UM1593
User manual

STSW-STM8A-LIN software package

Introduction

Purpose

This user manual provides application programmers with detailed information on how to use STMicroelectronics LIN driver (STSW-STM8A-LIN) for master and slave nodes. It gives a detailed description of the API implemented together with some examples of important files required for getting started and configuring the driver. It then explains how to configure the LIN driver to operate with STM8 microcontrollers.

Scope

STMicroelectronics implementation is compliant with the LIN API specification.
The LIN 2.1 software package supports all LIN standard versions 1.3, 2.0 and 2.1.

User profile

The users should be familiar with the concept of networks and in particular LIN networks. As STMicroelectronics LIN driver (STSW-STM8A-LIN) is written in C programming language, they should be experienced in the development of C applications.

References

- LIN specification package, revision 2.0, 23-September-2003
- LIN specification package, revision 2.1, 24-November-2006

STMicroelectronics STSW-STM8A-LIN software driver is available on the company website at *www.st.com*.

Contents

List of tables

1 Abbreviations

Table 1. Description of abbreviated forms

Abbreviation	Description
API	Application Programming Interface
CAN	Controller Area Network
LDF	LIN description
LIN	Local Interconnect Network
RSID	Response Service Identifier
SID	Service Identifier

2 Overview

2.1 LIN concept

LIN (local interconnect network) is a concept that has been developed by a group of well-known car manufacturers to produce low-cost automotive networks that complement existing networks such as CAN. It is based on single-wire serial communications using SCI (UART) interfaces that are commonly available on microcontrollers. LIN is intended to be used together with CAN to build a hierarchical vehicle network. LIN is usually used for local subsystems where a low bit rate (up to 20 kbit/s) is acceptable and no safety-critical functions are required. These applications are typically used for car body electronics such as door and seat control, air conditioning, etc. These subunits are connected to a CAN network through a LIN/CAN gateway.

A LIN cluster comprises one master node and one or several slave nodes. LIN has a special feature that performs the synchronization of slave nodes via the bus thus allowing to implement low-cost nodes without the need of quartz clocking. In addition, since accesses to the bus are controlled by the master node, no collision management is required between the slave nodes. This also means that a worst-case transmission time for signals can be guaranteed.

The slave nodes do not need any information about the LIN cluster. As a result, additional slave nodes can be added to the LIN without modifying existing slave nodes. The master node requests information from all slaves and must be re-built if new nodes are added.

The LIN standard includes the specification of the transmission protocol, the transmission medium, the system definition language and the interface for software programming.

2.2 LIN communications

Communications between the application software embedded on LIN nodes are performed by exchanging signals. The LIN driver software is responsible for managing the exchange of signals at low level. The information are transferred between the nodes by using frames.

The driver is consequently responsible for processing application signals, packing them into the frame data section, and initiating the transfers. The frames are then transferred via the microcontroller serial interface. Using this communication technique, signal reading and writing operations are performed asynchronously of the frame transfers. An overview of communications between LIN nodes is described in *Figure 1*.

All transfers are initiated by the master node. Slave nodes only transmit data when requested to do so. The message frame consists of a header and a frame body. The master node sends the frame message header. The frame body can be sent either by the master or by a slave node. Since the publisher for any given frame is configured before building the application, only one node will send the frame body.

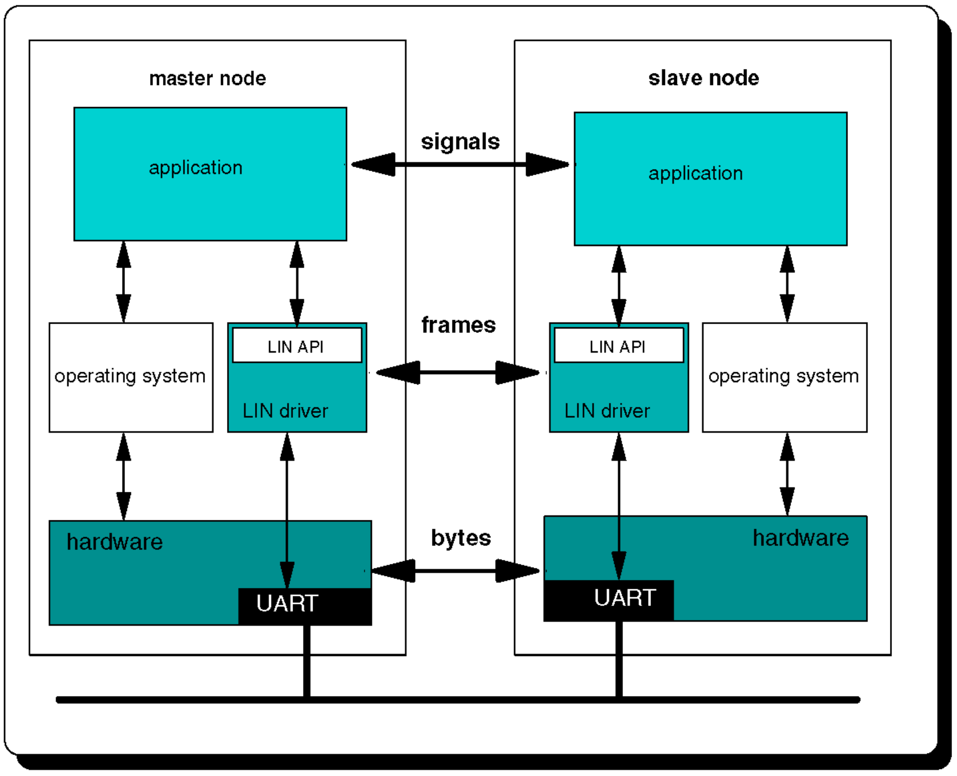
The frame message identifier denotes the message content and not the destination. This communication concept means that data are exchanged between nodes as follows:

- From the master node to one or more slave nodes,
- From a slave node to the master and/or to other slave nodes.

As a result, communications are possible from slave to slave without routing through the master and the master can broadcast to all slaves belonging to the LIN subsystem.

The order in which frames are sent is determined by a schedule table used by the master. Several tables may be configured but only one table can be active at a time. Switching between tables can be carried out either by the application or internally by the driver. The schedule tables required by an application must be configured by the user in the LIN description file.

Figure 1. Master-slave node communications



2.3 Signal management

Signals are transferred in the frame data bytes. Several signals can be packed into a frame as long as they do not overlap each other or extend beyond the data area of the frame.

Each signal has only one producer. This means that it is always written by the same node in a given cluster. Each signal that is issued by a node must be configured by the user.

A signal is described either by a scalar value or by a byte array:

- Scalar signal ranging between 1 and 16 bits. A 1 bit signal is called a boolean signal. 2-bit to 16-bit scalar signals are treated as unsigned values.
- A byte array ranging from 1 to 8 bytes.

Signal self-consistency is ensured by the driver (a partially updated 16-bit signal must never be passed to an application), while the consistency between signals is managed by the application.

The signal LSB is transmitted first. Scalar signals may cross a byte boundary at most once. The driver maps each byte in the byte array to a byte in the frame.

2.4 Using the driver

The driver must be configured and built before use. Details on the configuration steps are given in *Section 4: Driver configuration*, and architecture specific details are provided in *Section 7: Architecture notes*.

The STMicroelectronics LIN driver includes the diagnostic layer as described in LIN 2.1 specification. The diagnostic API is divided between a RAW API and a COOKED API. The RAW API allows a diagnostic application to control the contents of every frame sent while the COOKED API provides a full transport layer. The user can choose to include or not the diagnostic functions when building the driver (see *Section 4.4.3: Diagnostic functions configuration*).

Before using the driver functionalities, the driver itself must be initialized by calling the `l_sys_init()` API function. Before using any interface-related function the microcontroller interfaces must be initialized using the `l_ifc_init()` API function, and then connected using `l_ifc_connect()`.

Note: `l_ifc_connect()` and `l_ifc_disconnect()` functions are obsolete for LIN 2.1 protocol, and will not be compiled. These functions will be compiled only for LIN 2.0 or LIN 1.3 protocols.

The following naming convention that has been adopted in STMicroelectronics driver:

Table 2. LIN naming conventions

item type	item name	LIN name
signal	sigName	LIN_SIGNAL_sigName
frame	frameName	LIN_FRAME_frameName
flag	flagName	LIN_FLAG_flagName
schedule table	tabName	LIN_TAB_tabName
node	nodeName	LIN_NODE_nodeName

The application must use the “LIN name” format except when calling the API static functions. For example, if a signal named `sigMstatus` has been configured in the LDF file, the application must use the form `LIN_SIGNAL_sigMstatus` for dynamic function calls:

```
my_sig = l_u8_rd(LIN_SIGNAL_sigMstatus);
```

or the form `sigMstatus` as used in the generation of static function names:

```
my_sig = l_u8_rd_sigMstatus();
```

If a master node is configured to use multiple interfaces, an optional tag can be specified by the user to avoid naming conflicts. This tag will prefix the “item name” given above. See *Section 4.3: Cluster configuration* for details and examples.

2.5 Driver version

The driver comprises several source and header files that are revisioned with a version number which is only updated on change. A given driver release consequently contains files with varied version numbers. The list of the file versions used to build a given driver release is contained in the file `lin_version_control.h` located at the root of the source directory. This information is used to ensure that only consistent files are included when building the driver.

3 API

3.1 Data

The following data types must be defined for the driver:

l_bool

l_u8

l_u16

l_u32

l_ioctl_op.

l_irqmask

l_ifc_handle

Since these types are hardware dependent, they are defined in the architecture specific file `lin_def_`*`archname`*`_gen.h` located in the architecture specific directory.

3.2 Functions

The numbering in the description sections below refers to the LIN API specification section where the corresponding function is described.

3.3 CORE API

3.3.1 Driver and cluster management

Table 3. System Initialization

l_sys_init(void)	
Prototype	`l_bool l_sys_init(void);`
Availability	Master and slave nodes
Include	lin.h
Description	This function performs the initialization of the LIN core (LIN API 7.2.1.1). The scope of the initialization is the physical node i.e. the complete node (see LIN 2.1 section 9.2.3.3). The call to `l_sys_init` is the first call the user application must issue in the LIN core before calling any other API functions.
Parameters	None
Return value	zero if initialization succeeded non-zero if initialization failed

3.3.2 Signal interaction

Table 4. Scalar signal read

l_bool_rd, l_u8_rd, l_u16_rd	

Prototype (dynamic)	`l_bool l_bool_rd (l_signal_handle signalId);`
	`l_u8 l_u8_rd (l_signal_handle signalId);`
	`l_u16 l_u16_rd (l_signal_handle signalId);`

Availability	Master and slave nodes
Include	lin.h
Description	These functions read and return the current value of the specified signal (see LIN 2.1 section 7.2.2.2)
Parameters	`signalId`: name of the signal to be read e.g. for the configured signal `status` then `LIN_SIGNAL_status`
Return value	l_bool: boolean signal value or 0 if `signalId` is invalid l_u8: 8 bit signal value or 0 if `signalId` is invalid l_u16: 16 bit value or 0 if `signalId` is invalid

Prototype (static)	`l_bool l_bool_rd_sss (void);`
	`l_u8 l_u8_rd_sss (void);`
	`l_u16 l_u16_rd_sss (void);`
	where `sss` denotes the name of the signal that is to be read e.g. for the configured boolean signal `status` then the prototype:
	`l_bool l_bool_rd_status(void);`

Table 5. Scalar signal write

l_bool_wr, l_u8_wr, l_u16_wr	

Prototype (dynamic)	`void l_bool_wr (l_signal_handle signalId, l_bool val);`
	`void l_u8_wr (l_signal_handle signalId, l_u8 val);`
	`void l_u16_wr (l_signal_handle signalId, l_u16 val);`

Availability	Master and slave nodes
Include	lin.h
Description	These functions set the current value of the specified signal to `val` (see LIN 2.1 section 7.2.2.3)
Parameters	`signalId`: the signal to be set e.g. for the configured signal `status` then `LIN_SIGNAL_status` `val`: the value to which the signal is to be set
Return value	None

Prototype (static)	`void l_bool_wr_sss (l_bool val);`
	`void l_u8_wr_sss (l_u8 val);`
	`void l_u16_wr_sss (l_u16 val);`
	where `sss` denotes the name of the signal which value is to be set to `val` e.g. for the configured boolean signal `status` then the prototype:
	`void l_bool_wr_status (l_bool val);`

Table 6. Byte array read

l_bytes_rd	
Prototype (dynamic)	`void l_bytes_rd (l_signal_handle signalId, l_u8 start, l_u8 count, l_u8* const data);`
Availability	Master and slave nodes
Include	lin.h
Description	This function reads and returns the current value of the selected bytes in the specified signal (see LIN 2.1 section 7.2.2.4). The sum of start and count must never be greater than the length of the byte array.
Parameters	`signalId`: the signal to be read e.g. for the configured signal `user_data` then `LIN_SIGNAL_user_data` `start:` the first byte to be read `count`: the number of bytes to be read `data`: the area where the bytes will be written
Return value	None

Prototype (static)	`void l_bytes_rd_sss (l_u8 start, l_u8 count, l_u8* const data);`
	where `sss` denotes the name of the signal to be read e.g. for the configured signal `user_data` then the prototype:
	`void l_bytes_rd_user_data(l_u8 start, l_u8 count, l_u8* const data);`

Table 7. Byte array write

l_bool_wr, l_u8_wr, l_u16_wr	
Prototype (dynamic)	`void l_bytes_wr (l_signal_handle signalId, l_u8 start, l_u8 count, const l_8* const data);`
Availability	Master and slave nodes
Include	lin.h
Description	This function sets the current value of the selected bytes in the specified signal to the values given in `data` (see LIN 2.1 section 7.2.2.5). The sum of start and count must never be greater than the length of the byte array.
Parameters	`signalId`: the signal to be written e.g. for the configured signal `user_data` then `LIN_SIGNAL_user_data` `start`: the first byte to be written `count`: the number of bytes to be written `data`: the area where the bytes will be read from
Return value	None

Prototype (static)	`void l_bytes_wr_sss (l_u8 start, l_u8 count, const l_u8* const data);`
	where `sss` denotes the name of the signal to be written e.g. for the configured signal `user_data` then the prototype:
	`void l_bytes_wr_user_data (l_u8 start, l_u8 count, const l_u8* const data);`

3.3.3 Notification

Table 8. Test flag

l_flg_tst	
Prototype (dynamic)	`l_bool l_flg_tst (l_flag_handle flag);`
Availability	Master and slave nodes
Include	lin.h
Description	This function returns the state of the flag specified i.e. zero if cleared, non-zero otherwise. (see LIN 2.1 section 7.2.3.1)
Parameters	`flag`: the flag which state is to be returned e.g. for the configured flag `Txerror` then `LIN_FLAG_Txerror`
Return value	Zero if flag is clear, non-zero otherwise

Prototype (static)	`l_bool l_flg_tst_fff (void);`
	where `fff` denotes the name of the flag to be tested e.g. for the configured flag `Txerror` then the prototype:
	`l_bool l_flg_tst_Txerror (void);`

Table 9. Clear flag

l_flg_clr	
Prototype (dynamic)	`void l_flg_clr (l_flag_handle flag);`
Availability	Master and slave nodes
Include	lin.h
Description	This function sets the value of the specified flag to zero (see LIN 2.1 section 7.2.3.2)
Parameters	`flag`: the flag to be cleared e.g. for the configured flag `Txerror` then `LIN_FLAG_Txerror`
Return value	None

Prototype (static)	`l_bool l_flg_clr_fff (void);`
	where `fff` denotes the name of the flag to be cleared e.g. for the configured flag `Txerror` then the prototype:
	l_bool l_flg_clr_Txerror (void);

3.3.4 Interface Management

Table 10. Initialize interface

l_ifc_init	
Prototype (dynamic)	`l_bool l_ifc_init (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h

Table 10. Initialize interface (continued)

l_ifc_init	
Description	This function initializes the specified interface (for example the baud rate). The default schedule set will be `L_NULL_SCHEDULE` where no frames will be sent or received. The interface names are listed in lin_def.h. See *Section 4.3* and *Section 4.4* for details. This function must be called before using any other interface related to API functions (see LIN 2.1 section 7.2.5.1)
Parameters	`ifc`: the interface to be initialized
Return value	Zero if initialization was successful, non-zero otherwise.

