Application Note, V 1.3, August 2004

AP16086

LIN-Driver

Software implemented LIN Protocol for Infineon's 16 bit microcontrollers

Microcontrollers

Never stop thinking.

Revision History:	2004-08	**V 1.3**
Previous Version:	-	

Page	Subjects (major changes since last revision)
All pages	Document Version 0.63 ported to Infineon corporate design
All pages	Document Version 1.0 changes included
All pages	Document Version 1.1 ported to new Infineon design
All pages	Document Version 1.2 changes to wording

Controller Area Network (CAN): License of Robert Bosch GmbH

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Table 1 Abbreviations

Abbreviation	**Meaning**
LIN	Local Interconnect Network
CAN	Controller Area Network
ECU	Electronic Control Unit
API	Application User Interface
IDE	Integrated Development Environment
UART	Universal Asynchronous Receiver Transmitter
ASC	Asyn/Synchronous Serial Interface
ID	Identifier
EMI	Electro Magnetic Isolation

Edition 2004-08

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents **Page**

1 Introduction

This application note gives an overview of the LIN protocol, showing typical applications for the LIN subbus and describing the operation of the LIN software driver for Infineon Technologies 16bit microcontrollers. The implementation of the LIN software driver will be discussed step by step. Moreover, two projects are attached to build a simple LIN network with a C16x device (master node and one slave node).

2 LIN Bus

The introduced serial communication protocol LIN is qualified to link distributed systems especially in automotive applications. LIN will be used as a standardized local subbus within clustered systems. Moreover, this bus means a low cost silicon implementation, because LIN is based on a standard UART data format. However, LIN is not going to replace the well-established CAN network. LIN will link several nodes to a central CAN access point. With the introduction of LIN the migration towards distributed systems, will speed-up. LIN enables new system partitioning especially in the field of body and convenience applications.

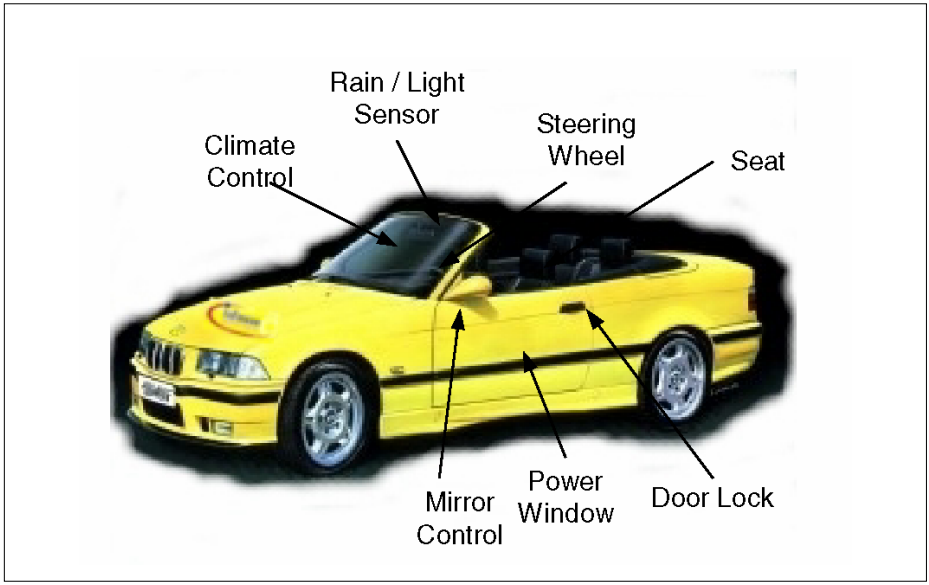

Figure 1 LIN Applications

Until now, high/low speed CAN and J1850 buses have been the standard in-vehicle networks. These buses provide high performance with baudrates up to 1Mbaud. The LIN subbus will not enter into competition with any of the high performance buses. On the contrary, the combination of both types of buses will give a lot of advantage to system designers. A huge number of body and convenience applications do not need the excellent performance of the CAN bus for local communication paths. What these applications really need is a standardized local subbus with a well-balanced price/performance ratio. The LIN bus addresses all these demands and provides many advantages:

- compatibility between different subbus applications from different developers
- avoids reengineering of the nearly same development
- easy monitoring/diagnostics and error detection of the whole system
- tools for homogeneously development
- pre-development possible through emulation
- easy integration of new nodes in an existing system
- cost reduction through wire replacement and minimized development-time

2.1 Evolution of Automotive Networking

LIN has a strong impact on vehicle system architecture. Today's centralized system use dedicated communication lines to pass analog/digital data from a central Electronic Control Unit (ECU) to actuator and sensor units or vice versa. Each relevant state or event is coded analog to a dedicated channel. Most of these centralized systems use the CAN bus as communication backbone.

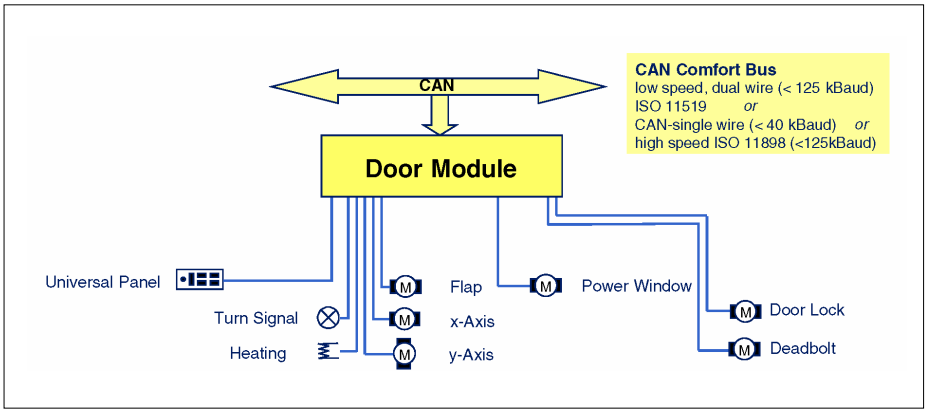

Figure 2 Centralized System

To migrate a system from centralized to a distributed system it has to be broken up into subsystems with common, standardized, and open interfaces. Some modules have to control actuators and sensors with low priority e.g. mirror positioning. All of these low-priority modules do not require the performance of CAN and could be controlled in a more cost effective way without using expensive CAN transceivers and CAN modules. The LIN subbus closes the gap between wire topology and CAN bus topology. This solution has the following advantages:

- cost optimized connection of all actuators and sensors to a standardized bus-system
- enhanced failure detection and bus diagnosis
- easy function extension

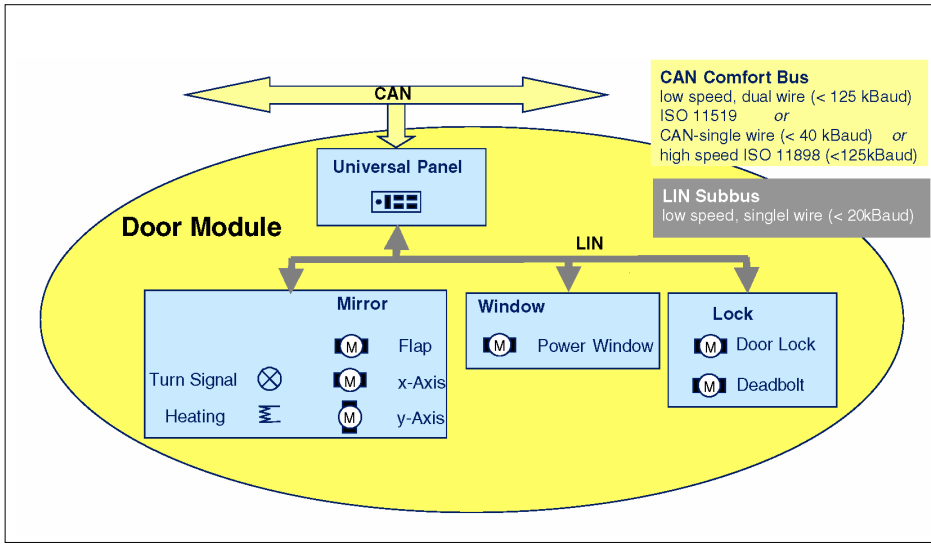

Figure 3 Distributed System with LIN

2.2 LIN bus Details

LIN is a single-master multiple-slave bus system. One master node and several slaves build up the bus system. The LIN bus is a single-wire bus system. The master node contains a master task and a slave task, whereas slave nodes only contain slave tasks only.

