UM1729
User manual

LIN 2.1 Driver Suite
Master Node User's Guide

Introduction

Purpose

The purpose of this user guide is to provide application programmers with detailed information about the use of the STMicroelectronics LIN 2.1 driver (STSW-SPC56002FW). A detailed description of the API implemented is provided together with some examples of important files required for getting started and for driver configuration.

Scope

The STMicroelectronics implementation is in accordance with the LIN 2.1 specification [1].

User profile

It is expected that users of this driver are familiar with the concept of networks and in particular LIN. As the STMicroelectronics driver is implemented in the C programming language, users should be experienced in the development of applications in C.

References

- [1] LIN specification package, revision 2.0, 23-September-2003
- [2] LIN specification package, revision 2.1, 24-November-2006

Contents

1 **Abbreviations** 6

2 **Overview** 7

2.1 LIN concept 7

2.2 LIN communication 7

2.3 Signal management 8

2.4 Using the driver 8

2.5 Driver version 9

3 **API** 10

3.1 Data 10

3.2 Functions 10

3.3 CORE API 10

3.3.1 Driver and cluster management 10

3.3.2 Signal interaction 11

3.3.3 Notification 13

3.3.4 Interface management 14

3.4 Diagnostic API 20

3.4.1 Node configuration (diagnostic) specific API 20

3.4.2 Diagnostic transport layer 21

3.4.3 Diagnostic transport layer: RAW API 23

3.4.4 Diagnostic transport layer: COOKED API 24

3.5 Slave specific API 26

3.5.1 Interface management 26

3.6 STMicroelectronics extensions 27

3.7 Implementation Notes 28

3.7.1 API data types 28

3.7.2 Notification flags 28

4 **Driver configuration** 30

4.1 File and directory structure 30

4.2 Makefiles 30

4.2.1 Top-level makefile 30

4.3 Cluster configuration 32
4.3.1 Cluster description 32
4.3.2 Lingen 33
4.4 User configuration 34
4.4.1 Timers 34
4.4.2 General settings 35
4.4.3 Diagnostic functions configuration 39
4.4.4 Diagnostic class 40
4.4.5 Callback functions 43
4.5 Interrupt configuration 45

5 Specification of lingen control file format 47
5.1 lingen control file 47
5.1.1 File definition 47
5.1.2 Interface specification 47
5.1.3 Default frame IDs 47
5.2 Specification syntax 48

6 Examples 49
6.1 Sample control file for lingen 49
6.2 LIN 2.0 LDF example 49
6.3 LIN 2.1 LDF example 56
6.4 Example implementation of IRQ callbacks 60

7 Revision history 61

List of tables

Table 1. Description of abbreviated forms . . . 6
Table 2. LIN naming conventions . . . 9
Table 3. System initialization . . . 10
Table 4. Scalar signal read . . . 11
Table 5. Scalar signal write . . . 11
Table 6. Byte array read . . . 12
Table 7. Byte array write . . . 12
Table 8. Test flag . . . 13
Table 9. Clear flag . . . 13
Table 10. Initialise interface . . . 14
Table 11. Wake-up . . . 15
Table 12. Interface control . . . 15
Table 13. Character received notification . . . 17
Table 14. Character transmitted notification . . . 18
Table 15. Read interface status . . . 18
Table 16. Description of l_ifc_read_status returned value . . . 20
Table 17. Read configuration . . . 20
Table 18. Set configuration . . . 21
Table 19. Initialization . . . 21
Table 20. Put raw frame . . . 23
Table 21. Get raw frame . . . 23
Table 22. Query raw transmit-queue status . . . 23
Table 23. Query raw receive-queue status . . . 24
Table 24. Send message . . . 24
Table 25. Receive message . . . 25
Table 26. Get transmit-queue status . . . 25
Table 27. Get receive-queue status . . . 25
Table 28. Slave synchronise . . . 26
Table 29. Read by ID callout . . . 26
Table 30. Software Timer Function . . . 27
Table 31. Protocol Switch . . . 27
Table 32. Set baud rate . . . 28
Table 33. Raw Tx Frame Delete . . . 28
Table 34. Directory structure . . . 30
Table 35. Top-level makefile predefined variable definition . . . 31
Table 36. Disable Interrupts . . . 43
Table 37. Restore Interrupts . . . 43
Table 38. Protocol switch function callback . . . 44
Table 39. ld_read_by_id callback . . . 44
Table 40. ld_data_dump callback . . . 44
Table 41. Baud rate detection callback . . . 45
Table 42. Handler for character rx . . . 46
Table 43. Handler for character tx . . . 46
Table 44. Syntax description . . . 48
Table 45. Document revision history . . . 61

List of figures

Figure 1. Master-slave node communications 8
Figure 2. Lingen workflow 33

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 in order to produce low-cost automotive networks that complement existing networks such as CAN. It is based on a single-wire serial communication using SCI (UART) interfaces that are commonly available on microcontrollers. LIN is intended to be used together with CAN forming a hierarchical vehicle network. Generally, it is used for local sub-systems where a low bit rate (up to 20kbit/s) is acceptable and no safety-critical functions are required. Typically these applications are used for car body electronics e.g. doors, seats, air conditioning etc. These sub-units are connected as units of a CAN network using a LIN/CAN gateway.

A LIN cluster comprises one master node and one or more slave nodes. A special feature of the LIN concept is the synchronisation of slave nodes via the bus which means that low-cost nodes without quartz clocking can be implemented. Also, access to the bus is controlled by the master node and so no collision management is needed in the slave nodes. This also means that a worst-case transmission time for signals can be guaranteed.

The slave nodes do not use any information about the LIN cluster. This means that further slave nodes can be added to the LIN without requiring a change in the existing slave nodes. The master node requires information for 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 communication

In abstract terms, communication between the application software in LIN nodes is achieved by exchange of signals. The driver software, responsible for achieving signal exchange at a lower level, exchanges information between nodes in terms of frames. Therefore, the driver is responsible for taking application signals and packing these into the data section of a frame and for initiating transfer. The frames are then transferred via the serial interface of the controller. Using this communication technique the reading and writing of signals is asynchronous to the transfer of frames. An overview of communication between LIN nodes is depicted in *Figure 1*.