Prototype (static)	`l_bool l_ifc_init_iii (void);`
	where `iii` denotes the interface to be initialized e.g. for the configured interface SCI0 then the prototype:
	`l_bool l_ifc_init_SCI0 (void);`

Table 11. Connect interface

l_ifc_connect	
Prototype (dynamic)	l`_ifc_connect()` function is obsolete for LIN 2.1 protocol and should not be used.

Table 12. Disconnect interface

l_ifc_disconnect	
Prototype (dynamic)	l`_ifc_disconnect()` function is obsolete for LIN 2.1 protocol and should not be used.

Table 13. Wakeup

l_ifc_wake_up	
Prototype (dynamic)	`void l_ifc_wake_up (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function issues a wakeup signal to the specified interface (see LIN 2.1 section 7.2.5.3). The wake-up signal (0xF0 byte, i.e. a dominant pulse lasting 250 µs to 5 ms depending on the configured bit rate) is transmitted directly when this function is called. It is the responsibility of the application to retransmit the wakeup signal according to the wakeup sequence (see LIN 2.1 section 2.6.2.).
Parameters	`ifc: interface handle`
Return value	None

Prototype (static)	`void l_ifc_wake_up_iii (void);`
	where `iii` denotes the interface to be woken up e.g. for the configured interface SCI0 then the prototype:
	`void l_ifc_wake_up_SCI0 (void);`

Table 14. Interface control

l_ifc_ioctl	
Availability	Master and slave nodes
Include	lin.h
Description	This function controls a functionality that is not covered by the other API calls. It is used to control protocol specific parameters or hardware specific functionalities, such as switching on/off the wakeup signal detection. The operations supported depend on the interface type. The parameter block `pParams` is optional. It is set to null if not needed. Otherwise it is interpreted as specified below (see LIN 2.1 section 7.2.5.4) This function is currently implemented to support the operations listed below.
Parameters	`ifc`: interface to which the operation is to be applied `operation`: the operation to be applied `pParams`: optional parameter block
Return value (master node)	`LIN_IOCTL_DRIVER_STATE`: This function returns a 16-bit values where the lower 8 bits represent the state of the driver, and the upper 8 bits either the protected identifier of the frame currently being transferred or 0xFF. The protected identifier is only returned if the state is neither LIN_STATE_IDLE nor LIN_STATE_BUSSLEEP. Note: The definition of driver states is currently located in the file *`lin_types.h`*.

Table 14. Interface control (continued)

l_ifc_ioctl	

Return value (slave node)	The return value depends on the operation requested:
	`LIN_IOCTL_DRIVER_STATE`:
	The function returns a 16-bit values where the lower 8 bits represent the state of the driver, and the upper 8 bits either the protected identifier of the frame currently being transferred or 0xFF. The protected identifier is returned if the state is either LIN_STATE_SEND_DATA or LIN_STATE_RECEIVE_DATA.
	Note: The definition of driver states is currently located in the file *lin_types.h*.
	
	`LIN_IOCTL_READ_FRAME_ID`
	The parameter referenced by *pParams must match the type l_frameMessageId_t defined in the file lin.h. The function sets the frame identifier pParams->frameId and the frame index pParams->frameIndex that matches the message ID pParams->messageId.
	The function returns 0 if it is successful, and 1 if the message was not found.
	
	`LIN_IOCTL_READ_MESSAGE_ID`
	The parameter referenced by *pParams must match the type l_frameMessage_t defined in the file lin.h. The function sets the message ID pParams->messageId and the frame index pParams->frameIndex that matches the message ID pParams->messageId. Returns 0 if successful or 1 if the message ID was not found.
	
	`LIN_IOCTL_READ_FRAME_ID_BY_INDEX`
	The parameter referenced by *pParams must match the type l_frameMessageId_t defined in the file lin.h. The function sets the frame ID pParams->frameId and the message ID pParams->messageId for the frame indexed by pParams->frameIndex. Returns 0 if successful or 1 if the index is invalid.
	
	`LIN_IOCTL_SET_FRAME_ID`
	The parameter referenced by *pParams must match the type l_frameMessageId_t defined in the file lin.h. The function sets the frame ID for the frame matching pParams->messageId to that given by pParams->frameId. Returns 0 if success otherwise 1.
	
	`LIN_IOCTL_FORCE_BUSSLEEP`
	Forces the driver into sleep mode.
	
	`LIN_IOCTL_SET_VARIANT_ID`
	Sets the Variant ID part of the Product ID in a slave node. The default Variant ID used for a slave node on startup is that which is given in the LDF.
	The parameter referenced by *pParams must be of type l_u8.
	
	`LIN_IOCTL_READ_VARIANT_ID`
	The function returns the current value of the Variant ID. The parameter given by pParams is not used and may be set to 0 in the function call.

Table 14. Interface control (continued)

l_ifc_ioctl	

Return value (slave node)	`LIN_IOCTL_READ_CONFIG_FLAGS`
	The function returns a 16-bit value indicating which configuration flags are set. These flags are set on successful completion of the corresponding diagnostic service. The flags are only cleared when read using this operation.
	Flags set are:
	`LIN_DIAG2_FLAGS_ASSIGN_FRAME_ID`
	`LIN_DIAG2_FLAGS_ASSIGN_NAD`
	`LIN_DIAG2_FLAGS_COND_CHANGE_NAD`
	`LIN_DIAG2_FLAGS_READ_BY_ID`
	`LIN_DIAG2_FLAGS_DATA_DUMP`
	
	`LIN_IOCTL_READ_NAD`
	The function returns a 16-bit value, the lower 8 bit being the diagnostic node address (NAD) currently configured. pParams is not used and may be set to 0 in the function call.
	
	`LIN_IOCTL_WRITE_NAD`
	Sets the diagnostic node address (NAD) of the slave node to the l_u8 value of *pParams. All values are accepted, values from 1 to 126 are specified by the standard as the values to be used for diagnostic node addresses. Always returns success i.e. 0.
	
	`LIN_IOCTL_WRITE_INITIAL_NAD`
	Sets the *initial* diagnostic node address (NAD) of the slave node to the l_u8 value of *pParams. All values are accepted, values from 1 to 126 are specified by the standard as values to be used for diagnostic node addresses. Always returns success i.e. 0.
	Note: This function shall be called after l_sys_init() but before l_ifc_init() otherwise the initial NAD set with the call will not be used by the driver to initialise the “current” NAD.

Prototype (static)	`l_u16 l_ifc_ioctl_iii (l_ioctl_op operation, void* pParams);` where `iii` denotes the interface to which the operation is to be applied e.g. for the configured interface SCI0 then the prototype: `l_u16 l_ifc_ioctl_SCI0 (l_ioctl_op operation, void* pParams);`

Table 15. Character reception notification

l_ifc_rx	
Prototype (dynamic)	`void l_ifc_rx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function must be called when the specified interface receives one character of data (see LIN 2.1 section 2.5.5). The application program is responsible for binding the interrupt and setting the correct interface handle if the interrupt is used. For UART based implementations, this function can be called from a user-defined interrupt handler triggered by a UART on reception of one character of data. In this case the function performs the required operations on the UART control registers. For more complex LIN hardware, this function can be used to indicate the reception of a complete frame (see also *Section 4.5*).
Parameters	`ifc`: the interface that received the data
Return value	None

Prototype (static)	`void l_ifc_rx_iii (void);`
	where `iii` denotes the interface that received data e.g. for the configured interface SCI0 then the prototype:
	`void l_ifc_rx_SCI0 (void);`

Table 16. Character transmission notification

l_ifc_tx	
Prototype (dynamic)	`void l_ifc_tx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function must be called when the specified interface transmits one character of data (see LIN 2.1 section 7.2.5.6). The application program is responsible for binding the interrupt and for setting the correct interface handle if the interrupt is used. For UART based implementations, this function can be called from a user-defined interrupt handler triggered by a UART on transmission of one character of data. In this case the function performs the required operations on the UART control registers. For more complex LIN hardware, it can be used to indicate the transmission of a complete frame (see also *Section 4.5*).
Parameters	`ifc`: the interface that sent the data
Return value	None

Prototype (static)	`void l_ifc_tx_iii (void);`
	where `iii` denotes the interface that transmitted data e.g. for the configured interface SCI0 then the prototype;
	`void l_ifc_tx_SCI0 (void);`

Table 17. Read interface status

l_ifc_read_status	
Prototype (dynamic)	`l_u16 l_ifc_read_status (l_ifc_handle ifc);`
Availability	Master and slave nodes. The behavior is different for master and slave nodes.
Include	lin.h
Description	Returns a 16-bit status frame for the specified interface.
Parameters	`ifc`: the interface which status is to be returned (see LIN 2.1 section 7.2.5.8)

Table 17. Read interface status (continued)

l_ifc_read_status	
Return value	Status of the previous communication: 16-bit word which value depends on the frame transmitted or received by the node (except bus activity). The call is a read-reset call. This means that after the call has returned, the status word is set to 0. For the master node, the status word is updated in the l_ifc_sch_tick() function. For slave nodes, the status word is updated later when the next frame is started. The status word returned by `l_ifc_read_status` is defined as follows (bit 15 is MSB, bit 0 is LSB) (see *Table 18*): **Bit 0**: Response error This bit is set when a frame error (such as checksum error or framing error) is detected in the frame response. An error in the header results in the header not being recognized and the frame being ignored. It is not set if there was no response on a received frame or if there is an error in the response (collision) of an event-triggered frame. **Bit 1**: Successful transfer This bit is set when a frame has been successfully transmitted/received. **Bit 2**: Overrun This bit is set when two or more frames have been processed since the last call to this function. If set, bit 0 and bit 1 represent the 'OR'ed values for all the frames processed. **Bit 3**: Go to sleep This bit is set when a go to sleep command has been received by a slave node, or when the go to sleep command has been successfully transmitted on the bus by a master node. Receiving a go to sleep command does not affect the node power mode. It is up to the application to do this. **Bit 4**: Bus activity This bit is set when the node has detected activity on the bus. A slave node is put in bus sleep mode after a period of bus inactivity. This can be implemented by the application by monitoring the bus activity. Note: The bus is inactive when there is no transition between recessive and dominant bit values. Bus activity is the opposite. **Bit 5**: Event-triggered frame collision This bit remains set as long as the collision resolving schedule is executed. It can be used in conjunction with the value returned by `l_sch_tick()`. For slave nodes, this bit is always 0. If the master node application switches schedule table while the collision is resolved, the event -triggered frame collision flag is set to 0. **Bit 6**: Save configuration This bit is set when the save configuration request has been successfully received. It is set only for slave nodes, and always remains 0 for the master node. **Bit 7**: always set to 0 **Bit [8:15]**: Last frame protected identifier This 8-bit value represents the protected identifier last detected on the bus and processed in the node. If the overrun bit (bit 2) is set, only the last value is maintained.

Prototype (static)	`l_u16 l_ifc_read_status_iii (void);`
	where `iii` denotes the interface which status is to be read e.g. for the configured interface SCI0 then the prototype:
	`void l_ifc_read_status_SCI0 (void);`

Table 18. Description of l_ifc_read_status returned value

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Last frame PID								0	Save configuration	Event-triggered frame collision	Bus activity	Go to sleep	Overrun	Successful transfer	Response error

3.4 Diagnostic API

3.4.1 Node Configuration specific API (diagnostic)

Table 19. Diagnostic module ready

ld_is_ready	
Prototype	`l_u8 ld_is_ready (l_ifc_handle ifc);`
Availability	Master node only
Include	lin.h
Description	This function returns the status of the last requested configuration service. The returned value values are interpreted as follows: LD_SERVICE_BUSY: Service is ongoing. LD_REQUEST_FINISHED: The configuration request has been completed. This is an intermediate status between configuration request and configuration response. LD_SERVICE_IDLE: The configuration request/response combination has been completed, i.e. the response is valid and can be analyzed. This value is also returned if no request has been called. LD_SERVICE_ERROR: The configuration request or response failed. This means that a bus error occurred, and that the slave node did not receive any negative configuration response (see LIN 2.1 section 7.3.1.1). *Figure 2* shows the case of a successful configuration request and configuration response. The change of state after the master request frame and slave response frame are complete may be delayed by up to one timebase period.
Parameters	`ifc`: the interface of the module to be queried
Return value	LD_SERVICE_BUSY: Service is ongoing LD_REQUEST_FINISHED: The configuration request has been completed. This is an 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 can be analyzed. This value is also returned if no request has been called. LD_SERVICE_ERROR: The configuration request or response failed. This means that a bus error occurred, and that the slave node did not receive any negative configuration response.

Figure 2. successful configuration request and response

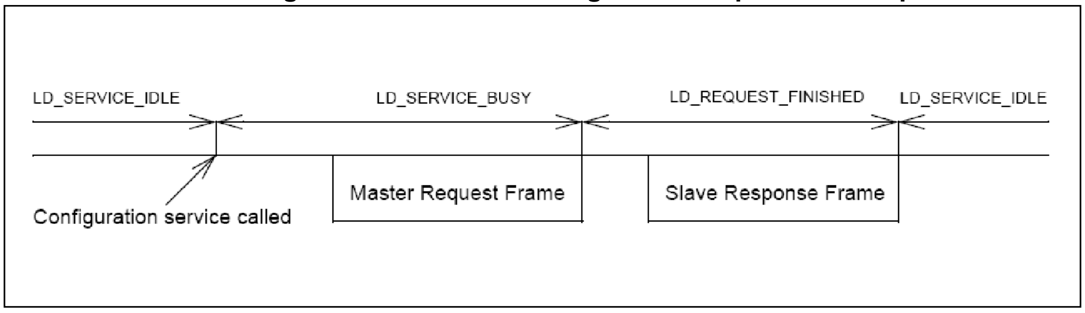


Table 20. Check configuration response

ld_check_response	
Prototype	`void ld_check_response(l_ifc_handle ifc, l_u8* const pRsid, l_u8* const pErrorCode);`
Availability	Master node only
Include	lin.h
Description	This function returns the result of the last node configuration call completed in the parameters `pRsid` and `pErrorCode`. A value is always returned in `pRsid`, but not always in pErrorCode. Default values for `pRsid` and `pErrorCode` are 0 (see LIN 2.1 section 7.3.1.2)
Parameters	`ifc`: interface of the node to be checked `pRsid`: location of the RSID returned `pErrorCode`: location of the error code returned
Return value	None

Table 21. Assign frame ID range

ld_assign_frame_id_range	
Prototype	`Void ld_assign_frame_id_range (l_ifc_handle ifc, l_u8 nad, l_u8 start_index, const l_u8* const PIDs);`
Availability	Master node only
Include	lin.h
Description	This function changes the PIDs of up to four frames in the slave node with the addressed NAD. The PIDs parameter must be four bytes long; each byte must contain a PID, don’t care (0xFF)or unassigned value (0x00) (see LIN 2.1 section 7.3.1.3)
Parameters	`ifc`: interface handle `nad`: diagnostic address of the node `start_index`: frame index from which to start to assign PIDs `PIDs`: Up to 4 bytes containing the new PIDs (PIDs values must be either a value listed in table 2.4 in section 2.8.2 of LIN 2.1 specifications, 0xFF, or 0x00).
Return value	None

Table 22. Assign diagnostic address

ld_assign_NAD	
Prototype	`void ld_assign_NAD (l_ifc_handle ifc, l_u8 initial_nad, l_u16 supplier_id, l_u16 function_id, l_u8 new_nad);`
Availability	Master node only
Include	lin.h
Description	This function changes the diagnostic address (NAD) of any slave nodes matching the specified `initial_nad`, `supplier_id` and `function_id` to the value in `new_nad` (See LIN2.1 section 7.3.1.4)
Parameters	`ifc`: addressed interface `initial_nad`: nad of the nodes to be re-assigned `supplier_id`: supplier id of the nodes `function_id`: function id of the nodes `new_nad`: the new nad to be assigned to the nodes
Return value	None

Table 23. Save configuration

ld_save_configuration	
Prototype	`void ld_save_configuration (l_ifc_handle ifc, l_u8 nad);`
Availability	Master node only
Include	lin.h
Description	This function issues a save configuration request to a specific slave node with the given NAD, or to all slave nodes if NAD is set to broadcast (see LIN 2.1 section 7.3.1.5)
Parameters	`ifc`: addressed interface `nad`: nad of the nodes to which the driver sends the request
Return value	None

Table 24. Assign frame ID

ld_assign_frame_id	
Prototype	`void ld_assign_frame_id (l_ifc_handle ifc, l_u8 nad,  l_u16 supplier_id, l_u16 message_id, l_u8 pid);`
Availability	Master node only. **This function is obsolete for LIN 2.1 and disabled by default. It can be used only to configure LIN 2.0 slaves in mixed clusters.**
Include	lin.h
Description	This function changes the protected identifier of a frame in the slave node corresponding to `nad` and `supplier_id`. The changed frame must be the specified `message_id` and the new protected identifier value given in `pid`.

