LIN - Protocol, Development Tools, and Software Interfaces for Local Interconnect Networks in Vehicles

LIN - Protokoll, Entwicklungswerkzeuge und Software-Schnittstelle für Lokale Datennetzwerke im Kraftfahrzeug

Dr.-Ing. **J. Will Specks,** Motorola GmbH, Munich
Antal Rajnák, Volcano Communications Technologies, Gothenburg (S)

Abstract – **LIN is a holistic communication concept for local interconnect networks in vehicles. The specification covers in addition to the definition of the protocol and the physical layer also the definition of interfaces for development tools and application software. LIN enables a cost-effective communication for smart sensors and actuators where the bandwidth and versatility of CAN is not required. The communication is based on the SCI (UART) data format, a single-master/multiple-slave concept, a single-wire 12V bus, and a clock synchronization for nodes without stabilized time base. The LIN specification is open and is driven by an automotive industry consortium.**

Zusammenfassung – LIN ist ein ganzheitliches Kommunikationskonzept für lokale Datennetzwerke im Kraftfahrzeug. Die Spezifikation umfasst zusätzlich zur Definition des Protokolls und der elektrischen Übertragungsebene auch die Definition der Schnittstellen für Entwicklungswerkzeuge and Anwendersoftware. LIN ermöglicht eine kostengünstige Kommunikation für intelligente Sensoren und Aktuatoren dort, wo die Bandbreite und Vielseitigkeit von CAN nicht erforderlich ist. Die Kommunikation basiert auf dem seriellen SCI (UART) Datenformat, einem Single-Master/Multi-Slave Konzept, einem 12V Eindrahtbus, und einer Taktsynchronisation für Busknoten ohne stabilisierte Zeitbasis. Die LIN-Spezifikation ist offengelegt und wird durch ein Konsortium der Automobilindustrie getrieben.

1. Introduction

The LIN Consortium started as workgroup in late 1998 as an initiative by the five car manufacturers Audi, BMW, DaimlerChrysler, Volvo and Volkswagen, the tool manufacturer VCT, and the semiconductor manufacturer Motorola. The objective of this workgroup is the specification of an open standard for low-cost local interconnect networks (LIN) in vehicles where the bandwidth and versatility of CAN are not required. Typical applications for LIN are smart sensors and actuators that require data communication from and to a network.

The LIN standard [1] specifies not only the data transmission but also gives provision for a highly automated tool chain. It addresses the needs of increasing complexity, implementation, and maintenance of software in distributed systems. For this very reason the LIN specification covers in addition to the definition of protocol and medium also the interfaces for the development tools and for a network-independent application software (see Figure 1).

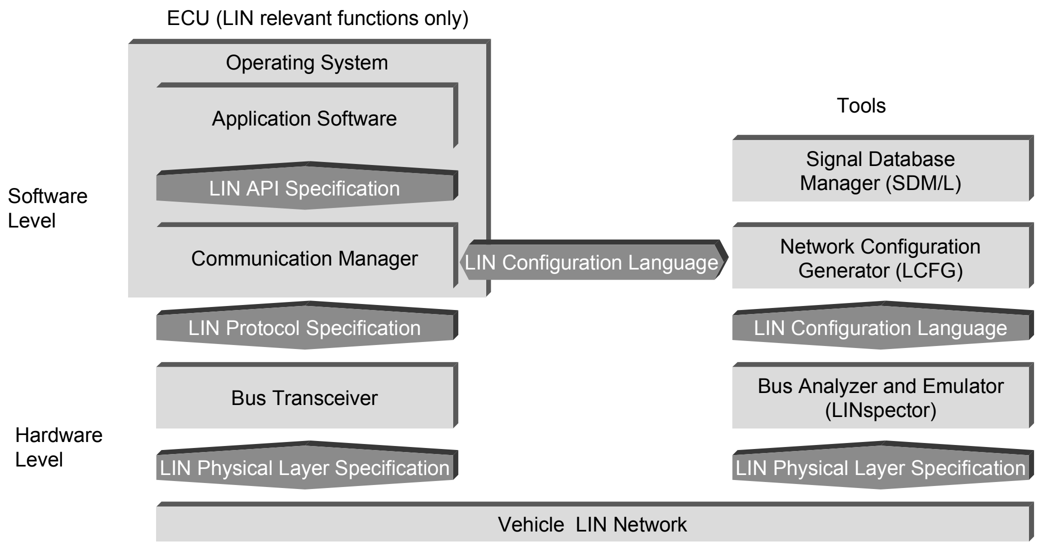

ECU (LIN relevant functions only)
Operating System
Application Software
LIN API Specification
Communication Manager
LIN Protocol Specification
Bus Transceiver
LIN Physical Layer Specification
Software Level
Hardware Level
Tools
Signal Database Manager (SDM/L)
Network Configuration Generator (LCFG)
LIN Configuration Language
LIN Configuration Language
Bus Analyzer and Emulator (LINspector)
LIN Physical Layer Specification
Vehicle LIN Network

Figure 1: Networking and tool interfaces for LIN

The LIN protocol (see Section 5) is based on the SCI (UART) serial data link format, which is supported by a wide products range. The communication concept is 'single-master/multiple-slave' with a message identification for multi-cast transmission between any network nodes. A particular feature of LIN is the synchronization for slave nodes that do not have a stabilized timebase for cost reasons. The physical layer (see Section 6) is a single-wire 12V bus interface, derived from the ISO9141 standard for automotive diagnostics, that has been adapted to the particular requirements by EMC, ESD, and noise under vehicle operation.

The LIN configuration language (see Section 3.4) is used to describe the network topology and objects, as e.g. nodes, interfaces, and latencies. This enables the integration of development tools from different vendors, such as the database manager, the network configuration manager, or the network analyzer/emulator. Finally, the application programmer's interface (API, see Section 4) defines common calls between application and communication software in an ECU. The definition of the API is simple but quite powerful as it provides the capability for automatic code generation.

1.1 The Need for LIN

The applications for network protocols in vehicles can be separated in four distinct areas, as illustrated by Figure 2. There is no single protocol, which could claim to represent a “one fits all” solution for all these areas. Each of them requires specific protocol features:

(a) Multimedia applications, calling for protocols providing high speed, high bandwidth, and even wireless interconnection, like MOST, D2B, or Bluetooth;

(b) Emerging safety critical applications in chassis and power train (x-by wire) calling for a fault tolerant, dependable protocol, like TTP/C, Byteflight, TT-CAN or others;

(c) Conventional body and powertrain applications, mainly using CAN;

(d) Mechatronic type applications such as smart sensors and actuator, or even complex ECUs with simple communications needs, being addressed by low-end protocols like LIN, TTP/A, J-1850, and quite a few other OEM or Tier-I in-house protocols.

The progress of development and consolidation of the various protocols is different in the four areas. In general, the evolution can be differentiated in two phases: the introduction and the consolidation. The introduction of protocols in a new field of application is driven by time to market. During this phase, a variety of protocols are developed in parallel and secrecy by several OEM. The standardization efforts are quite low, as the network requirements are difficulty to share without revealing details about the application itself.

The introduction phase is followed by the consolidation phase, which is driven by cost reduction. The need for a significant amount of resources and competence in application know-how, protocol design, tool-support, silicon-design and manufacturing drive the standardization through the industry. While the standardization in body electronics is far progressed with CAN, the consolidation process in the field of low-end multiplexing has started now.