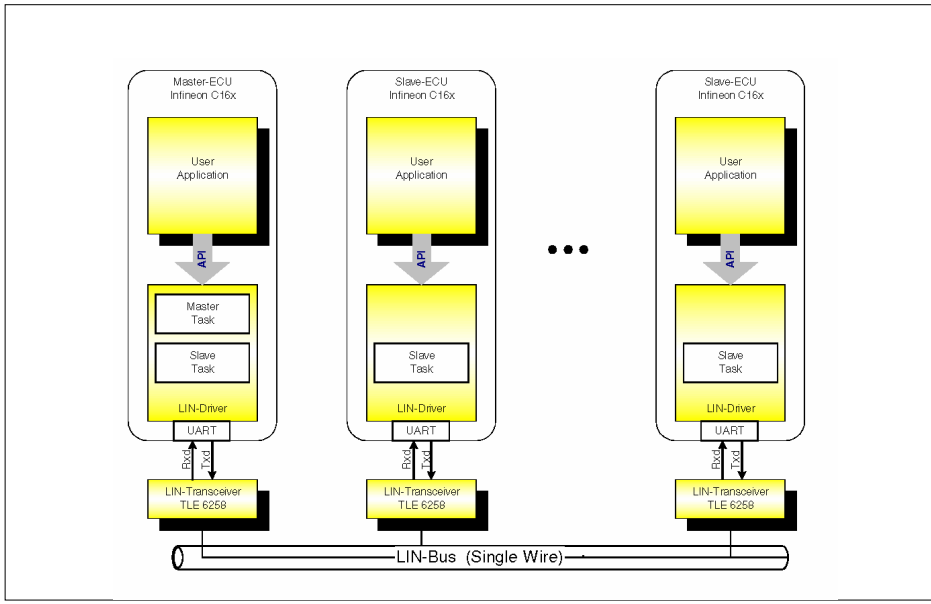

Figure 4 LIN bus architecture

A fixed message format is defined for the LIN protocol. Each LIN message starts with a header consisting of a synchbreak followed by a synchfield and an identifier-field. Only the master task may send this frame. Next 2, 4, or 8 bytes code the message data. A checksum field completes the LIN message (Figure 5). This response to the header is sent either by the slave task of the master node or by one of the several slaves nodes hooked up to the LIN bus.

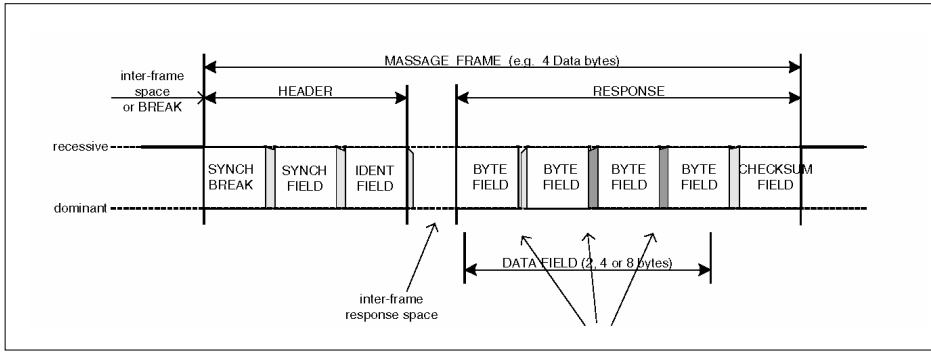

Figure 5 LIN message frame

Main features of the LIN bus

- Single-master / multiple-slave
- Use of an ordinary UART / SCI (supported by any Infineon Microcontroller with ASC)
- Self synchronization
- Low-cost single-wire (enhanced ISO 9141)
- Easy connection with Infineons' single-wire-transceivers (e.g. TLE6258 or TLE6259)
- Speed up to 20 kBits/sec
- Guaranteed latency times
- Message Frame contains 2,4 or 8 data bytes
- Multi-cast reception
- Checksum calculation (inverted modulo 256 checksum)
- Recognition of defective nodes
- Error-Detection
- Low cost

3 LIN Software Driver

The driver itself is written for the C16x, but the principal for the XC16x family is identical. In contrast to CAN or J1850, the LIN bus requires no dedicated on-chip microcontroller communication module. LIN utilizes the standard serial communication interface (USART). That is one major point for the well-balanced cost/performance ratio of this recently introduced Class A subbus. Data exchange is based on a common hardware peripheral (serial communication interface) controlled by a dedicated LIN software driver. Unlike the above mentioned in-vehicle communication protocols the driver software handles the basic communication layers and takes care of message transfers, message filtering, and error detection (protocol handling).

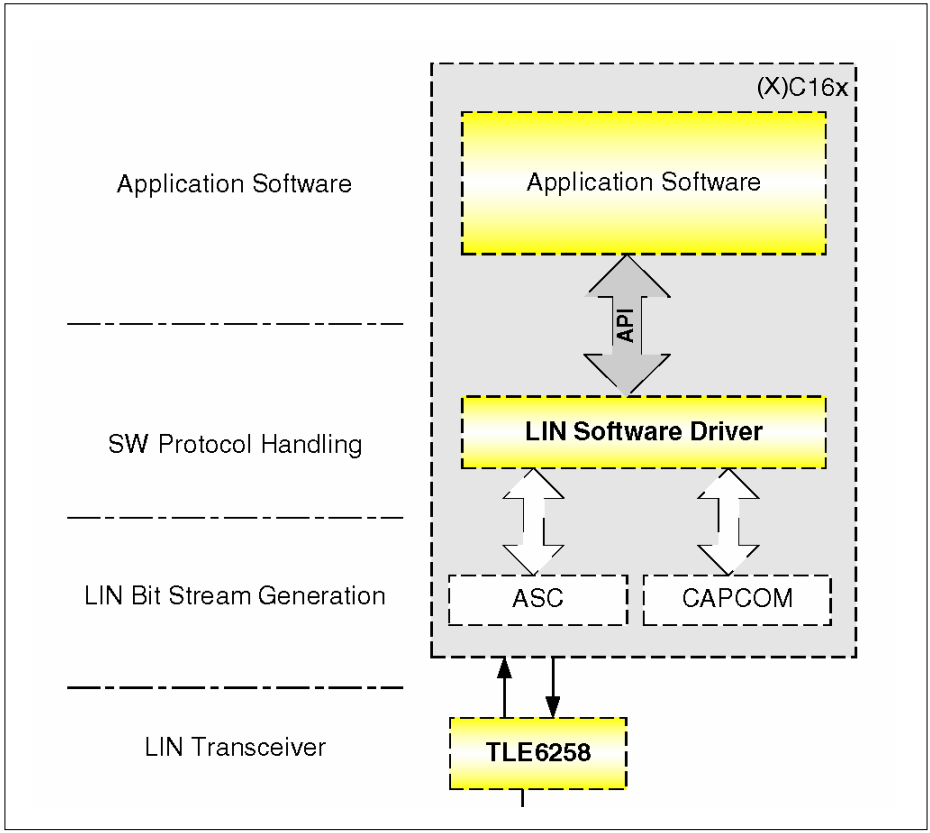

Figure 6 LIN message frame

The Infineon LIN software driver entirely encapsulates the hardware modules and exclusively handles the on-chip peripherals of the Infineon C16x and XC16x microcontrollers, which support LIN. Figure 6 illustrates this approach.

Configurable software building blocks handle the LIN protocol. LIN message frame handling is done autonomously by the LIN software driver. Operations on LIN are at disposal of the user and are initiated by API-function calls. A LIN network based on Infineon Technologies C16x microcontroller family can be easily realized by using the driver.

3.1 LIN Software Driver Services

The LIN driver provides several API-functions for LIN bus handling. The main services of the LIN software driver are:

- Message transmission
- Message reception
- Message filtering
- Connect/disconnect the LIN node to the LIN bus
- Sending "*go to sleepmode*" command
- Sending "*wake up*" command
- Bus timeout detection
- Frame monitoring
- ID field calculation
- Data length extraction
- Checksum calculation
- LIN message scheduler

3.2 LIN Software Driver Requirements

The LIN software driver requires CPU processing time as well as hardware resources like peripherals, code/data memory, and interrupts nodes. The exact requirements are listed next.

3.2.1 Hardware Requirements

LIN utilizes the serial interface for message frame generation and additionally occupies one timer channel for LIN bus timing monitoring e.g. for detection of not responding slaves or no bus activity. Both peripherals require an interrupt handler, with main processes of the LIN software driver being executed from the interrupt handler of the serial communication channel peripheral

3.2.2 Memory Requirements

The software driver occupies less than 3 kBytes of code memory and less than 100 bytes of data memory. These numbers include all the basic LIN software driver functions. Application specific user LIN message buffers require additional memory space.