All transfers are initiated by the master node -- a slave node will only transmit when required to do so. The master node sends a message header for a frame. The frame body can be sent either by the master or by a slave node. Together, the header and the frame body form one complete message frame. Since the publisher for any given frame is configured before system build, there is only one possible node that will send the frame body.

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

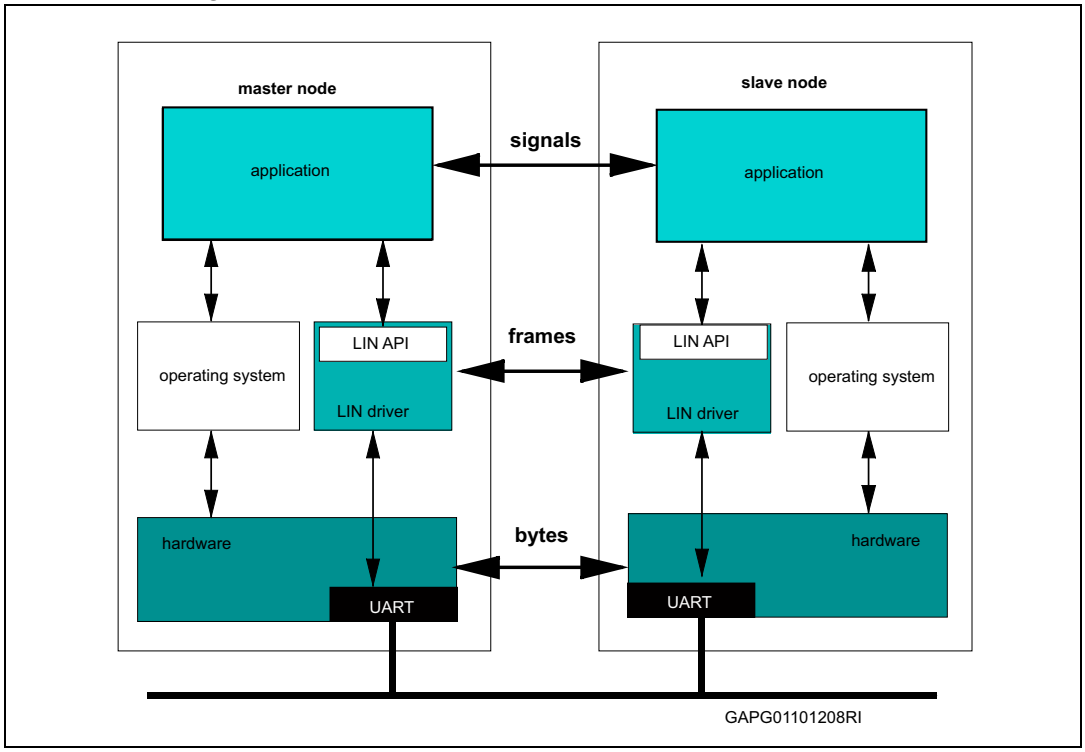
- from a master node to one or more slave nodes
- from a slave node to the master and/or to other slave nodes

This means that communication is possible from slave to slave without routing through the master and that the master can broadcast to all slaves in 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 may be active at a time. Switching between tables can be carried out 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 data bytes of a frame. 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 i.e. it is always written by the same node in a cluster. Each signal that is produced by a node must be configured by the user.

A signal is either a scalar value or a byte array. A scalar signal is between 1 and 16 bits long. A one bit signal is called a boolean signal.

- Scalar signals between 2 bits and 16 bits long are treated as unsigned.
- A byte array is an array of between one and eight bytes.

Signals must be kept consistent by the driver. A partially updated 16-bit signal must never be passed to an application. Consistency between signals is the responsibility of the application.

Signals are transmitted LSB first. Scalar signals may cross a byte boundary at most once. Each byte in a byte array must be mapped to a byte in a frame by the driver.

2.4 Using the driver

The driver must first be configured and built before use. Details of the steps for general configuration are given in this document in *Section 2.5: Driver version*. Architecture specific details are provided in the relevant in *Architecture Notes* document supplied in addition to this user guide.

The STMicroelectronics driver includes the diagnostic layer as specified in [1]. 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 diagnostic functions may be selectively included in the build of the driver. This is detailed in *Section 4.4.3: Diagnostic functions configuration*.

Before using the driver functionality the driver itself must be initialised by calling the

Note: *`l_sys_init` API function. Before using any interface related functions the controller interfaces must be initialised using the `l_ifc_init` API function and then connected using the `l_ifc_connect` function.*

In addition the user should note the following naming convention that has been adopted in the STMicroelectronics driver. The following table shows the scheme adopted:

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 static functions of the API. For example, if a signal named `sigMstatus` has been configured in the LDF file then 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 then an optional tag may be specified by the user to avoid naming conflicts. This tag will be prepended to the "item name" form 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 versioned and whose version number is only updated on change. Therefore, a given **driver** version will have files with varying version numbers. The definition of file versions used to build a particular driver version is contained in the file `lin_version_control.h` in the top level source directory. This information is used to ensure that only consistent files are included in driver build.

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 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	Performs the initialisation 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 [2] section 9.2.3.3).** The call to the l_sys_init is the first call a user must use in the LIN core before using any other API functions.
Parameters	None
Return	zero if initialisation succeeded non-zero if initialisation failed

3.3.2 Signal interaction

Scalar signal read

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	Reads and returns the current value of the signal specified (see [2] section 7.2.2.2)
Parameters	`signalId` – the name of the signal to be read e.g. for the configured signal `status` then `LIN_SIGNAL_status`
Return	l_bool – boolean signal value or 0 if `signalId` invalid l_u8 – 8 bit signal value or 0 if `signalId` invalid l_u16 – 16 bit value or 0 if `signalId` 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	Sets the current value of the signal specified to the value `val` (see [2] 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

Table 5. Scalar signal write (continued)

l_bool_wr, l_u8_wr, l_u16_wr	
Return	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 whose 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	Reads and returns the current value of the selected bytes in the specified signal (see [2] section 7.2.2.4). The sum of start and count shall 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	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

Table 7. Byte array write (continued)

l_bool_wr, l_u8_wr, l_u16_wr	
Description	Sets the current value of the selected bytes in the signal specified to the values given in `data` (see [2] section 7.2.2.5). The sum of start and count shall 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	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	Returns the state of the flag specified i.e. zero if cleared, non-zero otherwise. (see [2] section 7.2.3.1)
Parameters	`flag` – the flag whose state is to be returned e.g. for the configured flag `Txerror` then LIN_FLAG_Txerror
Return	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