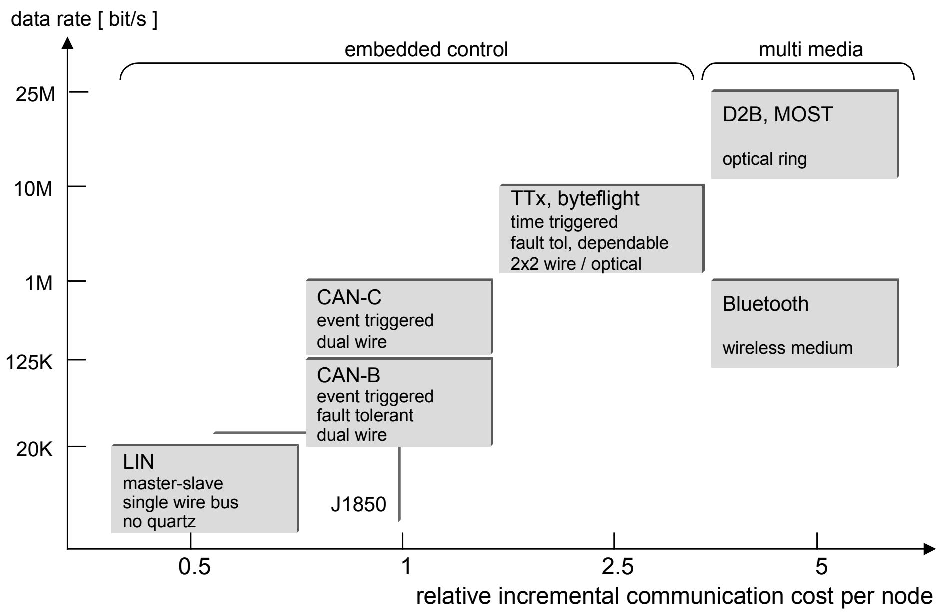


Figure 2: Hierarchy of the major multiplex networks in vehicles

There are virtually dozens of different SCI based protocols defined within the industry today, almost being proprietary in-house standards of either an OEM or a supplier. The approach of LIN to find the consensus for a standard during is based on the holistic addressing of the needs of development, configuration, programming, signal transmission, and interconnection, and not only the specification of another SCI protocol. The definition of LIN regards the lean development process in the automotive industry of tomorrow and provides well-defined tool interfaces, functional APIs and a properly designed and verified protocol.

Typical candidates for LIN nodes are door control (window lift, lock, mirror), roof control (light/rain sensor, fond control unit), steering wheel and steering column, seat control and heating, switch panels, motors and sensors in climate control, smart wiper motor, RF-receiver for remote control, or the intelligent alternator.

1.2 Performance and resource requirements of LIN versus CAN

LIN targets to low end applications where the communication cost per node must be two to three times lower compared to CAN but where the performance, bandwidth, and versatility of CAN is not required. The main saving factors of LIN versus CAN are the single-wire transmission, the low cost of implementation as hardware or software in silicon, and the avoidance of quartz or ceramics resonator in slave nodes. These advantages are compromised by a lower bandwidth and the restrictive single-master bus access scheme. The main features of LIN and CAN protocol, as well as the typical memory and CPU requirements of LIN and CAN nodes are compared for body applications in Table 1 and Table 2, respectively.

	LIN	**CAN**
medium access control	single master	multiple master
typical bus sped	2.4 … 19.6 kbps	62.5 … 500 kbps
multicast message routing	6-bit identifier	11 / 29-bit identifier
typical size of network	2 … 10 nodes	4 … 20 nodes
bit / byte coding	NRZ 8N1 (UART)	NRZ w/ bit stuffing
data byte per frame	2, 4, 8 byte	0 … 8 byte
transmission time for 4 data bytes	3.5 ms at 20 kbps	0.8 ms at 125 kbps
error detection (data field)	8-bit checksum	15-bit CRC
physical layer	single wire, 13.5V	twisted pair, 5V
quartz/ceramic resonator	no (except master)	yes
relative cost per network connection	x 0.5	x 1

Table 1: Comparison of the main features of LIN and CAN protocol in body applications

	network speed	**CPU clock**	**CPU load**	**memory flash/ROM**	**memory RAM**
	[kbps]	[MHz]	[%]	[byte]	[byte]
LIN 16-bit master	19.2	4	10	1200	25
LIN 8-bit slave w/o quartz	19.2	4	15	750	22
LIN 8-bit slave with quartz	19.2	4	6	650	20
CAN 16-bit node	125	8	15	3000	150

Table 2: Typical memory and CPU requirements for LIN and CAN microcontroller (C)

2. The Vehicle EE Architecture with LIN

The objective for the design of the vehicle electrical and electronic (EE) network is to enable the exchange of signals between nodes meeting the latency requirements and communication security at lowest system cost. The optimization of the network requires the consideration of criteria such as

- number of signal to be communicated;
- signal latencies and other real-time requirements of functions;
- speed, bandwidth, and medium of the bus;
- electro-magnetic compatibility (EMC);
- fault tolerance or fail safety;
- cost per electrical interconnection versus cost of local intelligence;

- ECU variety cost by customer options and over vehicle platforms, system scalability;
- cost and reliability of systems integration in silicon and mechatronics;
- availability of tools and software;
- existing infrastructures and development skills, architecture and networking legacies.

The network architecture depends on the weight and balance of the above items. The possible outcomes of such an optimization are illustrated by the two examples in Figure 3 and Figure 4. The examples show a central comfort ECU that controls actuators and sensors in a seat. The ECU exchanges signals via a CAN link with other main ECUs as for example the instrumentation cluster. The CAN link between the main ECUs is required due to the amount of signals and latency requirements between these nodes.

The network architecture in Figure 3 is based on the intensive usage of the CAN backbone, which links the zone modules with the central ECU. This architecture is today common for highly functional zone ECUs as in door, seat, or roof modules [2]. The actuators and sensors are hard-wired to the zone modules. This partitioning is chosen if the complexity of the system requires a high bandwidth for signal exchange between the main ECUs, and if the local actuators and sensors require a high computing performance. This architecture is cost effective only if the diversity of zone ECUs can be kept low because any change in the peripheral electric requires another ECU design and qualification.

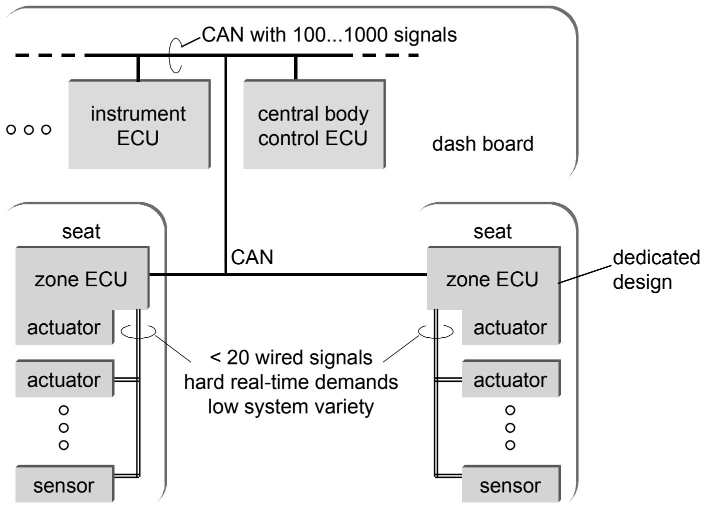


Figure 3: Network with CAN zone ECUs and locally hard-wired actuators and sensors

The architecture in Figure 4 shows an alternative distributed system that is based on smart actuators and sensors. The zone ECU is almost dissolved and replaced by mechatronics that are linked to the central comfort ECU via several LIN links. This partitioning is chosen in order to achieve a scaleable EE architecture with universally applicable mechatronic components. This architecture will be cost effective if the additional cost for local intelligence and networking can be compensated by cost savings in production and development due to a lower variety of electronic components. The key enablers for this architecture are a sub-bus standard, low-cost mechatronic assembly, and semiconductor systems integration.

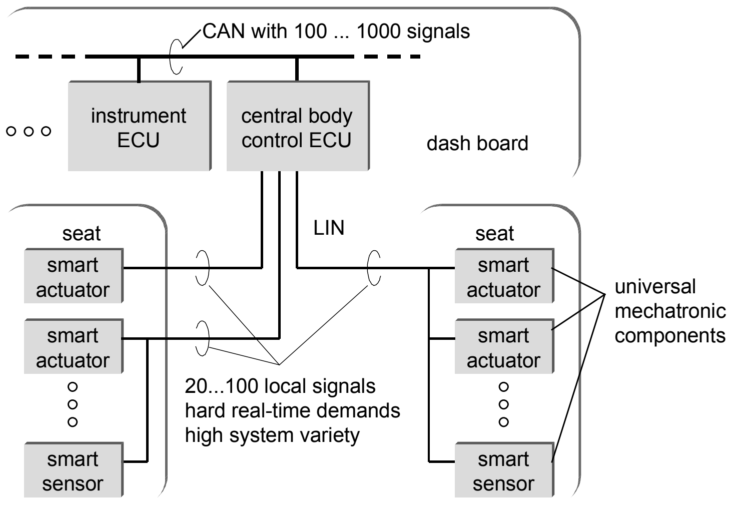


Figure 4: Network with smart actuators and sensors directly linked to the main ECU via LIN

2.1 Case Study: Seat, Steering, and Mirror Systems with LIN

This case study shall illustrate the philosophy of a LIN architecture and shall also give the background for the description of the network configuration and management as well as software programming in Section 3. It is the body comfort system comprising seat, steering, and mirror nodes in Figure 5. This example includes four different LIN networks with one master ECU that also acts as a gateway between the different networks. The master node is also connected to another CAN network.

