LIN08EY16 Driver User Manual

HC08
Microcontrollers

LIN08EY16DUM
Rev. 2
10/2005

freescale.com

freescale™
semiconductor

LIN08EY16 Driver User Manual

To provide the most up-to-date information, the revision of our documents on the World Wide Web will be the most current. Your printed copy may be an earlier revision. To verify that you have the latest information available, refer to http://www.freescale.com.

The following revision history table summarizes changes contained in this document. For your convenience, the page number designators have been linked to the appropriate location.

Revision History

Date	Revision Level	Description	Page Number(s)
10/2005	2	Converted to Freescale template.	N/A

Contents

Chapter 1
Overview

Chapter 2
Notations

2.1 Manual Structure . . . 11
2.2 Typographical Conventions . . . 11
2.3 Definitions, Acronyms and Abbreviations . . . 12
2.4 References . . . 13

Chapter 3
LIN Concepts

3.1 General Description . . . 15
3.2 LIN Concept . . . 15
3.3 Message Frame . . . 16
3.3.1 Break Field . . . 17
3.3.2 Synchronization Field . . . 17
3.3.3 Identifier Field . . . 17
3.3.3.1 Reserved Identifiers . . . 19
3.3.4 Data Field . . . 20
3.3.5 Checksum Field . . . 20
3.4 Error Detection . . . 20
3.5 Synchronization . . . 20
3.6 Wakeup Signal Frame . . . 20

Chapter 4
LIN Driver

4.1 Driver Configuration . . . 23
4.1.1 LIN API Configuration . . . 23
4.1.2 Freescale API Configuration . . . 23
4.1.2.1 Driver Configuration File (LINCFG.H) . . . 24
4.1.2.2 Message Configuration File (LINMSGID.H) . . . 24
4.2 Error Handling . . . 25
4.2.1 Bit Error . . . 26
4.2.2 Checksum Error . . . 26
4.2.3 Inconsistent-Sync-Field Error . . . 26
4.3 Timeout Handling . . . 27
4.3.1 No-Bus-Activity . . . 27

Chapter 5
Freescale API

5.1 General . . . 29
5.2 Data Types . . . 29
5.3 Constant Definition . . . 29
5.4 Driver Services . . . 31
5.4.1 LIN_Init . . . 31
5.4.2 LIN_Wakeup . . . 32
5.4.3 LIN_GotoRun . . . 32
5.4.4 LIN_DriverStatus . . . 33
5.4.5 LIN_GetMsg . . . 34
5.4.6 LIN_PutMsg . . . 35
5.4.7 LIN_MsgStatus . . . 36
5.4.8 LIN_GetRxErr . . . 37
5.4.9 LIN_GetTxErr . . . 37
5.4.10 LIN_ClearRxErr . . . 38
5.4.11 LIN_ClearTxErr . . . 38
5.4.12 LIN_IdleClock . . . 38
5.4.13 LIN_GetSyncPD . . . 39
5.5 Call-back Services . . . 40
5.5.1 LIN_Command . . . 40

Chapter 6
LIN API

6.1 General . . . 41
6.2 Data Types . . . 41
6.3 Driver Services . . . 42
6.3.1 l_sys_init . . . 42
6.3.2 l_bool_rd . . . 42
6.3.3 l_u8_rd . . . 43
6.3.4 l_u16_rd . . . 43
6.3.5 l_bool_wr . . . 43
6.3.6 l_u8_wr . . . 44
6.3.7 l_u16_wr . . . 44
6.3.8 l_flg_tst . . . 44
6.3.9 l_flg_clr . . . 45
6.3.10 l_ifc_init . . . 45
6.3.11 l_ifc_connect . . . 45
6.3.12 l_ifc_disconnect . . . 46
6.3.13 l_ifc_ioctl . . . 46
6.3.14 l_ifc_rx . . . 48
6.3.15 l_ifc_tx . . . 49
6.4 Call-back Services . . . 49
6.4.1 l_sys_irq_disable . . . 49
6.4.2 l_sys_irq_restore . . . 49

Chapter 7
Platform Specific

7.1 General 51
7.2 MCU Resources Usage 51
7.3 Physical Interface Connection 51
7.4 Disabled Interrupt Code Sections 51
7.5 Break Signal Detection 51
7.6 Zero Page Usage 52

Chapter 8
Building Application

8.1 General 53
8.2 Compilation 53
8.3 Linking 54
8.3.1 CodeWarrior 54

Appendix A
Sample Application

A.1 Sample Description 55
A.2 Sample Building and Running 55
A.3 CodeWarrior Project 55
A.4 Troubleshooting 56
A.4.1 Environment settings 56
A.4.2 Startup Files 56
A.4.3 LinSigFlags Size 56

Appendix B
Performance Characteristics

B.1 Performance Characteristics 57
B.2 Memory Consumption 57

Chapter 1
Overview

This user manual describes a LIN driver for the Freescale HC08EY16 microcontroller.

LIN (Local Interconnect Network) is a serial communications protocol that efficiently supports the control of mechatronic nodes in distributed automotive applications. The protocol is applicable to buses with a single master node and a set of slave nodes.

The main properties of the LIN bus are:

- Single-master, multiple-slave concept
- Low-cost silicon implementation based on common UART/SCI interface hardware, or a software equivalent, or as pure state machine
- Self synchronization without quartz or ceramic resonators in the slave nodes
- Deterministic signal transmission
- Low-cost single-wire implementation
- Speeds up to 20 kbps

The driver is supplied as source code and header files.

The supported toolchain is the CodeWarrior compiler V3.1 or later, with ‘C’ header files for user defined parameters.

Chapter 2
Notations

2.1 Manual Structure

This user's manual comprises the following sections.

Chapter 1 Overview provides the LIN bus overview and describes its main features.

Chapter 2 Notations includes a description of the structure of the document, typographical conventions used, references to other documents, technical support information, and a list of acronyms.

Chapter 3 LIN Concepts provides a general description of the LIN protocol. It explains the basic concepts of the LIN network communication model.

Chapter 4 LIN Driver describes the LIN driver configuration and functionality.

Chapter 5 Freescale API provides a detailed description of Freescale Semiconductor LIN driver run-time services.

Chapter 6 LIN API provides a detailed description of LIN API services compatible with the LIN specifications.

Chapter 7 Platform Specific covers platform specific features that can be useful in the development of applications that use the LIN driver.

Chapter 8 Building Application describes the steps for compiling and linking the application.

Appendix A Sample Application contains the sequence of actions needed for creation, building and execution of sample application included into the LIN driver package.

Appendix B Performance Characteristics contains LIN driver performance characteristics, such as ROM and RAM usage, timing and CPU load.

2.2 Typographical Conventions

This manual employs the following typographical conventions:

- *italic* type is used for all names of directives, macros, constants, routines and variables. Also, this type is used for special terms.
- `courier` type is used for code examples in the text.

2.3 Definitions, Acronyms and Abbreviations

8N1	character coding with eight bits per char, one stop bit, and no parity bit
API	application program interface (a set of data types and functions)
bps	bits per second
connected state	logical state when driver with LIN API can transmit or receive LIN frames (headers and responses)
CPU	central processing unit
disconnected state	logical state when driver with LIN API ignores all LIN frames (headers and responses)
ELF/DWARF	extensible linking format/debugging with attribute record format
Frame Error	stop bit absence when SCI receives a data byte
ID	identifier
ISO	International Standards Organization
LDF	LIN description file
LIN	Local Interconnection Network
LIN API	signal oriented API, specified by the LIN Consortium
LSB	less/least significant bit/byte
Master node	network node that implements the functionality of a LIN bus master and a LIN bus slave
MCU	microcontroller unit (Freescale Semiconductor's microcontrollers)
Freescale API	message oriented API, specified by Freescale
MSB	more/most significant bit/byte
N_{bit}	number of target MCU timer ticks per T_{bit}
N_{idle}	user-defined number of special function calls (recognized as *No-Bus-Activity* condition)
OSI	Open Systems Interconnection
RAM	random access memory
ROM	read only memory
SCI	serial communication interface
T_{bit}	transmission time of one bit
UART	universal asynchronous receiver/transmitter

2.4 References

[1] LIN Specification Package, Revision 1.3, 12 December 2002

Chapter 3
LIN Concepts

3.1 General Description

The LIN protocol has the following properties.