Table 9. Clear flag (continued)

l_flg_clr	
Description	Sets the value of the flag specified to zero (see [2] section 7.2.3.2)
Parameters	`flag` – the flag to be cleared e.g. for the configured flag `Txerror` then `LIN_FLAG_Txerror`
Return	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. Initialise interface

l_ifc_init	
Prototype (dynamic)	`l_bool l_ifc_init (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	Initialises the interface specified (e.g. baud rate). The default schedule set will be L_NULL_SCHEDULE where no frames will be sent or received. The interfaces are listed by name in the file lin_def.h. See *Section 4.3: Cluster configuration* and *User configuration* for details. This function must be called before using any other interface related to API functions. (see [2] section 7.2.5.1)
Parameters	`ifc` – the interface to be initialised
Return	Zero if initialisation was successful, non-zero if failed.

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

Connect interface

l_ifc_connect() function becomes obsolete for LIN 2.1 protocol and will not be used.

Disconnect interface

l_ifc_disconnect() function becomes obsolete for LIN 2.1 protocol and will not be used.

Table 11. Wake-up

l_ifc_wake_up	
Prototype (dynamic)	`void l_ifc_wake_up (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	Issues a wake-up signal on the interface given. (See [2] section 7.2.5.3). The wake-up signal (0xF0 byte, i.e. a dominant pulse of between 250 µsec and 5 ms depending on the configured bit rate) will be transmitted directly when this function is called. It is the responsibility of the application to retransmit the wake up signal according to the wake up sequence (See [2] section 2.6.2.).
Parameters	`ifc` – interface handle
Return	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 12. Interface control

l_ifc_ioctl	
Prototype (dynamic)	`l_u16 l_ifc_ioctl (l_ifc_handle ifc, l_ioctl_op operation, void* pParams);`
Availability	Master and slave nodes
Include	lin.h
Description	Controls functionality that is not covered by the other API calls. It is used for protocol specific parameters or hardware specific functionality. Example of such functionality can be to switch on/off the wake up signal detection. It controls protocol and interface specific parameters. The operations supported depend on the interface type. The parameter block `pParams` is optional, set to null if not needed otherwise to be interpreted as specified below. (See [2] 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	This depends on the operation requested

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 12. Interface control (continued)

l_ifc_ioctl	
Operation	LIN_IOCTL_DRIVER_STATE Returns in 16 bits two values; in the lower 8 bits the state of the driver and in 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 that the definition of driver states is currently located in the file lin_types.h.
Operation	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. Returns 0 if successful or 1 if the message was not found.
Operation	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.
Operation	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.
Operation	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.
Operation	LIN_IOCTL_FORCE_BUSSLEEP Forces the driver into sleep mode.
Operation	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.
Operation	LIN_IOCTL_READ_VARIANT_ID Return 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 12. Interface control (continued)

l_ifc_ioctl	

Operation	`LIN_IOCTL_READ_CONFIG_FLAGS`
	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`

Operation	`LIN_IOCTL_READ_NAD` 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.
Operation	`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.
Operation	`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.*

Table 13. Character received notification

l_ifc_rx	
Prototype (dynamic)	`void l_ifc_rx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	To be called when the interface specified receives one character of data (See [2] section 2.5.5). The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used). For UART based implementations it may be called from a user-defined interrupt handler triggered by a UART when it receives one character of data. In this case the function will perform necessary operations on the UART control registers. For more complex LIN hardware it may be used to indicate the reception of a complete frame. See also *Section 4.5: Interrupt configuration*.
Parameters	`ifc` – the interface that received the data

Table 13. Character received notification (continued)

l_ifc_rx	
Return	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 14. Character transmitted notification

l_ifc_rx	
Prototype (dynamic)	`void l_ifc_tx (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h
Description	To be called when the interface specified transmits one character of data (See [2] section 7.2.5.6). The application program is responsible for binding the interrupt and for setting the correct interface handle (if interrupt is used). For UART based implementations it may be called from a user-defined interrupt handler triggered by a UART when it has transmitted one character of data. In this case the function will perform necessary operations on the UART control registers. For more complex LIN hardware it may be used to indicate the transmission of a complete frame. See also *Section 4.5: Interrupt configuration*.
Parameters	`ifc` – the interface that sent the data
Return	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 15. 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 behaviour 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 whose status is to be returned (See [2] section 7.2.5.8)

Table 15. Read interface status (continued)

l_ifc_read_status	
Return	The status of the previous communication. Returned value is a status word (16 bit frame): it's only set based on a frame transmitted or received by the node (except bus activity). The call is a read-reset call; meaning that after the call has returned, the status word is set to 0. In the Master node the status word will be updated in the l_ifc_sch_tick() function. In the slave node the status word is updated latest when the next frame is started. The status word returned is defined as follows (bit 15 is MSB, bit 0 is LSB) see *Table 16*: **bit0** – error in response: set if a frame error is detected in the frame response, e.g. checksum error, framing error, etc. An error in the header results in the header not being recognized and thus, the frame is ignored. It will not be set if there was no response on a received frame. Also, it will not be set if there is an error in the response (collision) of an event triggered frame. **bit1** – successful transfer: set if a frame has been transmitted/received without an error **bit2** – overrun: set if two or more frames are processed since the last call to this function. If set, bit0 and bit1 represent 'OR'ed values for all processed frames **bit3** – go to sleep: set in a slave node if a go to sleep command has been received, and set in a Master node when the go to sleep command is successfully transmitted on the bus. After receiving the go to sleep command the power mode will not be affected. This must be done in the application. **bit4** – bus activity: set if the node has detected bus activity on the bus. A slave node is required to enter bus sleep mode after a period of bus inactivity on the bus: this can be implemented by the application monitoring the bus activity. (*Bus inactivity* in response: set if a frame error is detected in the frame response, e.g. checksum error, framing error, etc. An error in the header results in the header not being recognized and thus, the frame is ignored. It will not be set if there was no response on a received frame. Also, it will not be set if there is an error in the response (collision) of an event triggered frame. **bit1** – successful transfer: set if a frame has been transmitted/received without an error **bit2** – overrun: set if two or more frames are processed since the last call to this function. If set, bit0 and bit1 represent 'OR'ed values for all processed frames **bit3** – goto sleep: set in a slave node if a go to sleep command has been received, and set in a Master node when the go to sleep command is successfully transmitted on the bus. After receiving the go to sleep command the power mode will not be affected. This must be done in the application. **bit4** – bus activity: set if the node has detected bus activity on the bus. A slave node is required to enter bus sleep mode after a period of bus inactivity on the bus: this can be implemented by the application monitoring the bus activity. (*Bus inactiv*configuration: is set when the save configuration request has been successfully received. It is set only in the slave node, in the Master node it is always 0 (zero). **bit7** – value 0 **bit8-bit15** – last frame protected identifier: the protected identifier last detected on the bus and processed in the node. If the overrun bit i.e. bit2 is set, only the last value is maintained.