Table 24. Assign frame ID (continued)

ld_assign_frame_id	
Parameters	`ifc`: interface handle `nad`: diagnostic address of the node `supplier_id`: supplier id of the node `message_id`: the message id to assign to the frame id `pid`: the new protected id value
Return value	None

Table 25. Conditional change diagnostic address

ld_conditional_change_NAD	
Prototype	`void ld_conditional_change_NAD (l_ifc_handle ifc,  l_u8 nad, l_u8 id,  l_u8 byte, l_u8 mask, l_u8 invert, l_u8 new_nad);`
Availability	Master node only
Include	lin.h
Description	This function changes the `nad` if the corresponding node properties fulfills the test specified by `id`, `byte`, `mask` and `invert` (see LIN 2.1 section 7.3.2) Refer to LIN Diagnostic Specification for further details
Parameters	`ifc`: interface handle `nad`: diagnostic address of the node `id`: 0-31 `byte`: 1-5, specifying the id byte `mask`: 0-255 `invert`: 0-255 `new_nad`: new node address
Return value	None

Table 26. Read configuration

ld_read_configuration	
Prototype	`l_u8 ld_read_configuration (l_ifc_handle ifc, l_u8 *const data, l_u8 *const length);`
Availability	Slave node only
Include	lin.h

Table 26. Read configuration (continued)

ld_read_configuration	
Description	This function serializes the current configuration and copies it to the area (data pointer) provided by the application. It does not transport any message on the bus (see LIN 2.1 section 7.3.1.6) Tis function must be called when the save configuration request flag is set in the status register (see LIN 2.1 section 7.2.5.8). When the call has completed, the application is responsible for storing the data in the appropriate memory area. Before calling this function, the caller must reserve an area allowing to store the number of bytes specified by `length`. The `ld_read_configuration` function sets the `length` parameter to the actual size of the configuration. If the data area is too small, the function returns without performing any action. If 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 consists of two bytes, the NAD followed by the PIDs, for all frames ordered in the same way as in the frame list in the LDF and NCF (see LIN 2.1 section 9.2.2.2 and section 8.2.5 respectively).
Parameters	`ifc`: addressed interface `data`: structure that contains the NAD and n PIDs for the frames of the specified NAD `length:` length of `data` (1+n, NAD+PIDs)
Return value	LD_READ_OK if the service was successful. LD_LENGTH_TOO_SHORT if the configuration size is greater than the size specified by `length`. It means that the data area does not contain a valid configuration.

Table 27. Set configuration

ld_set_configuration	
Prototype	`l_u8 ld_set_configuration (l_ifc_handle ifc, const l_u8 *const data, l_u16 length);`
Availability	Slave node only
Include	lin.h
Description	This function configures the NAD and the PIDs according to the configuration specified by data.It does not transport any message on the bus (see LIN 2.1 section 7.3.1.7) This function must be called to restore a saved configuration or set an initial configuration (for example coded by I/O pins). It must be called after calling `l_ifc_init()`. The caller must set the size of the data area before calling it. 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 PIDs are ordered in the same way as in the frame list in the LDF and NCF (see LIN 2.1 section 9.2.2.2 and section 8.2.5 respectively).
Parameters	`ifc`: the interface to address `data`: structure containing the NAD and n PIDs for the frames of the specified NAD `length:` length of `data` (1+n, NAD+PIDs)
Return value	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: Configuration setting could not be performed.

3.4.2 Node Identification specific API (diagnostic)

Table 28. Read by ID

ld_read_by_id	
Prototype	`void ld_read_by_id (l_ifc_handle ifc, l_u8 nad, l_u16 supplier_id, l_u16 function_id, l_u8 id, l_u8* const data);`
Availability	Master node only
Include	lin.h
Description	This function reads the data specified by `id` from the node identified by `nad` and `supplier_id`. When the next call to `ld_is_ready` returns LD_SERVICE_IDLE (after this function is called), the area specified by `data` contains the requested data (see LIN 2.1 section 7.3.3.1) The data returned is always in big-endian format. Note: The user must reserve the memory space to store 5 bytes when *`id`* is in the range given for user defined ids.
Parameters	`ifc`: interface handle `nad`: diagnostic address of the node `supplier_id`: supplier id of the node `function_id`: function id of the node `id`: id of the data requested as follows: 0: LIN product identification 1: serial number 2-31: reserved 32-63: user defined 64-255: reserved `data`: pointer to buffer for receiving the requested data
Return value	None

3.4.3 Diagnostic Transport Layer

Table 29. Initialization

ld_init	
Prototype	`void ld_init (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function (re)initializes the raw and the cooked layers on the ifc interface. All transport layer buffers are initialized (see LIN 2.1 section 7.4.2). If a diagnostic frame transporting a cooked or raw message is ongoing on the bus, it will not be aborted.
Parameters	`ifc`: the interface handle
Return value	None

3.4.4 Diagnostic Transport Layer: RAW API

Table 30. Put raw frame

ld_put_raw	
Prototype	`void ld_put_raw (l_ifc_handle ifc, const l_u8* const pData);`
Availability	Master and slave nodes
Include	lin.h
Description	This function queues a raw diagnostic frame for transmission (see LIN 2.1 section 7.4.3.1) The data area is copied in the call and the pointer is not memorized. Note: The application should check *`ld_raw_tx_status`* before attempting to queue a frame. If no space is available, the frame data is discarded.
Parameters	`ifc`: interface handle `pData`: pointer to the data to be queued
Return value	None

Table 31. Get raw frame

ld_get_raw	
Prototype	`void ld_get_raw (l_ifc_handle ifc, l_u8* const pData);`
Availability	Master and slave nodes
Include	lin.h
Description	This function copies the oldest frame on the receive-stack to the provided buffer (see LIN 2.1 section 7.4.3.2). `ld_raw_rx_status` should be checked first as the `ld_get_raw` function does not report whether a frame has been copied or not. No data are copied if the receive queue is empty.
Parameters	`ifc`: interface handle `pData`: pointer to the buffer into which the frame will be copied
Return value	None

Table 32. Query raw transmit-queue status

ld_raw_tx_status	
Prototype	`l_u8 ld_raw_tx_status (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function returns the status of the raw frame transmission queue (see [LIN 2.1 section 7.4.3.3)
Parameters	`ifc`: interface handle
Return value	LD_QUEUE_FULL: transmit-queue is full and cannot accept further frames LD_QUEUE_EMPTY: transmit-queue is empty i.e. all frames in the queue have been transmitted LD_QUEUE_AVAILABLE: transmit-queue contains entries, but is not full, and is consequently ready to receive additional frames for transmission LD_TRANSMIT_ERROR: LIN protocol errors occurred during transfer. Reinitialize and redo the transmission.

Table 33. Query raw receive-queue status

ld_raw_rx_status	
Prototype	`l_u8 ld_raw_rx_status (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function returns the status of the raw frame receive-queue (see LIN 2.1 section 7.4.3.4)
Parameters	`ifc:` interface handle
Return value	LD_DATA_AVAILABLE: receive-queue contains data that can be read LD_NO_DATA: receive-queue is empty LD_RECEIVE_ERROR: LIN protocol errors occurred during transfer. Reinitialize and redo the transmission.

3.4.5 Diagnostic Transport Layer: COOKED API

Table 34. Send message

ld_send_message	
Prototype	`void ld_send_message (l_ifc_handle ifc, l_u16 length, l_u8 nad, const l_u8* const, pData);`
Availability	Master and slave nodes
Include	lin.h
Description	This function packs the information given by `data` and `length` into one or more diagnostic frames and sends them. If this function is called from a master node, the frames are sent to the node which address is `nad`. If it is called from a slave node, the frames are sent to the master (see LIN 2.1 section 7.4.4.1) The call returns immediately. The buffer should not be changed by the application as long as `ld_rx_status` returns LD_IN_PROGRESS. If a message transfer is ongoing, the call returns without performing any action. Note: SID (or RSID) must be the first byte in the data area. It is included in the length.
Parameters	`ifc`: interface handle `length`: from 1 to 4095 bytes (including SID or RSID, i.e. message length plus one) `nad`: address of node (not used in slave nodes) `pData`: pointer to the data to be sent
Return value	None

Table 35. Receive message

ld_receive_message	
Prototype	`void ld_receive_message (l_ifc_handle ifc, l_u16* const length, l_u8 * const nad, l_u8 * const pData);`
Availability	Master and slave nodes
Include	lin.h
Description	This function prepares the module to receive a message and stores it in the given buffer. When the call is issued, `length` specifies the maximum allowed length. After the call, `length` specifies the actual length. If `ld_receive_message` is called from a master node, `nad` is assigned the value of the `nad` in the message (see LIN 2.1 section 7.4.4.2) The call returns immediately. The buffer should not be changed by the application as long as `ld_rx_status` returns LD_IN_PROGRESS. If the call is issued after the message transmission has started (i.e. the SF or FF is already transmitted), this message will not be received and the function waits until the next message transfer begins. The application must monitor the `ld_rx_status` and must not call this function until the status is LD_COMPLETED. Otherwise this function may return inconsistent data. Note: SID (or RSID) must be the first byte in the data area and is included in the length.
Parameters	`ifc`: interface handle `length`: ranges from 1 to 4095. It is smaller than the value originally set in the call (SID or RSID is included in the length) `nad`: address of node (not used in slave nodes) `pData`: pointer to buffer into which the data will be written
Return value	None

Table 36. Get transmit-queue status

ld_tx_status	
Prototype	l_u8 `ld_tx_status (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function returns the status of the last call made to `ld_send_message` (see LIN 2.1 section 7.4.4.3)
Parameters	`ifc`: interface handle
Return value	LD_IN_PROGRESS: transmission not yet completed LD_COMPLETED: transmission completed successfully. A new `ld_send_message` call can be issued. This value is also returned after the initialization of transport layer is complete. LD_FAILED: transmission completed with an error. Data have been partially sent. The transport layer should be reinitialized before processing further messages. Check the status management function `l_read_status` to find out why the transmission failed (see LIN 2.1 section 7.2.5.8). LD_N_AS_TIMEOUT: transmission failed because of a N_As timeout (see LIN 2.1 section 3.2.5).

Table 37. Get receive-queue status

ld_rx_status	
Prototype	l_u8 ld_rx_status (l_ifc_handle ifc);
Availability	Master and slave nodes
Include	lin.h
Description	This function returns the status of the last call made to `ld_receive_message` (see LIN 2.1 section 7.4.4.4)
Parameters	`ifc`: interface handle
Return value	LD_IN_PROGRESS: reception not yet complete LD_COMPLETED: reception completed successfully. All information are available (length, NAD, data). This value is also returned when the transport layer initialization is complete and a new `ld_receive_message` call can be issued. LD_FAILED: reception completed with an error. Data ware partially received and should not be trusted. The transport layer should be reinitialized before processing further messages. Check the status management function `l_read_status` to find out why the reception failed (see LIN 2.1 section 7.2.5.8). LD_N_CR_TIMEOUT: reception failed because of a N_Cr timeout (See LIN 2.1 section 3.2.5). LD_WRONG_SN: reception failed because of an unexpected sequence number.

3.5 Master specific API

3.5.1 Schedule management

Table 38. Schedule tick

l_sch_tick	
Prototype (dynamic)	`l_u8 l_sch_tick (l_ifc_handle ifc);`
Availability	Master node only
Include	lin.h
Description	This function follows a schedule table and initiates the transmission of frames when due. When the end of the current schedule is reached, it restarts from the beginning of the schedule (see LIN 2.1 section 7.2.4.1) This function must be called periodically and individually for each interface within the node and at a rate specified in the cluster configuration file (LDF). The frequency at which this function is called is given by the `time_base` value set as part of the master node configuration. Since the period at which this function is called effectively sets the timebase tick, it is essential that the timebase period is uphold with a minimum jitter (see *Section 6.2* for an example). A call to this function 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. A predefined schedule table, L_NULL_SCHEDULE, shall exist and may be used to stop all transfers on the LIN cluster.
Parameters	`ifc`: interface handle
Return value	Non-zero: if next call to `l_sch_tick` will start the transmission of the frame in the schedule table entry. The value will be the next schedule table entry number, ranging from 1 to *n* for a table *n* entries. 0: if the next call will not start a transmission

Prototype (static)	`l_u8 l_sch_tick_iii (void);`
	where `iii` denotes the interface for which transmission of frames is to be initiated e.g. for the configured interface SCI0 then the prototype:
	`l_u8 l_sch_tick_SCI0 (void);`

Table 39. Set schedule table

l_sch_set	
Prototype (dynamic)	`void l_sch_set (l_ifc_handle ifc, l_schedule_handle, schedule_iii, l_u8 entry);`
Availability	Master node only
Include	lin.h

Table 39. Set schedule table (continued)

l_sch_set	
Description	This function sets up the next schedule to be followed by the `l_sch_tick` function for the given interface. The entry given specifies the starting point in the new schedule (see LIN 2.1 section 7.2.4.2) The interface is optional. It can be used to solve naming conflicts when the node is a master on more than one LIN cluster. The entry specified must lie in the range of 0 to *n* for a table with *n* entries. Values of 0 or 1 specify the beginning of the schedule table. This function can also be used to set the next frame to be transmitted in the current schedule. A predefined schedule table, L_NULL_SCHEDULE, must exist and can be used to stop all transfers on the LIN cluster.
Parameters	`ifc`: interface handle `schedule_iii`: new schedule handle for the interface ifc `entry`: entry point in the new schedule
Return value	None

Prototype (static)	`void l_sch_set_iii (l_schedule_handle schedule, l_u8 entry);`
	where `iii` denotes the interface which schedule table is to be set e.g. for the configured interface SCI0 then the prototype:
	`void l_sch_set_SCI0 (l_schedule_handle schedule, l_u8 entry);`

Table 40. Goto sleep

l_ifc_goto_sleep	
Prototype (dynamic)	`void l_ifc_goto_sleep (l_ifc_handle ifc);`
Availability	Master node only
Include	lin.h
Description	This function sets all slave nodes in the cluster connected to the interface given to sleep mode (see LIN 2.1 section 7.2.5.2) The go to sleep command is scheduled later when the next schedule entry is due. This function does 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 is set in the status register (see LIN 2.1 section 7.2.5.8). Note: It is important that this function is not called if a frame transfer is ongoing. The current transmission could be interrupted or the master node driver may not send the go to sleep command as expected. The user should first check the current driver state by calling *`l_ifc_ioctl()`*, and call *`l_ifc_goto_sleep()`* only if the driver is in *`LIN_STATE_IDLE`* state.
Parameters	`ifc`: interface handle
Return value	None

Prototype (static)	`void l_ifc_goto_sleep_iii (void);`
	where `iii` denotes the interface to be sent to sleep e.g. for the configured interface SCI0 then the prototype:
	`void l_ifc_goto_sleep_SCI0 (void);`

3.6 Slave specific API

3.6.1 Interface management

Table 41. Slave synchronize

l_ifc_aux	
Prototype (dynamic)	`void l_ifc_aux (l_ifc_handle ifc);`
Availability	Slave node only
Include	lin.h
Description	This function performs the synchronization with the BREAK and SYNC characters sent by the master node on the specified interface (see LIN 2.1 section 7.2.5.7) Note: This function is redundant for the drivers currently delivered by ST. It is therefore implemented as a null function.
Parameters	`ifc`: interface handle
Return value	None

Prototype (static)	`void l_ifc_aux_iii (void);`
	where `iii` denotes the interface e.g. for the configured interface SCI0 then the prototype:
	`void l_ifc_aux_SCI0 (void);`

Table 42. Read by ID callout

ld_read_by_id_callout	
Prototype	`l_u8 ld_read_by_id_callout (l_ifc_handle ifc, l_u8 id, l_u8* data);`
Availability	Slave node only (optional: if it is used, the slave node application must implement this callout).
Include	lin.h
Description	This function can be used when the master node transmits a read-by-identifier request with an identifier in the user defined area. When such a request is received, the slave node application is called from the driver (see LIN 2.1 section 7.3.3.2)
Parameters	`ifc`: interface handle `id`: identifier in the user defined area (32 to 63), from the read-by- identifier configuration request. (see LIN 2.1 Table 4.19) `data`: pointer to a data area of 5 bytes. This area is used by the application to set up the positive response (see LIN 2.1, user defined area in Table 4.20).
Return value	The driver acts according to the following return values from the application: LD_NEGATIVE_RESPONSE: the slave node responds with a negative response (see LIN 2.1, Table 4.21). In this case the data area is taken into account. LD_POSTIVE_RESPONSE: the slave node sets up a positive response using the data provided by the application. LD_NO_RESPONSE: The slave node does not answer.