The general philosophy demonstrated in this case study is to build the network links around the system functionality rather than around mechanical assembly needs. This is why in this example the seat, the steering, and the mirror form a quasi-single system around the driver.

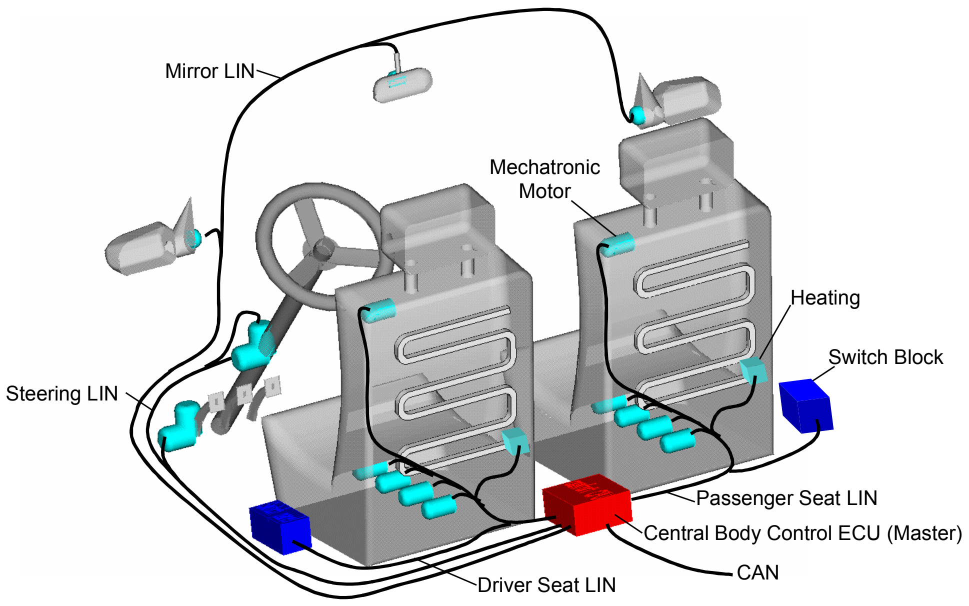


Figure 5: LIN case study with mechatronic seat, mirror, and steering nodes

3. Network Design and Configuration

3.1 Tool Chain and Workflow

The holistic concept of LIN supports the entire development, configuration, and maintenance of a network by the definition of interfaces not only between network nodes but also between automated tools and software modules. Tool chain and workflow are illustrated in Figure 6. The main tools are the signal database manager for LIN (SDML), the LIN configuration manager (LCFG), the software compiler and linker, and the bus analysis tool (LINspector).

The signal database manager is a tool for definition, configuration and maintenance of LIN networks. It is a PC Windows program that captures all properties of a LIN project including the definition of signals, node, interfaces, and latency requirements. With the database manager the project configuration comprising networks, functions, and gatewayed signals is created. The *frame/schedule packer* packs signals into frames, and creates the message schedules. A timing analysis is performed to ensure that all timing requirements of the signals are fulfilled, telling if a configuration is schedulable or not. Finally, the LIN configuration file is generated together with other reports describing entities of a specific node or network.

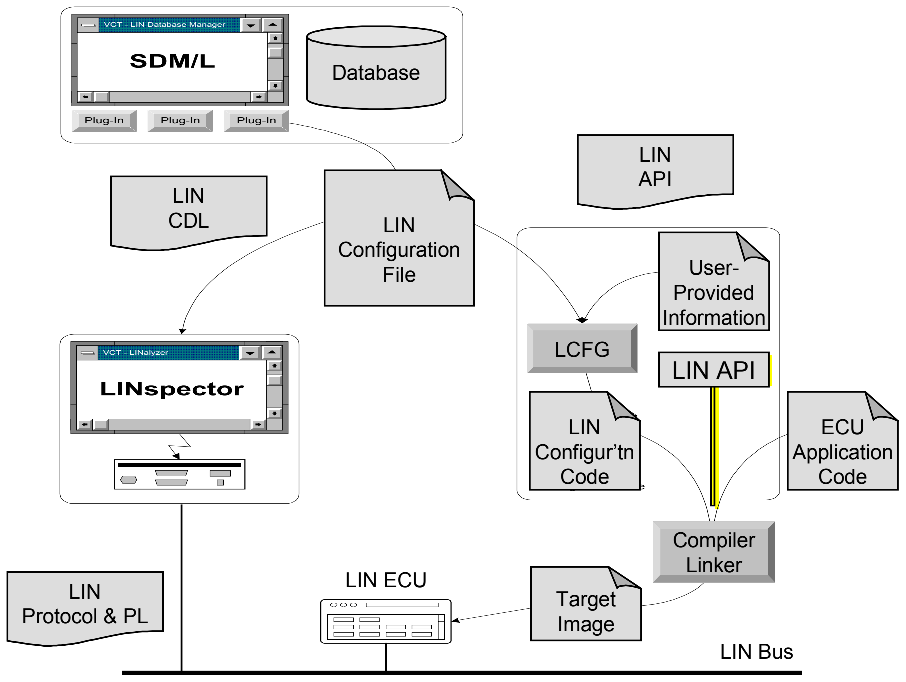


Figure 6: Network configuration and development workflow with LIN

The configuration file (see Section 3.4) contains all significant network information and is an input to the network analyzer and the configuration manager, which merges the network information with ECU-specific information and finally generates the LIN configuration C-code. This configuration code is compiled and linked with the ECU application code and loaded as target image into the node. The API (see Section 4) ensures a composable system for which the application code can be independently developed from the network definition.

3.2 Definition of Network Objects

3.2.1 Signals, Latencies, and Encoding Types

Signals, encoding types, nodes, and interfaces are entered as global objects into the signal database manager and can be used for different projects, releases, and configurations. For every node there is one or more interfaces defined. Though the SDML will only handle LIN networks, a node can of course also have interfaces to other networks as for example CAN.

For each interface the global signals that are to be published or subscribed are defined. A signal can only be published from one interface, but any signal can be subscribed by several interfaces on a network. A signal that is published in one network and is subscribed in another network is defined as a gatewayed signal. The encoding types describe how a signal is encoded and decoded in terms of bits that represent logical or physical values and that are carried by the message frame.

The timing analysis of the system requires the definition of the master time base, the master jitter, and various signal latency parameters. These values are used for the timing analysis of the network. The master time base is the interval between two consecutive calls to a ‘tic’ function according to the LIN API. The master jitter is calculated as the maximum difference between the master time base and the distance of two consecutive ‘tics’.

Three types of signal latencies are required for the network configuration: generation latency, consumption latency, and maximum age. The definitions and the corresponding timing model are illustrated in Figure 7. The generation latency is defined as the time between an event input (e.g. a button pressed) and the signal being placed in a buffer, ready for transmission. It has to be noted that some time might pass before the transport itself will take place, depending on the actual schedule table and the master’s call to the “tic” function.

The consumption latency is defined as the time from when a signal has been received from the network into a buffer until it has been read into the subscribing application and some action has taken place (e.g. a motor starts moving). Further, every subscribed signal is given a maximum age. This is the maximum allowed time between a user action in the publishing node until the subscribing node is actually carrying out an action.