3.2.3 CPU-Load

The LIN software driver builds on several state machines and functional blocks. The process flow is driven by a state machine invoked by the interrupt of the serial interface. Figure 7 below shows a typical LIN bus frame. Channel 2 shows the LIN pulse train and Channel 1 shows the CPU activity. Each spike at Channel 1 is a process of the LIN software driver. This figure shows the involvement of a LIN master node. Its master task sends out the message header and its slave task attaches 4 data bytes followed by the corresponding checksum field. Generally, the master task transmits synchbreak, synchfield, and identifier-field: This is done within state 1 to 3 (see Figure 7). If a slave task detects a matching identifier on the bus it will either receive or transmit data on the bus (state 4 to 9).

The user has to configure the system as such, avoiding several slave tasks concurrently responding to the same ID. Moreover, the user is responsible for arranging an application specific network management.

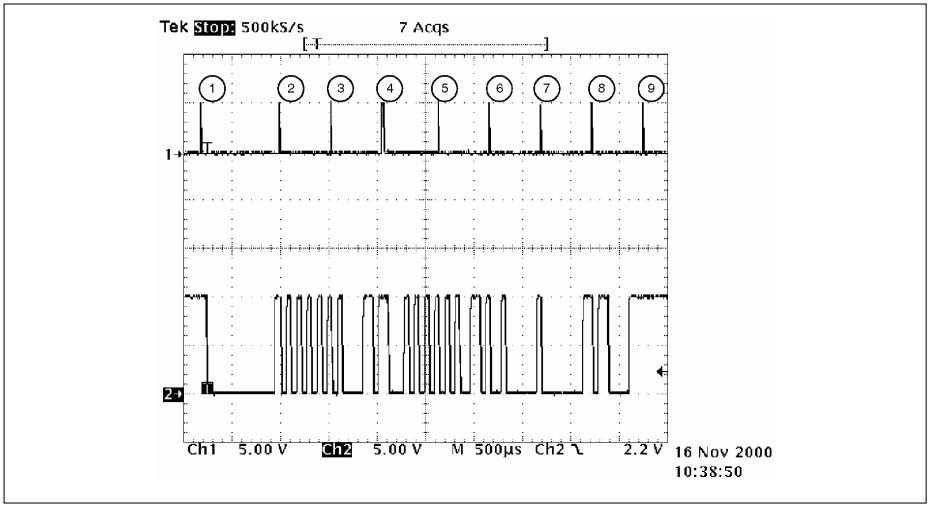

Figure 7 LIN message frame generation

Master Task

1. Send-synchbreak
2. Receive-synchbreak and send synchfield
3. Receive-synchfield and create / send ID-Field

Slave task:

1. Receive-ID-Field (message filtering), copy data to LIN transfer buffer and send first databyte
2. Receive first databyte and send second data byte
3. Receive second databyte and send third data byte
4. Receive third databyte and send fourth data byte
5. Receive fourth databyte and calculate / send checksum-field
6. Receive-Checksum

This bus configuration is running at a baudrate of 19.2 kBaud. The length of one message frame is about 4.5 ms. The required CPU time for a LIN frame generation depends strongly on the number of declared ID's. Within the example network, 16 identifiers are defined. During the states 1 to 9, the CPU of the master LIN node is busy processing LIN driver operations. The overall processing time for the LIN software driver is 110 µs (f_{CPU} = 20 MHz). While sending and receiving a LIN message frame the total CPU load of the LIN master node is below 3%.

4 How to use the LIN Software Driver

When using the software driver it takes only a few steps to get a running LIN network or LIN node. Figure 8 sketches the development flow of an LIN application. The application consists of two independent software parts: the pure user application and the LIN software driver. A dedicated header file (*LIN_InitNode.h*) enables customization.

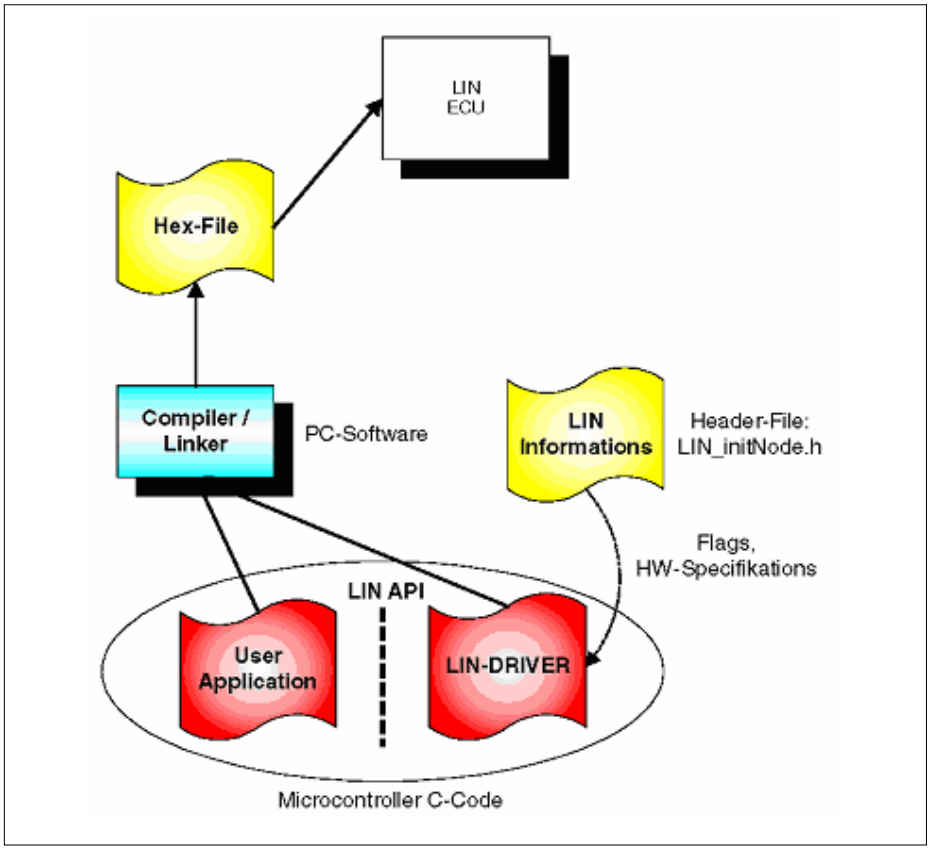

Figure 8 LIN-application development procedure

The LIN software driver consists of three different files, which all have to be included in the software project.

LIN driver Initialization File *lin_initnode.h*

This file allows LIN protocol specific and target specific settings

LIN protocol specific settings are:

- baudrate settings;
- configuration of Master or Slave LIN ECU;
- declaration of LIN messages (ID allocation for send and receive messages).

Target specific settings are:

- allocation of timer resources;
- allocation of serial interface resources;
- allocation of interrupt priorities

LIN driver C-File *lin_driver.c*

This file includes all functions required for the LIN communication

LIN driver Header-File *lin_driver.h*

This header-file includes the declarations of LIN-API-Functions.

Note: This header-file must be included in all C-files using API-Functions.

4.1 Function Overview

The software driver handles the basic communication layers and takes care of message transfers, message filtering, and error detection (protocol handling). All the following functions are elements of the software driver and can be found in the file *lin_driver.c*. The header-file *lin_driver.h* contains the corresponding function declarations.

Table 1 API-Function Overview:

Name		Short-Description
LIN_vInitNode	M / S	Initialization of the LIN-Node
LIN_vSchedule	M	scheduler function sends the header with defined IDS
LIN_vRxd_Interrupt	M / S	Must be called from within the UART-RxD-Interrupt-Function

Name		Short-Description
LIN_vCapture_Interrupt	M / S	Must be called from within the Capture-Interrupt-Function
LIN_vGoSleep	M	Send sleep-ID on the bus and set master node in sleepmode
LIN_vSendWakeUp	M / S	Send wakeup signal on the bus
LIN_vCopy_SendData	M / S	Copies databyte from user-buffer to transceiver buffer and starts send-procedure
LIN_vCopy_ReceiveData	M / S	Copies received databytes from transceiver buffer to user-buffer
LIN_vDefineUCBFunctionRxd	M / S	Defines the Rxd-ID User-Call-Back-Function
LIN_vDefineUCBFunctionTxd	M / S	Defines the TxD-ID User-Call-Back-Function
LIN_vDisconnect	M / S	Disconnects node from the LIN bus (Default-Node-Status)
LIN_vConnect	M / S	Connects node to the LIN bus
LIN_ucReadFlag_SingleError	M / S	Gets the status of a defined Error-Flag
LIN_ucRead_ErrorStatus	M / S	Gets the status of all error flags (user has to separate the responsible Error-Flag)
LIN_vClearFlag_SingleError	M / S	Resets a defined Error-Flag
LIN_vClear_ErrorStatus	M / S	Resets the complete Error-Status
LIN_ucReadFlag_ReadyForSynch break	M / S	Reads flag if the node is ready for a new synchbreak (very important in master node)
LIN_ucReadFlag_Sleep	M / S	Reads flag, if the bus is in sleepmode.
LIN_ucReadFlag_SleepPending	M	Reads flag, if node should send Sleep-Frame
LIN_ucReadFlag_WakeupPending	M / S	Reads flag, if node should send wakeup signal
LIN_ucReadFlag_MessageSent	M / S	Reads flag, if the message has been sent completely

Name		Short-Description
LIN_ucReadFlag_Disconnect	M / S	Reads flag, if the node is connected to the LIN bus
LIN_ucGetID	M / S	Separates and returns ID-Number. = ID-Field without Parity-Bits
LIN_ucGetNOD	M / S	Separates and returns Number-Of-Data. = Bit 4 and 5 in the ID-Field

Legend:

- M / S = used in Master and Slave ECU
- M = used only in the Master-ECU

Note: Table 1 summarizes all function overview but it is not intended to give information about parameters and return values.

