slave communication has been clarified.
- A chapter has been added, which specifies the bit sample timing of byte fields.
- The supply voltage reference has become unique. In this context the voltage reference of the battery and ground shift has been changed from the voltage across the vehicle battery connectors to the ECU supply voltage connectors.
- A chapter about the performance in non-operation range has been added.
- A chapter about the performance during fault modes has been added.

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.2 PHYSICAL LAYER COMPATIBILITY

Since the LIN physical layer is independent from higher LIN layers (e.g. LIN protocol layer), all nodes with a LIN physical layer according to this revision can be mixed with LIN physical layer nodes, which are according to older revisions (i.e. LIN 1.0, LIN 1.1, LIN 1.2, LIN 1.3 and LIN2.0), without any restrictions.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.3 BIT RATE TOLERANCE

The bit rate tolerance of the LIN medium describes the bit rate deviation from a reference bit rate. It is the sum of the following parameters:

- Bit time measurement failure of the slave node
- Inaccuracy of setting the bit rate (systematic failure due to granularity of the configurable bit rate)
- Clock source stability of the slave node starting from the end of the sync byte field up to the end of the entire LIN frame (last sampled bit)
- Clock source stability of the master node starting from the end of the sync byte field up to the end of the entire LIN frame (last transmitted bit)

On-chip clock generators can achieve a frequency tolerance of better than ±14% with internal-only calibration. Hence, a bit rate tolerance better than ±14% can be achieved. This accuracy is sufficient to detect a 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 the bit rate, such as temperature and voltage drift during operation.

The bit rates on the LIN bus are specified in the range of 1 to 20 kbit/s. The specific bit rate used on a LIN bus is defined as the nominal bit rate F_{Nom}.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

no.	bit rate tolerance	Name	$\Delta F / F_{Nom.}$
Param 1	Master node (deviation from nominal bit rate)	$F_{TOL_RES_MASTER}$	< ±0.5%
Param 2	Slave node without making use of synchronization (deviation from nominal bit rate)	$F_{TOL_RES_SLAVE}$	< ±1.5%
Param 3	Deviation of slave node bit rate from the nominal bit rate before synchronization; relevant for nodes making use of synchronization and direct break detection.	F_{TOL_UNSYNC}	< ±14%

Table 6.1: Bit rate tolerances relative to nominal bit rate

no.	bit rate tolerance	Name	$\Delta F / F_{Master}$
Param 4	Deviation of slave node bit rate relative to the master node bit rate 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 sync field.	F_{TOL_SYNC}	< ±2%

Table 6.2: Slave node bit rate tolerance relative to master node bit rate

no.	bit rate tolerance	Name	$\Delta F / F_{Master}$
Param 5	For communication between any two nodes (i.e. data stream from one slave to another slave) their bit rate must not differ by more than $F_{TOL_SL_to_SL}$. The following condition must be checked for: a) $\lvert F_{TOL_RES_SLAVE1} - F_{TOL_RES_SLAVE2} \rvert < F_{TOL_SL_to_SL}$ b) $\lvert F_{TOL_SYNCH1} - F_{TOL_SYNCH2} \rvert < F_{TOL_SL_to_SL}$ c) $\lvert (F_{TOL_RES_MASTER} + F_{TOL_SYNCH1}) - F_{TOL_RES_SLAVE2} \rvert < F_{TOL_SL_to_SL}$	$F_{TOL_SL_to_SL}$	< ±2%

Table 6.3: Bit rate tolerance for slave to slave communication

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.4 TIMING REQUIREMENTS

6.4.1 BIT TIMING REQUIREMENTS

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

6.4.2 SYNCHRONIZATION PROCEDURE

The sync byte 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}.

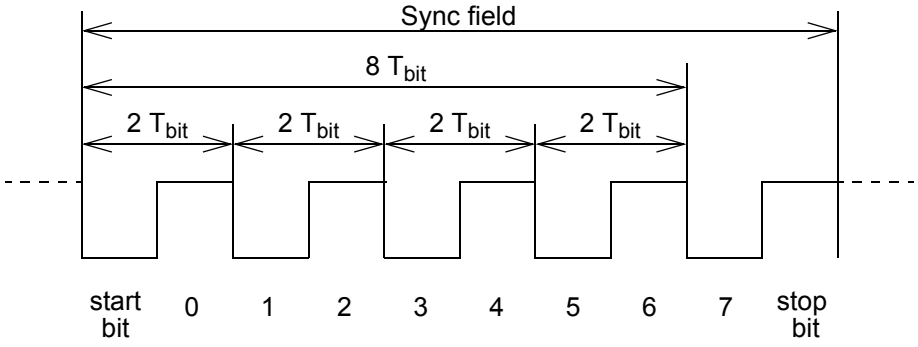


Figure 6.1: Sync byte field

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

6.4.3 BIT SAMPLE TIMING

The bits of a byte field shall be sampled according to the following specification. In Figure 6.2 the bit sample timing of a byte field is illustrated. The corresponding timing parameters are listed in Table 6.4.

A byte field shall be synchronized at the falling edge of the start bit. The byte field synchronization (BFS) shall have an accuracy of t_{BFS}.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

After the byte field synchronization on the falling edge of the start bit the start bit itself shall be sampled within the window between the earliest bit sample (EBS) time t_{EBS} and the latest bit sample (LBS) time t_{LBS}. The latest bit sample time t_{LBS} depends on the accuracy of the byte field synchronization t_{BFS}. The dependency between t_{LBS} and t_{BFS} is given in following equation:

$$t_{LBS} = 10/16 \quad T_{BIT} - t_{BFS} \qquad (11)$$

The following bits shall be sampled within the same range as the sample window of the start bit with the sample window repetition time t_{SR} respectively. The sample window repetition time t_{SR} is specified from the EBS of the former bit (n-1) to the EBS of the current bit:

$$t_{SR} = t_{EBS(n)} - t_{EBS(n-1)} = t_{LBS(n)} - t_{LBS(n-1)} = T_{BIT} \qquad (12)$$

no.	parameter	min.	typ.	max.	unit	Comment / condition
Param 6	t_{BFS}		1/16	2/16	T_{BIT}	Value of accuracy of the byte field detection
Param 7	t_{EBS}	7/16			T_{BIT}	Earliest bit sample time, $t_{EBS} \leq t_{LBS}$
Param 8	t_{LBS}				T_{BIT}	Latest bit sample (see Equation 11), $t_{LBS} \geq t_{EBS}$

Table 6.4: Bit Sample Timing

For devices, which make use of more than one sample per bit, the bit sample majority shall determine the bit data. Furthermore, the sample majority shall be between the EBS and the LBS.

In Table 6.5 bit sample timing examples are listed.

UART/SCI cycles per T_{BIT}	t_{BFS}	t_{EBS}	$t_{LBS} = 10/16\ T_{BIT} - t_{BFS}$
16	1/16 T_{BIT}	7/16 T_{BIT}	9/16 T_{BIT}
8	1/8 T_{BIT} (= 2/16 T_{BIT})	4/8 T_{BIT} (= 8/16 T_{BIT})	4/8 T_{BIT} (= 8/16 T_{BIT})

Table 6.5: Bit Sample Timing example

Registered copy for cbarbaud@cmaisonneuve.qc.ca

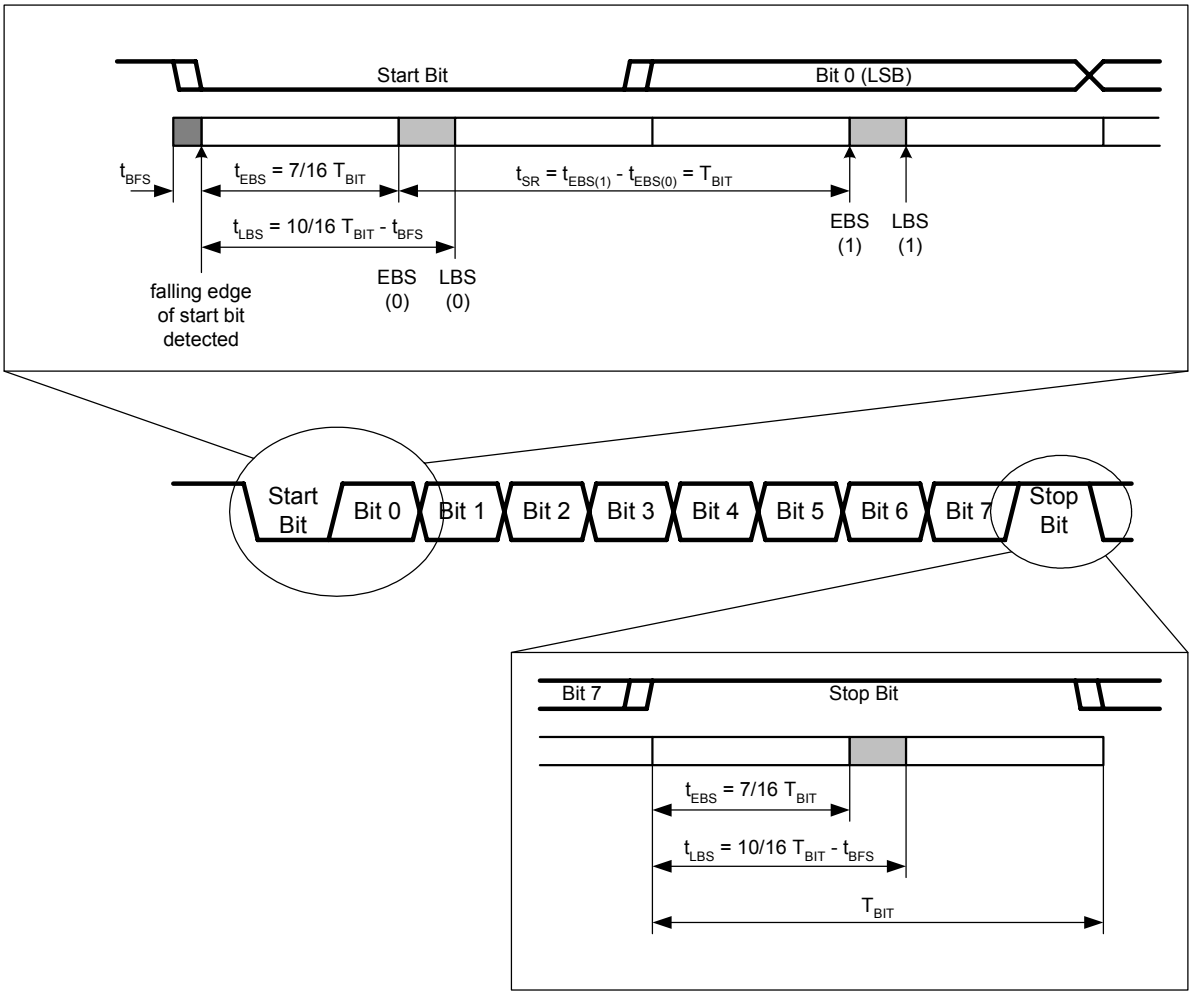


Figure 6.2: Bit Sample Timing

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.5 LINE DRIVER/RECEIVER

6.5.1 GENERAL CONFIGURATION

The bus line driver/receiver is based on 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 6.3). 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.