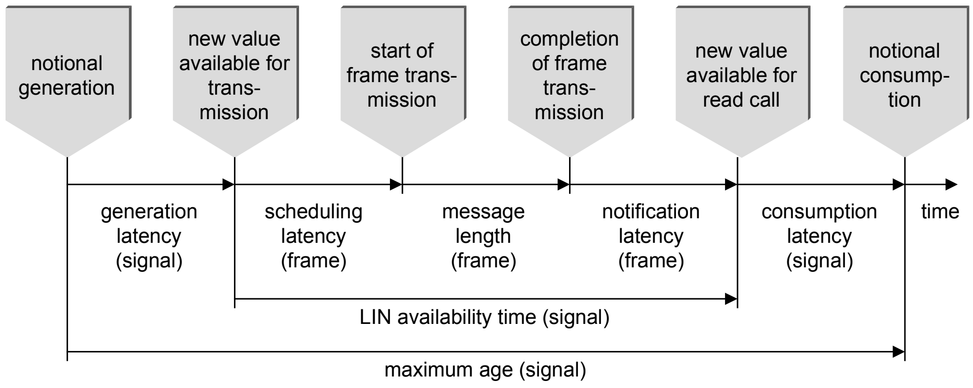
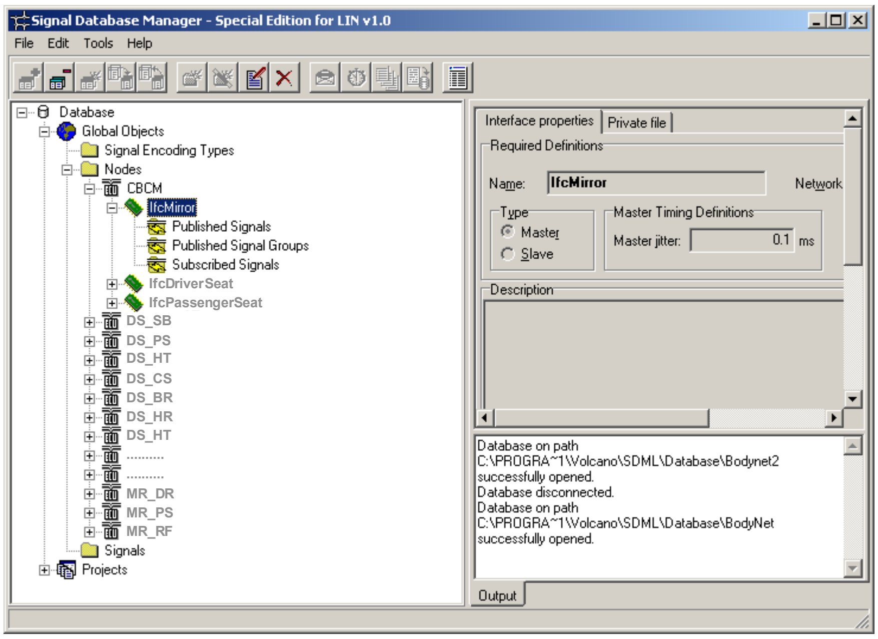

notional generation
new value available for trans-mission
start of frame trans-mission
completion of frame trans-mission
new value available for read call
notional consump-tion
generation latency (signal)
scheduling latency (frame)
message length (frame)
notification latency (frame)
consumption latency (signal)
time
LIN availability time (signal)
maximum age (signal)

Figure 7: The LIN timing model and definition of latencies

3.2.2 Nodes and Interfaces

The SDML interface consists of three windows that represent the tree structure of the objects, the list of object details, and an information window. With this interface, the network nodes are added or edited in a convenient form of ‘click-and-type’. If a node contains a master interface then a master time base (see Section 3.2) is assigned.

The screen shot in Figure 8 shows the list of nodes as defined for the case study in Section 2.1. The example includes the sub-tree of the interface ‘IfcMirror’, which has a 0.1 ms master jitter. For every node one or more interfaces have to defined. In the case that an interface is the master node then the previously defined master jitter should be entered.



Name: IfcMirror

Type: [✓ Master] Slave

Master jitter: 0.1 ms

```
Database on path
C:\PROGRA~1\Volcano\SDML\Database\Bodynet2
successfully opened.
Database disconnected.
Database on path
C:\PROGRA~1\Volcano\SDML\Database\BodyNet
successfully opened.
```

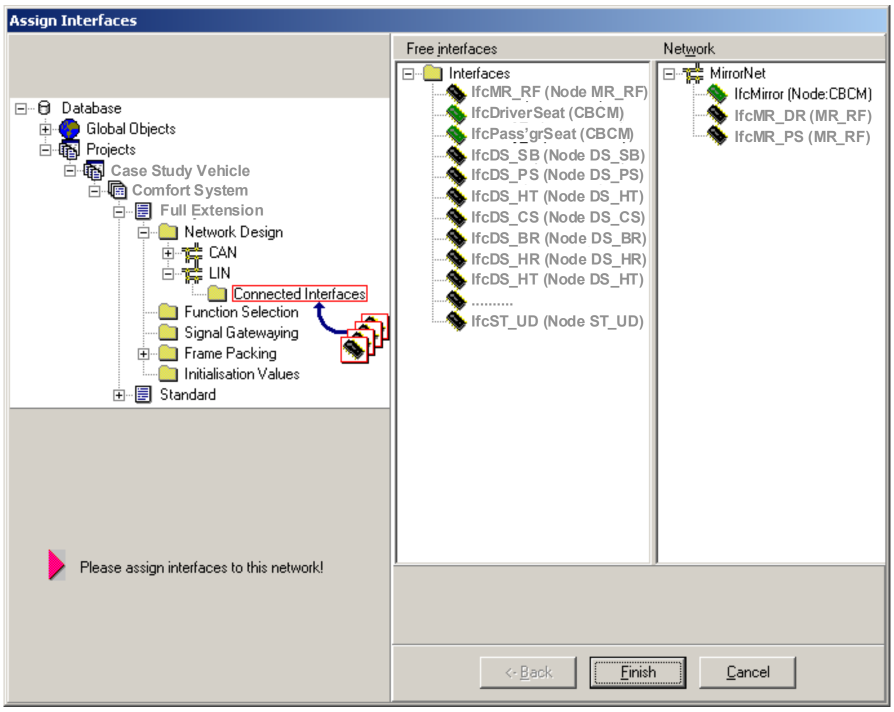
Figure 8: SDML interface with the list of nodes and mirror sub-tree used in the case study

Signals can be mapped to an interface as published signals with the generation latency or as subscribed signals with the specific consumption latency and the maximum age. In the particular case of published signals that are related to each other in a time critical way and that need to be mapped into one single frame, a 'published signal group' can be defined.

3.3 Definition of Network Topology and Generation of Message Schedules

3.3.1 Network and Gateways

After all global objects have defined the network itself is defined under a project. The case study uses four networks all using the *CBCM* module as the master node. The LIN networks will *DSeatNet*, *PSeatNet*, *SteeringNet*, and *MirrorNet* are defined together with their network the speed in kbits/sec. The interfaces are then added to the networks using a 'drag-and-drop' technique. The screen-shot in Figure 9 shows how the *MirrorNet* is created by dragging the master interface *IfcMirror* and the slave interfaces *IfcMR_DR* and *IfcMR_PS* to the right panel of the Assign Interfaces window.



Please assign interfaces to this network!

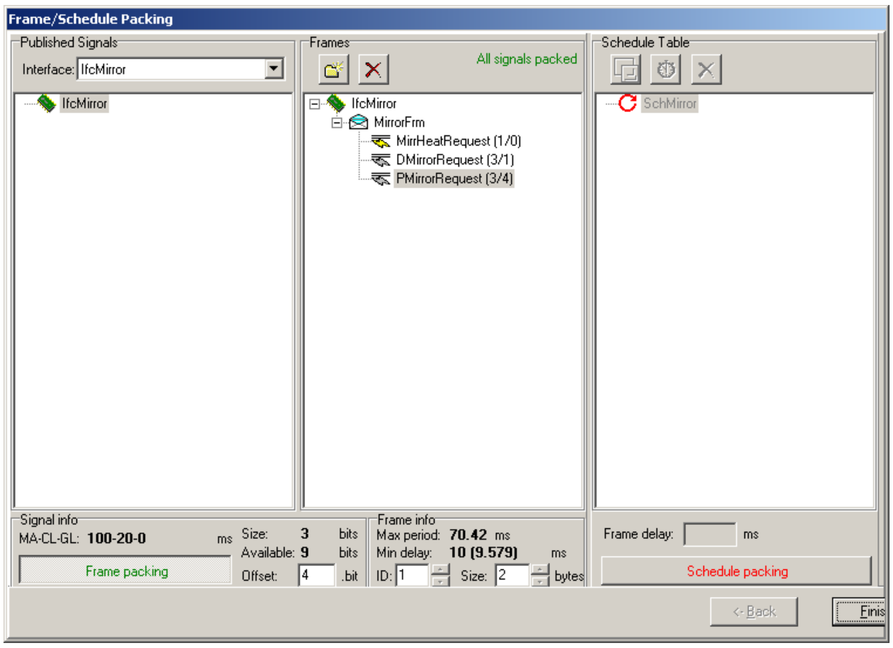
Figure 9: The network definition with the SDML interface

The SDML will note if a signal is published on one network and subscribed on another. The network designer defines such a signal as 'gatewayed' and assigns to it the gateway node, the source network with interface, and the target network with interface. For every gatewayed signal the maximum age needs to be distributed over the transporting networks.