- Single-master, multiple-slave organization (i.e. no bus arbitration)
- Guaranteed latency times for data transmission
- Variable length of message frame: 1 to 8 bytes
- Configuration flexibility
- Multi-cast reception with time synchronization, without quartz or ceramic resonator
- Data-checksum security
- Error detection
- Minimum cost for semiconductor components (die size)
- Transmission without acknowledgment
- *Sleep* mode control

3.2 LIN Concept

The LIN Protocol Specification [1] defines the Data Link Layer and the Physical Layer according to the ISO/OSI Reference Model.

The LIN Physical Layer is defined in LIN Protocol Specification [1]. When a signal is referenced in the manual, it is the logical level that is implied.

A typical LIN bus structure is presented in Figure 3-1.

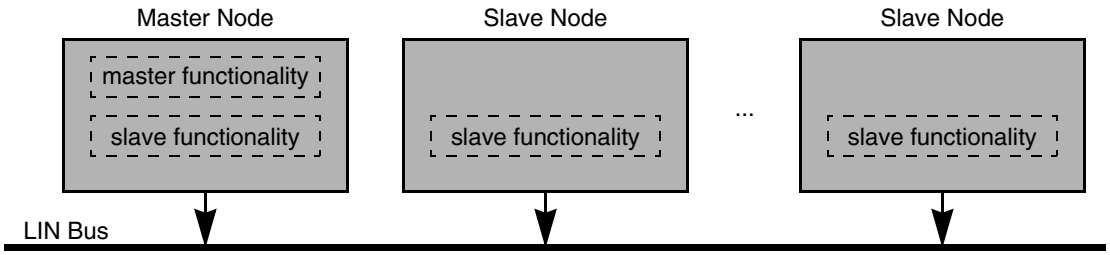

Figure 3-1. General Concept of LIN

The network consists of a *master node* and a number of *slave nodes*. There can be several slave nodes in the network, but only one master node. The bus traffic is controlled by the master node. All data transmission is asynchronous.

All nodes exchange data via *message frames*. The general format of a message frame is shown in Figure 3-2.

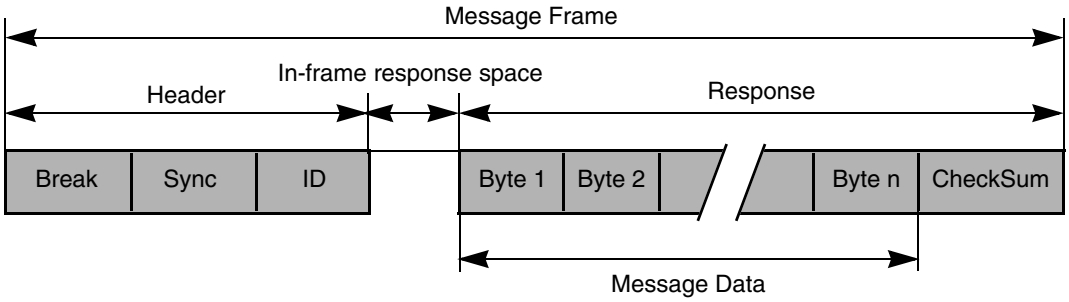

Figure 3-2. LIN Bus Frame Format

Master functionality (*master task*) on a node is responsible for *frame header* transmission. Slave functionality (*slave task*) is responsible for frame header reception, and transmission of the data in response to the master request.

To request data, the master task sends a message frame header. The frame header consists of a *break field (Break)*, followed by a *synchronization field (Sync)* and an *identifier field (ID)*.

The slave task sends back a *response*. The response consists of a *data field* and a *checksum field*. The data field can be from one to eight bytes long. The checksum field is one byte long. The checksum ensures data consistency during the transmission. There may be space between the fields of the message frame as well as between the data bytes. These spaces are limited only by the maximum length of the whole message frame, i.e. the sum of all spaces may not exceed 40% of the message frame length.

Only the master node containing the master task is allowed to start the transmission of a message frame. Only one slave task is allowed to answer to the identifier, as there is no arbitration procedure.

During reception, message filtering is based upon the whole identifier. The network configuration must ensure that not more than one slave task responds to a transmitted identifier.

3.3 Message Frame

As shown in Figure 3-2, the message frame consists of the header which is transmitted by the master task, and the response which is transmitted by the slave task.

The header comprises the following fields:

- Break
- Synchronization
- Identifier

The response comprises the following fields:

- Data
- Checksum

All these fields are described in detail in this section.

3.3.1 Break Field

To identify clearly the beginning of a message frame, its first field is a break field. The break field is always sent by the master task. This provides a regular opportunity for slave tasks to synchronize on the frame start.

The break field consists of two parts (see Figure 3-3). The first part consists of a logic zero state with a duration of 13 T_{bit}. The second part is the synchronization delimiter, which is presented as a logic one with a minimum duration of 1 T_{bit}. This second field is necessary to allow detection of the start bit of the following synchronization field.

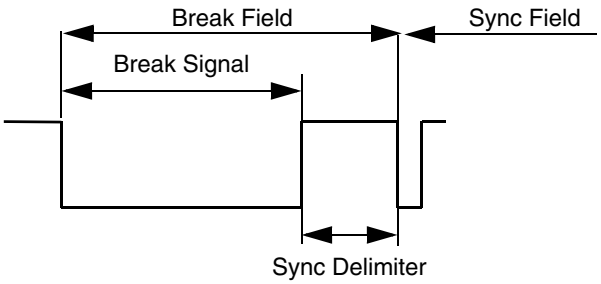

Figure 3-3. Break Field

A logic zero bus state is recognized as the first part of the break field if it is longer than any other regular zero bit-stream during communication. The first part of the break field is at least thirteen bits long in the master time base, so that a slave node with lost synchronization is able to distinguish between such a break and the maximum possible sequence of zero bits within a message frame. The slave node recognizes the break after ten logic zero bits on the bus.

3.3.2 Synchronization Field

The Sync field contains the information for the clock synchronization. The Sync field is the pattern ‘0x55’ which is characterized by five falling edges within a period of duration 8 T_{bit} (see Figure 3-4). Slave nodes with no crystal oscillators that have lost synchronization can identify the correct LIN bus frequency and adjust their T_{bit}, accordingly.

In LIN08EY16, this field is used only for bit error checking, and not for synchronization. (The LIN synchronization procedure is defined in 3.3.2 Synchronization Field.)

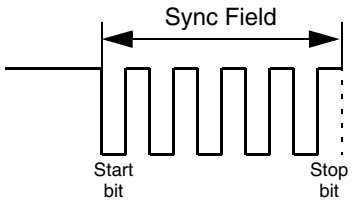

Figure 3-4. Synchronization Field

3.3.3 Identifier Field

The identifier field structure is presented in Figure 3-5. The field contains the content of a message. The content is represented by six *identifier bits* and two *parity bits*.