6.5.2 DEFINITION OF SUPPLY VOLTAGES FOR THE PHYSICAL INTERFACE

V_{BAT} denotes the supply voltage at the connector of the ECU. Electronic components within the unit may see an internal supply V_{SUP} being different from V_{BAT} (see Figure 6.3). 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

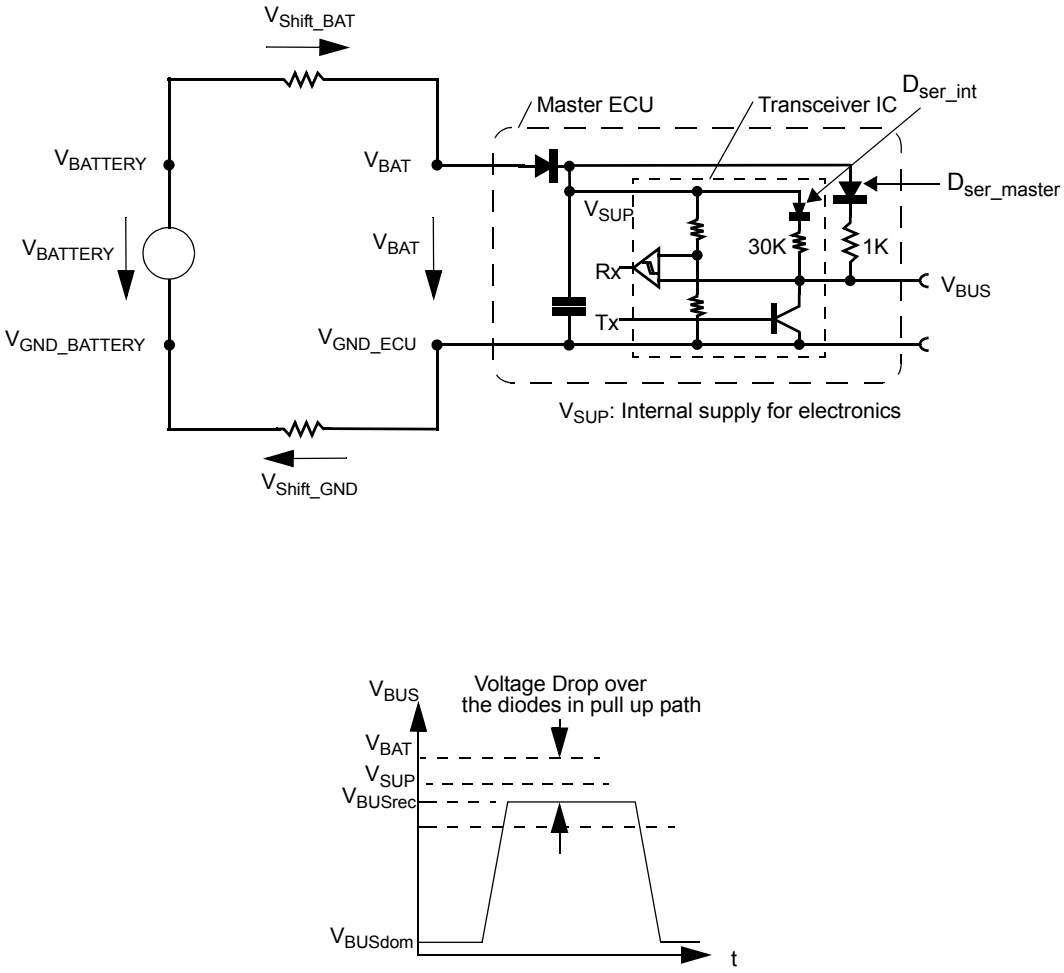


Figure 6.3: Illustration of the Difference between External Supply Voltage V_{BAT} and the Internal Supply Voltage V_{SUP}

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.5.3 SIGNAL SPECIFICATION

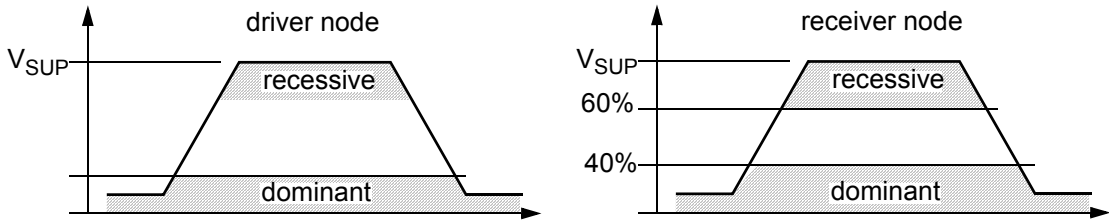


Figure 6.4: Voltage Levels on the Bus Line

For a correct transmission and reception of a bit, it has to be asserted 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 6.5 shows the timing parameters that impact the behavior of the LIN Bus.

The minimum and maximum values of the different parameters are listed in the following tables.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Timing diagram:

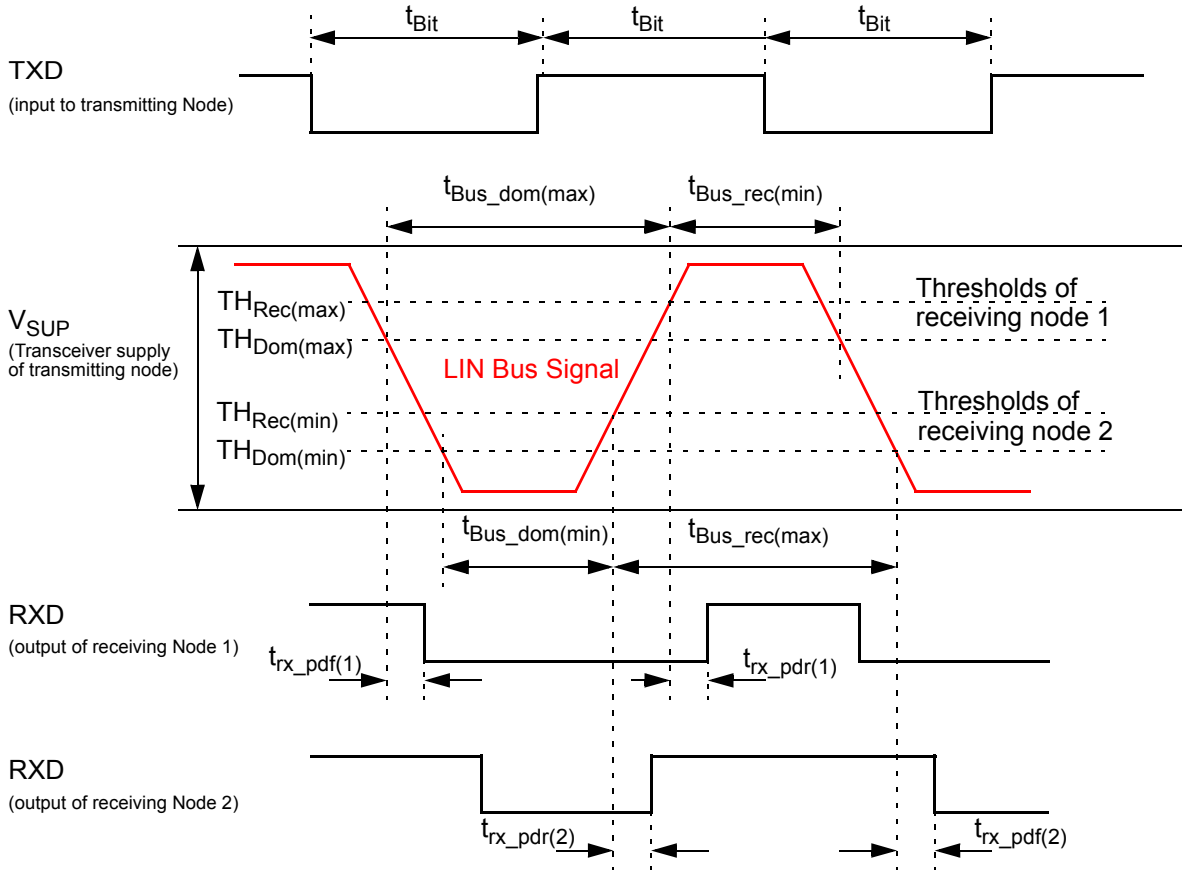


Figure 6.5: Definition of bus timing parameters

6.5.4 ELECTRICAL DC PARAMETERS

The electrical DC parameters of the LIN Physical Layer and the termination resistors are listed in Table 6.6 and Table 6.7, respectively. Unless otherwise specified, all voltages are referenced to the local ECU ground and positive currents flow into the ECU.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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 (VSUP), for example by ESD elements.

no.	parameter	min.	typ.	max.	unit	comment / condition
Param 9	V_{BAT} [a]	8		18	V	ECU operating voltage range
Param 10	V_{SUP} [b]	7.0		18	V	supply voltage range
Param 11	$V_{SUP_NON_OP}$	-0.3		40	V	voltage range within which the device is not destroyed
Param 12	I_{BUS_LIM} [c]	40		200	mA	Current Limitation for Driver dominant state driver on $V_{BUS} = V_{BAT_max}$ [d]
Param 13	$I_{BUS_PAS_dom}$	-1			mA	Input Leakage Current at the Receiver incl. Pull-Up Resistor as specified in Table 6.7 driver off V_{BUS} = 0V V_{BAT}= 12V
Param 14	$I_{BUS_PAS_rec}$			20	µA	driver off 8V<V_{BAT}<18V 8V<V_{BUS}<18V $V_{BUS} \geq V_{BAT}$
Param 15	$I_{BUS_NO_GND}$	-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.
Param 16	$I_{BUS_NO_BAT}$			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.
Param 17	V_{BUSdom}			0.4	V_{SUP}	receiver dominant state
Param 18	V_{BUSrec}	0.6			V_{SUP}	receiver recessive state
Param 19	V_{BUS_CNT}	0.475	0.5	0.525	V_{SUP}	$V_{BUS_CNT} = (V_{th_dom} + V_{th_rec})/2$ [e]
Param 20	V_{HYS}			0.175	V_{SUP}	$V_{HYS} = V_{th_rec} - V_{th_dom}$

Table 6.6: Electrical DC Parameters of the LIN Physical Layer

Registered copy for cbarbaud@cmaisonneuve.qc.ca

no.	parameter	min.	typ.	max.	unit	comment / condition
Param 21	$V_{SerDiode}$	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 6.3). $V_{SerDiode} = V_{ANODE} - V_{CATHODE}$ [f]
Param 22	V_{Shift_BAT}	0		11.5%	V_{BAT}	Battery-Shift $V_{Shift_BAT} = V_{BATTERY} - V_{Shift_GND} - V_{BAT}$ [g]
Param 23	V_{Shift_GND}	0		11.5%	V_{BAT}	GND-Shift $V_{Shift_GND} = V_{GND_ECU} - V_{GND_BATTERY}$ [g]
Param 24	$V_{Shift_Difference}$ [h]	0		8%	V_{BAT}	Difference between Battery-Shift and GND-Shift $V_{Shift_Difference} = \lvert V_{Shift_BAT} - V_{Shift_GND} \rvert$

Table 6.6: 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 6.5.2).
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 6.5.2).
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.
f. V_{ANODE}: voltage at the anode of the diode.
$V_{CATHODE}$: voltage at the cathode of the diode.
g. $V_{BATTERY}$: voltage across the vehicle battery connectors.
V_{GND_ECU}: voltage on the local ECU ground connector with respect to vehicle battery ground connector ($V_{GND_BATTERY}$).
h. This constraint refers to duty cycle D1 and D2 only.

no.	parameter	min.	typ.	max.	unit	comment
Param 25	R_{master}	900	1000	1100	Ω	The serial diode is mandatory (Figure 6.3).
Param 26	R_{slave}	20	30	60	KΩ	The serial diode is mandatory.