3.7 STMicroelectronics extensions

Table 43. Software Timer function

l_timer_tick	
Prototype	`void l_timer_tick (void);`
Availability	Master and slave nodes
Include	lin.h
Description	This function handles a software timer interrupt. Internal LIN timers are incremented and expired timers are evaluated. This function should be called by the user application every LIN_TIME_BASE_IN_MS ms when no hardware timer has been configured (see also *Section 4.4*).
Parameters	None
Return value	None

Table 44. Protocol Switch

l_protocol_switch	
Prototype	`void l_protocol_switch (l_ifc_handle ifc, l_bool, linEnable);`
Availability	Master and slave nodes
Include	lin.h
Description	This function switches the LIN protocol on or off for a specified interface. It provides the possibility to use an alternative protocol on a given interface. The ISR checks if LIN is enabled. If it is not, it calls a callback function that is provided as an entry point to the alternative protocol handler. This callback must be configured as described in *Section 4.4.6*.
Parameters	`ifc`: interface handle `linEnable`: if 1 then switch LIN on, if 0 then off
Return value	None

Table 45. Data Dump

l_data_dump	
Prototype	`void ld_data_dump (l_ifc_handle ifc, l_u8 nad, const l_u8 * const sendBuf, l_u8 * const receiveBuf);`
Availability	Master node only
Include	lin.h
Description	This function requests a data dump from a slave node. The slave may return up to 5 bytes via the buffer `receiveBuf`. The application must call `ld_is_ready()` after the call to `ld_data_dump`. When TRUE is returned, the application can access the data in the buffer. According to the LIN specification, this feature should not be used in live networks.
Parameters	`ifc`: interface handle `nad`: address of node `sendBuf`: holds the message to be sent to the slave `receiveBuf`: holds the response when `ld_is_ready()` next returns TRUE
Return value	None

Table 46. Set Cooling Break

l_ifc_set_cooling_break	
Prototype	`void l_ifc_set_cooling_break (l_ifc_handle ifc, l_bool on);`
Availability	Master node only
Include	lin.h
Description	This function sets the break length to the cooling value if the parameter `on` is set to TRUE, otherwise the break length is set to the standard break length.
Parameters	`ifc`: interface handle `on`: set to true if cooling break should be used, false if normal break should be used
Return value	None

Table 47. Set baud rate

l_change_baudrate	
Prototype	`void l_change_baudrate (l_ifc_handle ifc, l_u16, baudrate);`
Availability	Slave node only
Include	lin.h
Description	This function sets the baud rate for the specified interface. This function should only be called from the callback function that is invoked by the driver when an incorrect (too high) baud rate is detected. The callback function must be configured as described in *Section 4.4.6*.
Parameters	`ifc`: interface handle `baudrate`: the baud rate to set for the interface
Return value	None

Table 48. Raw Tx delete

ld_raw_tx_delete	
Prototype	`void ld_raw_tx_delete(l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	This function removes the oldest Raw Tx frame that has been put on the Tx stack using `ld_put_raw()`.
Parameters	`ifc`: interface handle
Return value	1 if a frame has been removed, 0 if no Raw Tx frame was on the stack.

3.8 Implementation notes

3.8.1 API data types

STLIN driver does not support some of the types defined in the standard API. This means that the application cannot directly define variables of these types. The types that are not defined are `l_signal_handle`, `l_frame_handle` and `l_flag_handle`.

When calling the dynamic interface, the application can only use the signals, frames and schedules by their name as defined in the LDF. It should use the interface name defined in the lingen control file. Flag names are based on the signals and frames defined in the LDF.

Refer to *Section 2.4: Using the driver* for details on the naming conventions used by the driver.

3.8.2 Notification flags

The notification flags are used to indicate signal or frame updates resulting from the transfer of signal or frames.

Since the application and driver executions are asynchronous, unexpected behavior may occur as described in the following example.

Description

1. The driver (master or slave) detects that a frame must be sent and builds its frame buffer. It copies the current signal values into the frame buffer and starts the transmission.
2. The user application writes a signal that is contained in the frame being transferred. It can also reset the Tx flag to be notified when the signal transmission is complete. However, the transfer of the frame is still in progress.
3. The driver completes successfully the transfer of the frame. It then marks the frame and all the signals it contains as transferred.
4. The user application polls the Tx flag and receives 1. It consequently assumes that the value just written has been transferred. Instead, the value that was originally valid has been transferred.

Workaround

Use the `l_ifc_ioctl()` function to query the driver state before writing new signal values. If a frame is being transferred, the query returns the pID and the driver state. The application can then check if the signal to be written is part of the current frame. See *Section 3.3.4: Interface Management* for further details of the `l_ifc_ioctl()` function.

4 Driver configuration

This section describes the configuration and the hardware specific settings required, and explains how to build the driver.

4.1 File and directory structure

The LIN driver consists of four different groups of source files:

- Generic files for all architectures
- Hardware specific files
- User configurable files
- Configuration files generated by the **lingen** tool (supplied with the driver)

The user configurable files, the **lingen** tool, the control file and the LDF files required to generate a LIN library must be located in a directory selected by the user. This must then be specified in the top-level makefile as described in *Section 4.2*.

The driver specific Make_LIN makefile (supplied with the driver) is based on a particular directory structure. The top-level directory is referred to by the variable LIN_SRC_PATH and must be configured in the top-level makefile (see *Section 4.2*). Its subdirectories should be compliant with *Table 49*:

Table 49. Directory structure

Top directory	Subdirectory	Comment
LIN_SRC_PATH	node_type	*node_type* is “master” if Master node otherwise “slave”
	general	as given
	diag	as given
	timer	as given
	arch/ *arch_name*	*arch_name* specifies the specific architecture for which the driver will be built

4.2 Makefiles

The LIN driver is delivered together with two makefiles that can be used to build a library containing the required functionalities:

- A top-level makefile example which includes the settings for environment variables (see *Section 4.2.1: Top-level makefile*).
- Make_LIN: this file controls the building process and is designed to be included in the top-level makefile.

4.2.1 Top-level makefile

Predefined variables

This file must include the definitions for the following variables:

Table 50. Top-level makefile predefined variable definition

Name	Description
LIN_NODE_IDENTITY	This variable must be set to MASTER_NODE for a master node driver or SLAVE_NODE for a slave node driver
LIN_CC	Compiler command
LIN_CC_OPT	Compiler options
LIN_CC_INC	Include directories for the compilation process
LIN_MAKE_PATH	Path to Make_LIN file
LIN_OBJ_PATH	Path in which to generate object files
LIN_LIB	Lib generation tool
LIN_TMP_FILE	Name for temporary file
LIN_SRC_PATH	Top-level directory of the LIN driver
LIN_CFG_PATH	Directory containing the configuration information for the cluster and the driver. The **lingen** control file and the files `lin_def.h`, `lin_def.c` and /`in_def_archname.h` must be located in this directory. The file `lin_def_archname.h` is the architecture specific user configuration file where –*archname* refers to the specific architecture name.
LIN_GEN_PATH	Directory in which generated files are written. This is used for the –o option for **lingen** in the file Make_LIN
LIN_LINGEN_BIN	Command used to invoke **lingen**
LIN_NODE	Name of the node as defined in the LDF. If multiple interfaces are defined for a master node then the name given in the associated LDF files should be the same throughout the application.
LIN_LINGEN_CONTROL	Name of the control file used by **lingen**
LIN_LINGEN_OPTS	Options to be used by the **lingen** tool. Details of options are given in *Section 4.3*

Optional variables

In addition, the optional makefile variable LIN_LDF_FILES can be set by the user. They can be used to list the LDF filename(s) to be included in the dependency checks during the make process.

The definition of the variables of the Make_LIN file should be included as follows:

```
include < path_to_MakeLIN > /Make_LIN
```

where < path_to_MakeLIN > specifies the location of Make_LIN.

The generation of the LIN library can then be included as follows:

```
make $(LIN_OBJ_PATH)/lin.lib
```

or by including $(LIN_OBJ_PATH/lin.lib in the target build instruction.

A sample makefile is delivered in the driver. This can be used as a basis for development purposes.

4.3 Cluster configuration

4.3.1 Cluster description

The description of the node and cluster must be provided in a LIN description file (LDF) in accordance with the LIN 2.1 specification. An LDF example is provided in *Section 6.2*. The **lingen** tool delivered with the driver suite can be used to convert the information given in the LDF into the appropriate format used internally by the driver.

In addition to the cluster description, the user must specify which hardware interface(s) are used. This information is specified in the lingen control file that is input to the **lingen** tool.

If the **lingen** tool is called from within the make process, then the name of the control file must be set by the user in the top-level makefile as described in *Section 4.2.1: Top-level makefile*.

Master node

A master node can support more than one interface. In this case a separate LDF file is needed for each interface. The lingen control file specifies which interfaces are used and the corresponding LDF file.

The format of the lingen control file is specified in *Section 5.1: Lingen control file* and is shown in the example below:

```
//
// lingen control file defining two interfaces
//
Interfaces
{
   SCI0: "/home/LIN/src/lin_config/lin_sci0.ldf", "IFC0";
   SCI1: "/home/LIN/src/lin_config/lin_sci1.ldf", "IFC1";
}
```

The interface entries consist of three parts: the interface name, the corresponding LDF file, and an optional tag field.

The location of the LDF file should be completely specified by giving the absolute path.

The tag entry is concatenated with all frame names and signal names when the **lingen** tool processes the LDF files. As an example, a signal name “s1_sig1” in the LDF file *lin_sci0.ldf* listed above will appear in the code as “LIN_SIGNAL_IFC0_sl_sig1”.

For the master node, the tag field must be used to resolve naming conflicts when two LDF files have common signal names. The number of interfaces that can be listed in the control file and their names depend on the hardware. The tag names can be freely chosen by the user.

Slave node

A slave node only supports one interface and therefore only one LDF file is required for a slave. The lingen control file then specifies this interface and the name of the corresponding LDF file. In addition to this interface definition, the user can also define default frame identifiers in the slave lingen control file. In this case, the default values given in the LDF file will be used for all slave frames. This means that the slave nodes can start communicating without having been first configured by the master node. This behavior is then no longer compliant with the standard.

The format of the lingen control file is specified in *Section 5.1: Lingen control file* and is shown in the examples below:

```
// lingen control file defining one interface
//
Interfaces
{
  SCI0: "/home/LIN/src/lin_config/lin_sci0.ldf", "IFC0";
}
//
//specify that slave nodes will start with default frame IDs
//
LIN_use_default_frame_ids;
```

The interface entries consist of three parts: the interface name, the LDF file to be associated with this interface and an optional tag field.

The location of the LDF file should be completely specified by giving the absolute path.

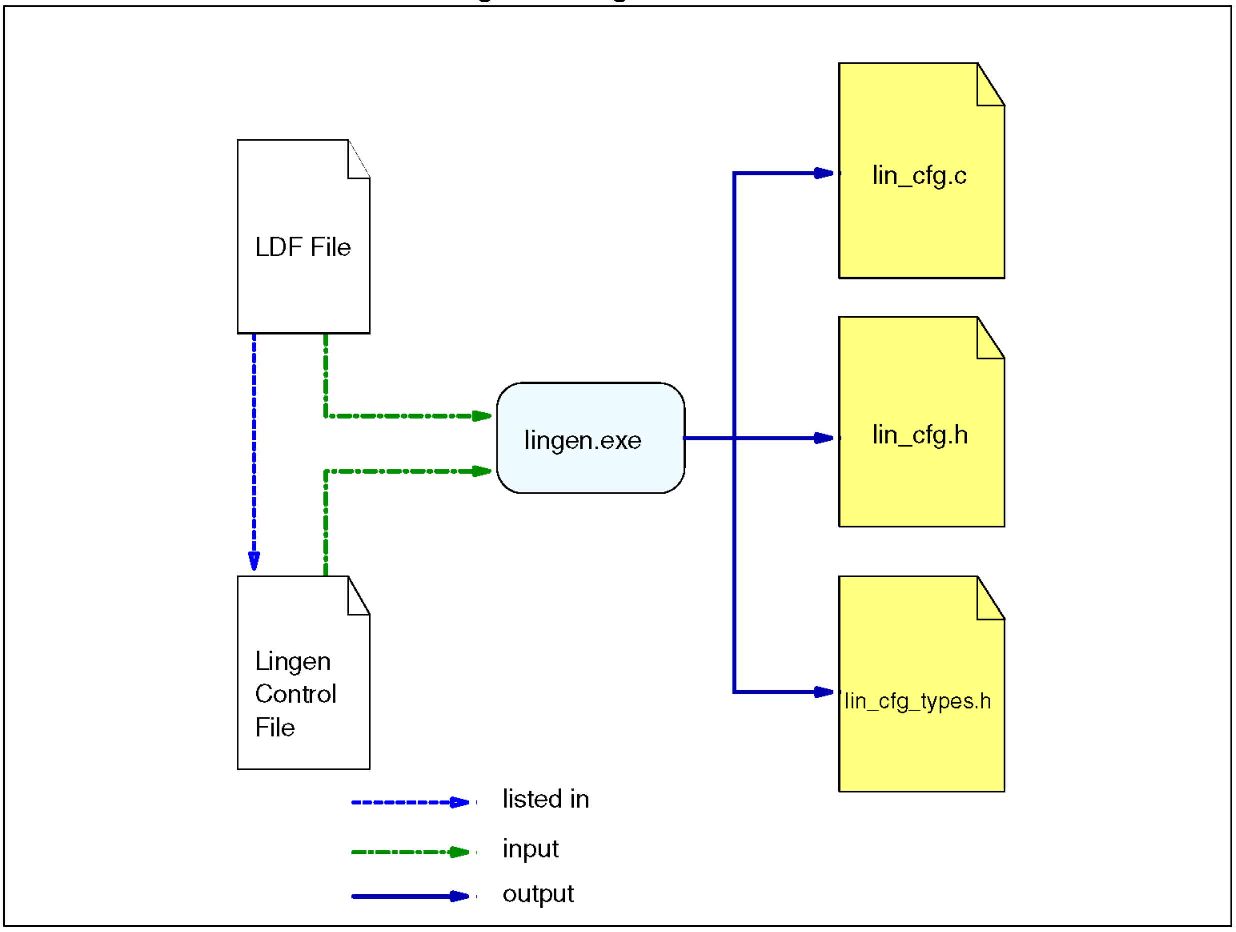
The tag entry is concatenated with all frame names and signal names when the **lingen** tool processes the LDF files. As an example, a signal name “s1_sig1” in the LDF file *lin_sci0.ldf* listed above will appear in the code as “LIN_SIGNAL_IFC0_sl_sig1”.

For slave nodes, the interface name given in the control file depends on the hardware. Following the interface definition the user can specify the use of default frame IDs as described earlier.

4.3.2 Lingen tool

A set of configuration files can be generated using the **lingen** tool provided by ST. It is based on the information given in the lingen control file and on the associated LDF file. Inputs and outputs for lingen are described in *Figure 3*.

Figure 3. Lingen workflow



The `lin_cfg_types.h` file contains the type definitions needed for the driver. `lin_cfg.h` contains static macros used for accessing configured signals and `lin_cfg.c` contains initialized data structures in accordance with the information given in the LDF.

The **lingen** tool is started automatically from the makefile but can be manually executed using the following command format:

```
lingen nodeIdentifier [options] lingen_control_file
```

where `nodeIdentifier` is the name of the node given in the LDF files for which the driver is to be built. For a master node, it must be the same in all ldfs, while there is only one ldf supported in the case of a slave node.