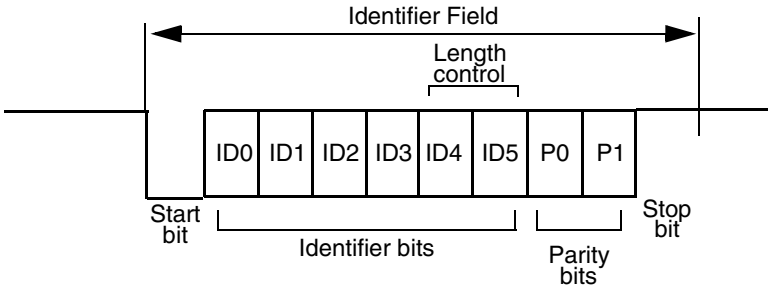

Figure 3-5. Identifier Field

If required (e.g. for compatibility with LIN Specification 1.1), the identifier bits ID4 and ID5 may define the number of data fields in a message. This divides the set of 64 identifiers into four subsets of sixteen identifiers, with 2, 4, and 8 data fields, respectively. The coding of the data length is presented in Table 3-1. In every case, the length of a data field is defined in the configuration description file.

Table 3-1. Standard Message Length

ID5	ID4	Message Length
0	0	2
0	1	2
1	0	4
1	1	8

The parity check bits of the identifier are calculated by a mixed-parity algorithm:

- $P0 = ID0 \oplus ID1 \oplus ID2 \oplus D4$ (even parity)
- $\overline{P1} = ID1 \oplus ID3 \oplus D4 \oplus D5$ (odd parity)

With this algorithm, no pattern with all zero bits or all one bits is possible.

The identifiers 0x3C, 0x3D, 0x3E, and 0x3F with their respective identifier fields 0x3C, 0x7D, 0xFE, and 0xBF are reserved for *command frames* (e.g. *Sleep* mode) and *extended frames*.

3.3.3.1 Reserved Identifiers

Command Frame Identifier

Two *command frame identifiers* are reserved to broadcast general command requests for service purposes from the master to all bus participants. The frame structure is identical to that of a regular 8-byte message frame and is distinguished only by the reserved identifiers:

- 0x3C: ID Field = 0x3C — *Master Request Frame (MasterReq)*,
- 0x3D: ID Field = 0x7D — *Slave Response Frame (SlaveResp)*.

The identifier 0x3C is a *master request frame* to send commands and data from the master to the slave node. The identifier 0x3D is a *slave response frame* that triggers one slave node (being addressed by a prior download frame) to send data to the master node.

Command frames with the first byte of their data field containing a value from 0x00 to 0x7F are reserved; their use will be defined by the LIN Consortium. The remaining command frames can be assigned by the user. First data byte of command frame:

- bit D7 = 0: reserved
- bit D7 = 1: available for use

Sleep Mode Command

The *Sleep* mode command is used to broadcast the *Sleep* mode to all bus nodes. There is no more bus activity after completion of this message until a *wakeup signal frame* on the bus ends the *Sleep* mode. The *Sleep* mode command is a download command frame with the first data field being 0x00.

Extended Frame Identifier

Two *extended frame identifiers* are reserved to allow the embedding of user-defined message formats and future LIN formats into the LIN protocol without violating the current LIN specification. This ensures the upward compatibility of LIN slaves with future revisions of the LIN protocol.

The extended frames are distinguished by the following reserved identifier fields:

- 0x3E: ID Field = 0xFE — user-defined extended frame,
- 0x3F: ID Field = 0xBF — future LIN extension.

The identifier 0x3E (ID field = 0xFE) indicates a user defined extended frame which is free for use. The identifier 0x3F (ID field = 0xBF) is strictly reserved for future extended versions of LIN and must not be used in current implementations.

The identifier can be followed by an arbitrary number of bytes in the data field. The frame length, the communication concept, and the data content are not specified here. The length coding of the *ID Field* does apply to these two frames.

A slave receiving the extended frame identifier, and not being in the position to make use of the content, must ignore all subsequent bytes in the data field until the reception of the next Sync Break.

LIN08EY16 drivers ignore any extended frames. The master node does not send frame headers with identifiers 0xFE, 0xBF. All nodes ignore any response fields for frames with identifiers 0xFE, 0xBF.

3.3.4 Data Field

The data field consists of data bytes of a message. It is transmitted by the slave task in response to the master request. The whole data byte is transmitted using the 8N1 scheme, with the LSB of the data byte first.

The number of data bytes can be defined by the user or chosen according to the Identifier Field.

3.3.5 Checksum Field

The checksum field contains the inverted modulo-256 sum over all data bytes. The sum is calculated by "ADD with Carry" where the carry from the addition is added to the LSB of the resulting sum. The checksum byte is transmitted using the 8N1 scheme, with the LSB first.

On reception, the sum of the modulo-256 sum, over all data bytes, and the checksum byte must be equal to 0xFF.

3.4 Error Detection

An acknowledgment procedure for a correctly received message is not defined in the LIN protocol. It can be implemented on the higher levels. The master node checks the consistency of a message being initiated by the master task and being received by its own slave task. In the event of an inconsistency (e.g., a missing slave response or an incorrect checksum), the master task can retransmit the message.

Where a slave has detected an inconsistency, the slave controller saves this information and can provide it on request to the master node in the form of diagnostics information.

This diagnostics information can be transmitted as a regular message frame with a certain identifier.

3.5 Synchronization

Each message frame starts with a synchronization break (Break), followed by a synchronization field (Sync), which includes several falling edges (i.e. logic one to logic zero transitions) in defined periods, which are multiples of the bit time. This period can be measured (e.g., by a timer capture function), and can be used by the slave nodes to calculate their internal timebase.

The synchronization break frame enables slave nodes that have lost synchronization to identify the synchronization field. This allows the slave nodes to have an oscillator tolerance up to 15% of the nominal bit rate.

3.6 Wakeup Signal Frame

The *Sleep* mode of the bus can be terminated by any node by sending a wakeup signal frame. A wakeup signal frame can be sent by any node, but only if the bus was previously in *Sleep* mode and a node-internal request for wakeup is pending.

The format of the LIN wakeup frame is presented in Figure 3-6. During the *wakeup delimiter*, the master node can not send any message header. The length of the wakeup delimiter is $4T_{bit}$ minimum, 64 T_{bit} maximum.

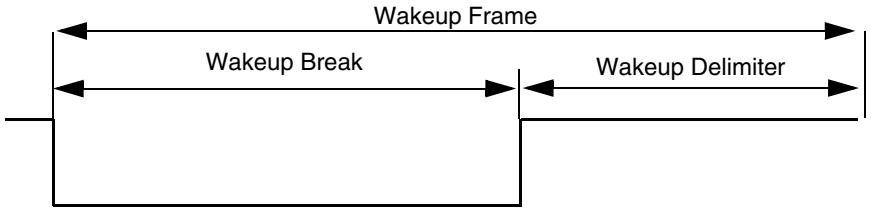

Figure 3-6. LIN Bus Wakeup Frame

Chapter 4
LIN Driver

4.1 Driver Configuration

The driver has a static configuration; it can not be changed during run-time.

The driver can be configured to use either:

- Freescale API for slave node, or
- LIN API for slave node.

This is chosen by the proper configuration files, header files and library. The LIN API is compatible with [1].

4.1.1 LIN API Configuration

To use the LIN API, the user must place `#include <l_api.h>` in the application file and use following files:

- l_gen.h
- l_gen.c

These files are generated from the LIN Description File (LDF) by a special utility. The LDF is written using a special configuration language. The description of the language can be found in [1].

4.1.2 Freescale API Configuration

To use the Freescale API, the user must place `#include <linapi.h>` in the application file and create following files:

- lincfg.h
- linmsgid.h

Samples of these files are located in the *inc* sub-directory of the driver installation directory. The files can be used several ways, as follows.

The user can edit the files in their origin location for every application. In this case, it is necessary to correct the files for every new application.

Alternatively, the user can create a separate file with any name for each different application. To bind the configuration files to the application, the user can define two macros in the compiler command line or compiler command file during the compilation phase:

- LINCFGH — for the file that substitutes the *lincfg.h* file
- LINMSGIDH — for the file that substitutes the *linmsgid.h* file.