Table 6.7: Parameters of the Pull-Up Resistors

6.5.4.1 Electrical AC Parameters

The electrical AC parameters of the LIN Physical Layer are listed in Table 6.8, Table 6.9, and Table 6.10, with the parameters being defined in Figure 6.5. The electrical AC-Characteristics of the bus can be strongly affected by the line characteristics

Registered copy for cbarbaud@cmaisonneuve.qc.ca

as shown in Section 6.5.3. The time constant τ (and thus the overall capacitance) of the bus (Section 6.5.5) has to be selected carefully in order to allow for a correct signal implementation under worst case conditions.

The following table (Table 6.8) specifies the timing parameters for proper operation at 20 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						
Param 27	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 = $t_{Bus_rec(min)} / (2 \times t_{Bit})$
Param 28	D2 (Duty Cycle 2)			0.581		$TH_{Rec(min)} = 0.422 \times V_{SUP}$; $TH_{Dom(min)} = 0.284 \times V_{SUP}$; V_{SUP} = 7.6V...18V; t_{Bit} = 50µs; D2 = $t_{Bus_rec(max)} / (2 \times t_{Bit})$

Table 6.8: 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 6.9), 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						
Param 29	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 = $t_{Bus_rec(min)} / (2 \times t_{Bit})$
Param 30	D4 (Duty Cycle 4)			0.590		$TH_{Rec(min)} = 0.389 \times V_{SUP}$; $TH_{Dom(min)} = 0.251 \times V_{SUP}$; V_{SUP} = 7.6V...18V; t_{Bit} = 96µs; D4 = $t_{Bus_rec(max)} / (2 \times t_{Bit})$

Table 6.9: Driver Electrical AC Parameters of the LIN Physical Layer (10.4kBit/s)

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Application specific implementations (ASICs) shall meet the parameters in Table 6.8 and/or Table 6.9. 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 behavior: $R_{pull-up}$ = 2.4kΩ)						
Param 31	t_{rx_pd}			6	µs	propagation delay of receiver
Param 32	t_{rx_sym}	-2		2	µs	symmetry of receiver propagation delay rising edge w.r.t. falling edge

Table 6.10: 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.

6.5.5 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 as:

$$C_{BUS} = C_{MASTER} + n \cdot C_{SLAVE} + C'_{LINE} \cdot LEN_{BUS} \quad (13)$$

the RC time constant τ is calculated as:

$$\tau = C_{BUS} \cdot R_{BUS} \quad (14)$$

with:

$$R_{BUS} = R_{Master} \,||\, R_{Slave1} \,||\, R_{Slave2} \,||\, \ldots \,||\, R_{Slave_n} \quad (15)$$

Registered copy for cbarbaud@cmaisonneuve.qc.ca

under consideration of the parameters given in Table 6.11.

no.	description	parameter	min	typ.	max	unit
Param 33	total length of bus line	LEN_{BUS}			40	m
Param 34	total capacitance of the bus including slave and master capacitances	C_{BUS}	1	4	10	nF
Param 35	time constant of overall system	τ	1		5	µs
Param 36	capacitance of master node	C_{MASTER}		220		pF
Param 37	capacitance of slave node	C_{SLAVE}		220	250	pF
Param 38	line capacitance	C'_{LINE}		100	150	pF/m

Table 6.11: 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. 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)

6.5.6 PERFORMANCE IN NON-OPERATION SUPPLY VOLTAGE RANGE

For V_{BAT} > 18V or V_{BAT} < 8V the ECU may still operate, but communication is not guaranteed. If an ECU is not intending to transmit on the LIN bus (e.g. transmit input of a LIN transceiver is recessive), the LIN driver shall not drive the LIN bus to dominant state. If the LIN bus is in recessive state, the LIN receiver output shall provide a recessive state.

6.5.7 PERFORMANCE DURING FAULT MODES

6.5.7.1 Loss of supply voltage connection or ground connection

ECUs with loss of connection to either supply voltage or ground shall not interfere with normal communication among the remaining LIN participants. Upon return of connection, normal operation shall resume without any intervention on the LIN bus line.

6.5.7.2 Bus wiring short to battery or ground

The network data communication may be interrupted but there shall be no damage to any ECU when the LIN bus line is shorted to either positive battery with less than 26.5 V or ground. Upon remove of the fault, normal operation shall resume without any intervention on the LIN bus line.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

6.5.8 ESD/EMI COMPLIANCE

Semiconductor Physical Layer devices must comply with requirements for protection against human body discharge according to IEC 61000-4-2:1995. The minimum discharge voltage level ± 2000 V.

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Application Program Interface Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.1 INTRODUCTION

The LIN API is a network software layer that hides the details of a LIN network configuration (e.g. how signals are mapped into certain frames) for a user making an application program for an arbitrary ECU. Instead the user will be provided an API, which is focused on the signals transported on the LIN network. A tool is used to take care of the step from network configuration to ready made program code. This will provide the user with configuration flexibility.

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.

7.1.0.1 LIN cluster generation

Normally the LDF (see Configuration Language Specification) is parsed by a tool and generates a configuration for the LIN device driver. The NCF (see Node Capability Language Specification) is normally not used in this process since its intention is to describe an hardware slave node, and therefore, does not need the API.

See Section 1.1.3 for a description of the workflow and the roles of the LDF and NCF.

7.1.1 CONCEPT OF OPERATION

The API is split in three sections:

- LIN core API
- LIN node configuration and identification API
- LIN transport layer API (optional)

7.1.1.1 LIN core API

The LIN core API handles initialization, processing and a signal based interaction between the application and the LIN core. This implies that the application does not have to bother with frames and transmission of frames. Notification exists to detect transfer of a specific frame if this is necessary, see Section 7.2.3. Of course, API calls to control the LIN core also exist.

Two versions exist of most of the API calls:

- Static calls embed the name of the signal or interface in the name of the call.
- Dynamic calls provide the signal or interface as a parameter.

The choice between the two is a matter of taste.

The behavior of the LIN core API is defined in the Protocol Specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Note that the named objects (signals, schedules) defined in the LDF may extend their names with the channel postfix name, see Section 9.2.1.4.

7.1.1.2 LIN node configuration and identification API

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

The behavior of the LIN node configuration and identification API is covered in the Node configuration and Identification Specification.

7.1.1.3 LIN transport layer API

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

The behavior of the LIN transport layer API is covered in the Transport Layer Specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.2 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 - 0 is false, and non-zero (>0) is true
- l_ioctl_op - Implementation dependent
- l_irqmask - Implementation dependent
- l_u8 - Unsigned 8 bit integer
- l_u16 - Unsigned 16 bit integer

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.).

7.2.1 DRIVER AND CLUSTER MANAGEMENT

7.2.1.1 l_sys_init

Prototype

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

Availability

Master and slave nodes.

Description

l_sys_init performs the initialization of the LIN core. The scope of the initialization is the physical node (i.e. the complete node), see Section 9.2.2.3.

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.

The function returns:

Zero	If the initialization succeeded.
Non-zero	If the initialization failed.

7.2.2 SIGNAL INTERACTION

In all signal API calls below the sss is the name of the signal, e.g. l_u8_rd_EngineSpeed ().

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.2.2.1 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 2 - 8 bits
l_u16	for signals of the size 9 - 16 bits

7.2.2.2 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 prototype

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

Availability

Master and slave nodes.

Description

Reads and returns the current value of the signal.

Reference

Protocol Specification, Section 2.2.

7.2.2.3 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 prototype

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

Availability

Master and slave nodes.

Description

Sets the current value of the signal to v.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Reference

Protocol Specification, Section 2.2.

7.2.2.4 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 prototype

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

Availability

Master and slave nodes.

Description

Reads and returns the current values of the selected bytes in the signal.

The sum of start and count shall never be greater than the length of the byte array.

Example

Assume that a byte array is 6 bytes long, numbered 0 to 5. Reading byte 2 and 3 from this array 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].

Reference

Protocol Specification, Section 2.2.

7.2.2.5 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 (..).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability

Master and slave nodes.

Description

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

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

Example

Assume that a byte array is 7 bytes long, numbered 0 to 6. Writing byte 3 and 4 from this array 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].

Reference

Protocol Specification, Section 2.2.

7.2.3 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. Flags may be attached to all types of frames. A flag is set when the frame/signal is considered to be transmitted respectively received, see Section 2.2.4.

Three types of flags can be created:

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

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

7.2.3.1 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 ().

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability
Master and slave nodes.

Description
Returns a C boolean indicating the current state of the flag specified by the name fff, i.e. returns zero if the flag is cleared, non-zero otherwise.

Reference

No reference, flags are API specific and not described anywhere else.

Example
A flag, named txconfirmation, is attached to a published signal valve_position stored in the IO_1 frame. The static implementation of the l_flg_tst will be:

```
l_bool l_flg_tst_txconfirmation (void);
```

The flag will be set when the IO_1 frame (containing the signal valve_position) is successfully transmitted from the node.

7.2.3.2 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 flag, e.g. l_flg_clr_RxEngineSpeed ().

Availability
Master and slave nodes.

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

Reference

No reference, flags are API specific and not described anywhere else.

7.2.4 SCHEDULE MANAGEMENT

7.2.4.1 l_sch_tick

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Static implementation

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

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

Availability

Master nodes only.

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 periodically and individually for each interface within the node. The period is the time base, see Section 2.4, set in the LDF, see Section 9.2.2.1. The period of the l_sch_tick call effectively sets the time base tick, see Section 2.4. Therefore it is essential that the time base period is uphold with minimum jitter.

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, see Section 2.2.4.

The function 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.

Reference

Protocol Specification, Section 2.4.