`lingen_control_file` is the name of the control unit input to **lingen.**

The options listed in *Table 51* are currently supported:

Table 51. Options

Name	Description
-c configuration	Specifies which of the possible configurations given in the LDF will be used to build the driver.
-o outputDirectory	Specifies the destination for the configuration files that are to be generated
-r receiveChecksum	Selects the checksum model to be used for receiving frames. Possible values are: `ldf`(default): the lingen tool determines the checksum model from the information given in the LDF. `both`: the driver accepts either model for all frames
-s sendChecksum	Selects the checksum model to be used for sending frames. Possible values are: `ldf` (default): the lingen tool determines the checksum model from the information given in the LDF. `classic`: the driver sends all frames using the classic checksum
- v verbose mode	Details from lingen will be output to the shell

Note: The -o option is set in the file `Make_LIN` and should not be set in the top-level makefile. It must be set to LIN_GEN_PATH. It is recommended not to change this setting unless the file `Make_LIN` is replaced.

4.4 User configuration

There are two header files that contain user configurable settings, `lin_def.h` and `lin_def_archname.h` where `_archname` refers to a specific architecture:

- `lin_def_archname.h` contains the architecture dependent settings (see *Section 7: Architecture notes*).
- `lin_def.h` contains other configurable settings (see *Section 4.4.1*, *Section 4.4.2*, *Section 4.4.3*, *Section 4.4.4*, *Section 4.4.5*, and *Section 4.4.6*).

4.4.1 Timers

The LIN driver can use either a hardware timer or a software timer to monitor bus activity when sending frames or checking bus sleep conditions. This timer must be configured by the user in the architecture specific configuration file `lin_def_archname.h`.

If a hardware timer is selected, the timer number must also be configured according to the architecture (see *Section 7: Architecture notes* for the possible values).

The timer timebase is configured in `lin_def.h`. This timebase specifies the frequency at which the driver timer routine is called.

For a software timer, this timebase gives the frequency at which the API function `l_timer_tick()` is called by the user application or operating system. If a hardware timer interrupt has been configured, the driver sets the timer so that the timer ISR is called at this frequency.

The recommended value for the timebase is either 1 or 2 ms. It is set as follows:

```
/**************************************************************
 *
 * Set the time base of the lin timer in ms.
 * This is the time base of the timer ticks of the application
 * driven timer or the time base of the hardware timer
 *
 *************************************************************/
#define LIN_TIME_BASE_IN_MS    1
```

Further details concerning the use of a hardware timer are given in *Section 7: Architecture notes*.

4.4.2 General settings

The user must define additional settings as described below.

Checking function parameters

The driver can be built either for development or for production purposes. The development version includes a more extensive check on the parameters passed to the functions. This may not be necessary for a production version. In this case, checking operations can be reduced by changing the following switch:

```
/**************************************************************
* Set the driver for development or production:
*
* For development:
* #define LIN_DEVELOPMENT, several checks of input parameters
* are performed. This will be quite useful for debugging
* during the development phase.
*
* For production:
* #undef LIN_DEVELOPMENT, only a few checks on the input
* parameters of the functions are performed. Activate this
* for smaller and faster code for the production phase after
* development.
*************************************************************/
#define LIN_DEVELOPMENT
```

Setting the frame maximum duration

The maximum duration for a frame transfer can be configured as a percentage of the nominal transfer time (the default setting of 140 corresponds to that specified in the LIN 2.1 specification:

```
/**************************************************************
 *
 * select the maximum frame transfer time in multiples of the
 * nominal time (*100)
 *
 *************************************************************/
#define LIN_FRAME_TIME_MULTIPLIER    140
```

Setting the number of bits for the BREAK signal

The number of bits to be used for a normal BREAK signal can be configured by changing the following setting. However, it is not normally recommended to change this value from the LIN standard value of 13 bits.

```
/***************************************************************
 *
 * length of the break signal in bit times (nr of bits)
 * recommended is 13
 * Please adjust LIN_FRAME_TIME_MULTIPLIER if necessary
 *
 ***************************************************************/
#define LIN_BREAK_DURATION_BITS       13
```

According to the LIN 2.1 standard, a slave node is able to detect a BREAK at any time. The current frame processing is interrupted and the new frame is then processed. A BREAK is detected by the driver through a framing error. This may occur at any time during the transmission of a data byte or between transmissions of data bytes. The following switch allows to decide if all framing errors should be treated as a possible BREAK or not. Defining the switch will force the driver to examine the information last received over the bus. Only a BREAK character that does not occur during the transmission of a data byte will be accepted as a valid BREAK.

```
/***************************************************************
 * The slave driver is able to detect a new BREAK character
 * during an ongoing frame transfer (if supported by
 * hardware). This is detected through a framing error and may
 * occur at any time.
 * If LIN_FORCE_STANDALONE_BREAK is *not* defined, any framing
 * error will be considered as a possible BREAK character,
 * even if this occurs during the transmission of a data byte.
 * Otherwise a framing error will only be considered as a
 * possible break if it occurs between transmission of data
 * bytes.
 ***************************************************************/
#undef LIN_FORCE_STANDALONE_BREAK
```

Additionally, a longer break signal is required for drivers in a Cooling v2.1 network and so the standard 13 bit break signal can be lengthened to 36 bits for a 19200 baud rate network or 18 bits in a 9600 baud rate network or equivalent.

The Cooling feature must be activated before being used:

```
/***************************************************************
 *
 * Activate the Cooling option with  #define LIN_USE_COOLING
 * Deactivate it with                #undef  LIN_USE_COOLING
 *
 ***************************************************************/
#define LIN_USE_COOLING
```

and the break length set:

```
#ifdef LIN_USE_COOLING
```

```
/************************************************************
*
* length of the break signal in bit times (nr of bits)
* Please adjust LIN_FRAME_TIME_MULTIPLIER if necessary
*
***********************************************************/
#define LIN_COOLING_BREAK_DURATION_BITS    36

#endif
```

Activating the Cooling feature provides the user with an additional interface function `l_ifc_set_cooling_break` that can be called from the application when a longer break is required. This interface function can be used to toggle the length of the break between the configured cooling break length and the configured normal break length as required.

Configuring LIN node startup behavior

The startup behavior of LIN nodes can be influenced by two settings. The first option is to start the slave node bus sleep timer going when a slave node connects to the network. The slave then enters sleep mode if no activity is detected.

Additionally, a master node can be set up to send a wakeup signal when connecting to the network. Note that if a slave is set up to start the bus sleep timer on connection then the master should be configured to send a wakeup. If it is not the case, and the master does not start transmitting within 4 seconds, then the slave node enters sleep mode. In this case the slave node will not be ready to receive frames as it expects to receive a wakeup signal first.

The following two settings can be used for this purpose:

```
/*************************************************************
 *
 * select whether the slave node will start the bus sleep timer
 * on connect
 * Note: The slave may lose the first frame if the master
 *       node does not start with a Wakeup signal followed by
 *       100ms silence
 *
 ************************************************************/
#define LIN_START_BUSSLEEP_TIMER_ON_CONNECT

/*************************************************************
 *
 * select whether the master node should start a wakeup
 * sequence on connect
 *
 *************************************************************/
#define LIN_SEND_WAKEUP_SIG_ON_CONNECT
```

For a master node, the initial delay is undefined when switching from L_NULL_SCHEDULE to a valid schedule table after startup. Use the following definition to set the initial value of the delay in timebase ticks:

```
#define LIN_DELAY_INIT            2
```

On slave nodes, when receiving frames, pIDs are validated against stored pIDs. However, there is no validation of pIDs assigned by the master or by the slave application. Therefore an option to validate pIDs on assignment is provided. The following definition can be used for this purpose:

```
#define LIN_INCLUDE_PID_PARITY_CHECK
```

Note: Validation is only carried out on assignment and not on reception of each frame by the slave node.

In LIN 1.2/1.3 nodes, the API functions `l_ifc_goto_sleep()` and `l_ifc_wake_up()` were not defined. Use the following definition if LIN 1.2/1.3 nodes need to include these two API functions:

```
/***************************************************************
 * LIN 1.2/1.3 specific setting
 * #define this if you want to use LIN 2.x goto sleep/wakeup
 * API for LIN 1.2/1.3
*************************************************************/
#define LIN_INCLUDE_2x_SLEEP_MODE_API
```

The LIN 2.x standard describes the behavior of nodes when receiving a wakeup request from a slave. All slave nodes must be ready to receive frames within 100 ms. A master must send frames within 150 ms after the wakeup request.

This means that a slave node may take up to 100 ms before being ready to receive frames and so a master node should not start transfer before this delay has elapsed. However, if slave nodes are ready to receive frames in a shorter time frame, then the master may start sending earlier.

This time delay must be configured using the following definition:

```
/***************************************************************
 * Master specific setting
 * Here the delay after wakeup before sending frames is
 * specified for LIN 2.x.
 * The value is configurable here (in milliseconds). The
 * default value is 100 milli secs.
 * Here you can specify other values, for example shorter
 * times if you know your slaves are responding faster.
 * In particular the CANoe.lin conformance test needs a
 * shorter time, recommended in this case is 75.
 *************************************************************/
#define LIN_MASTER_WAKEUP_TIMER_VALUE            100
```

On slave nodes, the default value for bus sleep timeout is specified in the LIN2.x standard as 4 s. When a wakeup request is issued by a slave node, the period between consecutive retries is 150 ms. After three failed attempts the node must wait 1.5 s before issuing another wakeup request. These values can be configured by the user as follows:

```
/***************************************************************
 * LIN 2.x specific setting
 * The value for the bus sleep timeout is configurable here (in
 * milliseconds). The recommended default value given in the
 * standard is 4secs.
 * The other two definitions give the period between the
 * signals in milliseconds, the standard demands 150 and 1500
 * msecs
 ***********************************************************/
#ifndef LIN_13
  #define LIN_BUSSLEEP_TIMEOUT_VAL(IFC)      (l_u16) 4000
  #define LIN_WAKEUP_TIMEOUT_VAL_SHORT(IFC) (l_u16)  150
  #define LIN_WAKEUP_TIMEOUT_VAL_LONG(IFC)  (l_u16) 1500
#endif
```

The maximum number of attempts can also be configured:

```
/************************************************************
 * The number of wakeup retries to send
 * If after a wakeup signal from the slave the master does not
 * start to send frame headers, the slave may retry to send
 * the wakeup signal. The define gives the maximum number of
 * retries
 ***********************************************************/
#define LIN_WAKEUP_RETRIES_MAX   3
```

Note: Setting this value to zero means that the driver does not stop sending wakeup signals when there is no response from the master.

4.4.3 Diagnostic functions configuration

The diagnostic module functions can be individually selected as described below. The driver default settings reflect the standard requirements. The following definitions are used for this purpose:

```
/************************************************************
 *
 * Configuration features. Select by define'ing.
 * Default values match the mandatory services defined by the
 * standard
 *
 ***********************************************************/

/************************************************************
 * service Assign Frame Id (mandatory for LIN 2.0, obsolete
 * for LIN 2.1)
 ***********************************************************/
#undef LIN_INCLUDE_ASSIGN_FRAME_ID

/************************************************************
 * service Assign NAD (optional for LIN 2.x)
 ***********************************************************/
#define LIN_INCLUDE_ASSIGN_NAD
```

The “Assign NAD” service is optional. It is enabled since it is called in the Initialization table of the demo present in the LDF file of the LIN package and is required to configure slave nodes.

```
/************************************************************
 * service Read By Id (mandatory for LIN 2.x)
 **************************************************************/
#define LIN_INCLUDE_READ_BY_ID

/**************************************************************
 *service Conditional Change NAD (optional for LIN 2.x)
 **************************************************************/
#undef LIN_INCLUDE_COND_CHANGE_NAD

/**************************************************************
 * service Data Dump (optional for LIN 2.x))
 * Note: The standard strongly discourages use of this service
 *       in operational LIN clusters
 **************************************************************/
#undef LIN_INCLUDE_DATA_DUMP

/**************************************************************
 *
 * choose Serial Number (optional for Slave node)
 * Slave node may have a serial number to identify a specific
 * instance of a slave node product. The serial number is 4
 * bytes long.
 **************************************************************/
#define SERIAL_NUMBER                    0xFFFF
```

The **Save Configuration** and **Assign Frame Id Range** services are valid only for LIN 2.1:

```
#ifdef LIN_21
/**************************************************************
 * service Save Configuration (optional for LIN 2.1)
 **************************************************************/
#define LIN_INCLUDE_SAVE_CONFIGURATION


/**************************************************************
 * service Assign Frame Id Range (mandatory for LIN 2.1)
 **************************************************************/
#define LIN_INCLUDE_ASSIGN_FRAME_ID_RANGE
```

The **Save Configuration** service is also optional. It is enabled since it is called in the Initialization table of the demo present in the LDF file of the LIN package.

4.4.4 Diagnostic class

The Diagnostic class is valid and mandatory only for LIN 2.1 slave nodes. It can be used to perform the following actions:

- Do a crosscheck between LDF and the same class to understand if the slave node is able to perform the diagnostic
- Understand which diagnostic services the slave node is able to respond to
- Know which configuration and identification services is supported
- Understand if the slave node is able to support the Transport Protocol
- Understand if the slave node is able to be reprogrammed (only class 3).

```
/************************************************************
 * choose Diagnostic Class (mandatory for LIN 2.1 slave node)
 * LIN 2.1 slave nodes must have a Diagnostic Class value
 * defined.
 * This value can be: 1,2 or 3 (other values will involve in
 * an error).
 ***********************************************************/
/*
 * DIAGNOSTIC_CLASS 1: Only the node configuration services
 * are supported.
 * The slave does not support any other diagnostic services.
 * Single frames (SF) transport protocol support is
 * sufficient.
 * Node Identification is limited to the mandatory read by
 * identifier service.
 *
 * DIAGNOSTIC_CLASS 2: Node configuration and identification
 * services are supported.
 * Full transport layer implementation is required to support
 * multi-frame transmissions.
 * Node Identification is extended to all the Read By Id
 * services. Slave-nodes will support a set of ISO 14229-1
 * diagnostic services like Node identification (SID 0x22),
 * Reading data parameter (SID 0x22) if applicable, Writing
 * parameters (SID 0x2E) if applicable.
 *
 * DIAGNOSTIC_CLASS 3: Node configuration and identification
 * services are supported.
 * Full transport layer implementation is required to support
 * multi-frame transmissions.
 * Node Identification is extended to all the Read By Id
 * services. Slave-nodes shall support all services as of
 * Class II.
 * Additionally, other services may be supported depending on
 * the features which are implemented by the slave node:
 * for example Session control (SID 0x10), I/O control by
 * identifier (0x2F), Read and clear DTC (SID 0x19, 0x14).
 * Only class 3 slave nodes can reprogram the application via
 * the LIN bus.
 ***********************************************************/
#define LIN_DIAGNOSTIC_CLASS 1

 #ifdef LIN_SLAVE_NODE
   #ifndef LIN_DIAGNOSTIC_CLASS
     #error "For a LIN 2.1 slave node, LIN Diagnostic Class is
             mandatory and must be defined!"
   #endif

   #if ((LIN_DIAGNOSTIC_CLASS < 1) ||
        (LIN_DIAGNOSTIC_CLASS > 3))
     #error "LIN 2.1 Diagnostic Class value can be 1,2 or 3!"
   #endif
 #else
   #if ((!defined(LIN_MASTER_ONLY)) ||
        (!defined(LIN_SLAVE_ONLY)))
     #error "A master node must support the Interleaved
             Diagnostics schedule Mode (mandatory)!"
   #endif
 #endif

#endif /* LIN_21 */


/*************************************************************
 *
 * LIN TP features. Select by define'ing.
 * TP is disabled by default
 *
 *************************************************************/


/*************************************************************
 * the cooked diagnostic TP
 *************************************************************/
#undef LIN_INCLUDE_COOKED_TP


/*************************************************************
 * the raw diagnostic TP
 *************************************************************/
#undef LIN_INCLUDE_RAW_TP


For the Raw TP the size of the Rx and Tx FIFO stack can be
configured using the definition:
/*************************************************************
 * define the stack size of the raw tp fifo stacks
 * (in numbers of frames)
 *************************************************************/
#define LIN_DIAG3_FIFO_SIZE_MAX                        1
```

4.4.5 Transport Protocol handling in LIN 2.1 Master

Transport Protocol handling requires that:

- The master node supports a diagnostic master request schedule table that contains a single master request frame.
- The master node supports a diagnostic slave response schedule table that contains a single slave response frame.
- When no diagnostic communication is active, the master node does not execute diagnostic schedules tables (default).
- A master node supports the following different scheduling modes:
  - Interleaved diagnostics mode (mandatory)
  - Diagnostics-only mode (optional)

Master node supporting a diagnostic master request schedule table containing a single master request frame

The diagnostic master request schedule table is executed each time a master request frame is transmitted (see *Figure 4*).