This technique is used in the sample application that is described in Appendix A Sample Application.

It is also possible to have only one configuration file, which contains all the information from both.

To use slave node, the user should define the macro SLAVE in the compiler command line or compiler command file during the compilation phase:

4.1.2.1 Driver Configuration File (LINCFG.H)

File *lincfg.h* contains the general LIN configuration. It contains the definition of some constants used by the driver. It is usually the same for all nodes, if they all use the same target hardware.

The file configures the following parameters:

LIN_BAUDRATE

This definition configures the LIN bus baud rate. This value must be set according to the usage of the target MCU SCI register. This means that, for the HC08EY16, this 8-bit value will be masked by 0x37 and put into the SCBR register.

LIN_IDLETIMEOUT

This definition configures the number of calls of the *LIN_IdleClock()* function, after which the *No-Bus-Activity* timeout is considered to have expired (see 4.3.1 No-Bus-Activity). The value of the parameter can not exceed 0xFFFF.

LIN_SCIPRESCALERDIVISOR

This definition configures the ESCI prescaler divisor. The driver will compute the ESCI Prescaler register value during the pre-compilation time and set this value in the register during driver initialization.

LIN_SYNC_SLAVE

When defined this forces the LIN driver to use the self-synchronization capabilities of the ESCI module.

4.1.2.2 Message Configuration File (LINMSGID.H)

File *linmsgid.h* contains the definition of direction for each message received or transmitted by the node and the user-defined length of each message with non-standard length.

Message Direction

All messages sent or received by the node should be defined in the message configuration file. This file is usually different for each node.

The format of the message direction definition is

```
#define LIN_MSG_xx direction
```

where “xx” stands for the hexadecimal message identifier (00 through 3F), and “direction” defines the direction of message transfer:

- *LIN_RECEIVE* — message is received by the node
- *LIN_SEND* — message is always transmitted by the node
- *LIN_SEND_UPDATED* — message is transmitted by the node only if response data has changed by the user since the last transmission of this message (this type of transmission may be used for *event triggered frames and slave response command frames*).

If the message is not defined in the file it will be ignored by the node, i.e. all the API services for this message will return *LIN_NO_ID* status.

Example of message direction definition:

Message with identifier 0x1A is received by the node; message with identifier 0x1B is always transmitted by the node; message with identifier 0x3D is transmitted by the node if its data has been changed. The following statements in the message configuration file should be used.

```
#define LIN_MSG_1A LIN_RECEIVE
#define LIN_MSG_1B LIN_SEND
#define LIN_MSG_3D LIN_SEND_UPDATED
```

After the direction of a message is defined in the file, the user should use the same 6-bit message identifier to refer to a certain message in the API services, e.g.

```
LIN_PutMsg (0x1A, LIN_data);
```

Message Length

The LIN driver has the capability to use user-defined data lengths, as specified in [1] and LIN specification v1.1 standard length coding (see Table 3-1).

All user-defined lengths of messages transmitted or received by the slave node should be defined in the message configuration file for this slave node.

All user-defined lengths of messages processed in the network should be defined in the message configuration file. The format of message length definition is:

```
#define LIN_MSG_xx_LEN len
```

where “xx” stands for the hexadecimal message identifier (00 through 3D), and “len” defines the number of the data bytes in the message with this identifier (in the range from 1 through 8).

The user-defined length for the particular message must be defined identically on each slave node that transmits or receives this message and on the master node on a LIN network.

It is not necessary to define LIN specification v1.1 standard lengths of messages (see Table 3-1). They are defined automatically by the driver.

Example of message length definition: Message with identifier 0x1A has five data bytes; message with identifier 0x1B has eight data bytes. The following statements in message configuration file should be used.

```
#define LIN_MSG_1A_LEN 5
#define LIN_MSG_1B_LEN 8
```

4.2 Error Handling

Error handling is performed via shift counters. There are two counters *Rx* and *Tx* for receive and transmit errors, respectively. Both counters contain history on the eight last receive or transmit attempts.

If there was an error during frame reception or transmission, the Rx or Tx counter is shifted to the right and the MSB is set to 1. Otherwise, it is shifted to the right, and the MSB is set to 0. In either case, only one counter is shifted; it depends on whether the node transmitted or received the data.

Table 4-1. Error Counter Processing

Situation	Rx Counter	Tx Counter
Message response received without errors	0	n
Message response transmitted without errors	n	0
Error during message response reception	1	n
Error during message response transmission	n	1
Error during message header reception by the slave node	1	n
Message response is ignored by the node and processed without errors	0	n
Error during processing of the message Response that is ignored by the *Slave node*	0	n

NOTE

'0' - error counter is shifted to the right and MSB is set to 0;

'1' - error counter is shifted to the right and MSB is set to 1;

'n' - error counter is not shifted.

The following message error types are detected by the LIN driver.

4.2.1 Bit Error

A node that is sending a bit on the bus also monitors the bus. A *bit error* is detected when the byte value that is sent to the bus is different from the byte value that is received.

Also, a node that is receiving a *message response* and discovers a *frame error* (i.e. stop bit is zero) detects this error.

4.2.2 Checksum Error

A *checksum error* is detected by a node that is configured to receive the message data. The error condition occurs if the modulo-256 sum over all received message data bytes and the checksum does not result in '0xFF'.

4.2.3 Inconsistent-Sync-Field Error

An *inconsistent-sync-field error* is detected when a received Sync field byte value is different from 0x55.

An *inconsistent-sync-field error* is detected when the received Sync field can not be used for synchronization, i.e. the measured T_{bit} differs from the statically configured T_{bit} value by more than 15%.

4.3 Timeout Handling

One timeout is maintained on the master and slave nodes. This timeout is described below.

4.3.1 No-Bus-Activity

When the Freescale API is used, a *No-Bus-Activity* condition must be detected if no Break fields were recognized on the bus for more than N_{idle} times of the *LIN_IdleClock()* service calls since:

- *LIN_Init()* call, or
- *LIN_GotoRun()* call, or
- the end of previous valid Break field,

whichever occurs last.

When the LIN API is used, the *No-Bus-Activity* condition has to be detected on the particular node if no Break fields were recognized on the bus for more than N_{idle} times of *l_ifc_ioctl_iii (l_op_idleclock, NULL)* service calls since:

- *l_sys_init()* call, or
- *l_ifc_connect_iii ()* call, or
- the end of previous valid Break,

whichever occurs last.

Chapter 5
Freescale API

5.1 General

This section provides a detailed description of Freescale LIN driver run-time services, with appropriate examples. All predefined LIN driver data types can be found in 5.2 Data Types.

5.2 Data Types

It is considered that LIN driver services use the following naming conventions for data types:

...Type: describes the values of individual data.

...RefType: describes the identifier referencing an object(1).

The following standards are used for variable types.

Table 5-1. Data Types

Mnemonic	C type	HC08 Implementation
LINStatusType	unsigned char	unsigned 8 bits
LINDriverStatusType	unsigned char	unsigned 8 bits
LINMsgIdType	unsigned char	unsigned 8 bits
LINMsgRefType	unsigned char	unsigned 16 bits
LINErrCounterType	unsigned char	unsigned 8 bits
LINSyncPDType	unsigned char	unsigned 8 bits

5.3 Constant Definition

There are predefined constant values for some of the data types.

Table 5-2. LIN Status Constant Values

Constant Value	Description
LIN_OK	No error; service call has succeeded.
LIN_MSG_NODATA	The message data buffer is empty (data has not been initialized or received yet).
LIN_MSG_NOCHANGE	The message data has not changed since last read.