7.2.4.2 l_sch_set

Dynamic prototype

```
void l_sch_set (l_ifc_handle      iii,
                l_schedule_handle schedule_iii,
                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).

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability
Master node only.

Description
Sets up the next schedule to be followed by the l_sch_tick function for a certain interface iii. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point. The extension "_iii" is the interface name. It is optional and the intention is to solve naming conflicts when the node is a master on more than one LIN cluster.

The entry defines the starting entry point in the new schedule table. The value 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.

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

Reference

Protocol Specification, Section 2.4.

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

7.2.5 INTERFACE MANAGEMENT

These calls manages the specific interfaces (the logical channels to the bus). Each interface is identified unique by its interface name, denoted by the iii extension for each API call. How to set the interface name (iii) is not in the scope of this specification.

7.2.5.1 l_ifc_init

Dynamic prototype

```
l_bool l_ifc_init (l_ifc_handle iii);
```

Static implementation

```
l_bool_ifc_init_iii (void);
```

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

Availability
Master and slave nodes.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Description

l_ifc_init initializes the controller specified by the name iii, i.e. sets up internal functions such as the baud rate. The default schedule set by the l_ifc_init call will be the L_NULL_SCHEDULE where no frames will be sent and received.

This is the first call a user must perform, before using any other interface related LIN API functions.

The function returns zero if the initialisation was successful and non-zero if failed.

Reference

A general description of the interface concept is found in Section 1.1.5.

7.2.5.2 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 ().

Availability

Master node only.

Description

This call requests slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command, see Section 7.2.5.8.

The go to sleep command will be scheduled latest when the next schedule entry is due.

The l_ifc_goto_sleep will not affect the power mode. It is up to the application to do this.

If the go to sleep command was successfully transmitted on the cluster the go to sleep bit will be set in the status register, see Section 7.2.5.8.

Reference

Protocol Specification, Section 2.6.3.

7.2.5.3 l_ifc_wake_up

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Static implementation

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

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

Availability

Master and slave nodes.

Description

The function will transmit one wake up signal. The wake up signal will be transmitted directly when this function is called. It is the responsibility of the application to retransmit the wake up signal according to the wake up sequence defined in Section 2.6.2.

Reference

Protocol Specification, Section 2.6.2.

7.2.5.4 l_ifc_ioctl

Dynamic prototype

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

Static implementation

```
l_u16 l_ifc_ioctl_iii (l_ioctl_op op, void* pv);
```

Where iii is the name of the interface, e.g. l_ifc_ioctl_MyLinIfc (MyOp, &MyPars).

Availability

Master and slave nodes.

Description

This function controls functionality that is not covered by the other API calls. It is used for protocol specific parameters or hardware specific functionality. Example of such functionality can be to switch on/off the wake up signal detection.

The iii is the name of the interface to which the operation defined in op should be applied. The pointer pv points to an optional parameter that may be provided to the function.

Exactly which operations that are supported are implementation dependent.

Reference

No reference, the behavior is API specific and not described anywhere else.

7.2.5.5 l_ifc_rx

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Static implementation

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

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

Availability

Master and slave nodes.

Description

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

For UART based implementations it may 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.

For more complex LIN hardware it may be used to indicate the reception of a complete frame.

Reference

No reference, the behavior is API specific and not described anywhere else.

7.2.5.6 l_ifc_tx

Dynamic prototype

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

Availability

Master and slave nodes.

Static implementation

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

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

Description

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

For UART based implementations it may 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.

For more complex LIN hardware it may be used to indicate the transmission of a complete frame.

Reference

Registered copy for cbarbaud@cmaisonneuve.qc.ca

No reference, the behavior is API specific and not described anywhere else.

7.2.5.7 l_ifc_aux

Dynamic prototype

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

Static implementation

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

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

Availability

Master and slave nodes.

Description

This function may be used in the slave nodes to synchronize to the break/sync field sequence transmitted by the master node 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.

l_ifc_aux may only be used in a slave node.

This function is strongly hardware connected and the exact implementation and usage is implementation dependent.

This function might even be empty in cases where the break/sync field sequence detection is implemented in the l_ifc_rx function.

Reference

No reference, the behavior is API specific and not described anywhere else.

7.2.5.8 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 ().

Availability

Master and slave nodes. The behavior is different for master and slave nodes, see description below.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Description

This function will return the status of the previous communication. The call returns the status word (16 bit value), as shown in Table 7.1.

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

Table 7.1: Return value of l_ifc_read_status (bit 15 is MSB, bit 0 is LSB).

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

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

In the master node the status word will be updated in the l_ifc_sch_tick function. In the slave node the status word is updated latest when the next frame is started.

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

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

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

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

Bus activity will be set if the node has detected bus activity on the bus. See Section 2.6.3 for definition of bus activity. A slave node is required to enter bus sleep mode after a period of bus inactivity on the bus, see Section 2.6.3. This can be implemented by the application monitoring the bus activity. Note the difference between bus activity and bus inactivity.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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

Save configuration is set when the save configuration request, see Section 4.2.5.4, has been successfully received. It is set only in the slave node, in the master node it is always 0 (zero).

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

The combination of the two status bits successful transfer and error in response is interpreted according to Table 7.2.

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

Table 7.2: Node internal error interpretation.

It is the responsibility of the node application to process the individual status reports.

Reference

Protocol Specification, Section 2.7.

Example 1

The l_ifc_read_status is designed to allow reading at a much lower frequency than the frame slot frequency, e.g. once every 50 frame slots. In this case, the last frame PID has little use. Overrun is then used as a check that the traffic is running as 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.

Example 2

This example shows how the event triggered flag behaves in case of a collision resolving.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The normal schedule table is depicted in Table 7.3.

frame	delay	frame type
UF1	10 ms	unconditional
IO_check	10 ms	event triggered
UF2	10 ms	unconditional

Table 7.3: Event triggered frame example schedule table

The IO_1 and IO_2 unconditional frames are associated with check_IO. The collision solving schedule table contains the unconditional frames IO_1 and IO_2 (with delays set to10 ms). The collision will be handled as shown in Figure 7.1. The time base in this example is set to 5 ms.

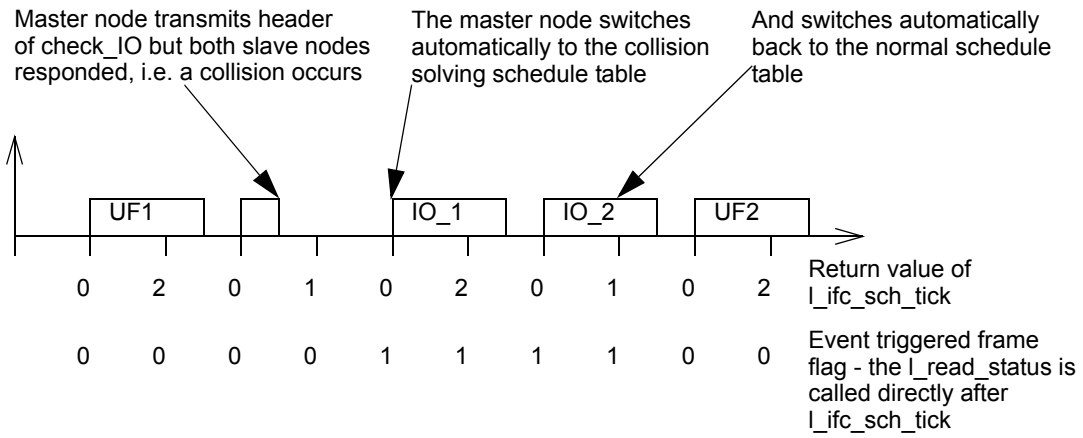


Figure 7.1: Event triggered frame collision solving example

7.2.6 USER PROVIDED CALL-OUTS

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

7.2.6.1 l_sys_irq_disable

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability
Master and slave nodes.

Description
The user implementation of this function must achieve a state in which no interrupts from the LIN communication can occur.

Reference

No reference, the behavior is API specific and not described anywhere else.

7.2.6.2 l_sys_irq_restore

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

Availability
Master and slave nodes.

Description
The user implementation of this function must restore the interrupt level identified by the provided l_irqmask previous.

Reference

No reference, the behavior is API specific and not described anywhere else.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.3 NODE CONFIGURATION AND IDENTIFICATION

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

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

7.3.1 NODE CONFIGURATION

7.3.1.1 ld_is_ready

Dynamic prototype

```
l_u8 ld_is_ready (l_ifc_handle iii);
```

Availability

Master node only.

Description

This call returns the status of the last requested configuration service. The return values are interpreted as follows:

LD_SERVICE_BUSY	Service is ongoing.
LD_REQUEST_FINISHED	The configuration request has been completed. This is a intermediate status between the configuration request and configuration response.
LD_SERVICE_IDLE	The configuration request/response combination has been completed, i.e. the response is valid and may be analyzed. Also, this value is returned if no request has yet been called.
LD_SERVICE_ERROR	The configuration request or response experienced an error. Error here means error on the bus, and not a negative configuration response from the slave node.

The following Figure 7.2 shows the situation where a successful configuration request and configuration response is made. Note that the state change after the master request frame and slave response frame are finished may be delayed up to one time base.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

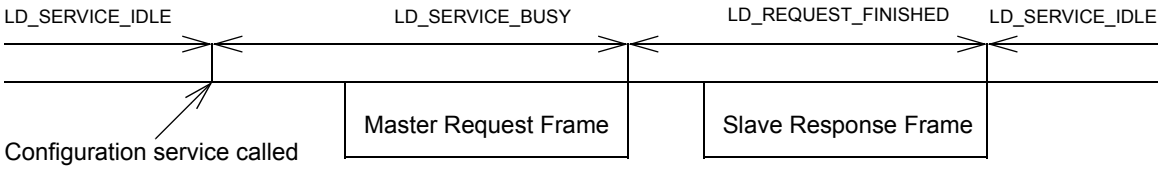


Figure 7.2: Successful configuration request and configuration response

Reference

No reference, the behavior is API specific and not described anywhere else.

7.3.1.2 ld_check_response

Dynamic prototype

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

Availability

Master node only.

Description

This call returns the result of the last node configuration service, in the parameters RSID and error_code. A value in RSID is always returned but not always in the error_code. Default values for RSID and error_code is 0 (zero).

Reference

No reference, the behavior is API specific and not described anywhere else.

7.3.1.3 ld_assign_frame_id_range

Dynamic prototype

```
void ld_assign_frame_id_range (l_ifc_handle      iii,
                               l_u8              NAD,
                               l_u8              start_index,
                               const l_u8* const PIDs);
```

Availability

Master node only.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Description

This call assigns the protected identifier of up to four frames in the slave node with the addressed NAD. The PIDs parameter shall be four bytes long, each byte shall contain a PID, do not care or unassign value.

Reference

See the definition of the service assign frame id range, Section 4.2.5.5.

7.3.1.4 ld_assign_NAD

Dynamic prototype

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

Availability

Master node only.

Description

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

Reference

See the definition of the service assign NAD, Section 4.2.5.1.

7.3.1.5 ld_save_configuration

Dynamic prototype

```
void ld_save_configuration (l_ifc_handle iii,
                            l_u8         NAD);
```

Availability

Master node only.

Description

This call will make a save configuration request to a specific slave node with the given NAD, or to all slave nodes if NAD is set to broadcast.

Reference

See the definition of the service save configuration, Section 4.2.5.4. API call l_ifc_read_status, Section 7.2.5.8. See also the example in Section 7.5.2.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.3.1.6 ld_read_configuration

Dynamic prototype