3.3.2 Frame Packing and Schedule Table, and Timing Analysis

The *Frame/Schedule Packer* tool is used to pack signals into frames and to create schedule tables for each network. The screen-shot in Figure 10 shows the example for the definition of the *MirrorFrm* frame *SchMirror* schedule table. The left and the middle panel are used for frame packing, and the middle and right panel for schedule table generation.

The signals of the actual interface shown in the left panel are 'drag-and-dropped' into the selected frame in the middle panel. Each signal is shown with information about the signal size and offset in the frame. The size of a frame is selectable between 2, 4, or 8 data bytes, depending on the number and size of the included signals. The frame ID is also assigned during this step. After the signals are packed into frames, the schedule tables for these frames are automatically generated with delays that optimize the network's bandwidth usage.



Interface: IfcMirror

IfcMirror

- IfcMirror
  - MirrorFrm
    - MirrHeatRequest (1/0)
    - DMirrorRequest (3/1)
    - PMirrorRequest (3/4)

SchMirror

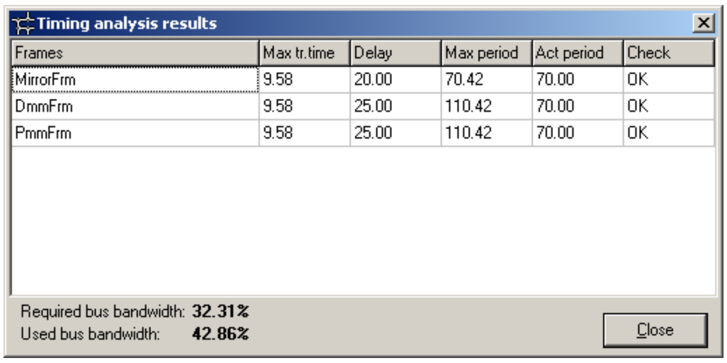
Signal info
MA-CL-GL: **100-20-0** ms
Size: 3 bits
Available: 9 bits
Offset: 4 .bit

Frame info
Max period: **70.42** ms
Min delay: **10 (9.579)** ms
ID: 1 Size: 2 bytes

Frame delay: ms

Figure 10: Packing of signals into *MirrorFrm* frame and definition of schedule table *SchMirror*

The timing analysis will determine if the system is schedulable or not. If it is not, appropriate actions like changing one or more maximum age settings should be taken until the system is proved to be schedulable. The timing results and the bandwidth usage for the mirror case study are shown in Figure 11.



Frames	Max tr.time	Delay	Max period	Act period	Check
MirrorFrm	9.58	20.00	70.42	70.00	OK
DmmFrm	9.58	25.00	110.42	70.00	OK
PmmFrm	9.58	25.00	110.42	70.00	OK

Required bus bandwidth: **32.31%**
Used bus bandwidth: **42.86%**

Figure 11: Timing analysis of the schedule table for the *IfcMirror* frames

3.4 LIN Configuration File and Language

After the creation and verification of the network schedules, a network configuration file is automatically generated. The LIN Configuration Language defines the syntax for the objects and topology of a network (see Section 3.3). The specification of a common description format for LIN networks enables the integration of tools from different vendors into a homogeneous development flow. The format of the configuration language is illustrated by Listing 1.

```
LIN_description_file;   // example: mirror network
LIN_protocol_version =  "1.1";
LIN_language_version =  "1.1";
LIN_speed =             9.6 kbps;

Nodes     { Master:     CBCM, time base 5 ms, jitter 0.1ms;
            Slave:      MR_DR, MR_PS, MR_RF;}

Frames    { frame_name, frame_id, published_by {
                 {signal_name, signal offset } }

Signals   { signal_name : signal_size, init_value,
            published_by, subscribed_by }

Schedule_tables  { schedule_table_name {
                   frame_name delay frame_time }

Signal_groups {}               // optional
Signal_encoding_types {}       //optional
Signal_representation {}       //optional
```

Listing 1: Format of the LIN configuration language for network description

4. Software Configuration Manger and API

The LIN application programmer's interface (API) is a network software layer that hides the details of a LIN network configuration (e.g. how signals are mapped into frames) to the programmer of the application program in an ECU. The programmer uses the API to write and read signals from other nodes via the network, without caring about the details of the data transport.

The configuration manager (LCFG) converts the LIN configuration files (one per node interface) and one node private file into the ANSI-C compliant source and header files ‘`l_gen.c`’ and ‘`l_gen.h`’ (see Figure 12). The private file contains typically flags attached to signals, local renaming, or interface descriptions. The generated source and header files are compiled together with the application source file. The resulting object code is then linked with provided LIN library functions into the final target image. The library contains re-locatable object modules and is generated for a specific target hardware and a certain type of linker.

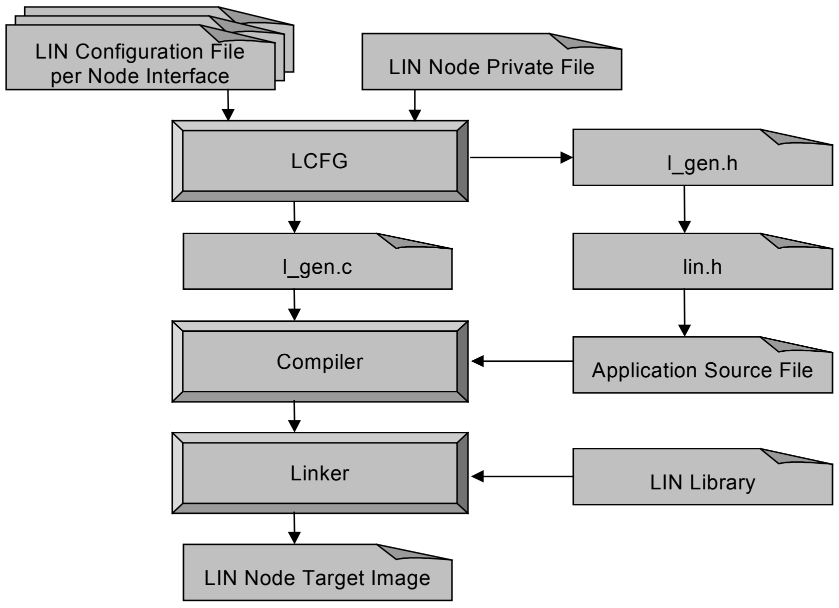


Figure 12: Data flow of the automated code generation with the configuration manager

The LIN API specification defines a set of functions that are used for system initialization, read and write calls for signals and flags, scheduler calls, initialization and connection of node interfaces, and interrupt management for the controller. An example for the most frequently called API - the signal read/write calls – is shown in Listing 2.

```
void intDrMirrorPosition(void) {     // position driver mirror
    uint8 u8DrMirrorPos;
    if (l_flg_tst_FlgDrMirrorReq()) {
        l_flg_clr_FlgDrMirrorReq();
        if (l_bool_rd_DrMirrorReq() &&
            l_bool_rd_DrMirrorSwitchStatus()) {
            u8DrMirrorPos = l_u8_rd_DrMirrorPosition();
        } else {
            u8DrMirrorPos = 0;
        }
        l_u8_wr_DrMirrorPos(u8DrMirrorPos);
    }
}  // end DrMirrorPosition
```

Listing 2: Usage of signal read/write calls and flags in a LIN application software

In this example, the specified LIN API calls are printed bold. The calls '`l_flg_tst`' and '`l_flg_tst`' return and set a C boolean variable that represents a flag status. The call '`l_bool_rd`' reads a one-bit signal, while the calls '`l_u8_rd`' and '`l_u8_wr`' read and write a signal of 1...8 bit, respectively.