4.2 Getting the Master ECU started

The next example will show how to implement the LIN software driver on a LIN master node. In addition to the afore mentioned files *lin_initnode.h, lin_driver.c, and lin_driver.h*, the attached example application contains the files *User_LIN_functions.c, User_interrupt_management.c* and *main.c.* All the following steps refer to the example application.

Note: For correct operation of the software driver Txd and Rxd Pin of the microcontroller must be connected to a dedicated LIN transceiver e.g. TLE6258.

Table 2 Steps to get the master ECU started

Step	Explanation
1	**Goal** Prepare initialization header file **Where** LIN_initNode.h **How** See description 5.1 “Initialization” on page 26 **Description** This must be done separately for each node with LIN driver. See example in Step 2.

Step	Explanation

2	**Goal**
	Call the LIN-Node initialization function
	Where
	Main.c (above the Main-Loop, one of the first functions)
	How
	Call: `LIN_vInitNode()`
	Description
	Handles on the initialization header file prepared in Step 1
	
	EXAMPLE:
	`Void main (void)`
	`{`
	`  LIN_vInitNode();  // initialize LIN-ECU`
	`  USER_vInitNode();  // initialize User-Application`
	`  // (depends on user-application)`
	`  LIN_vConnect();  // connect ECU to the LIN-Network`
	
	`  While (1)`
	`  {`
	`  …`
	`  #ifdef MASTER`
	
	`  LIN_vSchedule(); // only in master node`
	`  // check if button is pressed`
	`  if(USER_ucCheckButtonStatus(4) == 1)`
	`  {`
	`  // if button is pressed set LIN bus in sleepmode`
	`  LIN_vGoSleep();}`
	`  #endif`
	`  }`
	`}`

3	**Goal** Create an user-buffer **Where** e.g. User_LIN_functions.c (user C-file, where the LIN-functions are handled) **How** e.g. call: `USER_vInitNode()`–function in which the buffer will created

Step	Explanation
	Description This is a buffer, where the LIN databytes should be stored, because the LIN driver includes no buffer for this. It is recommended to save default values in this buffer, once created.

4	**Goal**
	Create Txd-User-Call-Back Function
	Where
	e.g. User_LIN_functions.c (user C-file, where the LIN-functions are handled)
	How
	Depends on user-buffer, function could be called:
	`USER_vUCB_TxdId_Received(…)`
	See Description
	Description
	This function will be called if a new LIN message has been received with an ID, defined as TxdID.
	Following function should be called from within the UCB-function:
	`Void LIN_vCopy_SendData(unsigned char* pucLocalData)`
	See Step 7 for an UCB-Txd-function example

5	**Goal**
	Include Data-Copy function
	Where
	In the UCB-Txd-function (e.g. `USER_vUCB_TxdId_Received(…)`)
	How
	Call: `LIN_vCopy_SendData()`
	Description
	The function copies data from the user-buffer to the LIN transceiver-buffer. It is up to the user to keep the databytes that will be sent, up-to-date.
	
	EXAMPLE:
	`void USER_vUCB_TxdId_Received(unsigned char ucLocalId)`
	`{`
	`  stUSER_MESSAGE* pucLocHelp1;  // help variable`
	`  // get address of array-element in which are the data-`
	`  // bytes that should be sent (dependent on ID)`
	`  pucLocHelp1 = USER_pstGetArrayAddress(ucLocalId);`

Step	Explanation

	`// call function which copies the databytes from the`
	`// user-buffer to the LIN transceiver-buffer (parameter is`
	`// address of first Data-Byte).`
	`LIN_vCopy_SendData(&(pucLocHelp1->ucdatabyte[0]));`
	`}`

6	**Goal**
	Create Rxd-User-Call-Back Function
	Where
	e.g. User_LIN_functions.c (user C-file, where the LIN-functions are handled)
	How
	Depends on user-buffer, function could be called:
	`USER_vUCB_RxdId_Received(…)`
	See Description
	Description
	This function will be called if a new LIN message has been received with an ID, defined as RxdID.
	Following function should be called from within the UCB-function:
	`Void LIN_vCopy_ReceiveData(unsigned char* pucLocalData)`
	See Step 5 for an UCB-Rxd-function example

7	**Goal**
	Include Data-Copy function
	Where
	In the UCB-Rxd-function (e.g. `USER_vUCB_RxdId_Received(…)`)
	How
	Call: `LIN_vCopy_ReceiveData()`
	Description
	Copies received Data from the LIN transceiver-buffer to the user-buffer.
	
	EXAMPLE:
	`void USER_vUCB_RxdId_Received(unsigned char ucLocalId)  //`
	`parameter = received ID`
	`{`
	`stUSER_MESSAGE* pucLocHelp1;  // help variable`
	
	`// get address of array-element where the`
	`// databytes should be stored (dependent on ID)`
	`pucLocHelp1 = USER_pstGetArrayAddress(ucLocalId);`

Step	Explanation

	`// call function which copies the databytes from the`
	`// LIN transceiver-buffer to the user-buffer (parameter`
	`// is address of first Data-Byte).`
	`LIN_vCopy_ReceiveData(&(pucLocHelp1->ucdatabyte[0]));`
	`}`

8	**Goal**
	Define User-Call-Back Functions
	Where
	e.g. in the `USER_vInitNode()`–function in the file: User_LIN_functions.c
	How
	Call: `LIN_vDefineUCBFunctionRxd(…)`
	&
	Call: `LIN_vDefineUCBFunctionTxd(…)`
	Description
	The created `UCB_Rxd()` and `UCB_Txd()` Function must be defined with this “UCB-define”-Functions. See Step 4 and Step 6
	
	EXAMPLE:
	`void USER_vInitNode(void)`
	`{`
	`// Create Buffer`
	`USER_vCreateBuffer();`
	
	`// Define UCBs`
	`LIN_vDefineUCBFunctionTxd(USER_vUCB_TxdId_Received);`
	`LIN_vDefineUCBFunctionRxd(USER_vUCB_RxdId_Received);`
	`}`

9	**Goal** Prepare UART Rxd Interrupt-function **Where** e.g. in the `USER_viRxd1_Interrupt()`–Function in the file: User_interrupt_management.c **How** Include the function `LIN_vRxd_Interrupt()` in the Rxd-Interrupt-function **Description** It is recommended to place only this function in the “Rxd-Interrupt” –function.

Step	Explanation

	EXAMPLE:
	`void USER_viRxd1_Interrupt(void)  interrupt 0x49`
	`{`
	`  LIN_vRxd_Interrupt();`
	`}`

10	**Goal**
	Prepare CapCom Interrupt-function
	Where
	e.g. in the `USER_viCapture15_Interrupt()`–Function in the file: User_interrupt_management.c
	How
	Include the function `LIN_vCapture_Interrupt()` in the CapCom-Interrupt-function
	Description
	It is recommended to place only this function in the "`Capture-Interrupt`" –function.
	
	EXAMPLE:
	`void USER_viCapture15_Interrupt(void) interrupt 0x1F`
	`{`
	`  LIN_vCapCom_Interrupt();`
	`}`

11	**Goal** Connect the LIN-Node to the LIN bus **Where** e.g. in the `USER_vInitNode()` –Function in the file: User_LIN_functions.c or in the `main()`–function **How** Call: `LIN_vConnect(…)` **Description** Default Node-Status after the initialization function (see Step 2) • disconnected See example in Step 2
12	**Goal** Place scheduler function

Step	Explanation
	Where in the Main-Loop in the `main()`–function **How** Call: `LIN_vScheduling()` **Description** This function is responsible for the LIN-Message handling on the bus. (Sends LIN-header if the bus is free). See function description in section 5.2 “Function Description” on page 31 See also example in Step 2
13	**Goal** Place “go sleep” -function **Where** e.g. User_LIN_functions.c (user C-file, where the LIN-functions are handled) **How** Call: `LIN_vGoSleep()` **Description** Depends on the event (user-application) where from this function should be called to set the node (bus) in sleepmode. See example in Step 2
14	**Goal** Place “wake up”–function **Where** e.g. User_LIN_functions.c (user C-file, where the LIN-functions are handled) **How** Call: `LIN_vSendWakeUp()` **Description** Depends on the event (user-application) where from this function should be called to wake up the node (bus).