```
l_u8 ld_read_configuration (l_ifc_handle iii,
                            l_u8* const  data,
                            l_u8* const  length);
```

Availability

Slave node only.

Description

This function will not transport anything on the bus.

This call will serialize the current configuration and copy it to the area (data pointer) provided by the application. The intention is to call this function when the save configuration request flag is set in the status register, see Section 7.2.5.8. After the call is finished the application is responsible to store the data in appropriate memory.

The caller shall reserve bytes in the data area equal to length, before calling this function. The function will set the length parameter to the actual size of the configuration. In case the data area is too short the function will return with no action.

In case the NAD has not been set by a previous call to ld_set_configuration or the master node has used the configuration services, the returned NAD will be the initial NAD.

The data contains the NAD and the PIDs and occupies one byte each. The structure of the data is: NAD and then all PIDs for the frames. The order of the PIDs are the same as the frame list in the LDF, Section 9.2.2.2, and NCF, Section 8.2.5.

The function returns:

LD_READ_OK	If the service was successful.
LD_LENGTH_TOO_SHORT	If the configuration size is greater than the length. It means that the data area does not contain a valid configuration.

Reference

See the definition of the service save configuration, Section 4.2.5.4. Function l_ifc_read_status, Section 7.2.5.8. See also the example in Section 7.5.2.

7.3.1.7 ld_set_configuration

Dynamic prototype

```
l_u8 ld_set_configuration (l_ifc_handle      iii,
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
                                const l_u8* const data,
                                l_u16             length);
```

Availability

Slave node only.

Description

This call will not transport anything on the bus.

The function will configure the NAD and the PIDs according to the configuration given by data. The intended usage is to restore a saved configuration or set an initial configuration (e.g. coded by I/O pins). The function shall be called after calling l_ifc_init.

The caller shall set the size of the data area before calling the function.

The data contains the NAD and the PIDs and occupies one byte each. The structure of the data is: NAD and then all PIDs for the frames. The order of the PIDs are the same as the frame list in the LDF, Section 9.2.2.2, and NCF, Section 8.2.5.

The function returns:

LD_SET_OK	If the service was successful.
LD_LENGTH_NOT_CORRECT	If the required size of the configuration is not equal to the given length.
LD_DATA_ERROR	The set of configuration could not be made.

Reference

See the definition of the service save configuration, Section 4.2.5.4. Function l_ifc_read_status, Section 7.2.5.8. See also the example in Section 7.5.2.

7.3.2 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);
```

Availability

Master node only.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Description

This call changes the NAD if the node properties fulfil the test specified by id, byte, mask and invert.

Id shall be in the range 0 to 31, see Table 4.20, 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.

Reference

See the definition of the service conditional change NAD, Section 7.3.2.

7.3.3 IDENTIFICATION

7.3.3.1 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);
```

Availability

Master node only.

Description

The call requests the slave node selected with the NAD to return the property associated with the id parameter, see Table 4.19 in the Node configuration and Identification Specification, for interpretation of the id. When the next call to ld_is_ready returns LD_SERVICE_IDLE (after the ld_read_by_id is called), the RAM area specified by data contains between one and five bytes data according to the request.

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

Reference

Node configuration and Identification Specification, Section 4.2.6.1.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.3.3.2 ld_read_by_id_callout

Dynamic prototype

```
l_u8 ld_read_by_id_callout (l_ifc_handle iii,
                            l_u8         id,
                            l_u8*        data);
```

Availability

This callout is optional and is available in slave node only. In case the user defined read by identifier request is used, the slave node application must implement this call-out.

Description

This callout is used when the master node transmits a read by identifier request with an identifier in the user defined area. The slave node application will be called from the driver when such request is received.

The id parameter is the identifier in the user defined area (32 to 63), see Table 4.19, from the read by identifier configuration request.

The data pointer points to a data area with 5 bytes. This area will be used by the application to set up the positive response, see the user defined area in Table 4.20.

The driver will act according to the following return values from the application:

LD_NEGATIVE_RESPONSE	The slave node will respond with a negative response as defined in Table 4.21. In this case the data area is not considered.
LD_POSTIVE_RESPONSE	The slave node will setup a positive response using the data provided by the application.
LD_NO_RESPONSE	The slave node will not answer.

Reference

Node configuration and Identification Specification, Section 4.2.6.1.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.4 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 Transport Layer Specification.

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

7.4.1 RAW AND COOKED API

Since ISO 15765-2 [3] PDUs on CAN are quite similar to LIN diagnostic frames, a raw API is provided. The raw API is frame/PDU based and it is up to the application to manage the PCI information. The idea of the raw API is to interface to the CAN transport layer. With small efforts and resources the raw API can be used to gateway diagnostic requests/responds between CAN and LIN.

The cooked API is message based. The application will provide a pointer to a message buffer. When the transfer commences, the LIN 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.

Both raw API and the cooked API uses the same structure of the diagnostic frames, i.e. PCI, SID, NAD etc.

It is possible to use both the raw API and cooked API in a node. However, The behavior of the system is undefined in the case where the application tries to process frames using both the raw and the cooked API.

7.4.2 INITIALIZATION

Dynamic prototype

```
void ld_init (l_ifc_handle iii);
```

Availability

Master and slave nodes.

Description

This call will (re)initialize the raw and the cooked layers on the interface iii.

All transport layer buffers will be initialized.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

If there is an ongoing diagnostic frame transporting a cooked or raw message on the bus, it will not be aborted.

Reference

No reference, the behavior is API specific and not described anywhere else.

7.4.3 RAW API

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

7.4.3.1 ld_put_raw

Dynamic prototype

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

Availability

Master and slave nodes.

Description

The call queues the transmission of 8 bytes of data in one frame.

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

The data area will be copied in the call, the pointer will not be memorized.

If no more queue resources are available, the data may be jettisoned and the appropriate error status will be set.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.3.2 ld_get_raw

Dynamic prototype

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

Availability

Master and slave nodes.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

Description

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

The data returned is received from master request frame for slave nodes and slave response frame for master nodes.

If the receive queue is empty no data will be copied.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.3.3 ld_raw_tx_status

Dynamic prototype

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

Availability

Master and slave nodes.

Description

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

LD_QUEUE_EMPTY	The transmit queue is empty. In case previous calls to ld_put_raw, all frames in the queue have been transmitted.
LD_QUEUE_AVAILABLE	The transmit queue contains entries, but is not full.
LD_QUEUE_FULL	The transmit queue is full and can not accept further frames.
LD_TRANSMIT_ERROR	LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.3.4 ld_raw_rx_status

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability

Master and slave nodes.

Description

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

LD_NO_DATA	The receive queue is empty.
LD_DATA_AVAILABLE	The receive queue contains data that can be read.
LD_RECEIVE_ERROR	LIN protocol errors occurred during the transfer; initialize and redo the transfer.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.4 COOKED API

Cooked processing of diagnostic messages manages one complete message at a time.

7.4.4.1 ld_send_message

Dynamic prototype

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

Availability

Master and slave nodes.

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 application the frames are transmitted to the slave node with the address NAD. If the call is made in a slave node application the frames are transmitted to the master node with the address NAD. The parameter NAD is not used in slave nodes.

The value of the SID (or RSID) shall be the first byte in the data area.

Length must be in the range of 1 to 4095 bytes. The length shall also include the SID (or RSID) value, i.e. message length plus one.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The call is asynchronous, i.e. not suspended 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 transmitted in suitable frames (master request frame for master nodes and slave response frame for slave nodes).

If there is a message in progress, the call will return with no action.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.4.2 ld_receive_message

Dynamic prototype

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

Availability

Master and slave nodes.

Description

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

SID (or RSID) will be the first byte in the data area.

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

The parameter NAD is not used in slave nodes.

The call is asynchronous, i.e. not suspended until the message has been received, and the buffer may not be changed by the application as long as calls to ld_rx_status returns LD_IN_PROGRESS. If the call is made after the message transmission has commenced on the bus (i.e. the SF or FF is already transmitted), this message will not be received. Instead the function will wait until next message commence.

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

The application shall monitor the ld_rx_status and shall not call this function until the status is LD_COMPLETED. Otherwise this function may return inconsistent data in the parameters.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.4.3 ld_tx_status

Dynamic prototype

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

Availability

Master and slave nodes.

Description

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

LD_IN_PROGRESS	The transmission is not yet completed.
LD_COMPLETED	The transmission has completed successfully (and you can issue a new ld_send_message call). This value is also returned after initialization of the transport layer.
LD_FAILED	The transmission ended in an error. The data was only partially sent. The transport layer shall be reinitialized before processing further messages. To find out why a transmission has failed, check the status management function l_read_status, see Section 7.2.5.8.
LD_N_AS_TIMEOUT	The transmission failed because of a N_As timeout, see Section 3.2.5.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

7.4.4.4 ld_rx_status

Dynamic prototype

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

Availability

Master and slave nodes.

Description

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

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). This value is also returned after initialization of the transport layer.
LD_FAILED	The reception ended in an error. The data was only partially received and should not be trusted. Initialize before processing further transport layer messages. To find out why a reception has failed, check the status management function l_read_status, see Section 7.2.5.8.
LD_N_CR_TIMEOUT	The reception failed because of a N_Cr timeout, see Section 3.2.5.
LD_WRONG_SN	The reception failed because of an unexpected sequence number.

Reference

The raw and cooked is not differentiated outside the API. A general description of the transport layer can be found in Transport Layer Specification.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

7.5 EXAMPLES

In the following chapters a very simple example is given in order to show how the API can be used. The examples are not complete, there are functions that are not implemented.

7.5.1 MASTER NODE EXAMPLE

```
/*****************************************************************
* Description  : Example code for using the LIN API in a LIN master node
*                The static LIN API is used
*/
#include <lin.h>

#define INT_ENABLE_LEVEL 1

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


/*****************************************************************
* Procedure      : l_sys_irq_disable
* Description    : Disable the UART interrupts of the controller and
*                  return the interrupt level to be able to restore it
*                  later
* In parameters  : None
* Out parameters : None
* Return value   : The interrupt level before disable
*/
l_irqmask l_sys_irq_disable (void)
{
  l_irqmask interrupt_level;
  /* Store the interrupt level and then disable UART interrupts */
  return interrupt_level;
} /* l_sys_irq_disable */
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
/*****************************************************************
* Interrupt      : lin_char_rx_handler
* Description    : UART recieve character interrupt handler for the
*                  interface i1
* In parameters  : None
* Out parameters : None
* Return value   : void
*/
void __INTERRUPT /* Compiler intrinsic */ lin_char_rx_handler (void)
{
  /* Just call the LIN API provided function to do the actual work */
  l_ifc_rx_i1 ();
} /* lin_char_rx_handler */


/*****************************************************************
* Procedure      : main
* Description    : Main entry of application
* In parameters  : None
* Out parameters : None
* Return value   : function will never return
*/
int main (void)
{
  /* Initialize the LIN interface */
  if (l_sys_init ()) {
    /* The init of the LIN software failed - call error routine */
  }

  /* Initialize the interface */
  if (l_ifc_init_i1 ()) {
    /* Initialization of the LIN interface failed - call error routine */
  }

  /* Now is the first time the LIN interrupts can be enabled */
  l_sys_irq_restore (INT_ENABLE_LEVEL);