The lingen tool checks if a table containing only a single master request frame exists in the LDF file.

- If a table is found (if there are more than one table, it checks only the first table found in the LDF file), it includes the following define statement in the `lin_cfg_types.h` (located under ..\demo\stm8\lin_basic_demo\master\obj\):

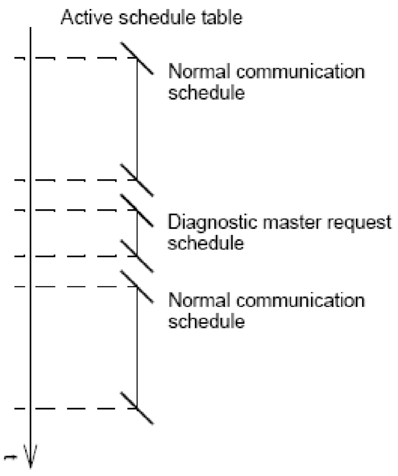
  ```
  #define LIN_MASTER_ONLY                  LIN_TAB_MasterOnly
  ```

  In the example below the lingen tool has found the MasterOnly table in the LDF file:

```
MasterOnly {
//This is the Master Diagnostic interleaved Mode (mandatory)
MasterReq delay 20 ms;

}
```

- If the lingen tool does not find a table complying with the mandatory requirements, it does not include the LIN_MASTER_ONLY define statement. This will cause a compilation error since a master node must support the Interleaved Diagnostics schedule mode which is mandatory (see also *Master node supporting either Interleaved diagnostics or Diagnostics-only mode*).

Figure 4. Interleaving of a diagnostic master request schedule table


Active schedule table
Normal communication schedule
Diagnostic master request schedule
Normal communication schedule

Master node supporting a diagnostic slave response schedule table containing a single slave response frame

The diagnostic slave response schedule table is run between the executions of the normal communication schedules whenever a slave response frame is transmitted (see *Figure 5*).

The lingen tool checks if a table containing only a single slave response frame exists in the LDF file.

- If a table is found (if there are more than one table, it checks only the first table found in the LDF file), it includes the following define statement in the `lin_cfg_types.h` (located under ..\demo\stm8\lin_basic_demo\master\obj\):

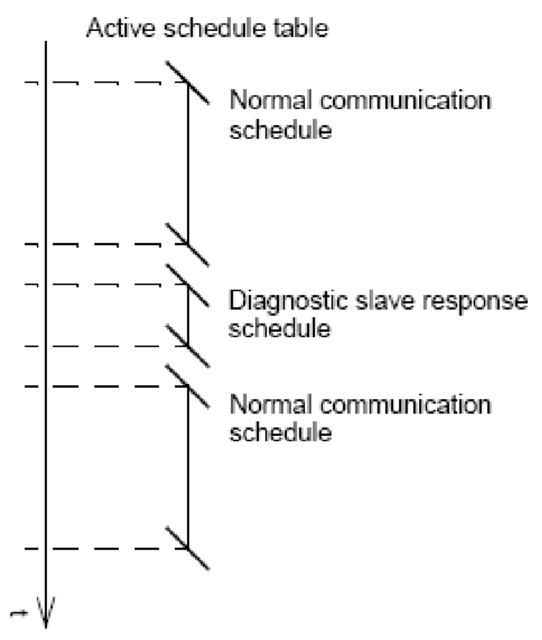
```
#define LIN_SLAVE_ONLY                  LIN_TAB_SlaveOnly
```

  In the example below the lingen tool has found the SlaveOnly table in the LDF file:

```
  SlaveOnly {
  //the Slave Diagnostic Interleaved Mode (mandatory)
  //the default scheduled table in Diagnostic Only Mode
  //and no active transmission from master node must be sent
  SlaveResp delay 20 ms ;
}
```

- If the lingen tool does not find any table complying with the mandatory requirements, it does not include the LIN_SLAVE_ONLY define statement. This will cause a compilation error since a master node must support the Interleaved Diagnostics schedule Mode which is mandatory (see also *Master node supporting either Interleaved diagnostics or Diagnostics-only mode*).

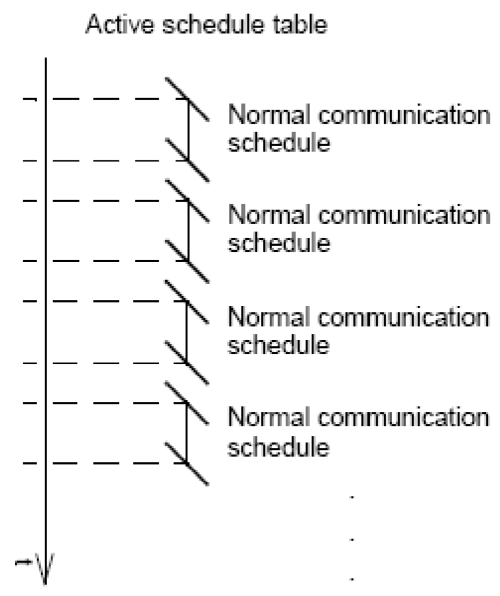
Figure 5. Interleaving of a diagnostic slave response schedule table


Active schedule table
Normal communication schedule
Diagnostic slave response schedule
Normal communication schedule

No diagnostic schedule tables executed by the master node when no diagnostic communication is active

This is the master node default behavior. The master application must not switch to tables such as the LIN_TAB_SlaveOnly or LIN_TAB_MasterOnly tables described in the previous examples before it receives a diagnostic request from an external tool (see *Figure 6*).

Figure 6. No diagnostic communication


Active schedule table
Normal communication schedule
Normal communication schedule
Normal communication schedule
Normal communication schedule

Master node supporting either Interleaved diagnostics or Diagnostics-only mode

Upon request from an external diagnostic test tool, the master node can operate each of its connected LIN clusters either in Interleaved Diagnostics Interleaved mode or in Diagnostics-only mode(see *Figure 5*):

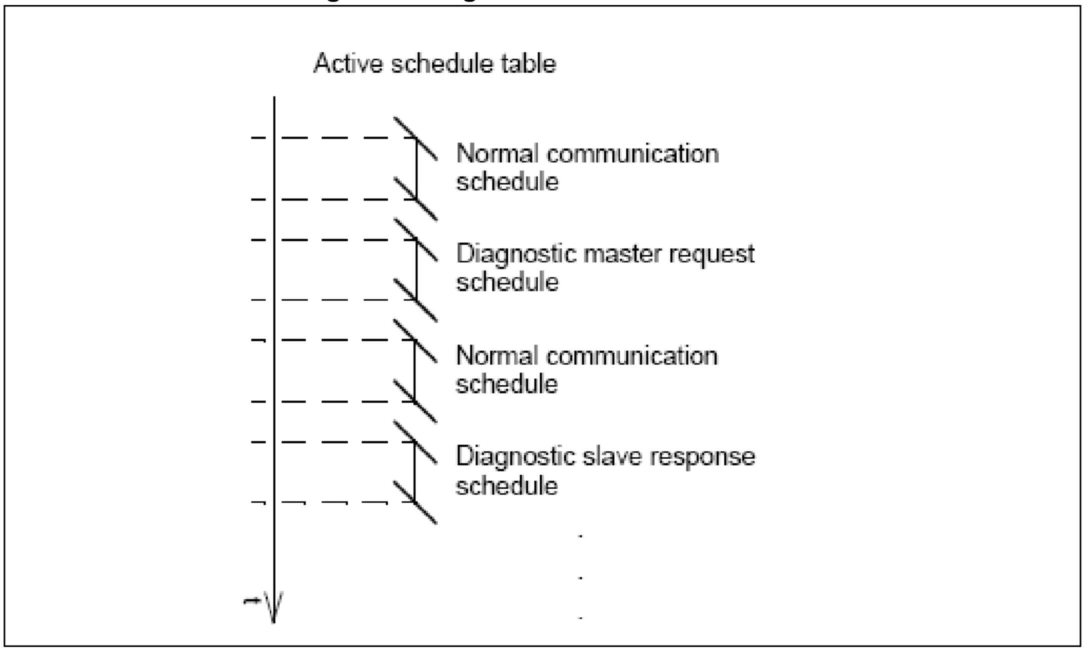
- Diagnostics Interleaved mode

  This is the master node default mode.

  Before executing diagnostic schedules, the master node completed the ongoing normal communication schedules (see *Figure 4* and *Figure 5*). After executing of a diagnostic schedule, the master node runs a normal communication schedule before executing the new diagnostic schedule (see *Figure 7*).

  In Diagnostics Interleaved mode, the master application ensures via normal communication schedule design that the time between two subsequent diagnostic schedules complies with the OEM specific diagnostic requirements.

Figure 7. Diagnostic Interleaved mode


Active schedule table
Normal communication schedule
Diagnostic master request schedule
Normal communication schedule
Diagnostic slave response schedule

The number of executions of the diagnostic master request schedule table depends on the amount of data that needs to be transmitted. It is determined by the master node in compliance with the LIN transport protocol. As an example, the schedule must be executed twice to transmit 10 user data bytes using the LIN transport protocol.

Subsequent executions of the interleaved diagnostic slave response schedule table depend on the amount of data to be transferred. They are performed by the master node until the transmissions have successfully completed or a transport protocol timeout occurs.

If a diagnostic transmission from slave node to master has started, the master node executes the diagnostic slave response schedule (even when one or several slave response frame headers have not been answered) until ((see *Figure 8*):

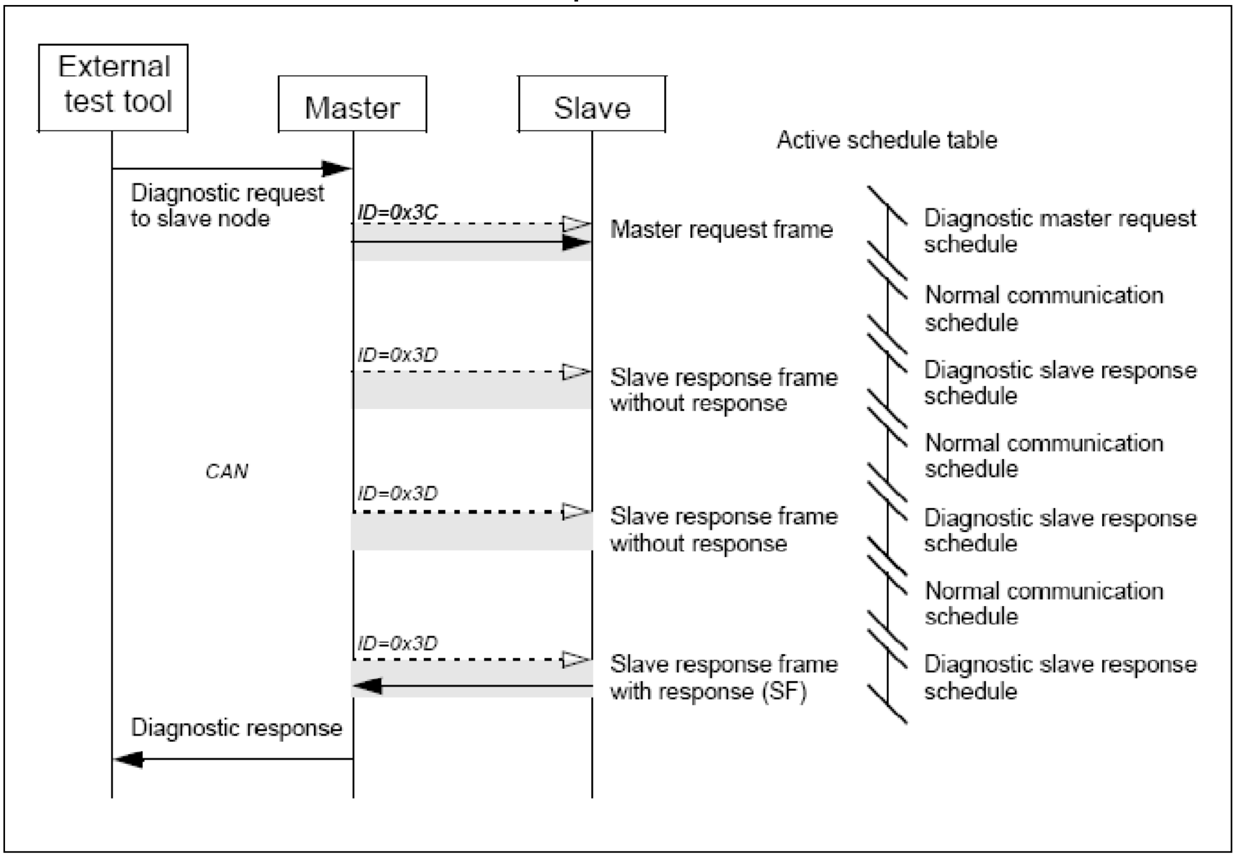
– A P2max / P2*max timeout occurs (See section 5.6 of LIN 2.1 specification);

– A transport protocol timeout occurs (See section 3.2.5 of LIN 2.1 specification).

The Diagnostic interleaved mode is configured by the Lingen tool by enabling the following define statement:

```
#define LIN_MASTER_ONLY LIN_TAB_TableName1
#define LIN_SLAVE_ONLY LIN_TAB_TableName2
```

Figure 8. Continued execution of diagnostic slave response schedule table until response received

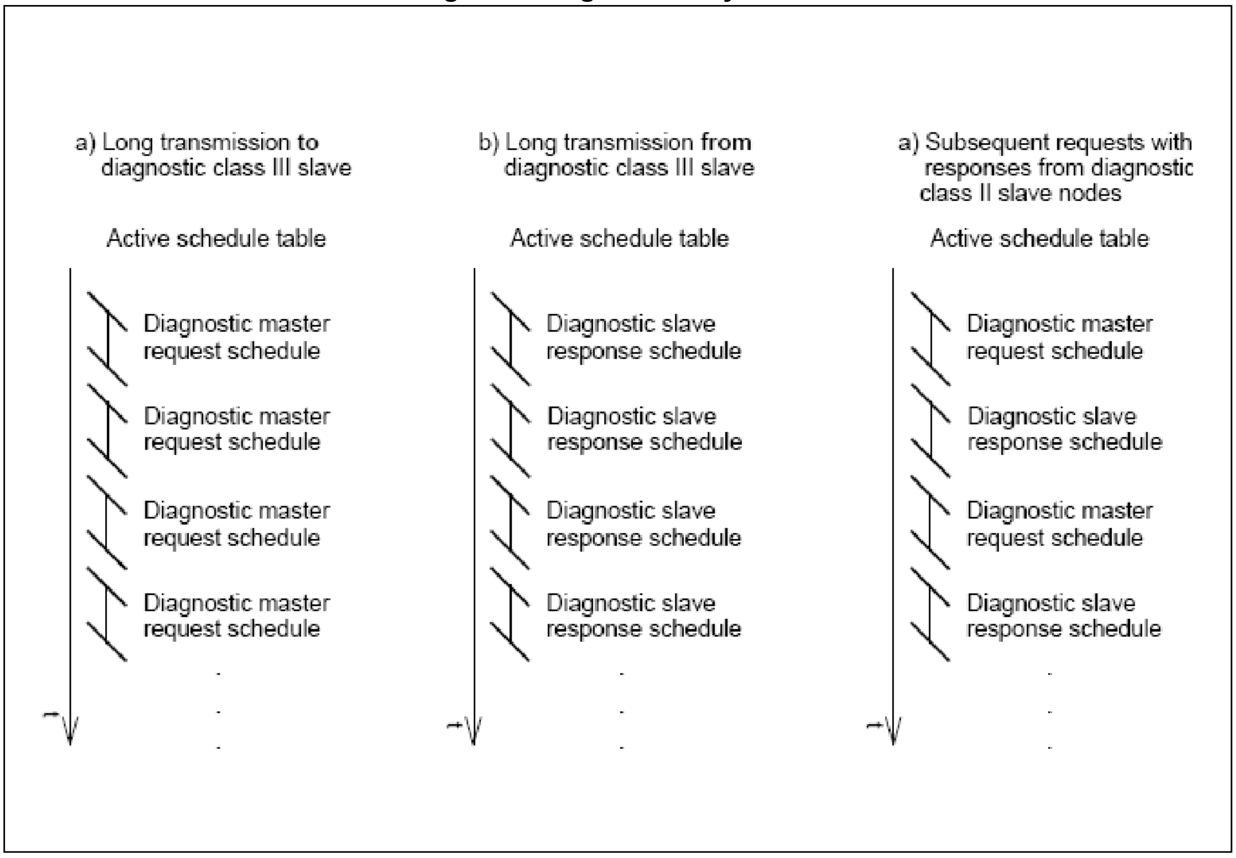

External test tool
Master
Slave
Active schedule table
Diagnostic request to slave node
ID=0x3C
Master request frame
Diagnostic master request schedule
Normal communication schedule
ID=0x3D
Slave response frame without response
Diagnostic slave response schedule
Normal communication schedule
CAN
ID=0x3D
Slave response frame without response
Diagnostic slave response schedule
Normal communication schedule
ID=0x3D
Slave response frame with response (SF)
Diagnostic slave response schedule
Diagnostic response

- Diagnostics-only mode (optional)

  In Diagnostics-only mode only the diagnostic schedules. No normal communication schedules are executed. The basic principles when using master request frame schedule tables and slave response frame schedule tables are similar as for the Diagnostics interleaved mode except that no normal communication schedules are interleaved between the diagnostic schedule tables.