5. LIN Protocol

5.1 Features and Objectives

LIN is a single-wire serial communications protocol based on the common SCI (UART) byte-word interface. UART interfaces as available as low cost silicon module on almost all micro-controller and can also be implemented as equivalent in software or pure state machine for ASICs. The medium access in a LIN network is controlled by a master node so that no arbitration or collision management in the slave nodes is required, thus giving a guarantee of the worst-case latency times for signal transmission.

A particular feature of LIN is the synchronization mechanism that allows the clock recovery by slave nodes without quartz or ceramics resonator (see Section 5.6). The specification of the line driver and receiver is following the ISO 9141 single-wire standard [3] with some enhancements. The maximum transmission speed is 20 kbit/s, resulting from the requirements by electromagnetic compatibility (EMC) and clock synchronization.

A node in LIN networks does not make use of any information about the system configuration, except for the denomination of the master node. Nodes can be added to the LIN network without requiring hardware or software changes in other slave nodes. The size of a LIN network is typically under 12 nodes (though not restricted to this), resulting from the small number of 64 identifier and the relatively low transmission speed.

The clock synchronization, the simplicity of UART communication, and the single-wire medium are the major factors for the cost efficiency of LIN.

5.2 Communication Concept

A LIN network comprises one master node and one or more slave nodes. All nodes include a slave communication task that is split in a transmit and a receive task, while the master node includes an additional master transmit task. The communication in an active LIN network is always initiated by the master task as illustrated in Figure 13: the master sends out a message header which comprises the synchronization break, the synchronization byte, and the message identifier.

Exactly one slave task is activated upon reception and filtering of the identifier and starts the transmission of the message response. The response comprises two, four, or eight data bytes and one checksum byte. The header and the response part form one message frame (see Section 5.4).

The identifier of a message denotes the content of a message but not the destination. This communication concept enables the exchange of data in various ways: from the master node (using its slave task) to one or more slave nodes, and from one slave node two the master node and/or other slave nodes. It is possible to communicate signals directly from slave to slave without the need for routing through the master node, or broadcasting messages from the master to all nodes in a network. The sequence of message frames is controlled by the master and may form cycles including branches.

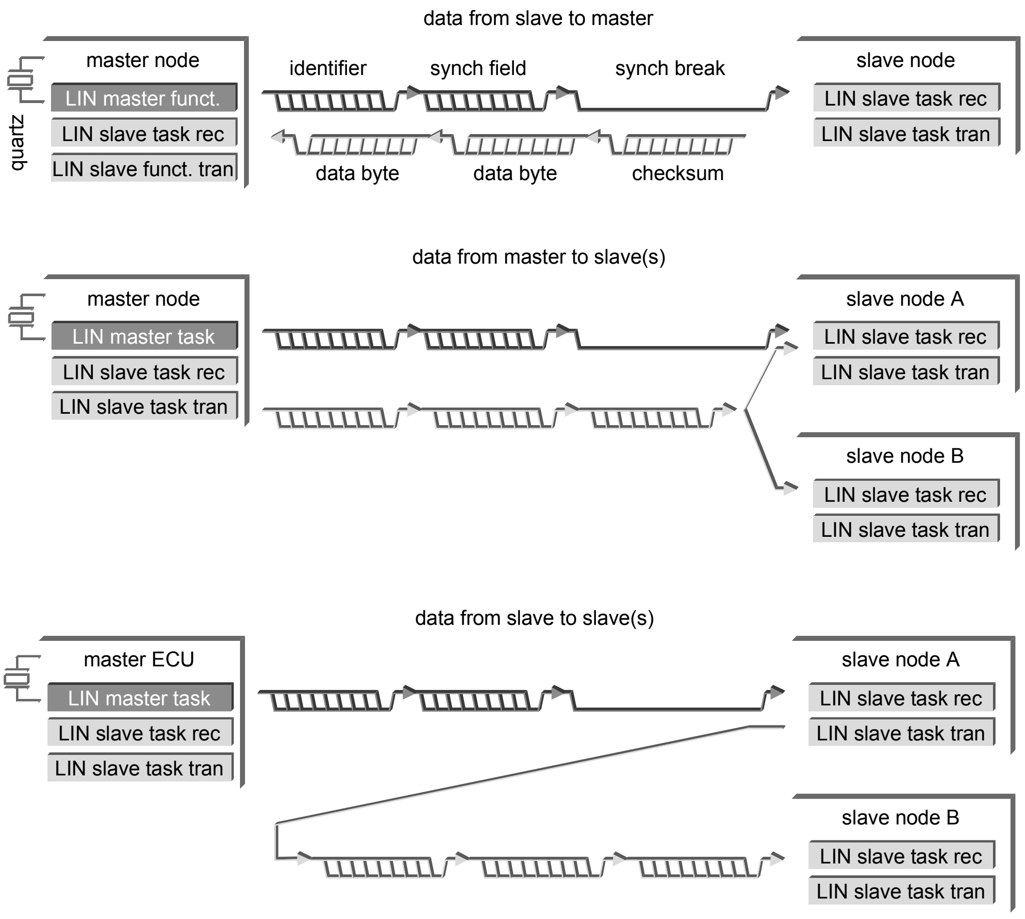


Figure 13: Various forms of data communication with LIN

5.3 Error Detection, Fault Confinement, and Data Protection

The actions that master and slave tasks undertake upon a corrupted communication (fault confinement) depend almost on the system requirements and have to be specified in the application layer. The LIN protocol defines only basic errors such as bit error (transmitted signal is different from monitored signal), checksum error, non-responding slave, and no bus activity. The fault confinement relies mainly on the master node that shall handle as much as possible of error detection and error recovery such as for example the re-scheduling a message.

An acknowledgment procedure for a correctly received message as known in CAN is not defined by the LIN protocol. Errors can not directly signaled by slaves but must be polled by the master. Local communication errors at the transmitter can be observed by comparing the outgoing message stream with the monitored message stream. If a slave node has detected an inconsistency it saves this as diagnostics information and provides it on request to the master node. Checksum errors can detected by all network nodes (global error).

The identifier and the data fields in a LIN message are error protected by parity and checksum information, respectively (see Section 5.4). The particular protection of the identifier is necessary as message header and response may origin from different sources. Thus, the identifier can not be protected by the checksum in the message response.

5.4 Message Frame

The LIN communication is based on message frames in a fixed format of selectable length. A message frame is built upon 8-bit characters (byte field) with 8N1-coding, known as SCI or UART serial data format. Every byte field has a length of ten bits, beginning with a dominant start bit, followed by eight data bits with the least significant bit (LSB) being sent first, and ending with a recessive stop bit (see Figure 14).

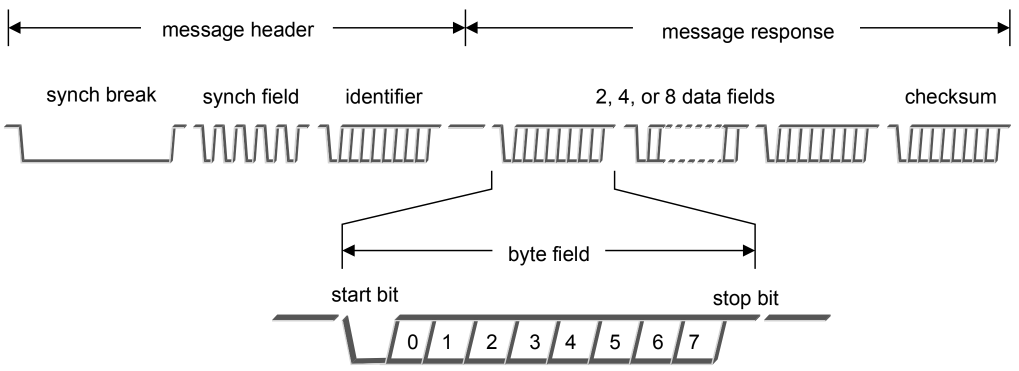


Figure 14: The LIN message frame is based on an 8N1 byte field.