  /* Set the normal schedule */
  l_sch_set_i1 (Normal_Schedule, 0);

  /* Start the OS */
  start_OS ();

  /* return code */
  return 1;
} /* main */
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
/*****************************************************************
* Procedure      : main_application_10ms
* Description    : Main 10 ms task of the application
* In parameters  : None
* Out parameters : None
* Return value   : void */
void main_application_10ms (void)
{
  /* In/output of signals. Call it first in the task to minimize jitter */
  (void) l_sch_tick_i1();

  /* Do some application specific stuff... */

  /* Just a small example of signal reading and writing */
  if (l_flg_tst_RxInternalLightsSwitch ())
  {
    l_flg_clr_RxInternalLightsSwitch ();
    l_u8_wr_InternalLightsRequest (l_u8_rd_InternalLightsSwitch());
  }

} /* main_application_10ms */
```

7.5.2 SLAVE NODE EXAMPLE

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

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

#define INT_ENABLE_LEVEL 1

/*****************************************************************
* Interrupt      : lin_char_rx_handler
* Description    : UART recieve character interrupt handler for the
*                  interface i1
* In parameters  : None
* Out parameters : None
* Return value   : void
*/
void __INTERRUPT /* Compiler intrinsic */ lin_char_rx_handler (void)
{
  /* Just call the LIN API provided function to do the actual work */
  l_ifc_rx_i1 ();
} /* lin_char_rx_handler */
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
/*****************************************************************
* Procedure      : main_task
* Description    : Main task covering LIN functionalities
* In parameters  : None
* Out parameters : None
* Return value   : void */
void main_task (void)
{
  /* Do some application specific stuff... */

  /* Just a small example of signal and flag handling */
  if (l_flg_tst_InternalLightsRequest_flag ())
  {
    l_flg_clr_InternalLightsRequest_flag ();
    if (l_u8_rd_InternalLightsSwitch () == 1) {
      /* turn on lights */
    }
  }
} /* main_task */

/*****************************************************************
* Procedure      : main
* Description    : Main entry of application
* In parameters  : None
* Out parameters : None
* Return value   : function will never return
*/
int main (void)
{
  l_u8 cfg[20];
  l_u8 len = 0;
  l_bool configuration_ok = 0;
  l_bool stored_configuration = 0;

  /* Initialize the LIN interface */
  if (l_sys_init ()) {
    /* The init of the LIN software failed - call error routine */
  }

  /* Initialize the interface */
  if (l_ifc_init_i1 ()) {
    /* Initialization of the LIN interface failed - call error routine */
  }

  /* Now is the first time the LIN interrupts can be enabled */
  l_sys_irq_restore (INT_ENABLE_LEVEL);
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
/* Configure the communication */
  configuration_ok = 0;
  stored_configuration = is_configuration_stored ();
  if (stored_configuration) {
    /* there is a stored configuration in NVRAM */
   read_from_NVRAM (cfg, &len);
    /* configure the communication */
    ld_set_configuration (i1, cfg, len);
    configuration_ok = 1;
  } else {
    /* wait for the master to configure me for 5 s*/
    l_u16 configuration_timeout = 1000;
    do {
      if (l_ifc_read_status_i1 () & SAVE_CONFIGURATION) {
        /* The master node is finished with the configuration */
        configuration_ok = 1;
        /* save configuration in NVRAM */
        ld_read_configuration (i1, cfg, len);
        write_to_NVRAM (cfg, len);
      }
      delay_5ms ();
      configuration_timeout--;
    } while (configuration_timeout || !configuration_ok);
  }
  if (!configuration_ok) {
    /* Timeout - no configuration from master, enter limp home */
  }

  while (1) {
    /* Call the only task */
    main_task ();
  }

  /* return code */
  return 1;
} /* main */
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Node Capability Language Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.1 INTRODUCTION

The intention of a 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 description file (LDF), see Configuration Language Specification, and initialization code (configuring the cluster, e.g. reconfigure conflicting frame identifiers) for the master node.

If the setup and configuration of any 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 cluster as a single CPU node with the physical devices connected directly to the node.

8.1.1 PLUG AND PLAY WORKFLOW

Figure 8.1 shows the development of a cluster split in three areas; design, debugging and the LIN physical cluster. This specification focuses on the design phase.

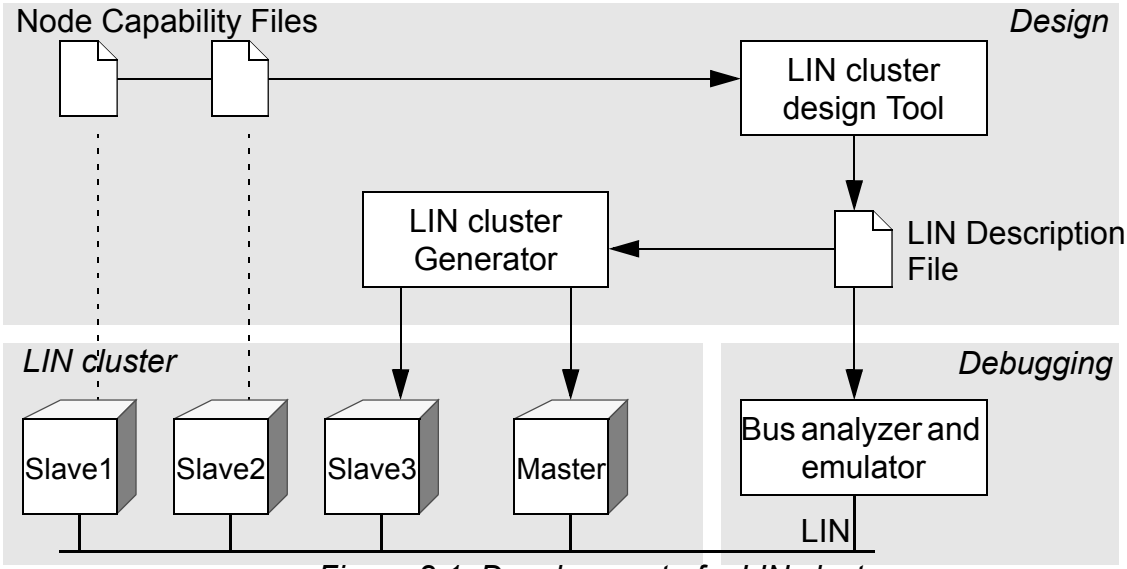


Figure 8.1: Development of a LIN cluster.

8.1.1.1 LIN cluster 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 LIN cluster generation. All signals and frames of the cluster are declared in this file.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.1.1.2 LIN cluster design

The process of creating the LDF file is named LIN cluster design. 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 cluster contains slave node address conflicts or frame identifier conflicts.

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

If you want to create new slave nodes as well, (Slave3 in Figure 8.1) the process becomes somewhat more complicated. The steps to perform depend on the LIN cluster design 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 slave nodes or frames must have been resolved before activating the cluster traffic.

8.1.1.3 Debugging

Debugging and node emulation is based on the LDF file produced in the LIN cluster design. 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 LIN cluster design tool.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.2 NODE CAPABILITY FILE DEFINITION

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

8.2.1 GLOBAL DEFINITION

Global definition data defines general properties of the file.

8.2.1.1 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 2.1.

8.2.2 NODE DEFINITION

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

<node_name> ::= identifier
```

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

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

8.2.3 GENERAL DEFINITION

```
<general_definition> ::=
  general {
    LIN_protocol_version = <protocol_version> ;
    supplier = <supplier_id> ;
    function = <function_id> ;
    variant = <variant_id> ;
    bitrate = <bitrate_definition> ;
    sends_wake_up_signal = "yes" | "no" ;
  }
```

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

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.2.3.1 LIN protocol version number definition

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

This specifies the protocol used by the slave node and it shall be in the range of "0.01" to "99.99".

8.2.3.2 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 Section 4.2.1.

8.2.3.3 Bit rate

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

Three kinds of bitrate_definition are possible:

- **automatic**, the slave 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 slave 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, slave nodes are encouraged to build automatic slave nodes since this gives the most flexibility to the cluster builder.

```
<bitrate> ::= real_or_integer kbps
```

The bit rates are specified in the range of 1 to 20 kbps.

8.2.3.4 Sends wake up signal

This parameter is set to yes if the slave is able to transmit the wake up signal. Otherwise it is set to no.

8.2.4 DIAGNOSTIC DEFINITION

```
<diagnostic_definition> ::=
  diagnostic {
    NAD = integer ([, integer]) ; | (integer to integer) ;
    diagnostic_class = integer ;
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
    (P2_min = real_or_integer ms ;)
    (ST_min = real_or_integer ms ;)
    (N_As_timeout = real_or_integer ms ;)
    (N_Cr_timeout = real_or_integer ms ;)
    (support_sid { integer ([, integer]) } ;)
    (max_message_length = integer ;)
}
```

The diagnostic_definition specifies the properties for transport layer and configuration.

The NAD property defines the initial node address; the value shall be set according to Section 4.2.3.2. Either a list of values or a range can be given. The range is inclusive, i.e. both values are included in the range. If more than one value is given, the slave will dynamically select one of the values within the given NAD set based on a physical property.

The diagnostic class defines the supported class 1, 2 or 3.

The default values of P2_min is 50 ms and ST_min are 0 ms, see Section 5.6.

The default values of N_As_timeout and N_Cr_timeout are 1000 ms, see Section 3.2.5.

Above timing parameters are only relevant for diagnostic class II and class III slave nodes.

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

The support_sid lists all SID values (node configuration, identification and diagnostic services) that are supported by the slave node. Default: 0xB2, 0xB7. Note that the supported identifiers for the read by identifier are not given.

8.2.5 FRAME DEFINITION

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

Frames listed here shall be all unconditional frames and event triggered frames processed by the slave node. Event triggered frames means the event triggered frame header, it will therefore not contain any signals. The diagnostic frames will always be supported and therefore not listed.

```
<single_frame> ::=
  <frame_kind> <frame_name> {
    <frame_properties>
    (<signal_definition>)
  }
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
<frame_kind> ::= publish | subscribe
<frame_name> ::= identifier
```

Each frame published or subscribed is declared as defined above. The frame_name is the symbolic name of the frame. The frame_kind is determined from the slave node point of view (e.g. a transmitted frame shall be a published frame).

8.2.5.1 Frame properties

```
<frame_properties> ::=
  length = integer ;
  (min_period = integer ms ;)
  (max_period = integer ms ;)
  (event_triggered_frame = identifier;)
```

The length is 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.

The event_triggered_frame refers to a event triggered frame, in case that the described frame is associated with it.

Several restrictions apply when a frame is also event triggered, see Section 2.3.3.2.

8.2.5.2 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_name> ;)

<init_value> ::= <init_value_scalar> | <init_value_array>

<init_value_scalar> ::= init_value = integer

<init_value_array> ::= 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 init_value_scalar is used for scalar signals and the init_value_array is used for byte array signals. The init_value_array is given in big-endian order.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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).

For a byte array, both size and offset must be multiples of eight.

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.

The encoding_name is a reference to a encoding defined in encoding clausal, defined below.