1. For example, a pointer or an index.

Table 5-2. LIN Status Constant Values (Continued)

Constant Value	Description
LIN_MSG_OVERRUN	The message data has not been read and was overwritten.
LIN_REQ_PENDING	The message request is already pending.
LIN_INVALID_MODE	The service could not be called in the current driver state.
LIN_INVALID_ID	The message identifier is invalid, i.e. the message direction differs from the configured one.
LIN_NO_ID	The message identifier is absent, i.e. there is no such identifier configured on this node.

Table 5-3. LIN Driver Status Constant Values

Constant Value	Description
LIN_STATUS_RUN	Driver is in *Run* state
LIN_STATUS_IDLE	*No-Bus-Activity* timeout has expired
LIN_STATUS_PENDING	LIN bus frame is pending

5.4 Driver Services

5.4.1 LIN_Init

Syntax: `void LIN_Init( void );`

Applicable: Slave

Parameters: None.

Return: None.

Description: The *LIN_Init* service performs software initialization of the LIN driver:

- sets the current driver state to *Run*,
- clears error counters,
- resets *No-Bus-Activity* condition counter,
- changes all message buffers status as it does not contain data,
- sets implementation specific internal states and variables in initial state.

The *LIN_Init* service also performs hardware initialization of the LIN driver:

- sets statically configured baud rate,
- sets Tx pin to idle (recessive) state.

Notes: This service should be called before any another LIN driver API service call. Otherwise, the result of any another LIN services and the LIN driver behavior will be unpredictable.

This service call aborts immediately all other LIN driver activity that was in progress, as soon as physical implementation allows but not later than after the end of the current LIN bus frame element (break field, sync delimiter, sync field, ID field, data field, checksum field, wakeup break) transmission or reception.

5.4.2 LIN_Wakeup

Syntax: `LINStatusType LIN_Wakeup( void );`

Applicable: Slave

Parameters: None.

Return:
- *LIN_OK* — the LIN bus wakeup frame has been issued successfully.
- *LIN_INVALID_MODE* — the current driver state is *Sleep.*
- *LIN_REQ_PENDING* — another LIN bus frame is currently being processed.

Description: The *LIN_Wakeup* service issues the LIN bus wakeup frame transmission. No LIN bus wakeup frame is issued if:
- current driver state is *Sleep*; in this case *LIN_INVALID_MODE* is returned;
- another LIN bus frame is being transmitted or received by the node, or another wakeup frame is begin transmitted by the node; in this case *LIN_REQ_PENDING* is returned.

Notes: The end of wakeup frame transmission on a slave node is just after the end of wakeup break. Therefore, on a slave node, the user should not call the *LIN_Wakeup* service again before the end of the wakeup delimiter.

5.4.3 LIN_GotoRun

Syntax: `void LIN_GotoRun( void );`

Applicable: Slave

Parameters: None.

Return: None.

Description: The *LIN_GotoRun* service changes the current driver state from *Sleep* to *Run* and resets the *No-Bus-Activity* condition counter. If the current driver state is *Run*, this service call does nothing.

5.4.4 LIN_DriverStatus

Syntax: `LINDriverStatusType LIN_DriverStatus( void );`

Applicable: Slave

Parameters: None.

Return: Actual driver status bit array. Particular status should be found by applying the mask constants specified in Table 5-3.

Description: The *LIN_DriverStatus* service returns the actual status of the LIN driver in bit array form.

Example:

```
void main( void )
{
LIN_Init();
while( 1 )
{
if(LIN_DriverStatus() != LIN_STATUS_RUN)
   {
   do
     {
     /*sleep mode*/
     } while(LIN_DriverStatus() != LIN_STATUS_RUN);
   }
}
```

5.4.5 LIN_GetMsg

Syntax: `LINStatusType LIN_GetMsg( LINMsgIdType <MsgId>, LINMsgRefType <Data> );`

Applicable: Slave

Parameters: <MsgId> — defines message identifier.

<*MsgData> — pointer to the user memory buffer where received data will be copied.

Return:
- *LIN_OK* — data has been successfully transferred to the memory buffer.
- *LIN_NO_ID* — the message identifier is absent, i.e. there is no such identifier configured on this node.
- *LIN_INVALID_ID* — the message identifier is invalid, i.e. the message with this identifier is configured on this node for transmission, but the current content has been successfully retrieved from the message buffer.
- *LIN_MSG_NODATA* — the message data has not been received since driver initialization.

Description: The *LIN_GetMsg* service retrieves the current content of the specified message to the specified memory location. This service also updates the status information of the message accordingly.

Notes: If this service returns code *LIN_NO_ID* or *LIN_MSG_NODATA*, then the memory buffer addressed by the input parameter is unchanged.

The user can read the data from the buffer configured for transmission, but if this buffer was not written by the user, the service returns unpredictable data.

Example:

```
#define MSG_TEMP    0x12
#define MSG_SPEED   0x13

unsigned char msgSpeedBuf[2];
unsigned char msgTempBuf[2];

void  main( void )
{
          LIN_Init();

          while( 1 )
          {
               LIN_PutMsg( MSG_SPEED, msgSpeedBuf );
               while( LIN_GetMsg( MSG_TEMP, msgTempBuf ) != LIN_OK )
               ;
               msgSpeedBuf[0] = msgTempBuf[0] * 2 + 10;
          }
}
```

5.4.6 LIN_PutMsg

Syntax: `LINStatusType LIN_PutMsg( LINMsgIdType <MsgId>, LINMsgRefType <Data> );`

Applicable: Slave

Parameters: <MsgId> — defines message identifier.

<Data> — pointer to the user memory buffer from where data will be transmitted.

Return: *LIN_OK* — data has been successfully transmitted from the memory buffer to the message buffer.

LIN_NO_ID — the message identifier is absent, i.e. there is no such identifier configured on this node.

LIN_INVALID_ID — the message identifier is invalid, i.e. the message with this identifier is configured on this node for reception.

Description: The *LIN_PutMsg* service transmits the current content of the specified memory location to the specified message. This service also updates the status information of the message accordingly.

Notes: This data will be transmitted to the bus on request from the master during the next poll. The service itself does not cause transmission on the bus. The message buffer addressed by the input parameter remains unchanged.

Example: See the example in 5.4.5 LIN_GetMsg.

5.4.7 LIN_MsgStatus

Syntax: `LINStatusType LIN_MsgStatus( LINMsgIdType <MsgId> );`

Applicable: Slave

Parameters: <MsgId> — defines message identifier.

Return:

- *LIN_NO_ID* — the message identifier is absent, i.e. there is no such identifier configured on this node.

When message is specified as received:

- *LIN_OK* — new message has been successfully received since the last read via the *LIN_GetMsg* service call.
- *LIN_MSG_NOCHANGE* — the message data has not been changed since the last read via the LIN_GetMsg service call.
- *LIN_MSG_NODATA* — no message data has been received since driver initialization.
- *LIN_MSG_OVERRUN* — the message data was not been read via the *LIN_GetMsg* service call and was overwritten by a further message with the same identifier.

When message is specified as always transmitted:

- *LIN_OK* — master request for this message has been received by the node since the last message update via the *LIN_PutMsg* service call (it does not matter if response transmission was successful or not)
- *LIN_MSG_NOCHANGE* — header for this message has not been received by the node since last message update via the *LIN_PutMsg* service call
- *LIN_MSG_NODATA* — message data has not been updated via the *LIN_PutMsg* service call since driver initialization.

Description: The *LIN_MsgStatus* service returns the current status of the specified message. This service has not updated the status information of the message.

Notes: The return code values depend on specified message direction (if this message transmitted or received).

5.4.8 LIN_GetRxErr

Syntax: `LINErrCounterType LIN_GetRxErr( void );`

Applicable: Slave

Parameters: None.

Return: Actual receive error bit queue.