A message frame is composed from a header being sent by the master task and a response being sent by a slave task. The message header again is formed by a synchronization break, a synchronization field, and the identifier field, while the message response is formed by two, four, or eight data fields and one checksum field. All fields can be transmitted without spacing, giving the minimum length of a message as $T_{msg,min} = (44 + 10*N_{field}) * T_{bit}$. For the simplicity of implementation, the LIN specification does not define the maximum spacing between single fields but defines a time budget for the transmission of a message. This maximum length is specified as $T_{msg,max} = 140\% * T_{msg,min}$.

5.4.1 Message Header

The synchronization break is the only exception from the 8N1 coded byte stream and is transmitted only by the master node. This particular pattern is longer than any regular sequence of dominant bits and can unambiguously be identified as start of a message (see Figure 15). The synchronization break provides a regular opportunity for slave nodes to synchronize to the bus clock, or to enter the LIN communication at any point in time, for example after a controller reset.

The synchronization break must be at least 13 bit long to ensure proper synchronization of salve node (see Section 5.6) but can be longer. UART modules on microcontroller that can generate only generate multiples of 10-bit breaks will typically transmit 20 dominant bit. This increases the minimum message length by 6% to 11%, depending on the number of data bytes.

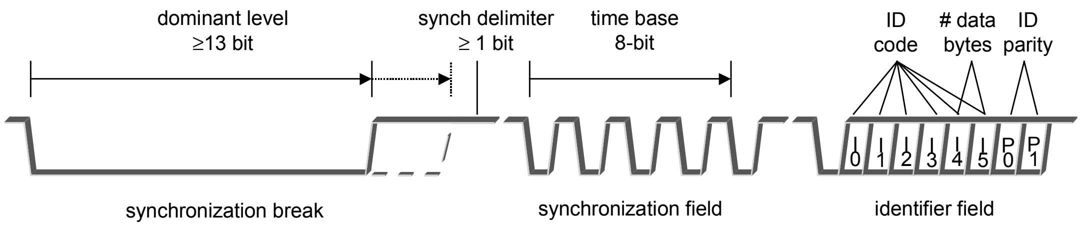


Figure 15: Synchronization break, synchronization field, and identifier field

The synchronization field is a bit pattern with the hexadecimal equivalent of 0x55. A slave node with in-accurate time base (e.g. on-chip oscillator without quartz stabilization) can evaluate this pattern to retrieve the communication clock. One synchronization method is the measurement of the period between first and last falling (i.e. recessive to dominant) edge and dividing the result by 8, giving the master bit time T_{bit}. Depending on the hardware and software resources in a slave node, other approaches might be more appropriate. In any case the synchronization should be based only on the falling signal edges as those are actively driven by the line driver.

The identifier field denotes the content and length of a message. The field comprises six identifier bits and two parity check bits (see Figure 15), forming a set of 64 identifiers. The parity bits are calculated by a mixed-parity algorithm as

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \text{ (even) and } P1 = \overline{ID1 \oplus ID3 \oplus ID4 \oplus ID5} \text{ (odd) parity.}$$

The bits ID4 and ID5 define the number of data fields in the message response. The coding splits the set of 64 identifiers in three sub-sets of 32 identifiers for 2-byte data response, 16 identifier for 4-byte data response, and 16 identifier for 8-byte data response.

5.4.2 Message Response

The message response is composed of 2, 4, or 8 data fields (depending on the length code in the identifier field) and one checksum field. The checksum is the inverted modulo-256 sum over all data bytes. The sum is calculated by an 'add with carry' operation with the carry bit of every addition being added to the LSB of the resulting sum. The addition with revolving carry improves the protection against MSB failures. The sum of modulo-256 sum over all data bytes and the checksum byte is '0xFF'. Nodes for which message is irrelevant – e.g. in case of an unknown identifier - may skip the checksum calculation.

5.5 Special Purpose Messages

Four identifier from the set with 8-byte response are reserved for particular message frames: two command frames and two extended frames. The two command frames include an 8-byte response and are used for data up- and downloads from the master to slave nodes and vice versa. This feature is used for software updates, network configuration, and diagnostic purposes. The frame structure is identical with a regular message. The response fields contain user-defined command fields instead of data fields that put the slave nodes for example in a service mode or in the sleep mode, as illustrated in Figure 16.

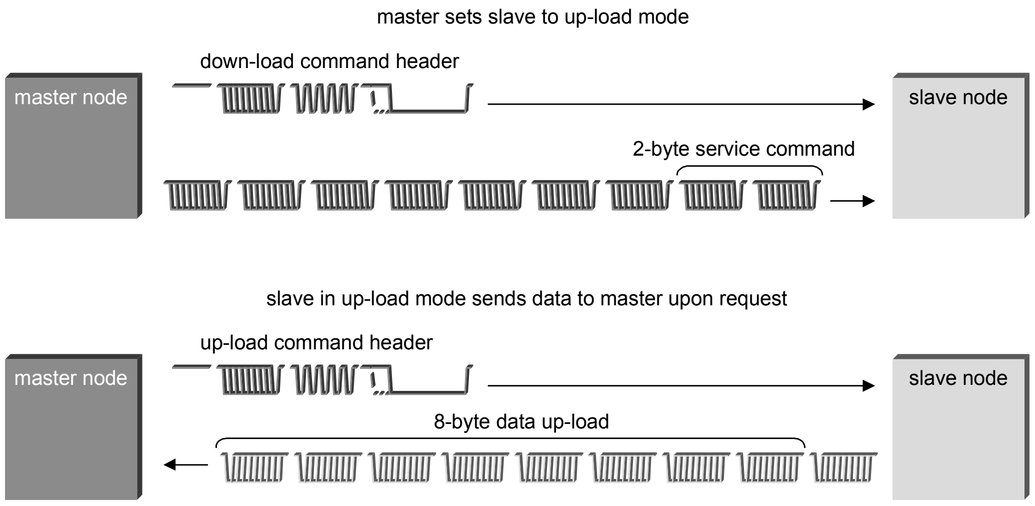


Figure 16: Illustration of software up- and downloads in service mode

The extended frame identifiers are defined to ensure upward compatibility of LIN slaves with future revisions of the LIN protocol if required so. These identifiers will be used to embed future extended formats of the LIN protocol into the existing specification. The extended frame identifier announce an unspecified frame format to all bus participants. The identifier can be followed by an arbitrary number of LIN byte fields. A slave receiving such an identifier must ignore all subsequent byte fields until the next synchronization.

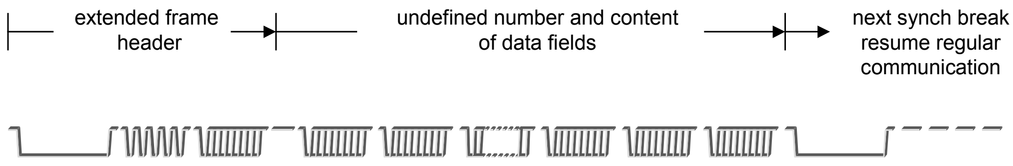

extended frame header
undefined number and content of data fields
next synch break resume regular communication

Figure 17: The extended frame ensures upward compatibility to future revisions of LIN

5.6 Clock Synchronization

The LIN protocol provides a dedicated synchronization pattern with the start of every message frame that allows slave nodes without quartz or ceramics resonator to synchronize their local time base to that of the master. Two levels of synchronization are specified for slave nodes in LIN:

'unsynchronized':	slave clock $F_{unsynch}$ differs less than ±15% from master clock F_{master}
'synchronized':	slave clock F_{synch} differs less than ±2% from master clock

A slave node with stabilized oscillator is always considered as synchronized, while a slave node without stabilized clock is assumed to be unsynchronized after the end of a message, after a reset, or after an exit from sleep mode. On-chip RC-oscillators can achieve a permanent clock tolerance of better than ± 15% with simple pre-calibration. After synchronization, the on-chip oscillator must be stable with $F_{unsynch}$ for the rest of the message, taking into account the impacts of temperature and voltage drift.

The actual capture range for synchronization is better than this specification value as will be shown by the following calculations. The synchronization is performed in two steps: firstly the detection of the synchronization break, and secondly the tuning of the local time base as described in Section 5.4.1. Two requirements have to be considered for the detection of the synchronization break, as illustrated in Figure 18:

(a) An unsynchronized slave sampling the bus fast with $F^{+}_{unsynch} = F_{master} + 15\%$ must not misconceive a regular '0x00' data as break.