8.2.5.3 Signal encoding type definition

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

```
<encoding_definition> ::=
  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 (16).

$$physical_value = (scale * raw_value) + offset \quad (16)$$

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.2.6 STATUS MANAGEMENT

```
<status_management> ::=
  status_management {
    response_error = identifier ;
    (fault_state_signals = identifier ([, identifier]) ;)
  }

<published_signal> ::= identifier
```

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

The identifiers above refer each to one unique published signal in the signal definition, see Section 8.2.5.2. See the definition of the response error signal in Section 2.7.3 and the fault state signals in Section 5.3.

8.2.7 FREE TEXT DEFINITION

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

The free_text_definition is used to bring up help text, limitations, etc., in the LIN cluster design tool, if desired.

Typical information provided in the free text definition is:

- Slave node purpose and physical world interaction, e.g. motor speed, power consumption etc.
- Deviations from the LIN standard.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.3 OVERVIEW OF SYNTAX

The syntax is described using a modified BNF (Bachus-Naur Format), as summarized in **Table 8.1** below.

Symbol	Meaning
::=	A name on the left of the ::= is expressed using the syntax on its right
<>	Used to mark objects specified later
|	The vertical bar indicates choice. Either the left-hand side or the right hand side of the vertical bar shall appear
Bold	The text in bold is reserved - either because it is a reserved word, or mandatory punctuation
[ ]	The text between the square brackets shall appear once or 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 or hexadecimal (prefixed with 0x) format.
real_or_integer	A real or integer number. A real number is always in decimal and has an embedded decimal point.

Table 8.1: BNF syntax used in this document.

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.

The reserved text and identifiers are case sensitive.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

8.4 EXAMPLE FILE

```
node_capability_file;
LIN_language_version = "2.1";

node step_motor {
  general {
    LIN_protocol_version = "2.1";
    supplier = 0x0005; function = 0x0020; variant = 1;
    bitrate = automatic min 10 kbps max 20 kbps;
    sends_wake_up_signal = "yes";
  }

  diagnostic {
    NAD = 1 to 3;
    diagnostic_class = 2;
    P2_min = 100 ms; ST_min = 40 ms;
    support_sid { 0xB0, 0xB2, 0xB7 };
  }

  frames {
    publish node_status {
      length = 4; min_period = 10 ms; max_period = 100 ms;
      signals {
        state       {size = 8; init_value = 0; offset = 0;}
        fault_state {size = 2; init_value = 0; offset = 9; fault_enc;}
        error_bit   {size = 1; init_value = 0; offset = 8;}
        angle       {size = 16; init_value = {0x22, 0x11}; offset = 16;}
      }
    }

    subscribe control {
      length = 1; max_period = 100 ms;
      signals {
        command {size = 8; init_value = 0; offset = 0; position;}
      }
    }
  }

  encoding {
    position {physical_value 0, 199, 1.8, 0, "deg";}
    fault_enc {logical_value, 0, "no result";
               logical_value, 1, "failed";
               logical_value, 2, "passed";}
  }

  status_management { response_error = error_bit;
                      fault_state_signals = fault_state; }

  free_text { "step_motor signal values outside 0 - 199 are ignored" }
}
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

LIN
Configuration Language Specification

Revision 2.1

© LIN Consortium, 2006.

This specification as released by the LIN Consortium is intended for the purpose of information only and is provided on an "AS IS" basis only and cannot be the basis for any claims. The LIN Consortium will not be liable for any use of this Specification. The unauthorized use, e.g. copying, displaying or other use of any content from this document is a violation of the law and intellectual property rights.

LIN is a registered Trademark ®. All rights reserved.
All distributions are registered.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.1 INTRODUCTION

The language described in this document is used in order to create a LIN description file. The LIN description file describes a complete cluster and also contains all information necessary to monitor the cluster. 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 cluster. An application program interface has been defined, see Application Program Interface Specification, in order to have a uniform way to access the cluster 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 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 cluster.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.2 LIN DESCRIPTION FILE DEFINITION

```
<LIN_description_file> ::=
  LIN_description_file ;
  <LIN_protocol_version_def>
  <LIN_language_version_def>
  <LIN_speed_def>
  (<Channel_name_def>)
  <Node_def>
  (<Node_composition_def>)
  <Signal_def>
  (<Diag_signal_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.

9.2.1 GLOBAL DEFINITION

Global definition data defines general properties of the cluster.

9.2.1.1 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".

9.2.1.2 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 2.1.

9.2.1.3 LIN speed definition

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

This sets the nominal bit rate for the cluster. It shall be in the range of 1 to 20 kbit/second.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.2.1.4 Channel postfix name definition

```
<Channel_name_def> ::=
  Channel_name = char_string ;
```

Postfix for all named objects in the LDF. The postfix is mandatory for master nodes that are connected too more than one cluster. It is used to avoid naming collision if a node is connected to several clusters (i.e. using several LDFs). If given all named objects shall add this postfix to its name.

9.2.2 NODE DEFINITION

The node definition sections identify 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.

9.2.2.1 Participating nodes

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

<node_name> ::= identifier
```

The nodes clause lists the physical nodes participating in the cluster. 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 shall be specified in milliseconds. For more information on time_base and jitter usage see Section 2.4.

9.2.2.2 Node attributes

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

```
<node_attributes_def> ::=
  Node_attributes {
    [<node_name> {
      LIN_protocol = <protocol_version> ;
      configured_NAD = <diag_address> ;
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
      (initial_NAD = <diag_address> ;)
      <attributes_def> ;
    }]
  }
```

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

All node_name identifiers shall exist within the node identifier set and refer to a slave node.

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

Shall be in the range of "0.01" to "99.99".

```
<diag_address> ::= integer
```

The diag_address specifies the diagnostic address for the identified slave node in the range as defined in Section 4.2.3.2. It shall specify the unique NAD used for the slave node after resolving any cluster conflicts, i.e. it shall be unique within the cluster.

In case the initial_NAD is not given the configured_NAD is the same as the initial_NAD.

In case of LIN 1.x the attributes_def contains no attributes.

In case of LIN 2.x, following attributes applies:

```
<attributes_def> ::=
  product_id = <supplier_id>, <function_id> (, <variant>) ;
  response_error = <signal_name> ;
  (fault_state_signals = <signal_name>([, <signal_name>]) ;)
  (P2_min = real_or_interger ms ;)
  (ST_min = real_or_interger ms ;)
  (N_As_timeout = real_or_integer ms ;)
  (N_Cr_timeout = real_or_integer ms ;)
  <configurable_frames_20_def> | <configurable_frames_21_def>
```

The product_id_def for LIN 2.x:

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

The supplier_id, function_id and variant_id ranges are defined in Section 4.2.1.

The variant ID is optional since it is a property of the slave node and not the cluster. It is set here for LIN 2.0 slave nodes.

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


Registered copy for cbarbaud@cmaisonneuve.qc.ca

All signal_name identifiers for the response error signals shall exist within the signal identifier set and refer to a one bit standard signal, see Section 9.2.3.1. The response error signal shall be published by the specified slave node. Refer to status management, Section 2.7, for more information.

The fault_state_signals is a property of LIN 2.1 slave nodes, and are used for diagnostic class I and II, see Section 5.3.

The default values of P2_min is 50 ms and ST_min is 0 ms, see Section 5.6.

The default values of N_As_timeout and N_Cr_timeout are 1000 ms, see Section 3.2.5. These values are only used in LIN 2.1 slave nodes.

Configurable frames shall list all frames (unconditional frames, event-triggered frames and sporadic frames) processed by the slave node. This section applies to LIN 2.x slave nodes only (not to LIN 1.x).

The configurable_frames_def for LIN 2.0:

```
<configurable_frames_20_def> ::=
  configurable_frames {
    [<frame_name> = <message_id> ;]
  }

<message_id> ::= integer
```

The message_id range is defined in the **LIN Diagnostic and Configuration Specification** of LIN 2.0.

The configurable_frames_def for LIN 2.1:

```
<configurable_frames_21_def> ::=
  configurable_frames {
     [<frame_name> ;]
  }
```

The order of the frames are important since the node configuration request assign frame PID range dependent on the order, see Section 4.2.5.5.

9.2.2.3 Node composition definition

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

```
<node_composition_def> ::=
  composite {
    [configuration <configuration_name> {
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
    [<composite_node> {
      <logical_node> ([, <logical_node>]) ;]
  }]
}
```

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

```
<configuration_name> ::= identifier
```

The configuration_name is used to handle different configurations of the composite slave nodes. A physical cluster is statically built according to one of the configuration_names. The used configuration must be set outside the LDF.

```
<composite_node> ::= identifier
```

The composite_node groups a number of logical slave nodes into one physical slave node. The composite_node is used in the list of participating slave nodes, Section 9.2.2.1.

```
<logical_node> ::= identifier
```

The logical slave node shall be listed in the node attributes section. Each logical slave node will keep its NAD. The master node will communicate to the logical slave node as if they were separated.

9.2.3 SIGNAL DEFINITION

The signal definition sections identify 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.

9.2.3.1 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> ::= <init_value_scalar> | <init_value_array>

<init_value_scalar> ::= integer
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
<init_value_array> ::= {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 init_value_scalar is used for scalar signals and the init_value_array is used for byte array signals. The initial_value for byte arrays shall be arranged in big-endian order (i.e. with the most significant byte first).

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.

9.2.3.2 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 ;
    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.

9.2.3.3 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.x cluster.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
<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
```

9.2.4 FRAME DEFINITION

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

9.2.4.1 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 identifier number in range 0 to 59. The frame identifier shall be unique for all frames within the frames identifier set.

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

The published_by identifier shall exist in the node identifier set.

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

The frame_size specifies the size of the frame in range 1 to 8 bytes.

```
<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
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

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

Example

Table 9.1 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).

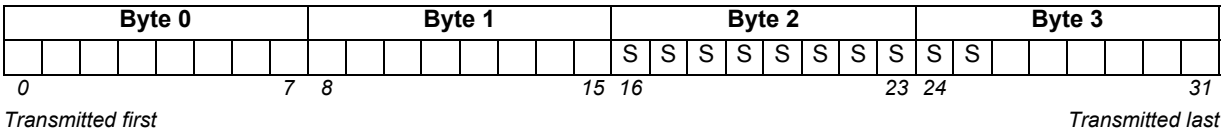


Table 9.1: Packing of a signal.

9.2.4.2 Sporadic frames

```
<sporadic_frame_def> ::=
  Sporadic_frames {
    [<sporadic_frame_name>: <frame_name> ([, <frame_name>]) ;]
  }

<sporadic_frame_name> ::= identifier
```

All sporadic_frame_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 unconditional frames published by the master node. Furthermore, they shall not be scheduled as unconditional frames directly in the same schedule table as the sporadic_frame_name.

9.2.4.3 Event triggered frames

```
<event_triggered_frame_def> ::=
  Event_triggered_frames {
    [<event_trig_frm_name>:
      <collision_resolving_schedule_table>,
      <frame_id>
      [, <frame_name>] ;]
  }

<event_trig_frm_name> ::= identifier
```

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

```
<collision_resolving_schedule_table> ::= identifier
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

This refers to a schedule table in the schedule table set. This schedule will be automatically activated after the collision. It shall minimum contain the associated unconditional frames.