Description: The *LIN_GetRxErr* service provides error information (receive error counter) for latest eight frames received or ignored by the LIN driver. It returns a bit queue, where each bit specifies presence (when the bit is set) or absence (when the bit is cleared) of receive errors for the particular received or ignored messages. The most significant bit presents the latest message status. See 4.2 Error Handling.

The following errors are counted:

- Bit error
- Checksum error
- *Inconsistent-sync-field* error

Notes: The error counter is cleared after driver initialization.

5.4.9 LIN_GetTxErr

Syntax: `LINErrCounterType LIN_GetTxErr( void );`

Applicable: Slave

Parameters: None.

Return: Actual transmit error bit queue.

Description: The *LIN_GetTxErr* service provides error information (transmit error counter) for the latest eight frames transmitted by the LIN driver. It returns a bit queue, where each bit specifies the presence (when the bit is set) or absence (when the bit is cleared) of transmit errors for the particular transmitted messages. The most significant bit represents the latest message status. See 4.2 Error Handling.

The following errors are counted:

- Bit error

Notes: The error counter is cleared after driver initialization.

5.4.10 LIN_ClearRxErr

Syntax: `void LIN_ClearRxErr( void );`

Applicable: Slave

Parameters: None.

Return: None.

Description: The *LIN_ClearRxErr* service clears the receive error counter.

5.4.11 LIN_ClearTxErr

Syntax: `void LIN_ClearTxErr( void );`

Applicable: Slave

Parameters: None.

Return: None.

Description: The *LIN_ClearTxErr* service clears the transmit error counter.

5.4.12 LIN_IdleClock

Syntax: `void LIN_IdleClock( void );`

Applicable: Slave

Parameters: None.

Return: None.

Description: The *LIN_IdleClock* service updates the *No-Bus-Activity* condition counter by one (See 4.3.1 No-Bus-Activity) and checks if the condition is met. When the counter reaches the limit, the corresponding bit is set in the driver status constant, according to Table 5-3.

5.4.13 LIN_GetSyncPD

Syntax: `LINSyncPDType LIN_GetSyncPD( void );`

Applicable: Slave

Parameters: None.

Return: Actual prescaler divisor..

Description: If the self-synchronization of the driver is enabled (define macro LIN_SYNC_SLAVE in driver configuration file) then the driver will return the latest synchronized prescaler divisor. With this value the application can compute the trim value for the ICG (Internal Clock Generator). If the macro LIN_SYNC_SLAVE is not defined then the driver will return the user defined prescaler divisor.

5.5 Call-back Services

5.5.1 LIN_Command

Prototype: `void LIN_Command( void );`

Applicable: Slave

Parameters None.

Return None.

Description: The *LIN_Command* must be defined in the user application to perform specific command actions. The *LIN_Command* service is called by the driver after successful reception of the master request frame (ID Field value 0x3C) if the node is configured to receive this frame.

From the *LIN_Command* call-back service, the user can call the *LIN_GetMsg(0x3C, data)* service and will get the data of the master request frame just received.

If the successfully transmitted command is the *Sleep* mode command, then the driver changes its state from *Run* to *Sleep* before calling the *LIN_Command* call-back.

Notes: The *LIN_Command* call-back is called from driver's interrupt. Therefore, the user must not enable CPU interrupts inside this call-back. Care must be taken when using any function call from this call-back, as it can use a large number of stack locations.

Chapter 6
LIN API

6.1 General

This section provides a detailed description of LIN run-time services, with appropriate examples. All predefined LIN driver data types can be found in 5.2 Data Types.

All services are applicable only to slave nodes.

6.2 Data Types

The following standard types are used for variables:

Table 6-1. Data Types

Mnemonic	C type	HC08 Implementation
l_bool	unsigned char	unsigned 8 bits
l_u8	unsigned char	unsigned 8 bits
l_u16	unsigned int	unsigned 16 bits
l_ioctl_op	unsigned char	unsigned 8 bits
l_irqmask	unsigned char	unsigned 8 bits

6.3 Driver Services

6.3.1 l_sys_init

Syntax: `l_bool l_sys_init( void );`

Parameters: None.

Return: Always returns 0.

Description: The *l_sys_init* service performs software initialization of the LIN driver, i.e.:

- clears all signal flags,
- clears error counters,
- resets *No-Bus-Activity* condition counter,
- changes all transmitted messages status so they do not contain data.

This service call aborts any LIN driver communication activity as soon as physical implementation allows, but not later than after the end of the current LIN bus frame element (break field, sync delimiter, sync field, ID field, data byte, checksum byte, wakeup frame) transmission or reception.

This service call puts the LIN driver into the disconnected state.

This routine must be executed as a first LIN driver API service call after power-on reset. It should be executed before any another LIN driver API service call. Otherwise the result of any another LIN services and the LIN driver behavior will be unpredictable.

Note: This service must not be called from any interrupt service routine.

6.3.2 l_bool_rd

Syntax: `l_bool l_bool_rd_sss( void )`

Parameters: None.

Return: The service returns zero if the current value of the *sss* signal is zero, and a non-zero value otherwise.

Description: The *l_bool_rd* service returns the current value of signal of size one bit, statically configured with the *sss* name. Service calls for configured signal names only are applicable.

6.3.3 l_u8_rd

Syntax: `l_u8 l_u8_rd_sss( void )`

Parameters: None.

Return: The service returns the current value of the *sss* signal. The return value is assumed to be right-aligned if the signal size is less than eight bits.

Description: The *l_u8_rd* service returns the current value of the signal of size from 2–8 bits, statically configured with the *sss* name. Service calls for configured signal names only are applicable.

6.3.4 l_u16_rd

Syntax: `l_u16 l_u16_rd_sss( void )`

Parameters: None.

Return: The service returns the current value of the *sss* signal. The return value is assumed to be right-aligned if the signal size is less than sixteen bits.

Description: The *l_u16_rd* service returns the current value of the signal of size 9–16 bits, statically configured with *sss* name. Service calls for configured signal names only are applicable.

6.3.5 l_bool_wr

Syntax: `void l_bool_wr_sss( l_bool <v> )`

Parameters: <v> — defines signal data.

Return: None.

Description: The *l_bool_wr* service sets the current value of the signal of size one bit, statically configured with *sss* name, to “0” if the value <v> is zero, and to “1” otherwise. Service calls for configured signal names only are applicable.

6.3.6 l_u8_wr

Syntax: `void l_u8_wr_sss( l_u8 <v> )`

Parameters: <v> — defines signal data.

Return: None.

Description: The *l_u8_wr* service sets the current value of the signal of size 2–8 bits, statically configured with *sss* name, to the value <v>. It is assumed that the <v> parameter is right-aligned for signal lengths less than eight bits. Service calls for configured signal names only are applicable.

6.3.7 l_u16_wr

Syntax: `void l_u16_wr_sss( l_u16 <v> )`

Parameters: <v> — defines signal data.

Return: None.

Description: The *l_u16_wr* service shall set the current value of the signal of the size 9–16 bits, statically configured with *sss* name, to the value <v>. It is assumed that the <v> parameter is right-aligned for signal lengths less than sixteen bits. Service calls for configured signal names only are applicable.

6.3.8 l_flg_tst

Syntax: `l_bool l_flg_tst_Rxsss( void )`

Parameters: None.

Return: The service returns non-zero value if signal *sss* is received by the node, and zero otherwise.

Description: The *l_flg_tst* service returns the current state of the signal flag, statically configured with *sss* name. The signal flag is set to “1” when the message frame contained by this signal arrives successfully, i.e. the signal value is updated by driver software. This service is applicable only to signals that are received at the node.

6.3.9 l_flg_clr

Syntax: `void l_flg_clr_Rxsss( void )`

Parameters: None.

Return: None.

Description: The *l_flg_clr* service sets the current state of the signal flag, statically configured with *sss* name, to zero. This service is only applicable to signals that are received at the node.