  This allows to optimize diagnostic data transmission when reading slave node Identifications or during Flash memory reprogramming (see *Figure 9* for the different cases).

Figure 9. Diagnostic-only mode


a) Long transmission to diagnostic class III slave
Active schedule table
Diagnostic master request schedule
Diagnostic master request schedule
Diagnostic master request schedule
Diagnostic master request schedule
b) Long transmission from diagnostic class III slave
Active schedule table
Diagnostic slave response schedule
Diagnostic slave response schedule
Diagnostic slave response schedule
Diagnostic slave response schedule
a) Subsequent requests with responses from diagnostic class II slave nodes
Active schedule table
Diagnostic master request schedule
Diagnostic slave response schedule
Diagnostic master request schedule
Diagnostic slave response schedule

The Diagnostics-only mode is enabled and disabled by issuing a diagnostic service request from an external test tool. For example, the "Communication Control" service in UDS disables normal communication on the LIN cluster and activates the Diagnostics-only mode.

When operating in Diagnostics-only mode and no transmission is active, the master node executes diagnostic slave response schedule tables (see *Figure 10*).

The lingen tool checks if a table containing only two frames, a master request frame and a slave response frame, exists in the LDF file:

If at least one table is found (if there are more than one table, it checks only the first table found in the LDF file), it includes the following define statement in the `lin_cfg_types.h` (located under ..\demo\stm8\lin_basic_demo\master\obj\):

```
#define LIN_DIAGNOSTIC_ONLY              LIN_TAB_Diagnostic
```

In the example below, the lingen tool has found the Diagnostic table in the LDF file:

```
Diagnostic {  //This is the Diagnostic Only Mode (optional)
  MasterReq delay 20 ms ;
  SlaveResp delay 20 ms ;
}
```

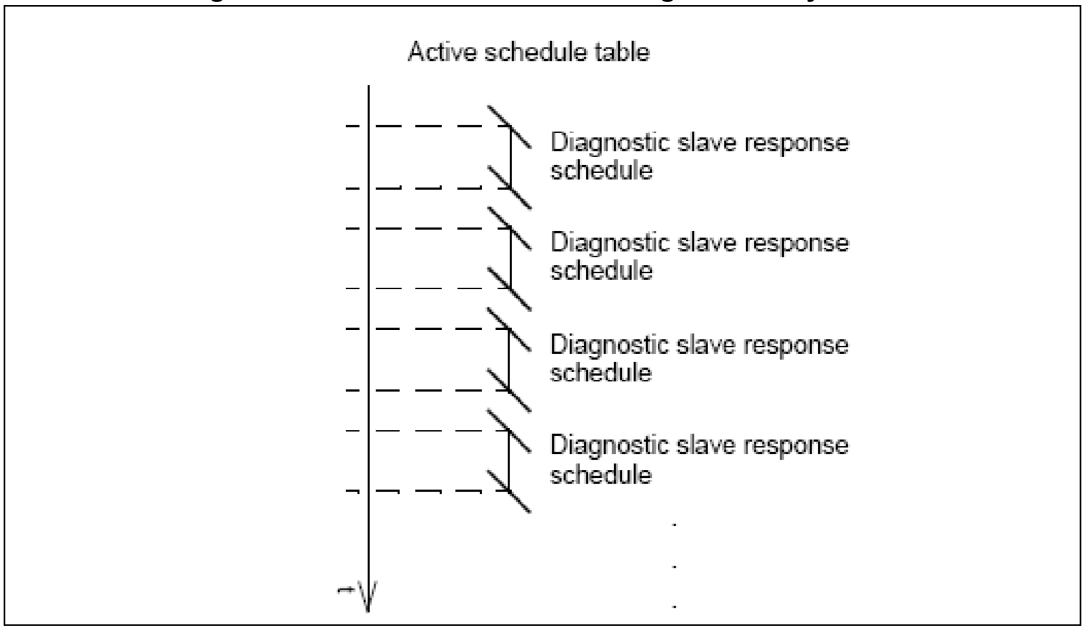
If the lingen tool does not find any table complying with the optional requirements, it does not include the LIN_DIAGNOSTIC_ONLY define statement. This does not generate any compilation since the Diagnostics-only mode is optional.

```
/**************************************************************
 * choose Diagnostic Class (mandatory for LIN 2.1 slave node)
 * LIN 2.1 slave nodes must have a Diagnostic Class value
 * defined.
 * This value can be: 1,2 or 3 (other values will involve in
 * an error).
 *************************************************************/
/*
 * DIAGNOSTIC_CLASS 1: Only the node configuration services
 * are supported.
 * The slave does not support any other diagnostic services.
 * Single frames (SF) transport protocol support is
 * sufficient.
 * Node Identification is limited to the mandatory read by
 * identifier service.
 *
 * DIAGNOSTIC_CLASS 2: Node configuration and identification
 * services are supported.
 * Full transport layer implementation is required to support
 * multi-frame transmissions.
 * Node Identification is extended to all the Read By Id
 * services. Slave-nodes will support a set of ISO 14229-1
 * diagnostic services like Node identification (SID 0x22),
 * Reading data parameter (SID 0x22) if applicable, Writing
 * parameters (SID 0x2E) if applicable.
 *
 * DIAGNOSTIC_CLASS 3: Node configuration and identification
 * services are supported.
 * Full transport layer implementation is required to support
 * multi-frame transmissions.
 * Node Identification is extended to all the Read By Id
 * services. Slave-nodes shall support all services as of
 * Class II.
 * Additionally, other services may be supported depending on
 * the features which are implemented by the slave node:
 * for example Session control (SID 0x10), I/O control by
 * identifier (0x2F), Read and clear DTC (SID 0x19, 0x14).
 * Only class 3 slave nodes can reprogram the application via
 * the LIN bus.
 *************************************************************/
#define LIN_DIAGNOSTIC_CLASS 1

 #ifdef LIN_SLAVE_NODE
   #ifndef LIN_DIAGNOSTIC_CLASS
     #error "For a LIN 2.1 slave node, LIN Diagnostic Class is
             mandatory and must be defined!"
   #endif

   #if ((LIN_DIAGNOSTIC_CLASS < 1) ||
        (LIN_DIAGNOSTIC_CLASS > 3))
     #error "LIN 2.1 Diagnostic Class value can be 1,2 or 3!"
   #endif
 #else
   #if ((!defined(LIN_MASTER_ONLY)) ||
        (!defined(LIN_SLAVE_ONLY)))
     #error "A LIN 2.1 master node must support the Interleaved
Diagnostics schedule Mode (mandatory)!"
   #endif
 #endif

#endif /* LIN_21 */


/**************************************************************
 *
 * LIN TP features. Select by define'ing.
 * TP is disabled by default
 *
 **************************************************************/


/**************************************************************
 * the cooked diagnostic TP
 **************************************************************/
#undef LIN_INCLUDE_COOKED_TP


/**************************************************************
 * the raw diagnostic TP
 **************************************************************/
#undef LIN_INCLUDE_RAW_TP
```

For the Raw TP the size of the Rx and Tx FIFO stack can be configured using the definition:

```
/**************************************************************
 * define the stack size of the raw tp fifo stacks
 * (in numbers of frames)
 **************************************************************/
#define LIN_DIAG3_FIFO_SIZE_MAX                        1
```

Figure 10. Default schedule in the Diagnostic Only mode


Active schedule table
Diagnostic slave response schedule
Diagnostic slave response schedule
Diagnostic slave response schedule
Diagnostic slave response schedule

4.4.6 Callback functions

Interrupt callback functions

Two callback functions must also be provided by the user. They allow the LIN driver to enable or disable system interrupts. The function prototypes are described below together with their implementation. They can be located in the `lin_def.c` file located in the user configuration directory. The function prototypes are defined as follows:

Table 52. Disable Interrupts

l_sys_irq_disable	
Prototype	`l_irqmask l_sys_irq_disable (void);`
Availability	Master and slave nodes
Description	This function achieves a state in which no interrupts from the LIN communication can occur
Parameters	None
Return value	Interrupt mask describing the state of the interrupts when this function is called.

Table 53. Restore Interrupts

l_sys_irq_restore	
Prototype	`void l_sys_irq_restore (l_irqmask irqMask);`
Availability	Master and slave nodes
Description	This function restores the interrupt level identified by the irqMask parameter
Parameters	`irqMask`: mask containing the state of the interrupts to be restored
Return value	None

An implementation example is given in *Section 6.4: Example implementation of IRQ callbacks (master and slave)*.

The LIN driver uses these user-defined functions each time an API function is called. Interrupts are disabled when entering the API function and re-enabled before returning. This means that the callback functions provided for the driver must handle nested calls. If an API function is called and interrupts have already been disabled, interrupts shall only be re-enabled at the outermost call to the `l_sys_irq_restore()` function.

The `SuspendOSInterrupts` and `RestoreOSInterrupts` OSEK functions described in *Section 6.4* meet this requirements.

Protocol switch callback function

The application can change the protocol for a given LIN interface by using the `l_protocol_switch()` function with its parameter set to disable LIN. When LIN is disabled, a callback function is used by the ISR as an entry to the alternative protocol. This callback function must be provided by the user and comply with the prototype given in *Table 54*.

To enable the use of this feature, include the following define statement in `lin_def.h`:

```
#define LIN_PROTOCOL_SWITCH
```

Table 54. Protocol switch function callback

l_protocol_callback_iii	
Prototype	`void l_protocol_callback_iii (void);` where `iii` denotes configured interfaces SCI0 .. SCIn
Availability	Master and slave nodes
Description	This function provides the entry point to an alternative protocol handler called by the ISR when an interrupt occurs after the application has called the `l_protocol_switch()` API function to disable LIN. The callback is interface specific and so for each interface the user must provide a corresponding callback.
Parameters	None
Return value	None

Diagnostic callback functions (slave only)

Two callback functions can be configured for slave nodes to be able to use the `ld_read_by_id()` diagnostic service (when used for user-defined ids) and the `ld_data_dump()` service. These have the following prototype forms:

For these two callbacks, empty implementations are included in the file `lin_def.c`. These must be replaced by the user to provide the functionality required.

Table 55. ld_read_by_id callback

ld_readByIdCallback	
Prototype	`l_bool ld_readByIdCallback(l_u8 id, l_u8* pBuffer);`
Description	This function provides a response in accordance with the id request sent from the master. This callback will be called by the slave driver when the id given lies in the range allocated for user-defined ids i.e. 32 – 63. If a non-zero value is returned, the driver sends the buffer back to the master. The user application receives the complete frame buffer and can write up to 5 bytes response into the buffer starting at location `pBuffer[3]`. The application is responsible for setting the PCI byte (`pBuffer[1])` correctly. It must be set to the number of valid data bytes written plus one. Since the buffer is pre-set to 0xFF, the unused bytes will have this value.
Parameters	`id`: the id sent by the master `pBuffer`: the buffer into which the user application must write a response
Return value	non-zero if buffer to be sent back to master

Table 56. ld_data_dump callback

ld_dataDumpCallback	
Prototype	`l_bool ld_dataDumpCallback(l_u8* sendBuf, l_u8* recBuf);`
Description	Provides a response to a data dump request from the master. This callback is called by the slave driver and must write 5 bytes in response starting at `recBuf[0]` and then return non-zero to the driver. If no response is to be sent then return zero to the driver. Note: When a response is to be returned, 5 bytes will always be transferred.
Parameters	`sendBuf`: the buffer sent by the master `recBuf`: the buffer into which the application can write its response
Return value	non-zero if the driver is to send a response back to the master

Baud rate detection callback function (slave only)

Baud rate detection for slave nodes can be configured. When this feature is enabled, a callback function is invoked by the LIN driver when an incorrect baud rate is detected. From this callback, the application can then call the `l_change_baudrate()` function to reduce the current baud rate.

The application starts with the highest possible baud rate and then repeatedly reduces the baud rate until communication is established.

This feature must be enabled in `lin_def.h` by:

```
#define LIN_BAUDRATE_DETECT
```

Table 57. Baud rate detection callback

l_baudrate_callback_iii	
Prototype	`void l_baudrate_callback_iii (l_u16 baudrate);`
Description	This function sets the baud rate for the given interface by calling the `l_change_baudrate()` API function. This callback is called if an incorrect (too high) baud rate is detected by the slave driver. The callback is interface specific and so for each interface the user must provide a corresponding callback.
Parameters	`ifc`: interface handle `baudrate`: the baud rate currently detected (i.e. the incorrect baud rate) on the interface
Return value	None

4.5 Interrupt configuration

The STMicroelectronics LIN driver provides functions to handle the interrupts occurring when a character is received or transmitted on a specific interface. These functions must be called from the user-defined interrupt handlers which are called when an interrupt is triggered. Since the driver functions completely handle the interrupts, any further handling should not be implemented by the user.

These driver functions are architecture dependent. As an example, the application user should not call the Tx handler when only the Rx handler is used. Refer to *Section 7: Architecture notes* for exact details.

The functions have the following interfaces:

Table 58. Handler for character rx

l_ifc_rx	
Prototype	`void l_ifc_rx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Description	This function handles the interrupt when a character is received
Parameters	`ifc`: the interface on which the interrupt occurred
Return value	None
Interface specific prototype	`void l_ifc_rx_iii (void);`
Description	handles the interrupt for the interface given by `iii`
Include	lin.h

Table 59. Handler for character tx

l_ifc_tx	
Prototype	`void l_ifc_tx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Description	This function handles the interrupt
Parameters	`ifc`: the interface on which the interrupt occurred
Return value	None
Interface specific prototype	`void l_ifc_tx_iii (void);`
Description	handles the interrupt for the interface given by `iii`
Include	lin.h

5 Lingen control file specifications

5.1 Lingen control file

The lingen tool uses a specific control file to determine which interfaces have to be configured and which LDF file is associated with the selected interfaces. The following sections specify the content of this control file.

C/C++ style comments can be used in the lingen control file. They are not described in the sections below. An explanation of the syntax is given in *Section 5.2: Syntax specification*.

5.1.1 File definition

```
<lingen_control_file> ::=Interfaces
            {
              [<interface_spec>]
            }
            (LIN_use_default_frame_ids ;)
```

5.1.2 Interface specification$

The interface is defined as follows:

```
<interface_spec> ::= <interface_id>:<ldf_file_name> (,<tag_id>);
<interface_id>       ::=   identifier
```

The driver currently supports the interface identifiers from SCI0 to SCI9. However, the identifiers should match the specific interface as defined in *Section 7: Architecture notes*.

`<ldf_file_name>` ::= string

The string should specify the filename of the LIN description file. It must include the full path specification for the file. It can be either the relative path to the file or the filename if the file is located in the current directory. It is recommended to specify the full path especially if the lingen tool is executed from a makefile:

`<tag_id>` ::= “tag_identifier”

The tag identifier is intended for avoiding naming conflicts and will be concatenated with C identifiers internally. Therefore it should include a legal sequence of characters following C identifier rules (see *Section 4.3.1: Cluster description* for an example).