```
<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.

Remark

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

9.2.4.4 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 (see Section 2.3.3.4) and shall be unique in the frame identifier set.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.2.5 SCHEDULE TABLE DEFINITION

The schedule table describes the frames and the timing of the frames transmitted on the bus. Valid frames in the schedule tables are the frame types defined in Section 2.3.3 (except the reserved frames) and the node configuration commands listed below.

```
<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 |
  AssignNAD {<node_name>} |
  ConditionalChangeNAD {<NAD>, <id>, <byte>, <mask>, <inv>, <new_NAD>} |
  DataDump {<node_name>, <D1>, <D2>, <D3>, <D4>, <D5>} |
  SaveConfiguration {<node_name>} |
  AssignFrameIdRange {<node_name>, <frame_index> (, <frame_PID>,
                      <frame_PID>, <frame_PID>, <frame_PID>)} |
  FreeFormat {<D1>, <D2>, <D3>, <D4>, <D5>, <D6>, <D7>, <D8>}
  (| AssignFrameId { <node_name>, <frame_name> })
```

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

```
<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.

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

The node_name refers to one slave node, see Section 9.2.2.2.

MasterReq and **SlaveResp** are either defined as frames in Section 9.2.4.4 or, if this clause is left out, automatically defined. The contents of these frames is provided via the services in the Node configuration and Identification Specification and Diagnostic specification.

AssignNAD generates a assign NAD request, see Section 4.2.5.1.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

ConditionalChangeNAD generates an conditional change NAD request, see Section 4.2.5.2.

DataDump generates a data dump request, see Section 4.2.5.3.

SaveConfiguration generates a save configuration request, see Section 4.2.5.4.

AssignFrameIdRange generates an assign frame PID range request with the contents based on the parameters: NAD and the order of the frames (frame_index) are taken from the node attributes of the node_name.

```
<frame_index> ::= integer
```

The frame_index sets the index to the first frame to assign a PID, see Section 9.2.2.2.

```
<frame_PID> ::= identifier
```

If the optional four frame_PID are given the request will include these values. If frame_PID are not given the PIDs for the four frames are taken from the frame definition for frame_name, see Section 9.2.4.

AssignFrameId generates an Assign_frame_id request 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 9.2.2.2 and the protected_id is taken from the frame definition for frame_name, see Section 9.2.4.

All data in this frame is fixed and determined during the processing of the LDF file. This service is only supported if the master node also support this configuration service. This service is optional since it is only defined in the LIN 2.0 specification.

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

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

The frame_time specifies the duration of the frame slot, see Section 2.4.2. The frame_time value shall be specified in milliseconds.

The handling and switching of schedule table is controlled by the master application program, see description in Section 2.4 and the schedule table handling API in Section 7.2.4.

Example

Figure 9.1 shows a time line that corresponds to the schedule table VL1_ST1. It is assumed that the time_base (see Section 9.2.2.1) is set to 5 ms.

```
schedule_tables {
  VL1_ST1 {
    VL1_CEM_Frm1 delay 15 ms;
    VL1_LSM_Frm1 delay 15 ms;
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
      VL1_CPM_Frm1 delay 15 ms;
      VL1_CPM_Frm2 delay 20 ms;
    }
  }
```

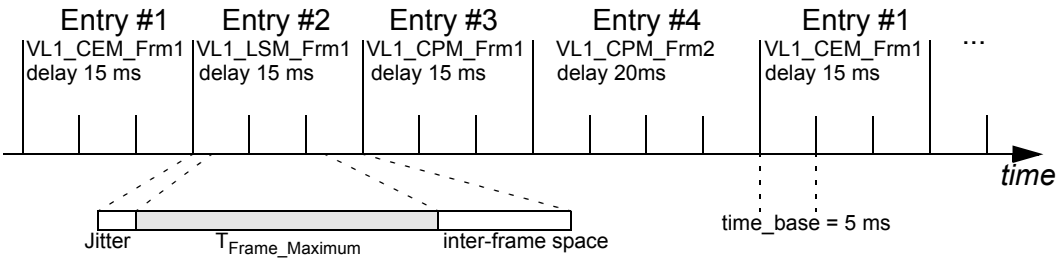


Figure 9.1: Time line for the VL1_ST1 schedule table.

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

9.2.6 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.

9.2.6.1 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 the node application, those API routines would require quite powerful nodes. The main purpose of the signal encoding type declarations is in bus traffic analyzing 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.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
<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 (17).

physical_value = (scale * raw_value) + offset. (17)

Example

The V_battery signal is an eight bit representation that follows the graph in Figure 9.1, 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";
  }
}
```


Registered copy for cbarbaud@cmaisonneuve.qc.ca

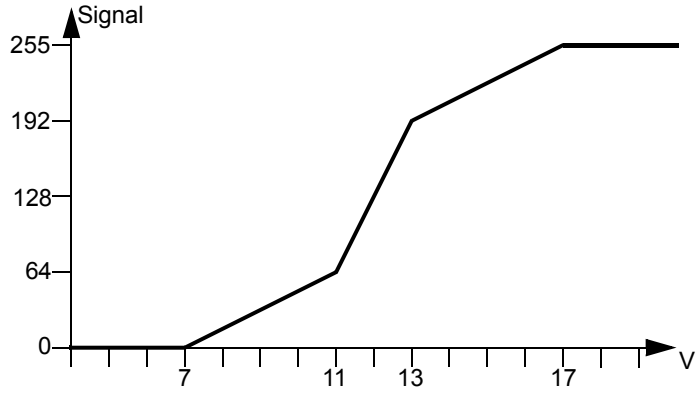


Figure 9.1: Representation of V_battery.

9.2.6.2 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 (both scalar and byte array signals are applicable). Each signal may only be associated with one signal_encoding_type_name and may not be nested in a signal_group_name.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.3 OVERVIEW OF SYNTAX

The syntax is described using a modified BNF (Bachus-Naur Format), as summarized in Table 9.2 below.

Symbol	Meaning
::=	A name on the left of the ::= is expressed using the syntax on its right
<>	Used to mark objects specified later
|	The vertical bar indicates choice. Either the left-hand side or the right hand side of the vertical bar shall appear
Bold	The text in bold is reserved - either because it is a reserved word, or mandatory punctuation
[ ]	The text between the square brackets shall appear once or 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 or hexadecimal (prefixed with 0x) format
real_or_integer	A real or integer number. A real number is always in decimal and has an embedded decimal point.

Table 9.2: BNF syntax used in this document.

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.

The reserved text and identifiers are case sensitive.

Registered copy for cbarbaud@cmaisonneuve.qc.ca

9.4 EXAMPLES

9.4.1 LIN DESCRIPTION FILE

```
LIN_description_file;
LIN_protocol_version = "2.1";
LIN_language_version = "2.1";
LIN_speed = 19.2 kbps;
Channel_name = "DB";

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

Node_attributes {
  LSM {
    LIN_protocol = "2.1";
    configured_NAD = 0x20;
    initial_NAD = 0x01;
    product_id = 0x4A4F, 0x4841;
    response_error = LSMerror;
    fault_state_signals = IntTest;
    P2_min = 150 ms;
    ST_min = 50 ms;
    configurable_frames {
      CEM_Frm1; LSM_Frm1; LSM_Frm2;}
  }

  RSM {
    LIN_protocol = "2.0";
    configured_NAD = 0x20;
    product_id = 0x4E4E, 0x4553, 1;
    response_error = RSMerror;
    P2_min = 150 ms;
    ST_min = 50 ms;
    configurable_frames {
      CEM_Frm1 = 0x0001; LSM_Frm1 = 0x0002; LSM_Frm2 = 0x0003;}
  }
}

Signals {
  IntLightsReq: 2, 0, CEM, LSM, RSM;
  RightIntLightsSwitch: 8, 0, RSM, CEM;
  LeftIntlLightsSwitch: 8, 0, LSM, CEM;
  LSMerror, 1, 0, LSM, CEM;
  RSMerror, 1, 0, LSM, CEM;
  IntTest, 2, 0, LSM, CEM;
}
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
Frames {
  CEM_Frm1: 0x01, CEM, 1 {
    InternalLightsRequest, 0;
  }
  LSM_Frm1: 0x02, LSM, 2 {
    LeftIntLightsSwitch, 0;
  }
  LSM_Frm2: 0x03, LSM, 1 {
    LSMerror, 0;
    IntError, 1;
  }
  RSM_Frm1: 0x04, RSM, 2 {
    RightIntLightsSwitch, 0;
  }
  RSM_Frm2: 0x05, RSM, 1 {
    RSMerror, 0;
  }
}

Event_triggered_frames {
  Node_Status_Event : Collision_resolver, 0x06, RSM_Frm1, LSM_Frm1;
}

Schedule_tables {
  Configuration_Schedule {
    AssignNAD {LSM} delay 15 ms;
    AssignFrameIdRange {LSM, 0} delay 15 ms;
    AssignFrameId {RSM, CEM_Frm1} delay 15 ms;
    AssignFrameId {RSM, RSM_Frm1} delay 15 ms;
    AssignFrameId {RSM, RSM_Frm2} delay 15 ms;
  }

  Normal_Schedule {
    CEM_Frm1 delay 15 ms;
    LSM_Frm2 delay 15 ms;
    RSM_Frm2 delay 15 ms;
    Node_Status_Event delay 10 ms;
  }

  MRF_schedule {
    MasterReq delay 10 ms;
  }

  SRF_schedule {
    SlaveResp delay 10 ms;
  }

  Collision_resolver { // Keep timing of other frames if collision
    CEM_Frm1 delay 15 ms;
    LSM_Frm2 delay 15 ms;
    RSM_Frm2 delay 15 ms;
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca

```
    RSM_Frm1 delay 10 ms;  // Poll the RSM node
    CEM_Frm1 delay 15 ms;
    LSM_Frm2 delay 15 ms;
    RSM_Frm2 delay 15 ms;
    LSM_Frm1 delay 10 ms;  // Poll the LSM node
  }
}

Signal_encoding_types {
  Dig2Bit {
    logical_value, 0, "off";
    logical_value, 1, "on";
    logical_value, 2, "error";
    logical_value, 3, "void";
  }

  ErrorEncoding {
    logical_value, 0, "OK";
    logical_value, 1, "error";
  }

  FaultStateEncoding {
    logical_value, 0, "No test result";
    logical_value, 1, "failed";
    logical_value, 2, "passed";
    logical_value, 3, "not used";
  }

  LightEncoding {
    logical_value, 0, "Off";
    physcial_value, 1, 254, 1, 100, "lux";
    logical_value, 255, "error";
  }
}

Signal_representation {
  Dig2Bit: InternalLightsRequest;
  ErrorEncoding: RSMerror, LSMerror;
  FaultStateEncoding: IntError;
  LightEncoding: RightIntLightsSwitch, LefttIntLightsSwitch;
}
```

Registered copy for cbarbaud@cmaisonneuve.qc.ca