4.3 Getting the Slave-ECU started

The steps in the slave node are nearly the same as the steps in the master node. See section 4.2 Getting the Master ECU started, and go through following steps.

Table 3 Steps to get the slave ECU started

Step	Explanation
Step 1 until Step 11	See above.
Step 14	See above.

4.4 LIN message Buffers

The LIN driver provides an abstract Buffer for LIN-Message sending/receiving. This buffer can store the data bytes of one LIN message.

- **Sending a LIN-Message**: The databytes must be copied from the user buffer to the LIN buffer. The content of the LIN-Buffer will be sent by the UCB-Txd-Function.
- **After receiving a LIN-Message**: Within the UCB-Rxd-Function, the content of the LIN-Buffer is transferred to the respective user-buffer.

The LIN driver does not indicate new received databytes, because this is part of the User-Application and should be done in the UCB function.

4.5 Read Error-Status

Each Slave task has its own Error-Status-Byte. If an error occurs, the responsible Error-Flag will be automatically set in the Error-Status-Byte. The User-Application can read and clear the Error-Flag within the Status-Byte. There are some functions to perform this described below:

- `LIN_ucRead_ErrorStatus(void)`
- `LIN_vClear_ErrorStatus(void)`
- `LIN_ucReadFlag_SingleError()`
- `LIN_vClearFlag_SingleError()`

5 Conclusion

The LIN bus will have a major impact on in-vehicle network topology and empowers the migration towards a distributed system architecture. A LIN implementation bases on three elements. Firstly, a microcontroller with an on-chip serial communication interface, secondly a LIN bus transceiver and last but not least a dedicated LIN software driver. Infineon Technologies addresses all these hardware and software demands. Infineon is offering a family of LIN bus transceivers like the TLE6258 or TLE6259
(http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_cat.jsp?oid=-8419)
and the powerful 16 bit microcontroller architecture C16x / XC16x
(http://www.infineon.com/cgi/ecrm.dll/ecrm/scripts/prod_cat.jsp?oid=-8984).
The introduced LIN software driver completes this package and allows the easy implementation of a LIN network.

6 Appendix A: LIN driver Details

6.1 Initialization

The header file ***lin_initnode.h*** contains dedicated settings for application specific configurations. Each LIN-Node requires its own dedicated initialization file.

This header must be included in the following project files:

- LIN_driver.c
- Main.c
- USER_lin_functions.c

6.2 ECU-Mode

There are two different modes for a LIN-Node:

- MASTER
- SLAVE

For master node `#define MASTER`

For slave node `#define SLAVE`

Note: Avoid both definitions

6.3 Identifier-Settings

An ID can have 3 different functions:

1. Node should send the databytes: ID must be defined as SEND_ID
7. Node should receive the databytes: ID must be defined as REC_ID
8. For master node only: Entries in the scheduler must be defined as MON_ID

There are some “define-names” which have to be used for ID-Definitions:

MAX_LIN_ID:	The maximum number of all used IDs (NUM_SEND + NUM_REC + number of MON-IDs) Valid Values: 0, 1, …15
NUM_SEND:	Number of SEND_IDs
NUM_REC:	Number of REC_IDs

It is possible to define up to 16 send-IDs and up to 16 receive-IDs per node.

To define a Send-ID you have to use the define-name:

SEND_IDx yy	x = {0,1,…15}	(internal used counter)
	yy = {1,2,…63}	is the ID (not the ID-Field)

To define a Receive-ID you have to use the define-name:

REC_IDx yy	x = {0,1,…15}	(internal used counter)
	yy = {1,2,…63}	is the ID (not the ID-Field)

To define a Monitor-ID you have to use the define-name:

MON_IDx yy	x = {0,1,…15}	(internal used counter)
	yy = {1,2,…63}	is the ID (not the ID-Field)

Note: Do never define the ID0. The corresponding ID-Field to the identifier 0 is the Sleep-Signal 0x80. So if you would define e.g. SEND_ID2 0, then the LIN bus would be set in sleepmode after sending the respective message-header.

Example

```
#define MAX_LIN_ID   6     // the absolute number of all used IDs

#define NUM_SEND  1        // number of send_IDs
#define SEND_ID0  51       // Send-ID definition: x=0, y=51
                           // The first ID has the Value 51

#define NUM_REC   2        // number of receive_IDs
#define REC_ID0   12       // example for the first Receive ID= 12
#define REC_ID1   10       // example for the second Receive ID= 10

Monitor IDs only for Master. These IDs will be sent by the scheduler
function.

#ifdef MASTER
   #define MON_ID0   1     // ID of the first monitor ID
   #define MON_ID1   2     // ID of the second monitor ID
   #define MON_ID2   3     // ID of the next monitor ID
#endif
```

Figure 9 LIN driver initialization example. The header files allows individual allocation of required on-chip resources e.g. capture compare channels, serial interfaces and interrupt priorities.

6.4 Capture Compare settings

```
////////////////////////////////////////////////////////////////////////
// CAPTURE COMPARE Settings
////////////////////////////////////////////////////////////////////////

// define the CapCom Mode Register: CCMx
#define CC_MODE_REGISTER            CCM3

// define the CapCom Register: CCxx
#define CC_REGISTER            CC15
```

6.5 Capture Compare TIMER -Settings

```
////////////////////////////////////////////////////////////////////////
// CAPTURE COMPARE TIMER Settings
////////////////////////////////////////////////////////////////////////

// define the CapCom Timer Control Register: TxxCON
#define CC_TIMER_CONTROL_REGISTER            T01CON
// define the CapCom Timer Register: Tx
#define CC_TIMER_REGISTER            T0
```

6.6 ASC -Settings

```
////////////////////////////////////////////////////////////////////////
// ASC Settings
////////////////////////////////////////////////////////////////////////

// define the ASC Control Register:
#define ASC_CONTROL_REGISTER        S1CON
// define the ASC Baudrate Register:
#define ASC_BAUDRATE_REGISTER       S1BG
```

6.7 Port-Settings

```
////////////////////////////////////////////////////////////////////////
// Port Settings
////////////////////////////////////////////////////////////////////////

// LIN Port Register
#define LIN_PORT_REGISTER  P3

// LIN Port Direction Register
#define LIN_PORT_DIRECTION_REGISTER  DP3

// LIN TxD-Bit in LIN_PORT_REGISTER
#define LIN_TXD_BIT     0x0001

// LIN RxD-Bit in LIN_PORT_REGISTER
#define LIN_RXD_BIT     0x0002
```

6.8 Interrupt—Settings

```
////////////////////////////////////////////////////////////////////////
// Interrupt Priority Settings
////////////////////////////////////////////////////////////////////////

// Timer IRQ Priority Value
#define TIMER_PRIORITY_VALUE        0x0012   // ILV 6 GLV 2
// ASC Receive IRQ Priority Value
#define ASC_RECEIVE_PRIORITY_VALUE  0x0022   // ILV 5 GLV 2
// Capture Compare IRQ Priority Value
#define CC_PRIORITY_VALUE           0x0032   // ILV 4 GLV 2

////////////////////////////////////////////////////////////////////////
// Interrupt Settings
////////////////////////////////////////////////////////////////////////

// define the global Interrupt enable/disable
#define GLOBAL_INTERRUPT   IEN
```

6.9 Baudrate-Settings

```
// The LIN driver supports two different baudrates:
9600 bits/sec
19200 bits/sec
```

Example:

```
#define BAUDRATE_LIN    19200
```

TIME-Definitions

Some Timer and ASC values must be defined. The following example is for C16x Infineon Microcontroller and 20MHz Oscillator.

Please have a look in the LIN specification to get general information about the different times e.g. Synchbreak (T_{SYNBRK}) or maximum Frame-Time (T_{FRAME_MAX}), …

Example:

```
#if BAUDRATE_LIN == 19200
   #define BAUDRATE_ASC     0x0020    // Baudrate for LIN : 19200 Baud
   #define BAUDRATE_SYN     0x002F    // Baudrate for Synchbreak
   #define BITTIME_LIN      0x0047    // T0: resol.= 51,2us = 0x0047
   #define TIMEBASE_LIN     0x0000    // Time base = max = 3,36 sec

// == Reload-Value
   #define TIME_FRAMEMAX2  0x005C     // 91 bittime
   #define TIME_FRAMEMAX4  0x0079     // 119 bittime
   #define TIME_FRAMEMAX8  0x00B2     // 175 bittime
   #define TIME_OUT        0x6355     // 25000 bittime
   #define T_T0BRK         0x0082     // 128 bittime
   #define T_T3BRK         0x3B99     // 15000 bittime
#endif
```

Figure 10 Baudrate setting example

Table 4 Time-Descriptions

Definition	Description
BAUDRATE_ASC	Sets the baudrate in the baudrate register
BAUDRATE_SYN	Sets the baudrate in the baudrate register for the synchbreak field
BITTIME_LIN	bittime-Resolution
TIME_FRAMEMAX2	Maximum time for frame transmission with 2 databytes
TIME_FRAMEMAX4	Maximum time for frame transmission with 4 databytes
TIME_FRAMEMAX8	Maximum time for frame transmission with 8 databytes
TIME_OUT	Bus idle time out
T_T**0**BRK	Timeout After wakeup signal: *Note: bold sign is a zero and not an O-letter*
T_T3BRK	Timeout After Three Break

Note: All times, refer to the bittime of the master node.