6.3.10 l_ifc_init

Syntax: `void l_ifc_init_iii( void )`

Parameters: None.

Return: None.

Description: The *l_ifc_init* service performs hardware initialization of the LIN driver, i.e.:

- sets the statically configured baud rate,
- sets the Tx pin to the high voltage level,
- disables LIN bus frame transmission or reception until the *l_ifc_connect* call.

It is assumed that *iii* is the statically configured LIN interface name. This service call aborts any LIN driver communication activity, not later than after the end of current LIN bus frame element (break field, sync delimiter, sync field, ID field, data byte, checksum byte, wakeup frame) transmission or reception.

This service call puts the LIN driver into the disconnected state. The service only initializes the communication hardware; incoming frames are not processed until *l_ifc_connect* service is called.

6.3.11 l_ifc_connect

Syntax: `void l_ifc_connect_iii( void )`

Parameters: None.

Return: The service returns a non-zero value if it is called before the *l_ifc_init* service; otherwise it returns a zero value.

Description: The *l_ifc_connect* service enables LIN frame transmission, reception and error detection. It is assumed that *iii* is the statically configured LIN interface name.

This service call puts the LIN driver into the connected state.

If the *l_ifc_connect* service is called when the interface is already connected, then it performs nothing.

6.3.12 l_ifc_disconnect

Syntax: `void l_ifc_disconnect_iii( void )`

Parameters: None.

Return: The service returns non-zero value if it is called before *l_ifc_init* service; otherwise it returns zero value.

Description: The *l_ifc_disconnect* service disables LIN frame transmission, reception and error detection. It is assumed that *iii* is the statically configured LIN interface name.

This service call aborts any LIN driver communication activity not later than after the end of current LIN bus frame element (break field, sync delimiter, sync field, ID field, data byte, checksum byte, wakeup frame) transmission or reception. All actions are performed only if the driver is in the connected state.

This service call puts the LIN driver into the disconnected state. This allows the user to disable the LIN driver temporarily and use the communication hardware. When needed, the LIN driver can be enabled by the *l_ifc_connect* service. If, during the disconnected state, the user has changed any SCI hardware settings, then the *l_ifc_init* service must be called before the *l_ifc_connect* service.

If the *l_ifc_disconnect* service is called when the interface is already disconnected, then it performs nothing.

6.3.13 l_ifc_ioctl

Syntax: `void l_ifc_ioctl_iii( i_ioctl_op <op>, void *<pv> )`

Parameters: <op> — specifies required functionality.

<pv> — specifies the pointer to the returned parameter (see Table 6-2).

Return: None.

Description: The *l_ifc_ioctl* service provides protocol-specific functionality. See Table 6-2 for details. It is assumed that *iii* is the statically configured LIN interface name.

Table 6-2. *l_ifc_ioctl* Functionality Description

<op> parameter value	Functionality	*<pv>* parameter value
l_op_getrxerr	Provides error information (receive error counter) for the eight most recent frames received or ignored by the LIN driver. The counter represents as a bit queue, where each bit shall specify presence (when the bit is set) or absence (when the bit is cleared) of receive errors for the particular received or ignored messages. Most significant bit presents the latest message status. The following errors are counted: - Bit error; - Checksum error; - Inconsistent-sync-field Error.	Pointer to receive errors bit queue (*l_u8**)
l_op_gettxerr	Provide error information (transmit error counter) for latest 8 received or ignored frames by the LIN driver. The counter represents as a bit queue, where each bit shall specify presence (when the bit is set) or absence (when the bit is cleared) of transmit errors for the transmitted messages. Most significant bit presents the latest message status. The following errors are counted: - Bit error.	Pointer to transmit errors bit queue (*l_u8**).
l_op_clrrxerr	Clear the receive error counter of the LIN driver.	Not used
l_op_clrtxerr	Clear the transmit error counter of the LIN driver.	Not used
l_op_wakeup	Issues the LIN bus wakeup frame transmission. No LIN bus wakeup frames shall be issued until: - another LIN bus wakeup frame is transmitted; - another LIN bus frame header is received; - LIN bus frame response is received or transmitted by this node.	Pointer to operation result (*l_bool*). Operation result is zero, if LIN bus wakeup frame is transmitted successfully and non-zero otherwise.
l_op_getidle	Check *No-Bus-Activity* condition. (See 4.3.1 No-Bus-Activity)	Pointer to operation result (*l_bool*). Operation result is non-zero, if *No-Bus-Activity* condition is met and zero otherwise.
l_op_idleclock	Update *No-Bus-Activity* condition (See 4.3.1 No-Bus-Activity) counter by 1.	Not used
l_op_getsyncpd	Provide the actual prescaler divisor. If the self-synchronization of the driver is enabled (define macro LIN_SYNC_SLAVE in driver configuration file) then the driver will return the latest synchronized prescaler divisor. With this value the application can compute the trim value for the ICG (Internal Clock Generator). If the macro LIN_SYNC_SLAVE is not defined then the driver will return the user defined prescaler divisor..	Pointer to the prescaler divisor value.

Example:

```
l_u8 result;
l_ifc_ioctl_sci08 ( l_op_getrxerr, &result );
if ( result != 0 )
{
       /* Call the user's service for error processing */
       error_processing ();
}
else
{
       /* Clear Rx error counter */
       /* Variable <result> is not used by this service */
       l_ifc_ioctl_sci08 ( l_op_clrrxerr, &result );
}
```

6.3.14 l_ifc_rx

Syntax: `void l_ifc_rx_iii( void )`

Parameters: None.

Return: None.

Description: The l_ifc_rx service must be called from:

- a user-defined interrupt handler raised by an SCI when it has received data,
- a user-defined interrupt handler raised by an SCI when it detects a transmission or reception error.

It is assumed that *iii* is the statically configured LIN interface name.

This service always clears all SCI interrupt flags.

If the driver is in the connected state, then the *l_ifc_rx* service performs all actions required to process just received SCI bytes, SCI breaks or SCI errors. It is the user's responsibility to not perform any operations that might modify the state of the SCI before this service call.

If the driver is in the disconnected state, then the *l_ifc_rx* service only clears all SCI interrupt flags.

6.3.15 l_ifc_tx

Syntax: `void l_ifc_tx_iii( void )`

Parameters: None.

Return: None.

Description: The l_ifc_tx service can be called from a user-defined interrupt handler raised by a SCI. It is assumed that *iii* is the statically configured LIN interface name.

The user should call the *l_ifc_tx* or *l_ifc_rx* service. These services have identical functionality.

It is the user's responsibility to not perform any operations that might modify the state of the SCI before this service call.

6.4 Call-back Services

6.4.1 l_sys_irq_disable

Syntax: `l_irqmask l_sys_irq_disable( void )`

Parameters: None.

Return: l_irqmask — original interrupt mask (before disabling).

Description: The *l_sys_irq_disable* call-back is defined in the user application. The implementation of this function achieves a state in which no interrupts can occur.

6.4.2 l_sys_irq_restore

Syntax: `void l_sys_irq_restore( l_irqmask <previous> )`

Parameters: <previous> — previous interrupt mask.

Return: None.

Description: The *l_sys_irq_restore* call-back is defined in the user application. The implementation of this function restores a state identified by the input parameter. It is assumed that the input parameter contains value stored by the previous *l_sys_irq_disable* call.

Chapter 7
Platform Specific

7.1 General

This section covers features that can be helpful during the development of applications using the LIN driver.

For troubleshooting see Appendix A.4 Troubleshooting.