Table 15. Read interface status (continued)

l_ifc_read_status	

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

Table 16. 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 configura -tion	Event-triggered frame collision	Bus activity	Go to sleep	Overrun	Successful transfer	Response error

3.4 Diagnostic API

3.4.1 Node configuration (diagnostic) specific API

Table 17. 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
Description	It serializes the current configuration and copy it to the area (data pointer) provided by the application. It will not transport anything on the bus. (See [2] section 7.3.1.6) To be called when the save configuration request flag is set in the status register (See [2] section 7.2.5.8). After the call is finished the application is responsible to store the data in appropriate memory. The caller shall reserve bytes in the data area equal to length, before calling this function. The function will set the length parameter to the actual size of the configuration. In case the data area is too short the function will return with no action. In case the NAD has not been set by a previous call to ld_set_configuration or the master node has used the configuration services, the returned NAD will be the initial NAD. The data contains the NAD and the PIDs and occupies one byte each. The structure of the data is: NAD and then all PIDs for the frames. The order of the PIDs is the same as the frame list in the LDF and NCF (See [2] section 9.2.2.2 and section 8.2.5 respectively).

Table 17. Read configuration (continued)

ld_read_configuration	
Parameters	`ifc` – the interface to address `data` – structure that will contain the NAD and all the n PIDs for the frames of the specified NAD `length` – length of `data` (1+n, NAD+PIDs)
Return	LD_READ_OK If the service was successful. LD_LENGTH_TOO_SHORT If the configuration size is greater than the length. It means that the data area does not contain a valid configuration.

Table 18. 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	It configures the NAD and the PIDs according to the configuration given by data. It will not transport anything on the bus. (See [2] section 7.3.1.7) To be called when it wants to restore a saved configuration or set an initial configuration (e.g. coded by I/O pins).It shall be called after calling l_ifc_init(). The caller shall 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 order of the PIDs is the same as the frame list in the LDF and NCF (See [2] section 9.2.2.2 and section 8.2.5 respectively).
Parameters	`ifc` – the interface to address `data` – structure containing the NAD and all the n PIDs for the frames of the specified NAD `length` – length of `data` (1+n, NAD+PIDs)
Return	LD_SET_OK If the service was successful. LD_LENGTH_NOT_CORRECT If the required size of the configuration is not equal to the given length. LD_DATA_ERROR The set of configuration could not be made.

3.4.2 Diagnostic transport layer

Table 19. Initialization

ld_init	
Prototype	`void ld_init (l_ifc_handle ifc);`
Availability	Master and slave nodes
Include	lin.h

Table 19. Initialization (continued)

ld_init	
Description	(Re)Initialize the raw and the cooked layers on the interface ifc. All transport layer buffers will be initialized (See [2] section 7.4.2) If there is an ongoing diagnostic frame transporting a cooked or raw message on the bus, it will not be aborted.
Parameters	`ifc` – the interface handle
Return	None

3.4.3 Diagnostic transport layer: RAW API

Table 20. Put raw frame

ld_put_raw	
Prototype	`void ld_put_raw (l_ifc_handle ifc, const l_u8* const pData);`
Include	lin.h
Description	Queue a raw diagnostic frame for transmission. (LIN API 4.1.1) Note: the application should check `ld_raw_tx_status` before attempting to queue a frame – if no space is available data is discarded
Parameters	`ifc` – the interface handle `pData` – pointer to the data to be queued
Return	None

Table 21. Get raw frame

ld_get_raw	
Prototype	`void ld_get_raw (l_ifc_handle ifc, l_u8* const pData);`
Include	lin.h
Description	Copy the oldest frame on the receive-stack to the buffer provided (LIN API 4.1.2). `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.
Parameters	`ifc` – interface handle `pData` – pointer to the buffer into which the frame is to be copied
Return	None

Table 22. Query raw transmit-queue status

ld_raw_tx_status	
Prototype	`l_u8 ld_raw_tx_status (l_ifc_handle ifc);`
Include	lin.h
Description	Return the status of the raw frame transmission queue. (LIN API 4.1.3)
Parameters	`ifc` – interface handle
Return	LD_QUEUE_FULL – transmit-queue is full and cannot accept further frames LD_QUEUE_EMPTY – transmit-queue is empty i.e. all frames have been transmitted LD_QUEUE_READY – transmit-queue is ready to receive further frames for transmission LD_TRANSFER_ERROR – LIN protocol errors occurred during transfer, abort and re-try

Table 23. Query raw receive-queue status

ld_raw_rx_status	
Prototype	`l_u8 ld_raw_rx_status (l_ifc_handle ifc);`
Include	lin.h
Description	Return the status of the raw frame receive-queue. (LIN API 4.1.4)
Parameters	`ifc` – interface handle
Return	LD_DATA_AVAILABLE – receive-queue contains data that can be read LD_QUEUE_EMPTY – receive-queue does not contain any data LD_TRANSFER_ERROR – LIN protocol errors occurred during transfer, abort and re-try

3.4.4 Diagnostic transport layer: COOKED API

Table 24. 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);`
Include	lin.h
Description	Pack the information given by data and length into one or more diagnostic frames and send. If called from a master node the frames are sent to the node with address `nad`. If called from a slave node the frames are sent to the master. (LIN API 4.2.1) The call returns immediately.
Parameters	`ifc` – interface handle `length` – in range 1 - 4095 bytes `nad` – address of node `pData` – pointer to the data to be sent
Return	None