Bittime Calculation:

Baudrate: 19200 Hz → $\frac{1}{19200} = 52{,}08\mu\text{sec}$ bittime

The best UART resolution on 20MHz is $51{,}2\mu\text{sec}$.

That is a deviation of the $\frac{52{,}08\mu\text{sec}}{51{,}2\mu\text{sec}} = 1{,}01725$

To work with the correct times you have to multiply the time-values by the deviation-factor 1,01725.

```
TimeOut After WakeUp Signal:  T_T0BRK = 128 bittime in order to the
                                        LIN specification.
BaudRate = 19200 bits/sec
128 bittimes = 0x80 in hex = 6,66msec       (bittime: 52,08 µsec)
128 bittimes = 0x80 in hex = 6,55msec       (bittime: 51,2 µsec)
128 * 1,01725 = 130,208 bittimes = 0x82 in hex = 6,66msec
                                             (bittime: 51,2 µsec)
result:
#define T_T0BRK               0x0082         // 128 bittime
```

Figure 11 Bittime calculation example

7 Appendix B: Function Description

7.1 LIN_vInitNode

Prototype: `void LIN_vInitNode(void)`

Parameters: void

Return-Value: None

Description: The operation executes as follows:

Step 1: Initialization of Timer

Step 2: Initialization of ASC

Step 3: Initialization of capture compare register

Step 4: Creates internal used buffer for ID-handling

Step 5: Resets counters, flags and state-machines

(sets “disconnect-node”)

Step 6: Start timer for “No-Bus-Activity”

Note: This function must be implemented above the loop in the `main()`-function. Bus is disconnected after this function.

7.2 LIN_vSchedule

Prototype: `void LIN_vSchedule(void)`

Parameters: void

Return-Value: None

Description: The master node is responsible for sending all required IDs on the bus. If the ID is not defined as SEND-ID or REC-ID then the IDs must be defined in the master node as MON-ID (Monitor-ID) to provide the IDs needed in the other nodes.

The function sends the SEND-IDs first (beginning by Send_ID0) followed by the REC-IDs and MON-IDs. The LIN bus must be ready for a new message, for this, a flag (ready_for_synchbreak) will be checked before sending the next message.

The function sends only one message per call so it is up to the user to include this function in the `main()` function or in another loop-function.

Note: MASTER-Function only. It is not possible to change the chronological order of IDs.

7.3 LIN_vRxd_Interrupt

Prototype: `void LIN_vRxd_Interrupt(void)`

Parameters: void

Return-Value: None

Description: The user must provide a `RXD-Interrupt-Function`. The `LIN_vRxd_Interrupt()`-Function must be called from within the Rxd-Interrupt function for a correct LIN driver handling.

In the function is a state-machine, which handles the different states of LIN-message transfer.

This function will be called after each byte is received, this applies in every node (including the sending node). The self-received byte will be checked and if necessary, the next byte will be sent.

Note: In the LIN driver is no Txd-Interrupt needed.

7.4 LIN_vCapCom_Interrupt

Prototype: `void LIN_vCapCom_Interrupt(void)`

Parameters: void

Return-Value: None

Description: The user must provide a CapCom-Interrupt-Function. The `LIN_vCapCom_Interrupt()`-Function must be called from within the Capture/Compare-Interrupt function for correct LIN driver operation.

In the function is a state-machine, which mainly handles the different Time-Outs e.g. T_{FRAME_MAX} or T_{TIME_OUT}.

7.5 LIN_vGoSleep

Prototype: `void LIN_vGoSleep(void)`

Parameters: void

Return-Value: None

Description: This function sets the bus in sleepmode. A flag (sleep_pending) will be set in the nPENDING_STATUS-Register. A message-header with ID0 = 0 (ID-FIELD=0x80) will be sent if the scheduler function recognizes that the sleep_pending bit is set.

Note: MASTER-Function only.

The sleep-message is the same as other LIN messages. However, the user must not define the SleepID in the LIN-InitNode-header-file, because the SleepID in the master node will automatically be set as SEND-ID and in the slave node as REC-ID. The user has only to provide the databytes for the SleepID in the user-buffer and the Master is going on to send the databytes with the sleep-message. (Sending messages see Function-Descriptions: `LIN_vSendData` and `LIN_vDefineUCBFunctionTxd`)

All slave nodes are able to receive the sleep-message (and the included databytes). After receiving the whole message the node will be set in sleepmode.(Receiving message see Function-Descriptions: `LIN_vReceiveData` and `LIN_vDefineUCBFunctionRxd`)

7.6 LIN_vSendWakeUp

Prototype: `void LIN_vSendWakeUp(void)`

Parameters: void

Return-Value: None

Description: This function sends a Wake-Up-Signal on the bus. (Only if the node is in sleepmode)

Procedure: A flag (wakeup_pending) will be set in the nPENDING_STATUS-Register and a Wake-Up-Signal will be transmitted to the bus. A Compare-Timer (Value: T_{T0BRK}) is started in the event that the transmission failed or the MASTER-ECU does not answer. In this case, the Compare-Interrupt would send a Wake-Up-Signal again (see Figure 12 and LIN specification for details).

Note: See LIN-Protocol Specification!

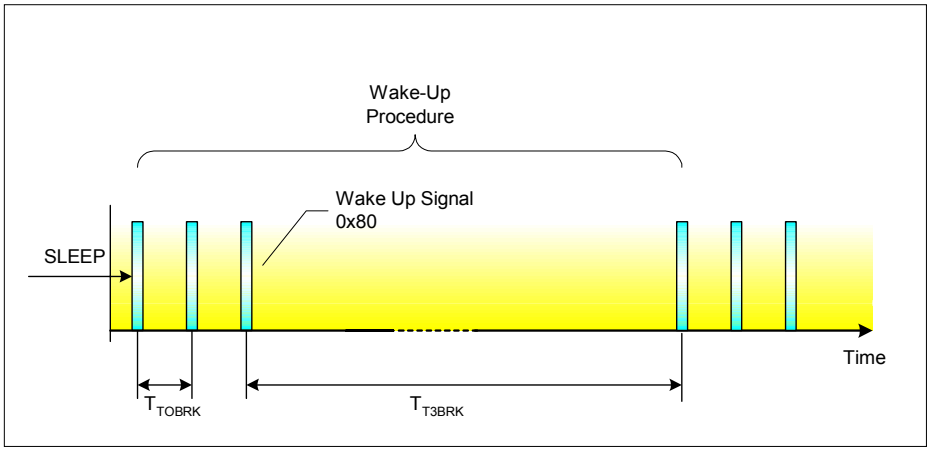

Figure 12 Wake-Up procedure and wakeup signals on the LIN bus.

7.7 LIN_vDefineUCBFunctionTxd

Prototype: `void LIN_vDefineUCBFunctionTxd((*)(unsigned char))`

Parameters: Pointer to function: The address of the User call-back-function (UCB), the UCB has an unsigned char as parameter and includes the received ID.

Return-Value: None

Description: This function defines the User Call Back Function, which will be called when a LIN message (with Send-ID) has been received. This function makes the definition of the UCB function very easy for the user.

It is up to the user to provide a UCB-function for the LIN driver.

The user-application must provide a Buffer for the data, as this is not part of the LIN driver. If a LIN message with SEND-ID is received then the UCB function will be called and the data will be sent from within the UCB function.

Note: Knowledge of “pointer to function” is helpful.

Example:

The user has a function named "`USER_vUCB_TxdId_Received(unsigned char ID)`". The function has an unsigned char as parameter for the ID. This function must be called if the LIN driver receives an ID, which was defined as SEND-ID. Before the LIN driver can call the UCB-function, the UCB must be defined in the LIN driver .

How?

The following statement in a user-init-function defines the UCB function for the LIN driver.

```
LIN_vDefineUCBFunctionTxd(USER_vUCB_TxdId_Received);
```

Note: The parameter in the "define-function" is only the UCB function name without parameter or parentheses detail.