5.1.3 Default frame Identifiers

The **LIN_use_default_frame_ids** optional keyword is intended to be used with slave nodes only. If included, the default frame identifiers (IDs) given in the LDF apply to all slave frames. Slave nodes can start communication without having been configured by the master.

5.2 Syntax specification

The following syntax has been used for the specification of the lingen control file. It must be consistent with the syntax used for specifying LIN description files as described in the LIN2.1 and LIN 2.0 standards for master and slave nodes, respectively.

Table 60. Syntax description

Symbol	Description
::=	Is defined to be
<>	Delimits an object specified later
[ ]	Delimits an object that shall appear one or more times
( )	Delimits an object that is optional
bold text	Keyword or symbol, use directly as given
identifier	Identifies an object, c-style identifier rules apply
string	Any c-style string
tag_identifier	Use to extend identifiers, c-style identifier rules apply

6 Examples

6.1 Sample control file for lingen

The lingen tool uses a specific control file to determine which interfaces have to be configured and which LDF file is associated with the selected interfaces. This example applies to a master configuration with two interfaces:

Master node

```
//
// lingen control file defining two interfaces
//
Interfaces
{
  SCI0: "/home/LIN/src/lin_config/lin_sci0.ldf", "IFC0";
  SCI1: "/home/LIN/src/lin_config/lin_sci1.ldf", "IFC1";
}
```

Slave node

```
//lingen control file defining one interface
//
Interfaces
{
  SCI0: "/home/LIN/src/lin_config/lin_sci0.ldf", "IFC0";
}

//
// specify that slave nodes will start with default frame IDs
//
LIN_use_default_frame_ids;
```

6.2 LIN 2.0 LDF example (master and slave)

The format and full details for a LIN description file are given in the LIN configuration language specification section of LIN 2.0 standard. This example shows a configuration with one master and two slave nodes. The first slave node is set up according to LIN 2.0, and the second according to LIN 1.2.

```
//
// global definitions
//
LIN_description_file;
LIN_protocol_version = "2.0";
LIN_language_version = "2.0";
LIN_speed = 19.2 kbps;

//
// node definition: participating nodes
//
```

```
Nodes
{
  Master: master, 10 ms, 0.1 ms;
  Slaves: slave1, slave2;
}

//
// signal definition: standard signals
//
Signals
{
  //
  // Master node signals
  //
  // the DIP state of the Master node board
  // masterDIPState1: DIPs 1 and 2
  // masterDIPState2: DIPs 3 to 6
  // masterDIPState3: DIP 7
  // masterDIPSTate4: DIP 8
  masterDIPState1: 2, 0, master, slave1, slave2;
  masterDIPState2: 4, 0, master, slave1, slave2;
  masterDIPState3: 1, 0, master, slave1, slave2;
  masterDIPState4: 1, 0, master, slave1, slave2;


  // slave1 node signals
  //
  // the DIP state of the slave1 node board
  // slave1DIPState1: DIPs 1 and 2
  // slave1DIPState2: DIPs 3 to 6
  // slave1DIPState3: DIP 7
  // slave1DIPSTate4: DIP 8
  slave1DIPState1: 2, 0, slave1, master, slave2;
  slave1DIPState2: 4, 0, slave1, master, slave2;
  slave1DIPState3: 1, 0, slave1, master, slave2;
  slave1DIPState4: 1, 0, slave1, master, slave2;
  slave1Toggle   : 1, 0, slave1, master;

  //
  // slave2 node signals
  //
  // the DIP state of the slave1 node board
  // slave2DIPState1: DIPs 1 and 2
  // slave2DIPState2: DIPs 3 to 6
  // slave2DIPState3: DIP 7
  // slave2DIPSTate4: DIP 8
  slave2DIPState1: 2, 0, slave2, master, slave1;
  slave2DIPState2: 4, 0, slave2, master, slave1;
  slave2DIPState3: 1, 0, slave2, master, slave1;
  slave2DIPState4: 1, 0, slave2, master, slave1;
  slave2Toggle   : 1, 0, slave2, master;
```

```
    // error signal
    errorSignalSlave1: 1, 0, slave1, master;
    errorSignalSlave2: 1, 0, slave2, master;
  }

  //
  // signal definition: diagnostic signals
  // (optional but recommended)
  //
  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;
  }


  //
  // frame definition: unconditional frames
  //
  Frames
  {
    //
    // frames published by the master
    //
    frmM1: 0, master, 2
    {
      masterDIPState1, 0;
    }

    frmM2: 1, master, 1
    {
      masterDIPState2, 0;
    }

    frmM3: 2, master, 1
    {
      masterDIPState3, 0;
      masterDIPState4, 1;
    }

    //
    // frames published by slave1
    //
    frmS11: 20, slave1, 2
    {
      slave1DIPState1, 0;
    }

    frmS12: 21, slave1, 2
    {
      slave1DIPState2, 8;
      slave1Toggle, 12;
    }

    frmS13: 22, slave1, 1
    {
      slave1DIPState3, 0;
      slave1DIPState4, 1;
      errorSignalSlave1, 2;
    }

    //
    // frames published by slave2
    //
    frmS21: 40, slave2, 2
    {
      slave2DIPState1, 0;
    }

    frmS22: 41, slave2, 2
    {
      slave2DIPState2, 8;
      slave2Toggle, 12;
    }

    frmS23: 42, slave2, 1
    {
      slave2DIPState3, 0;
      slave2DIPState4, 1;
      errorSignalSlave2, 2;
    }
  }

  //
  // frame definition: diagnostic frames
  // (optional but recommended)
  //
  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;
    }
  }

  //
  // node definition: node attributes
  //
  Node_attributes
  {
    slave1
    {
      LIN_protocol = 2.0;

      // the startup diagnostic address
      configured_NAD = 1;

      // product id is used to uniquely identify a slave node
      // within a cluster
      product_id = 0x1234, 0x5678, 0x03;

      // definition of the error signal of the slave
      response_error = errorSignalSlave1;

      // the list of configurable frames
      // all frames to be processed by the slave node
      // must get a message id in this section
      configurable_frames
      {
          frmM1    = 0x01;
          frmM2    = 0x02;
          frmM3    = 0x03;
          frmS11   = 0x04;
          frmS12   = 0x05;
```

```
            frmS13    = 0x06;
            frmS21    = 0x08;
            frmS22    = 0x09;
            frmS23    = 0x10;
        }
      }

      slave2
      {
        LIN_protocol = 1.2;

        // the startup diagnostic address
        configured_NAD = 2;
      }
    }

    //
    // schedule table definitions
    //
    Schedule_tables
    {
      //
      // this schedule table will configure the slave1 node to
      // participate in LIN communication
      //
      schTabConfig
      {
         AssignFrameId{slave1, frmM1   } delay 20 ms;
         AssignFrameId{slave1, frmM2   } delay 20 ms;
         AssignFrameId{slave1, frmM3   } delay 20 ms;
         AssignFrameId{slave1, frmS11  } delay 20 ms;
         AssignFrameId{slave1, frmS12  } delay 20 ms;
         AssignFrameId{slave1, frmS13  } delay 20 ms;
         AssignFrameId{slave1, frmS21  } delay 20 ms;
         AssignFrameId{slave1, frmS22  } delay 20 ms;
         AssignFrameId{slave1, frmS23  } delay 20 ms;
      }

      //
      // the normal signals are transferred using this schedule
      // table
      //
      schTab1
      {
        frmM1    delay 20 ms;
        frmS11   delay 20 ms;
        frmS21   delay 20 ms;

        frmM2    delay 20 ms;

        frmM3    delay 20 ms;
        frmS13   delay 20 ms;
        frmS23   delay 20 ms;
      }
    }
```

6.3 LIN 2.1 LDF example (master and slave node)

The format and full details of a LIN description file are given in the LIN configuration language specification section of LIN 2.1 standard. This example shows a configuration with one master and two slave nodes. Both slave nodes are set up according to LIN 2.1.

```
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;
}

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;
    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“;
    physical_value, 1, 254, 1, 100, “lux“;
    logical_value, 255, “error“;
  }
}

Signal_representation
{
  Dig2Bit: InternalLightsRequest;
  ErrorEncoding: RSMerror, LSMerror;
  FaultStateEncoding: IntError;
  LightEncoding: RightIntLightsSwitch, LefttIntLightsSwitch;
}
```

6.4 Example implementation of IRQ callbacks (master and slave)

The following example describes OSEK implementation:

```
l_irqmask l_sys_irq_disable (void)
{
   SuspendOSInterrupts();

   return 0;
}

void l_sys_irq_restore (l_irqmask irqmask)
{
   ResumeOSInterrupts();

   return ;
}
```

The user can locate these implementations in an application specific file that includes the corresponding operating system header file.

For an OSEK implementation, include `os.h`.

7 Architecture notes

These architecture notes explains how to configure the LIN driver to run on STM8 microcontrollers.

7.1 CPU frequency

The CPU frequency must be configured in `lin_def_stm8.h` as follows:

```
/*****************************************************************
 * define frequency of the cpu
****************************************************************/
#define LIN_BOARD_CPU_FREQ_HZ   8000000
```

7.2 Interfaces

Two serial interfaces are supported by STM8 microcontrollers, SCI1 and SCI2. The interface is configured in the lingen control file as described in *Section 5: Lingen control file specifications*.

In addition to the standard support for LIN, the interfaces offer extra functionalities:

- For a master node: both interfaces support the sending of BREAK directly. This is a fixed 13-bit BREAK signal and it is always used.
- For a slave node: the SCI1 interface allows to auto synchronize with the current bus baud rate when a SYNCH is detected (to within 15% of the nominal rate). This functionality can be selected by defining/undefining the following variable in `lin_def_stm8.h`:

```
#undef LIN_SLAVE_LINSCI_AUTOSYNC
```

or

```
#define LIN_SLAVE_LINSCI_AUTOSYNC
```

7.3 Timers

The timer used by the LIN driver can be either a hardware or a software timer. It is configured by the user application in `lin_def_stm8.h` by defining or undefining the `LIN_USE_HARDWARE_TIMER` variable.

```
The following example shows the selection of a hardware timer:
/*****************************************************************
 *
 * Activate the use of a hardware timer for LIN via
 * "#define LIN_USE_HARDWARE_TIMER"
 * Deactivate the use of a hardware timer for LIN via
 * "#undef  LIN_USE_HARDWARE_TIMER"
 *
****************************************************************/
#define LIN_USE_HARDWARE_TIMER
```

STM8 microcontrollers only support an 8-bit system timer (TIM4): it is referred to as 1. The timer must be configured as follows:

```
#ifdef LIN_USE_HARDWARE_TIMER
/*******************************************************************
***
  *
  * choose the timer used by the LIN driver
  * (valid values depend on the architecture used)
  *

********************************************************************
**/
  #define LIN_TIMER                    1
#endif
```

Note that the driver is delivered with the hardware timer configured as default timer in `lin_def_stm8.h`:

```
#define LIN_USE_HARDWARE_TIMER
```

When the hardware timer is used, its prescaler and reload values must be configured. For the 4 most common configurations, these values are already given in the `lin_stm8.h` file (see *Table 61*).

Table 61. Hardware timer configuration

CPU frequency (Hz)	Time base (ms)	Prescaler value	Reload value	Note
16,000,000	2	7	250	16 MHz/ 2^7/250 → 2 ms timebase
8,000,000	2	7	125	16 MHz/ 2^7/125 → 2 ms timebase
16,000,000	1	6	250	16 MHz/ 2^6/250 → 1 ms timebase
8,000,000	1	5	250	16 MHz/ 2^5/250 → 1 ms timebase

The driver uses a special callback function, `l_timerISR()`, to handle hardware timer interrupts. The user application must implement and bind a timer interrupt service routine. The user ISR will be called at the expiry of the hardware timer LIN_TIMER and it must call `l_timerISR()`. The timer must not be used for any other purposes.

Note: The user ISR should not handle the interrupt nor clear any flags – it should only call the driver interrupt service routine.

The description of `l_timerISR()` is given below:

Table 62. Hardware timer interrupt service routine

l_timerISR	
Prototype	`void l_timerISR (void);`
Description	This function handles the hardware timer interrupt, clears the interrupt flag, restarts the timer, and handles any software timeouts that may have occurred.
Include	lin.h
Parameters	None
Return value	None

7.4 Interrupt function configuration

Section 3.3.4: Interface Management provides a general description of the Rx and Tx interrupt functions defined by the LIN 2.x standard.

For STM8 architecture, only the Rx interrupt is used. The user defined interrupt handler must therefore call the `l_ifc_rx()` function. The `l_ifc_tx()` function should not to be called as it has no effect.

7.5 Using memory page 0

To improve the speed and the driver code size, some variables can be located in page 0 of the memory area. The user application must specify how many bytes to allocate for this purpose in the `lin_def_stm8.h` file. This is done as follows:

```
/*****************************************************************
 * Optimisation for STM8
 * Give size available in fastest/most efficient memory area
 * you can spare for the LIN driver in bytes
*****************************************************************/
#define LIN_ZERO_PAGE_SIZE                              5
```

The default value is of 5 bytes. The driver automatically selects which variables can be located in the available space, and tags them accordingly for the linker.

Additionally, the user can also choose to locate some additional data in page 0. This can be configured by changing the following settings in `lin_def_stm8.h`:

```
#undef LIN_TX_FLAGS_IN_ZERO_PAGE
#undef LIN_RX_FLAGS_IN_ZERO_PAGE
#undef LIN_CHANGED_FLAGS_IN_ZERO_PAGE
#undef LIN_FRAME_BUFFER_IN_ZERO_PAGE
#undef LIN_FRAME_IDS_IN_ZERO_PAGE
```

The space required to locate these data in page 0 is not automatically calculated by the driver, and must be defined by the user.

7.6 Toolchain support

The LIN driver for the STM8 has been developed with *GNU make v3.74* and *Cosmic compiler version 4.2.8* with the following components:

```
--------------------------------
STM8 COSMIC C Compiler
Version: 4.2.8 Date: 03 Dec 2008
--------------------------------
COSMIC Software STM8 C Cross Compiler V4.2.8 - 03 Dec 2008 - Win32-F
COSMIC Software STM8 C Parser V4.8.12 - 25 Nov 2008 - Win32-F
COSMIC Software STM8 Code Generator V4.2.8 - 03 Dec 2008 - Win32-F
COSMIC Software STM8 Optimizer V4.2.8 - 03 Dec 2008 - Win32
COSMIC Software STM8 Macro-Assembler V4.5.6 - 13 Oct 2008 - Win32-F
COSMIC Software Linker V4.7.4.1 - 27 Oct 2008 - Win32-F
COSMIC Software Hexa Translator V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software Absolute Listing V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software Librarian V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software Absolute C Listing V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software Object Inspector V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software Print Debug Info V4.3.13 - 27 Jun 2008 - Win32
COSMIC Software ELF/DWARF Converter V4.5.27 - 18 Nov 2008 - Win32
COSMIC Software IEEE695 Converter V7.0.7 - 23 Mar 2006 - Win32
```

To optimize the driver code size, the compiler `+compact` switch can be used. However, in this case, as in earlier Cosmic compiler versions, the driver may not be reliable. This risk must be considered by the user when choosing this option.

The driver can be built using stack or non-stack memory model. Since most driver functions need to be re-entrant, the `@stack` qualifier has been added to each one. As a result the memory model has little impact on the driver.

Using a memory model switch to build the driver with all global variables placed in memory page 0 is unlikely to work since the space required to store these variables probably exceeds the available space.

Note: A master node using multiple interfaces can only be compiled using the physical stack option.

If the driver has been configured to support the conditional change NAD function by defining LIN_INCLUDE_COND_CHANGE_NAD in the `lin_def.h` file, the memory model chosen when building the **slave** driver must use a physical stack.

8 Revision history

Table 63. Document revision history

Date	Revision	Changes
05-Dec-2012	1	Initial release replacing UM0941 Rev 1 14-Jun-2010 Changed document title. Updated *Introduction on page 1*. Added *Applicable products on page 1*.
16-May-2014	2	Updated the document title and the *Introduction* on the cover page to include STMicroelectronics LIN software package reference (STSW-STM8A-LIN). Removed the table “Applicable products” on the cover page. No other changes in the content.