Table 25. Receive message

ld_receive_message	
Prototype	`void ld_receive_message (l_ifc_handle ifc, l_u16* length, l_u8* nad, l_u8* const pData);`
Include	lin.h
Description	Prepare the module to receive one message and store it in the buffer given. When the call is made, length specifies the maximum length allowed. After the call, length specifies the actual length and if called from a master node, then `nad` is assigned the value of the `nad` in the message. (LIN API 4.2.2) The call returns immediately. The buffer should not be changed by the application as long as `ld_rx_status` returns LD_IN_PROGRESS. Note: SID (or RSID) must be the first byte in the data area and is included in the length.
Parameters	`ifc` – interface handle `length` – in range 1 - 4095 `nad` – address of node `pData` – pointer to buffer into which the data will be written
Return	None

Table 26. Get transmit-queue status

ld_tx_status	
Prototype	`l_u8 ld_tx_status (l_ifc_handle ifc);`
Include	lin.h
Description	Return the status of the last call made to `ld_send_message` (LIN API 4.2.3)
Parameters	`ifc` – interface handle
Return	LD_IN_PROGRESS – transmission not yet completed LD_COMPLETED – transmission completed successfully LD_FAILED – transmission ended with an error, data partially sent

Table 27. Get receive-queue status

ld_rx_status	
Prototype	`l_u8 ld_rx_status (l_ifc_handle ifc);`
Include	lin.h
Description	Return the status of the last call made to `ld_receive_message`. (See [2] section 7.2.5.7)
Parameters	`ifc` – interface handle
Return	LD_IN_PROGRESS – reception not yet complete LD_COMPLETED – reception completed successfully LD_FAILED – reception ended with an error, data partially received

3.5 Slave specific API

3.5.1 Interface management

Table 28. Slave synchronise

l_ifc_aux	
Prototype (dynamic)	`void l_ifc_aux (l_ifc_handle ifc);`
Availability	Slave node only
Include	lin.h
Description	Synchronizes to the BREAK and SYNC characters sent by the master on the interface specified. (see [2] section 7.2.5.7) *Note: This function is redundant for the currently delivered drivers and is therefore implemented as a null function.*
Parameters	`ifc` – interface handle
Return	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 29. 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's used, the slave node application must implement this callout).
Include	lin.h
Description	To be used when the master node transmits a read by identifier request with an identifier in the user defined area. The slave node application will be called from the driver when such request is received. (see [2] section 7.3.3.2)

Table 29. Read by ID callout (continued)

ld_read_by_id_callout	
Parameters	`ifc` – interface handle `id` – the identifier in the user defined area (32 to 63), from the read by id configuration request. (see [2] Table 4.19) `data` – pointer to a data area with 5 bytes. This area will be used by the application to set up the positive response (see [2], the user defined area in Table 4.20).
Return	The driver will act according to the following return values from the application: LD_NEGATIVE_RESPONSE The slave node will respond with a negative response (see [2] , Table 4.21). In this case the data area is not considered. LD_POSTIVE_RESPONSE The slave node will setup a positive response using the data provided by the application. LD_NO_RESPONSE The slave node will not answer.

3.6 STMicroelectronics extensions

Table 30. Software Timer Function

l_timer_tick	
Prototype	`void l_timer_tick (void);`
Include	lin.h
Description	Handles a software timer interrupt. Internal LIN timers are advanced and expired timers are evaluated. This function should be called every LIN_TIME_BASE_IN_MS ms by the user's application if a hardware timer has not been configured. See also *Section 4.4: User configuration*.
Parameters	None
Return	None

Table 31. Protocol Switch

l_protocol_switch	
Prototype	`void l_protocol_switch (l_ifc_handle ifc; l_bool linEnable);`
Include	lin.h
Description	Switches the LIN protocol on or off for a specified interface. This function provides the possibility to use an alternative protocol on a given interface. The ISR checks to see if LIN is enabled. If not, a callback function that is provided as an entry point to the alternative protocol handler will be called. This callback must be configured as described in *Section 4.4.5: Callback functions*.
Parameters	`ifc` – interface handle `linEnable` – if 1 then switch LIN on, if 0 then off
Return	None

Table 32. Set baud rate

l_change_baudrate	
Prototype	`void l_change_baudrate (l_ifc_handle ifc, l_u16 baudrate);`
Include	lin.h
Description	Sets the baudrate 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) baudrate is detected. The callback function must be configured as described in *Section 4.4.5: Callback functions*.
Parameters	`ifc` – interface handle `baudrate` – the baudrate to set for the interface
Return	None

Table 33. Raw Tx Frame Delete

ld_raw_tx_delete	
Prototype	`void ld_raw_tx_frame_delete(l_ifc_handle ifc);`
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	1 if a frame has been removed, 0 if no Raw Tx frame was on the stack.

3.7 Implementation Notes

3.7.1 API data types

Certain types defined as part of the standard API are not supported by the driver. This means that the application may not define variables to be of these types directly. The types that are not defined are `l_signal_handle`, `l_frame_handle` and `l_flag_handle`.

Note that the application may only use the signals, frames and schedules by their name as defined in the LDF when calling the dynamic interface. The interface name to be used is as defined in the lingen control file. Flag names are based on the signals and frames defined in the LDF.

Please also refer to *Section 2.4: Using the driver* for naming conventions used by the driver.

3.7.2 Notification flags

The notification flags are used to indicate signal or frame updates i.e. that the transfer of signal or frames has taken place.

Due to the asynchronous nature of the execution of the application and driver, it is possible that unexpected behaviour may occur as shown by the following example:

1. The driver (master or slave) detects that a frame must be sent and composes its frame buffer. It will copy the current values of the signals into the frame buffer and start transmission.
2. The user application writes a signal that is contained in the frame currently being transferred. Perhaps it even resets the Tx flag to be notified when the signal has been sent. However, the transfer of the frame is still in progress!
3. The driver finishes the transfer of the frame (successfully). It will then mark the frame and all signals within as transferred.
4. The user application polls the Tx flag and receives 1. It will then of course suppose that the value just written has been transferred. Instead, the value that was originally valid has been transferred.

A possible workaround would be to use the `l_ifc_ioctl()` function to query the driver state before writing new signal values. If a frame is in transfer then the query returns the pID as well as the driver state and so the application can check if the signal to be written is part of the current frame transfer or not. 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 build of the driver including hardware specific settings required.