Details of implementation:

The following Figure 13 will show the handling of Rxd-Interrupt, SEND-ID-UCB and LIN driver.

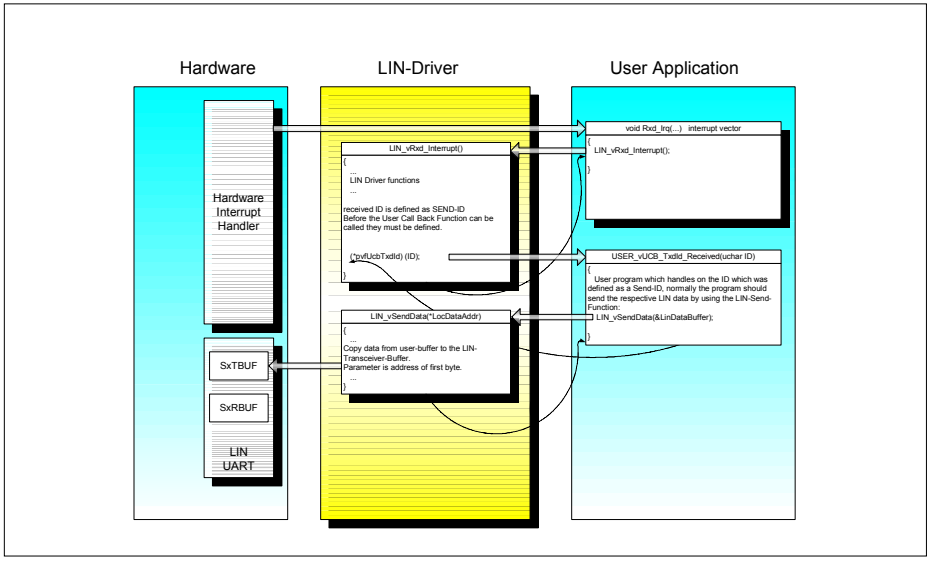

Figure 13 Handling of SEND-ID User Call Back function, LIN driver and UART-Interrupt.

7.8 LIN_vSendData

Prototype: `void LIN_vSendData(unsigned char*)`

Parameters: Pointer to unsigned char: The addresses of the first databyte to be sent.

Return-Value: None

Description: This function copies the databytes, which should be sent, from the user-buffer to the LIN transceiver buffer. It is up to the user to provide all databytes for the respective.

Note: See function `LIN_vDefineUCBFunctionTxd and Figure 13`

7.9 LIN_vDefineUCBFunctionRxd

Prototype: `void LIN_vDefineUCBFunctionRxd((*)(unsigned char))`

Parameters: Pointer to function: The address of the user-call-back-function (UCB), the UCB has an unsigned char as parameter and includes the received ID.

Return-Value: None

Description: This function defines the usercall-back-function, which will be called when a LIN message (with RECeive-ID) has been received. This function makes the definition of the UCB function very easy for the user.

It is up to the user to provide a UCB-function for the LIN driver.

The user-application has to provide a Buffer for the data, as this is not part of the LIN driver. The databytes of a LIN messages with REC-ID will be saved in the LIN transceiver buffer . If the slave task has received the last byte (checksum) correctly then the UCB function will be called and the databytes will be copied from the transceiver buffer into the user-buffer from within this UCB function.

Note: Knowledge of “pointer to function” is helpful.

Example:

The user has a function named "`USER_vUCB_RxdId_Received(unsigned char ID)`". The function has an unsigned char as parameter for the ID. This function should be called if the LIN driver receives an ID, which was defined as REC-ID. Before the LIN driver can call the UCB-function, the UCB must be defined in the LIN driver .

How?

The following statement in a user-init-function defines the UCB function for the LIN driver.

```
LIN_vDefineUCBFunctionRxd(USER_vUCB_RxdId_Received);
```

Note: The parameter in the "define-function" is only the UCB function name without parameter or parentheses detail.

Details of implementation:

The following will show the handling of Rxd-Interrupt, REC-ID-UCB and LIN driver.

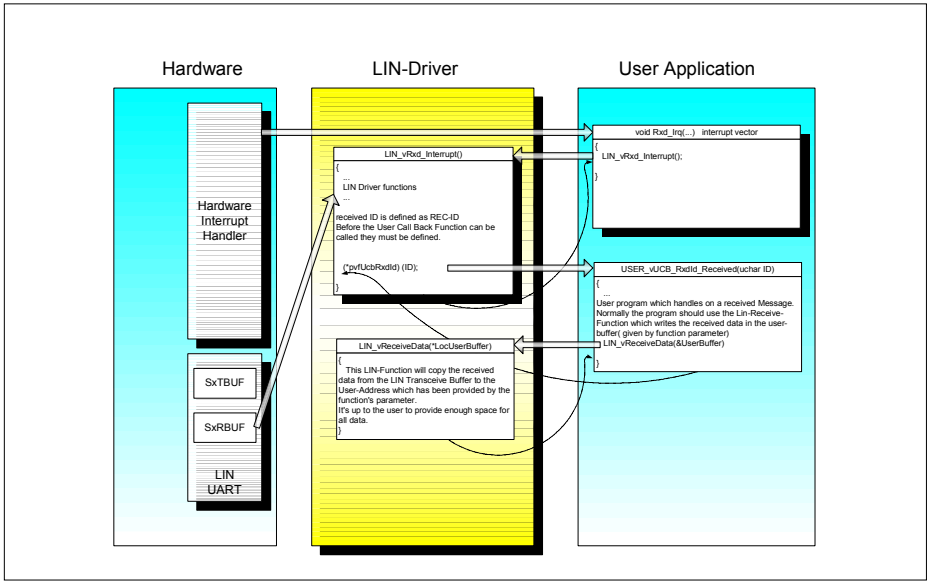

Figure 14 Handling of REC-ID usercall-back-function, LIN driver and UART-interrupt.

7.10 LIN_vReceiveData

Prototype: `void LIN_vReceiveData(unsigned char*)`

Parameters: Pointer to unsigned char: address of the first data byte, where the data should be stored in the USER-LIN-BUFFER

Return-Value: None

Description: This function copies all received data from the LIN transceiver buffer to the USER-LIN-BUFFER. The user application has to provide the address of the first byte in the USER-LIN-BUFFER. The Checksum-Byte will not be copied. The ID is already present<f in the LIN transceiver buffer .

Note: See function `LIN_vDefineUCBFunctionRxd` and Figure 14

7.11 LIN_vDisconnect

Prototype: `void LIN_vDisconnect(void)`

Parameters: void

Return-Value: None

Description: This function disconnects the LIN-ECU from the LIN bus. The node is unable to receive or send LIN data after this function call.

Note: Each LIN-ECU will be disconnected after the LIN is initialized. The user-application has to connect the LIN-ECU.

See also `LIN_vConnect()`

7.12 LIN_vConnect

Prototype: `void LIN_vConnect(void)`

Parameters: void

Return-Value: None

Description: This function connects the LIN-ECU to the LIN bus. The node is able to receive or send LIN data after this function call.

Note: Each LIN-ECU will be disconnected after the LIN is initialized. The user-application has to connect the LIN-ECU.

See also `LIN_vDisconnect()`

7.13 LIN_ucGetID

Prototype: `Unsigned char LIN_ucGetID(unsigned char)`

Parameters: Unsigned char: The ID-Field (ID=6bits + Parity=2bits)

Return-Value: Unsigned char: The ID (first 6 ID bits without parity-bits)

Description: The mention of “ID” or “Identifier” in the LIN specification and in the LIN driver normally means only the 6 bit-information and not the whole ID-Field.

Example: `ID = 0 → ID-Field = 0x80`

```
test = LIN_ucGetID(0x80)
```

result: `test = = 0`

rsp.

```
test = = 0x00
```

7.14 LIN_ucGetNOD

Prototype: `unsigned char LIN_ucGetNOD(unsigned char)`

Parameters: Unsigned char: The ID-Field (ID=6bits + Parity=2bits)

Return-Value: Unsigned char: The number of databytes. Valid values for the return value are 2, 4, or 8

Description: The information of the number of databytes (NOD) is part of the ID (bit 4 and 5).

Example:

```
test = LIN_ucGetNOD(0x80)
```

result: test = = 2

7.15 LIN_ucReadFlag_Sleep

Prototype: `unsigned char LIN_ucReadFlag_Sleep(void)`

Parameters: void

Return-Value: Unsigned char:

0x01: node is in sleepmode

0x00: node not in sleepmode

Description: This function returns the status of the sleep bit in the NODE_STATUS Register. Use this function to check if the LIN-bus is in sleepmode.