(b) The break signal must be long enough to be identified by an unsynchronized slave sampling the bus slowly with $F^{-}_{unsynch} = F_{master} - 15\%$.

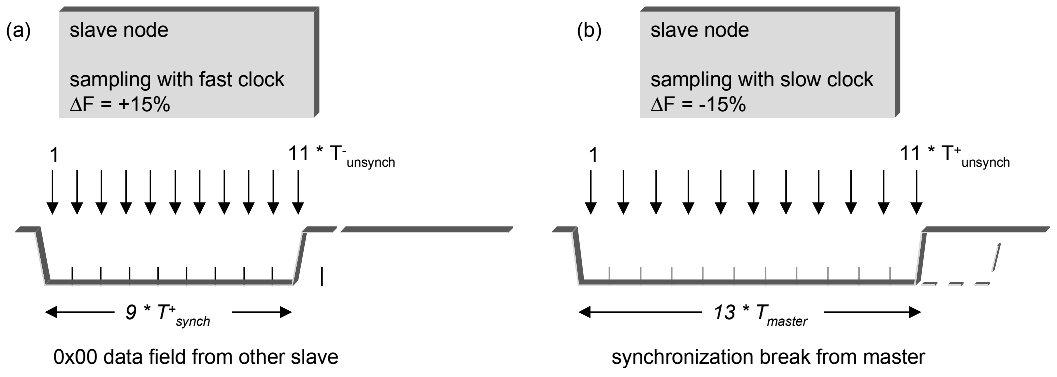


Figure 18: Illustration of the requirements (a) and (b) for the break detection

Requirement (a) leads to the specification of the break detection threshold T_{SBRKTS} for slaves. With the maximum length $T^{\max}_{0x00}$ of a regular 0x00 data pattern on the bus being

$$T^{\max}_{0x00} = 9 \cdot T^{+}_{unsynch} = \frac{9}{F^{-}_{synch}} = \frac{9}{F_{master} \cdot (1-2\%)} = 9.18 \cdot T_{master}, \tag{1}$$

where the term '(1-2%)' reflects that the 0x00 can be sent by a slave with -2% deviation. With the LIN specification of $T_{SBRKTS} = 11$ slave bit times, a fast slave can not misconceive a 0x00

$$11 \cdot T^{\min}_{unsynch} > T^{\max}_{0x00} = 9.18 \cdot T_{master}. \tag{2}$$

Rewriting (2) with $$T^{-}_{unsynch} = \frac{1}{F^{+}_{unsynch}} = \frac{1}{(1+\Delta F^{+}) \cdot F_{master}} = \frac{T_{master}}{(1+\Delta F^{+})} \tag{3}$$

results in $$11 \cdot \frac{T_{master}}{1+\Delta F^{+}} > 9.18 \cdot T_{master} \text{ or } \Delta F^{+} < \frac{11}{9} \cdot (1-2\%) - 1, \tag{4}$$

respectively. Solving Equation (4) results in $\Delta F^{+} < +19.7\%$. This is the maximum frequency deviation under which a slave will synchronize.

Requirement (b) leads to the minimum length of the synchronization break, which is specified in LIN as $T_{SYNBRK} = 13$ master bit times. A slow slave with will detect the break if

$$11 \cdot T^{+}_{unsynch} < 13 \cdot T_{master}. \tag{5}$$

Rewriting (5) with $$T^{+}_{unsynch} = \frac{1}{F^{-}_{unsynch}} = \frac{1}{\left(1 - \Delta F^{-}\right) \cdot F_{master}} = \frac{T_{master}}{\left(1 - \Delta F^{-}\right)} \tag{6}$$

results in $$11 \cdot \frac{T_{master}}{\left(1 - \Delta F^{-}\right)} < 13 \cdot T_{master} \text{ or } \Delta F^{-} < 1 - \frac{11}{13}, \tag{7}$$

respectively. Equation (7) results in $\Delta F^{-} < 15.3\%$. This is the minimum frequency deviation under which a slave will synchronize. The calculation shows that the average capturing range $\Delta F = \left(\Delta F^{+} + \Delta F^{-}\right) / 2 = \pm 17.5\%$ is wider than the specification, thus providing a safety margin.

6. LIN Physical Layer

The bus is a single line, wired AND bus being supplied via a termination resistor from the positive battery node (VBAT) as illustrated in Figure 19. The bus line driver/receiver ('transceiver') is an enhanced implementation of the ISO 9141 standard [3]. The bus can take two complementary logical values: the dominant value with an electrical voltage close to ground and representing a logical '0', and the recessive value with an electrical voltage close to the battery supply and representing a logical '1'.

The bus a terminated with a pull-up resistance of 1kΩ in the master node and 30kΩ in a slave node. A diode in series with the resistor is required to prevent the ECU from being powered by the bus in case of a local loss of battery.

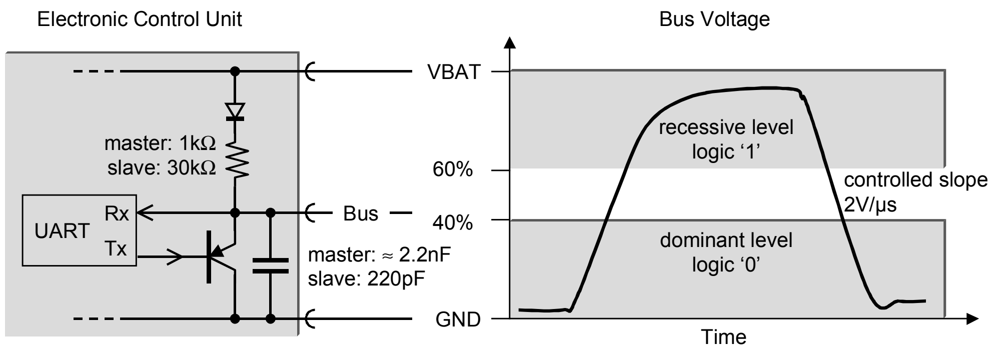


Figure 19: LIN physical layer and bus voltage levels

It has importantly to be noted that the specification of voltage levels in LIN refers to the interface of the ECU with wire harness and not to ECU-internal voltages. This must be considered when designing for example a LIN transceiver IC: due reverse polarity protection and supply buffer capacitors in the ECU, the local supply or reference voltage at the transceiver may be different from the external LIN connector voltages. Ideally, the design of the electronic component has to consider and compensate these differences.

The termination capacitance is typically $C_{slave} = 220pF$ in the slave nodes, while the capacitance of the master node is higher in order to make the total line capacitance less depended from the number of slave nodes. The termination capacitance for the master is calculated by

$$C_{master} = C_{bus} - n_{slave} \cdot C_{slave} - \overline{C}_{line} \cdot l_{line}, \tag{8}$$

where a typical target for total bus capacitance would be $C_{bus} = 9nF$ and the line capacitance $\overline{C}_{line} = 100...150 pF/m$.

The maximum baud rate is 20kbit/s due to the single wire medium. This value is a practical compromise between the opposing requirements of synchronization for high signal slew rates, and of slower slew rates for electromagnetic compatibility. The minimum baud rate is 1kbit/s to avoid conflicts with the practical implementation of time-out periods. The significant electrical parameter of the LIN physical interface are listed in Table 3.

parameter	**typical value(s)**
communication speed	2400, 9600, 19200 kbps
voltage level	13.5 V
signal slew rate	2 V/µs
termination resistor master / slave	1 kΩ / 30 kΩ
termination capacitance master /slave	220 pF / 2.2 nF
line capacitance	100 … 150 pF/m

Table 3: Major parameter of the LIN physical layer

7. References

[1] LIN Consortium, “LIN Specification, Version 1.1”, *www.lin-subbus.org*, March 2000

[2] W. Specks, A. Rajnák, “The Scaleable Network Architecture of the Volvo S80”, *8th Intl. Conference on Electronic Systems for Vehicles*, Baden-Bade, Oct 1998, pp. 597-641

[3] “Road Vehicles – Diagnostics Systems – Requirement for Interchange of Digital Information”, *International Standard ISO9141*, 1st Edition, 1989