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 tool **lingen** (supplied)

The user configurable files, the tool **lingen**, the control file and the LDF files required for generation may be located in a directory of the user's choice. This must then be specified in the top-level makefile as described in the *Section 4.2: Makefiles*.

The driver specific makefile Make_LIN (delivered) assumes 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: Makefiles*. Its sub-directories are expected in *Table 34*:

Table 34. 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 functionality.

- These are the files Make_LIN and an example top-level makefile. The top-level makefile includes the settings for environment variables, see *Section 4.2.1: Top-level makefile*.
- The file Make_LIN controls the build process and is designed to be included in the toplevel makefile.

4.2.1 Top-level makefile

This file must include definitions for the following variables:

Table 35. Top-level makefile predefined variable definition

Name	Description
LIN_NODE_IDENTITY	this 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	the lib generation tool
LIN_TMP_FILE	the name for temporary file
LIN_SRC_PATH	top-level directory of the LIN driver
LIN_CFG_PATH	the directory containing the configuration information for the cluster and the driver. The lingen control file and the files lin_def.h and lin_def.c and lin_def_*archname*.h must be located in this directory. The file lin_def_*archname*.h is the architecture specific user configuration file – *archname* refers to the specific architecture name.
LIN_GEN_PATH	the directory in which generated files are written. This is used for the –o option for **lingen** in the file Make_LIN
LIN_LINGEN_BIN	the command used to invoke **lingen**
LIN_NODE	the name of the node as is 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.
LIN_LINGEN_CONTROL	the 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: Cluster configuration*

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

Following the definitions of the variables the file Make_LIN should be included:

```
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 with 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 standard. An example LIN 2.0 LDF is provided in *Section 6.2: LIN 2.0 LDF example*. 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 to be used – this information is specified in the lingen control file that is used as input to the **lingen** tool.

A slave node only supports one interface and therefore only one LDF file is needed for a slave. The lingen control file is then used to specify this interface and the name of the LDF file to be used.

In addition to this interface definition, the user may also specify the use of default frame identifiers in the lingen control file for a slave. In this case, the default values given in the LDF file will be used for all slave frames. This means that the slave nodes may start communicating without having been first configured by the master node. This behaviour is then no longer in conformance with the standard.

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

```
//
// 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 i.e. the absolute path should be given.

The tag entry is concatenated with all frame names and signal names when **lingen** processes the LDF files. For example, a signal name “s1_sig1” in the LDF file `lin_sci0.ldf` listed above will appear in code as “LIN_SIGNAL_IFC0_s1_sig1”.

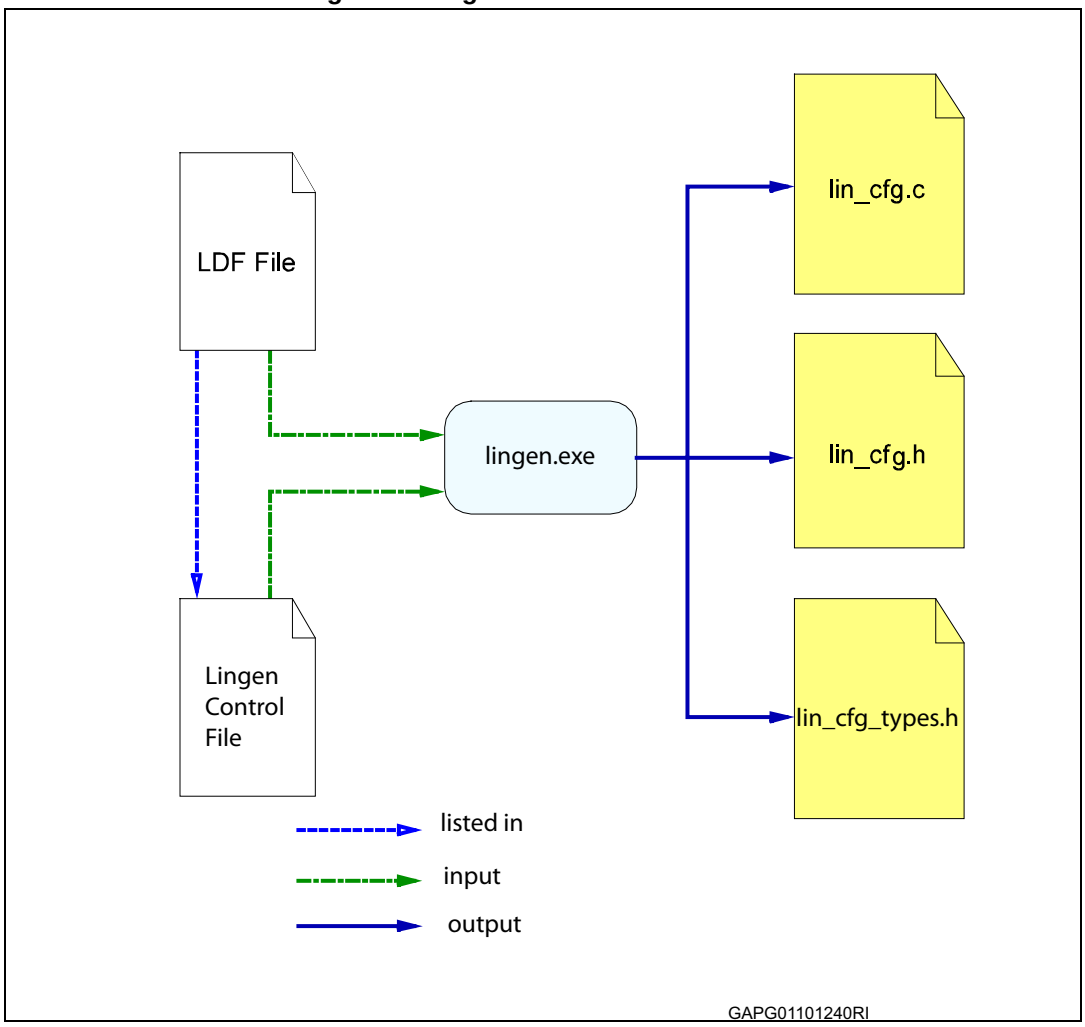
The interface name given in the control file depends on the actual hardware used. Following the interface definition the user may specify the use of default frame IDs as described earlier.

If the **lingen** tool is to be 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*.