7.2 MCU Resources Usage

The following MCU resources are used by driver and must not be used by the user application code:

- ESCI module registers and interrupt vectors
- PTE0/TxD pin (must be connected to the LIN Physical Interface Tx pin)
- PTE1/RxD pin (must be connected to the LIN Physical Interface Rx pin)

7.3 Physical Interface Connection

It is assumed that the MCU is connected to the LIN bus by means of the Physical Interface. The MCU to LIN Physical Interface connection is shown in Figure 7-1

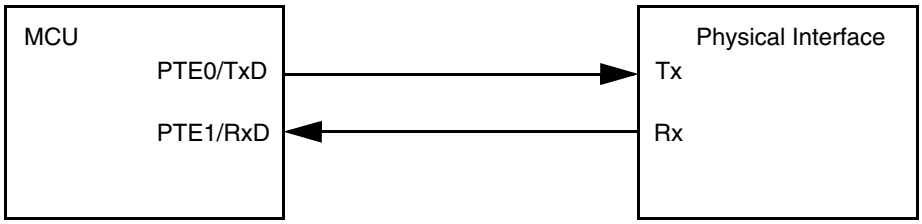

Figure 7-1. Physical Interface Connection

7.4 Disabled Interrupt Code Sections

As the driver code is very time sensitive and interrupt-driven, disabling interrupts in the user application code is not recommended. However, if disabling interrupts is absolutely necessary, make sure that interrupts are disabled for a time not longer than 5 T_{bit} transmission period. Otherwise, timeout accuracy is not guaranteed.

7.5 Break Signal Detection

On an MCU with a stable bus clock and the ESCI LIN driver implementation, the slave node recognizes the Break signal after ten logic zero bits on the bus.

The slave node recognizes an ESCI frame error, while receiving a byte from the bus, as a potential Break signal. If the next received byte is 0x55, the driver starts common message header processing.

7.6 Zero Page Usage

To speed up the driver and to decrease its ROM size some variable are located in the zero RAM page, i.e. addresses from 0x00 to 0xFF.

For information on how to use zero page setting for application building refer to the CodeWarrior sample project.

Chapter 8
Building Application

8.1 General

The driver code is supplied as source code targeted at CodeWarrior compilers. The following sections show general steps of building the user's application named `my_app.c`.

To know how to build sample application please refer to Appendix A.

8.2 Compilation

There are source files supplied with the library, which must be compiled:

Common files:

- Start08.c

LIN API files:

- l_cfg.c

Freescale API files:

- vector.c
- lincfg.c
- linmsgid.c

Driver files:

- lintmr.c
- linerr.c
- lininit.c
- linmsg.c
- linprot.c
- linsci.c
- linapi.c

The recommended compiler versions are defined in the `readme.txt` file in the installation root directory. If other versions are used, problems might arise.

To compile the files properly, the following macros should be defined in the compiler command line or command file at compilation time:

- HC08EY16
- HC08
- CW08

With the Freescale API, the user can use alternative files with LIN network and message configuration by specifying the following macros:

- LINCFGH
- LINMSGIDH

Refer to the CodeWarrior sample project and 4.1.2 Freescale API Configuration for details.

8.3 Linking

The driver object files are linked to the rest of the application by including the object files in the list of files to be linked. A file containing startup code is also required. A file containing basic startup code is supplied with the driver and may be modified and compiled as required.

8.3.1 CodeWarrior

To link the files properly the following segment should be defined:

- VECTORS_DATA — segment for vector table.

An example link file for a Freescale API slave node is shown below.

```
NAMES
  /* other object files to link are passed from the IDF with the linker -Add option */
END

SECTIONS
    LIN_ZRAM    = READ_WRITE 0x0040 TO 0x00FF;          /* zero page*/
    LIN_RAM     = READ_WRITE 0x0100 TO 0x01FF;          /* program data */
    LIN_STACK   = READ_WRITE 0x0200 TO 0x023F;          /* stack*/
    LIN_ROM     = READ_ONLY  0xC000 TO 0xFDFF;         /* program code & constants */
    LIN_VECTORS = READ_ONLY  0xFFDC TO 0xFFFF;        /* interrupt vectors (use your
vector.obj) */
END

PLACEMENT
    ZeroSeg, _DATA_ZEROPAGE INTO LIN_ZRAM;
    DEFAULT_ROM, ROM_VAR    INTO LIN_ROM;
    DEFAULT_RAM             INTO LIN_RAM;
    SSTACK                  INTO LIN_STACK;
    VECTORS_DATA            INTO LIN_VECTORS;
END

STACKSIZE 0x001F

ENTRIES
 _vectab
END
```

Appendix A
Sample Application

A.1 Sample Description

The sample application is located in the `sample` directory of the LIN installation.

This sample illustrates the operation of a simple LIN network and is based on Freescale's LINKits hardware. It contains master and slave nodes with Freescale API.

The sample runs using a master and any combination of up to sixteen slaves. Each slave controls four LEDs whose states can be controlled by a single push-button switch. The resulting four bits of data are returned to the master and displayed on four of its eight LEDs. The other four LEDs on the master are used to indicate the slave type and ID. Two LEDs show the slave type (GR, EY, QY or QL) and the other pair correspond to the four IDs allocated to that particular type. If more than one slave is connected, then the master's display cycles round all those present on the bus. The sending of header frames from the master can also be switched off, to demonstrate the slaves' ability to enter low-power *Sleep* mode in the absence of LIN activity.

Application Note AN2573 describes the sample in detail (see http://www.freescale.com).

A.2 Sample Building and Running

To build the sample, CodeWarrior software should be installed on the computer. For supported compiler and debugger versions, refer to the `readme.txt` file in the sample directory.

If a project based approach is *not* used, the following environment variables must be set before building the application:

For CodeWarrior C:

- HICROSSINC — path to CodeWarrior HC08 include files,
- HICROSSLIB — path to CodeWarrior HC08 library files in ELF 2.0 format.
- Add path to CodeWarrior executable files to the PATH environment variable.

NOTE

This sample includes direct support of the following HC08 derivatives:

– *MC68HC08EY16*

A.3 CodeWarrior Project

The sample comes with a CodeWarrior project that allows simple access to all files and ready configured code generation settings.

The project is called `sample.mcp` and contains slave target settings.

A.4 Troubleshooting

A.4.1 Environment settings

If the application does not compile, check that all required environment variables are set and the PATH variable includes path to the compiler.

A.4.2 Startup Files

The driver is supplied with startup files for the compilers listed in the *readme.txt* file. These files can be changed, to allow you to have different compiler versions, or to replace these startup files with your own versions. In these circumstances, you must use the driver at your own risk.

A.4.3 LinSigFlags Size

If there is no received signal on the node, and thus the size of the *LinSigFlag* array in the *l_gen.c* file is zero, the code will not compile. In this case the size of *LinSigFlag* array should be explicitly set to 1.

Appendix B
Performance Characteristics

B.1 Performance Characteristics

CPU performance is calculated as:

$L = T_{active} / T_{frame} * 100\%$

where:

- L is the percentage CPU load,
- T_{active} is the amount of CPU time expended in executing the driver code in a period T_{frame},
- T_{frame} is the amount of time taken to transmit or receive a regular LIN bus frame of maximum length, containing eight bytes of data.

The performance characteristics are presented in Table B-1.

Table B-1. Performance Characteristics

Node	LIN Baud Rate (bps)	MCU Bus Frequency (MHz)	MCU Load (%)
Slave (Freescale API)	19,200	3.6864	< 6
Slave (LIN API)	19,200	3.6864	< 7

B.2 Memory Consumption

The results on memory consumption are presented in Table B-2.

Table B-2. Memory Consumption

Item	RAM(1)	ROM(2)	Stack
Slave (Freescale API)	20 bytes	1245 bytes	< 21 bytes
Slave (LIN API)	20 bytes	1007 bytes	< 21 bytes

NOTES:
1. Plus 1 byte per message.
2. Plus 4 bytes per message.

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

LIN08EY16DUM
Rev. 2
10/2005