4.3.2 Lingen

From the information given in the lingen control file and the associated LDF file, a set of configuration files can be generated using the **lingen** tool provided. Inputs and outputs for **lingen** are depicted in *Figure 2: Lingen workflow*.

Figure 2. Lingen workflow



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

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

lingen nodeIdentifier [**options**] lingen_control_file

where

nodeIdentifieris the name of the node given in the LDF files for which the driver is to be built (in the case of a master this must be the same in all ldfs – in the case of a slave there is only one ldfsupported)

lingen_control_fileis the name of the control unit input to **lingen**

The following options are currently supported:

- options
- -c configurationspecifies which of the possible configurations given in the LDF is to be used for the build
- -o outputDirectoryspecifies the destination for the configuration files that are to be generated
- -r receiveChecksumselects the checksum model to be used for receiving frames. Possible values are: `ldf` – the lingen tool determines the checksum model from the information given in the LDF. This is the default. `both` – the driver accepts either model for all frames
- -s sendChecksumselects the checksum model to be used for sending frames. Possible values are: `ldf` – the lingen tool determines the checksum model from the information given in the LDF. This is the default. `classic` – the driver sends all frames using the classic checksum
- -vverbose mode, details from **lingen** will be output to the shell

Note: That the option -o is set in the file Make_LIN and should not be set in the top-level makefile. The -o option is set to LIN_GEN_PATH and it is not recommended that this setting be changed by the user unless the file Make_LIN is to be replaced.

4.4 User configuration

There are two header files that contain settings that must be configured by the user; `lin_def.h` and `lin_def_`*`archname`*`.h` where *`archname`* refers to a specific architecture.

The architecture dependent settings contained in `lin_def_`*`archname`*`.h` are described in the corresponding Architecture Notes document.

The following sections list the settings that must be configured and which are contained in the header file `lin_def.h`.

4.4.1 Timers

The LIN driver uses a timer for monitoring bus activity e.g. while sending frames or checking for bussleep conditions. This may be a hardware timer or a software timer and must be configured by the user in the architecture specific configuration file `lin_def_`*`archname`*`.h`.

If a hardware timer is selected then the timer number must also be configured according to the architecture in use. The architecture specific notes describe which values may be used.

A time base for the timer must be configured in the file `lin_def.h`. This time base specifies the frequency at which the driver's timer routine must be called.

If a software timer is used then this time base gives the frequency at which the API function `l_timer_tick()` must be called by the user's application or OS.

If a hardware timer interrupt has been configured then the driver sets the timer so that the timer ISR will be called at this frequency.

The recommended value for the time base is either 1 or 2ms and 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 described in the architecture specific notes.

4.4.2 General settings

Further settings that must be specified by the user are described below.

Checking function parameters

The driver may be built either for development or for production purposes.

The development version includes a more extensive check on parameters passed to functions. These may not be necessary for a production version and so the checking may be reduced if desired 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
```

The maximum time for transfer of a frame can also be configured as a percentage of the nominal transfer time (the default setting of 140 corresponds to that specified in the LIN2.0 standard):

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

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 2.0 standard, a slave node shall be able to detect a BREAK at any time. The current frame processing shall be interrupted and the new frame shall then be processed. A BREAK is detected by the driver through a framing error. This may occur at any time i.e. during the transmission of a data byte or between transmissions of data bytes. The following switch allows the user 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 V2.0 network and so the standard 13 bit break signal can be lengthened to 36 bits for a 19,200 baudrate network or 18 bits in a 9,600 baudrate network or equivalent.

If the Cooling feature is to be used then it must be activated:

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

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

The start-up behaviour of LIN nodes can be influenced by two settings. The first option is to start the slave node's bussleep timer going when a slave node connects to the network. If this is set then the slave will enter sleep mode if no activity is detected.

Additionally, a master node can be set up to send a wakeup signal on connecting to the network. Note that if a slave is set up to start the bussleep timer on connect then the master should be set to send a wakeup. If not, and the master does not start to send within 4 seconds, then the slave will enter 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 bussleep 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
```

When receiving frames, pIDs are validated against stored pIDs. However, there is no validation of pIDs when 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: That validation is only carried out on assignment and not on reception of each frame.

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 built should 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 default value for bussleep timeout is given in the LIN2.x standard as 4 seconds. For a wakeup request issued by a node, the period between successive retries is 150ms. After three failed attempts the node must wait 1.5s before issuing further wakeup requests. These values may be configured by the user as follows:

```
/*************************************************************
 * LIN 2.x specific setting
 * The value for the bussleep 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 retries that may be attempted may 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 will not stop sending wakeup signals when there is no response from the master.

4.4.3 Diagnostic functions configuration

The functions of the diagnostic module can be individually selected as detailed below. The default settings on delivery of the driver reflect the requirements in the standard. 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
```

Also if the “Assign NAD” service is optional, it’s enabled because it’s called in the Initialization table of the demo present in the LDF file of the LIN 2.1 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
```

Following services (**Save Configuration** and **Assign Frame Id Range**) 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

Also if the "Save Configuration" service is optional, it's enabled because it's called
in the Initialization table of the demo present in the LDF file of the LIN 2.1 package.
```

4.4.4 Diagnostic class

Diagnostic class is valid and mandatory only for LIN 2.1 slave nodes, and will be used to:

- Do a cross check between LDF and the same class, to understand if the slave node is able to do diagnostic;
- Understand which diagnostic services, the slave node will be able to respond to;
- Know which Configuration and Identification services will be 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 Callback functions

Interrupt callback functions

Two callback functions must also be provided by the user: these allow the driver to enable or disable interrupts in the system. The function prototypes are described below and their implementations must be provided by the user. This could be, for example, in the lin_def.c file located in the user's configuration directory. The function prototypes are defined as follows.

Table 36. Disable Interrupts

l_sys_irq_disable	
Prototype	`l_irqmask l_sys_irq_disable (void);`
Description	Achieves a state in which no controller interrupts may occur
Parameters	None
Return	interrupt mask describing the state of the interrupts at time of call

Table 37. Restore Interrupts

l_sys_irq_restore	
Prototype	`void l_sys_irq_restore (l_irqmask irqMask);`
Description	Restores the state of interrupts as given by irqMask
Parameters	`irqMask` – mask containing state of interrupts to be restored
Return	None

An example implementation that could be used is given in *Section 6.4: Example implementation of IRQ callbacks*.

The LIN driver will use these user-defined functions in each call to an API function. Interrupts will be disabled on entering the API function and then re-enabled before returning. This means that the callback functions provided for the driver must handle nested calls. In the case where 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 OSEK functions `SuspendOSInterrupts` and `RestoreOSInterrupts` shown in example *Section 6.4: Example implementation of IRQ callbacks* satisfy this criterion.

Protocol switch callback function

It is possible to change the protocol in use for a particular interface that is usually operating with LIN. This is done by the application 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 38*.

Table 38. Protocol switch function callback

l_protocol_callback_iii	
Prototype	`void l_protocol_callback_iii (void);` where `iii` denotes configured interfaces SCI0 .. SCIn
Description	Provides the entry point to an alternative protocol handler. This is 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	None

To enable the use of this feature the following line must also be included in lin_def.h:

```
#define LIN_PROTOCOL_SWITCH
```

Diagnostic callback functions

For the diagnostic service `ld_read_by_id()` (when used for user-defined ids) and for the service `ld_data_dump()` two callback functions must be configured for slave nodes. These have the following prototype forms:.

Table 39. ld_read_by_id callback

ld_readByIdCallback	
Prototype	`l_bool ld_readByIdCallback(l_u8 id, l_u8* pBuffer);`
Description	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 allocted for user-defined ids i.e. 32 – 63. If a non-zero value is returned then the driver sends the buffer back to the master. The user application receives the complete frame buffer and may write up to 5 bytes response into the buffer starting at location `pBuffer[3]`. The application is responsible for setting the PCI byte i.e. `pBuffer[1]` correctly. This 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	non-zero if buffer to be sent back to master

Table 40. 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.

Table 40. ld_data_dump callback (continued)

ld_dataDumpCallback	
Parameters	`sendBuf` – the buffer sent by the master `recBuf` – the buffer into which the application can write its response
Return	non-zero if the driver is to send a response back to the master

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.

Baud rate detection callback function

Baudrate detection for slave nodes can be configured. When this feature is enabled, a callback function is invoked by the driver when an incorrect baudrate is detected. From this callback, the application can then call the `l_change_baudrate()` function to reduce the current baudrate. The baudrate detection works by the principle that the application starts with the highest possible baudrate and then repeatedly tries by lowering the baudrate until communication is established.

Table 41. Baud rate detection callback

l_baudrate_callback_III	
Prototype	`void l_baudrate_callback_iii (l_u16 baudrate);`
Description	Sets the baudrate for the given interface by calling the `l_change_baudrate()` API function. This callback will be called if an incorrect (too high) baudrate 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 baudrate currently detected (i.e. the incorrect baudrate) on the interface
Return	None

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

```
#define LIN_BAUDRATE_DETECT
```

4.5 Interrupt configuration

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

Note: The driver's use of these functions is architecture dependent -- it may be the case that only the Rx handler is used and so the user should not call the Tx handler. The user should refer to the architecture notes for exact details.

The functions have the following interfaces:.

Table 42. Handler for character rx

l_ifc_rx	
Prototype	`void l_ifc_rx (l_ifc_handle ifc);`
Description	Handles the interrupt when a character is received
Parameters	`ifc` – the interface on which the interrupt occurred
Return	None
Interface specific protoytpe	`void l_ifc_rx_iii (void);`
Description	handles the interrupt for the interface given by `iii`
Include	lin.h

Table 43. Handler for character tx

l_ifc_tx	
Prototype	`void l_ifc_tx (l_ifc_handle ifc);`
Description	Handles the interrupt
Parameters	`ifc` – the interface on which the interrupt occurred
Return	None
Interface specific prototype	`void l_ifc_tx_iii (void);`
Description	handles the interrupt for the interface given by `iii`
Include	lin.h

5 Specification of lingen control file format

5.1 lingen control file

A special control file is used by lingen to determine which interfaces are to be used and which LDF file is associated with the interface chosen. The following specifies the content of this control file. Note that C/C++ style comments may be used in the lingen control file. The possibilities for these are not included in the specification below. An explanation of the syntax used is given in *Section 5.2: Specification syntax*.

5.1.1 File definition

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

5.1.2 Interface specification

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

<interface_id::= identifier

Currently the driver supports interface identifiers SCI0 to SCI9. However, the interface identifier used here should match the specific interface as defined in the architecture notes delivered with the driver.

<ldf_file_name> ::= string

The string should specify the filename of the LIN description file. This string may include the full path specification of the file, relative path to the file or just the filename if the file is located in the current directory. Note that it is recommended that the full path specification be used especially if lingen 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 for example *Section 4.3: Cluster configuration*.

5.1.3 Default frame IDs

The optional keyword **LIN_use_default_frame_ids** is intended for use with slave nodes only. If included, the default frame identifiers given in the LDF will be used for all slave frames. Slaves may start communication without having been configured by the master.

5.2 Specification syntax

The following syntax has been used for the specification of the lingen control file. This has been kept consistent with the syntax used for specifying LIN description files as described in the LIN2.0 standard [1].

Table 44. Syntax description

Symbol	Meaning
::=	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

A special control file is used by lingen to determine which interfaces are to be used and which LDF file is associated with the interface chosen. This example shows a slave interface configuration and the use of default frame IDs:

```
//
// 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

The format and full details for a LIN description file are given in the LIN configuration language specification part in [1]. This example shows a configuration with one master and two slaves. The first slave is set up according to LIN 2.0, 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
//
```

```
//
// 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
//
// 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
//
//
// 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
////
// 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
//
{
//
// 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
//
//
// 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
//
//
// 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
////
// 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
//
{
     LIN_protocol = 1.2;

     // the startup diagnostic address
     configured_NAD = 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
////
// 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
//

Assig//
// 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
//
nFr//
// 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
//
ameId{slave1, frmM3   } delay 20 ms;
      AssignF//
// 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
//
rameId{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

The format and full details for a LIN description file are given in the LIN configuration language specification part in [2]. This example shows a configuration with one master and two slaves. Both slaves 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

Example implementation for OSEK:

```
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 example, in an OSEK implementation include "os.h".

7 Revision history

Table 45. Document revision history

Date	Revision	Changes
04-Feb-2014	1	Initial release.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2014 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com