Example:

```
if(LIN_ucReadFlag_Sleep())
{
   // ...some code which should be handled in
sleepmode...
}
```

7.16 LIN_ucReadFlag_ReadyForSynchbreak

Prototype: `unsigned char LIN_ucReadFlag_ReadyForSynchbreak(void)`

Parameters: void

Return-Value: Unsigned char:

0x02: node is Ready-For-Synchbreak

0x00: node is not Ready-For-Synchbreak

Description: This function returns the status of the “ready_for_synchbreak” bit in the NODE_STATUS Register. Use this function to check if the node is ready for the next Synchronization Break.

Example:

```
if(LIN_ucReadFlag_ReadyForSynchbreak())
{
   // ...e.g. set sleepmode...
}
```

7.17 LIN_ucReadFlag_WakeupPending

Prototype: `unsigned char LIN_ucReadFlag_WakeupPending(void)`

Parameters: void

Return-Value: Unsigned char:

0x02: wakeup-mode is pending

0x00: wakeup-mode is not pending

Description: This function returns the status of the “wakeup-pending” bit in the PENDING_STATUS Register. Use this function to check if the node

will be set in wakeup-mode in the near future. For this, the node should have been in sleepmode.

Example:

```
If (LIN_ucReadFlag_WakeUpPending())
{
   // ...some code before going in awake-mode...
}
```

7.18 LIN_ucReadFlag_SleepPending

Prototype: `unsigned char LIN_ucReadFlag_SleepPending(void)`

Parameters: void

Return-Value: Unsigned char:

0x04: sleepmode is pending

0x00: sleepmode is not pending

Description: This function returns the status of the “sleep-pending” bit in the PENDING_STATUS Register. Use this function to check if the node will be set in sleepmode in the near future.

Example:

```
if(LIN_ucReadFlag_SleepPending())
{
   // ...some code before going in sleepmode...
}
```

7.19 LIN_ucRead_ErrorStatus

Prototype: `unsigned char LIN_ucRead_ErrorStatus(void)`

Parameters: void

Return-Value: Unsigned char: The Error-Status-Byte

Description: This function returns the Error-Status-Byte of the node. There are 6 different error flags.

```
typedef enum
{
    bit_error  = 0x01,
    checksum_error                    = 0x02,
    id_parity_error                   = 0x04,
    slave_not_responding_error        = 0x08,
    inconsistent_synch_field_error    = 0x10,
    no_bus_activity_error     = 0x20
```

```
}ERROR_STATUS;
```

Note: See functions: `LIN_vClear_ErrorStatus()` and `LIN_ucReadFlag_SingleError()`

See also LIN specification for error description.

Example:
```
if(LIN_ucRead_ErrorStatus())
{
   //...some code to check the Error-Status-Byte
}
```

7.20 LIN_vClear_ErrorStatus

Prototype: `void LIN_vClear_ErrorStatus(void)`

Parameters: void

Return-Value: none

Description: This function clears all error flags in the Error-Status-Byte of the node.

```
typedef enum
{
    bit_error  = 0x01,
    checksum_error                    = 0x02,
    id_parity_error                   = 0x04,
    slave_not_responding_error        = 0x08,
    inconsistent_synch_field_error    = 0x10,
    no_bus_activity_error     = 0x20
}ERROR_STATUS;
```

Note: See LIN specification error description and function: `LIN_ucRead_ErrorStatus()`

Example:
```
LIN_vClear_ErrorStatus();
```

all error flags cleared

7.21 LIN_ucReadFlag_SingleError

Prototype: `unsigned char LIN_ucReadFlag_SingleError(unsigned char)`

Parameters: Unsigned char: The bit, which should be read. See description.

Return-Value: Unsigned char: If the flag is set then return the Error-Bit else 0x00 will returned. See description.

Description: With this function, you can read a special Error-Flag in the Error-Status-Byte of the node.

Possible and valid values for the parameter and returnvalue are:

```
bit_error                          = 0x01,
checksum_error                     = 0x02,
id_parity_error                    = 0x04,
slave_not_responding_error         = 0x08,
inconsistent_synch_field_error     = 0x10,
no_bus_activity_error              = 0x20
```

Note: See LIN specification error description and function: `LIN_ucRead_ErrorStatus()`

Example:

```
if(LIN_ucReadFlag_SingleError(0x04))
{
  //  errorhandling on id_parity_error
}
```

7.22 LIN_vClearFlag_SingleError

Prototype: `void LIN_vClearFlag_SingleError(unsigned char)`

Parameters: Unsigned char: The bit, which should be cleared. See description.

Return-Value: void

Description: With this function, you can clear a special Error-flag in the Error-Status-Byte of the node.

Possible and valid values for the parameter are:

```
bit_error                          = 0x01
checksum_error                     = 0x02
id_parity_error                    = 0x04
slave_not_responding_error         = 0x08
inconsistent_synch_field_error     = 0x10
no_bus_activity_error              = 0x20
```

Note: See LIN specification for error description and function: `LIN_ucRead_SingleError()`

Example:

```
LIN_ucClearFlag_SingleError(0x04);
```

Result: id_parity_error-flag will be cleared.

7.23 LIN_ucReadFlag_MessageSent

Prototype: `unsigned char LIN_ucReadFlag_MessageSent(void)`

Parameters: void

Return-Value: Unsigned char: this function returns sent status. See description.

Description: This flag signals if a message has been sent. This flag should be read only if node is ready for synchbreak (read corresponding flag in the NODE_STATUS), otherwise it could be that the transfer is not complete and you will read a wrong status. Reading this flag is useful to check if an error has occurred while sending a message.

The flag will be reset automatically after receiving a new synchbreak.

Example:

```
if(LIN_ucReadFlag_ReadyForSynchbreak())
// no bus activity
{
  if(LIN_ucReadFlag_MessageSent())
        // message has been sent and
        // transmission complete
  {
    if(LIN_ucClearFlag_SingleError(0x08))
    // check Error
    {
        // ...send the same data again...
        // slave hasn't answered
    }
  }
}
```

7.24 LIN_ucReadFlag_Disconnect

Prototype: `unsigned char LIN_ucReadFlag_Disconnect(void)`

Parameters: void

Return-Value: Unsigned char: this function returns the disconnect flag status of the NODE_STATUS-Byte.

Valid Return-Values:

0x08 node is disconnected from the LIN bus

0x00 node is connected to the LIN bus

Description: If you want to know if the node is connected to the LIN bus then you can use this function shown in NOTE: Example

Example:

```
if(LIN_ucReadFlag_Disconnect())
{
   LIN_vSetFlag_Connect();
}
```

8 Appendix C: LIN driver state-machines

Five different state-machines are described in this chapter. See figure 13. Each state-machine consists of substates called STATEs (see description below). It could be that one state-machine consists of 4 or more states. Every time an external event occurs, the counter of the state will be incremented.

Note: The state-machines are always described for a message with 2 databytes. The handling of the states for 4 or 8 databytes will be the same

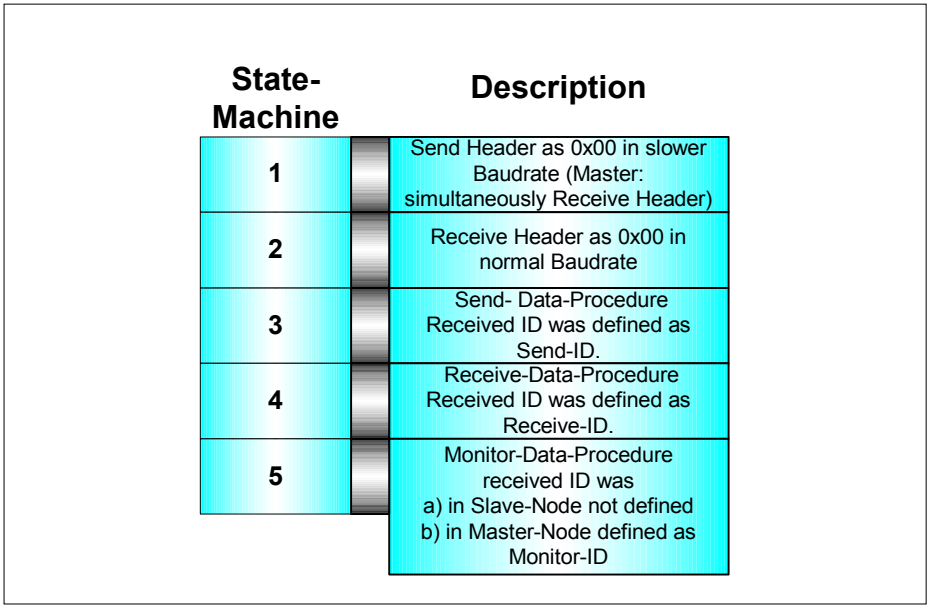

Figure 15 Overview of LIN driver state-machines.

8.1.1 State-Machine 1: Send Header

Figure 16 shows an overview of the state-machine, which will be used when sending a header on the LIN bus. For more details please have a look at the Appendix.

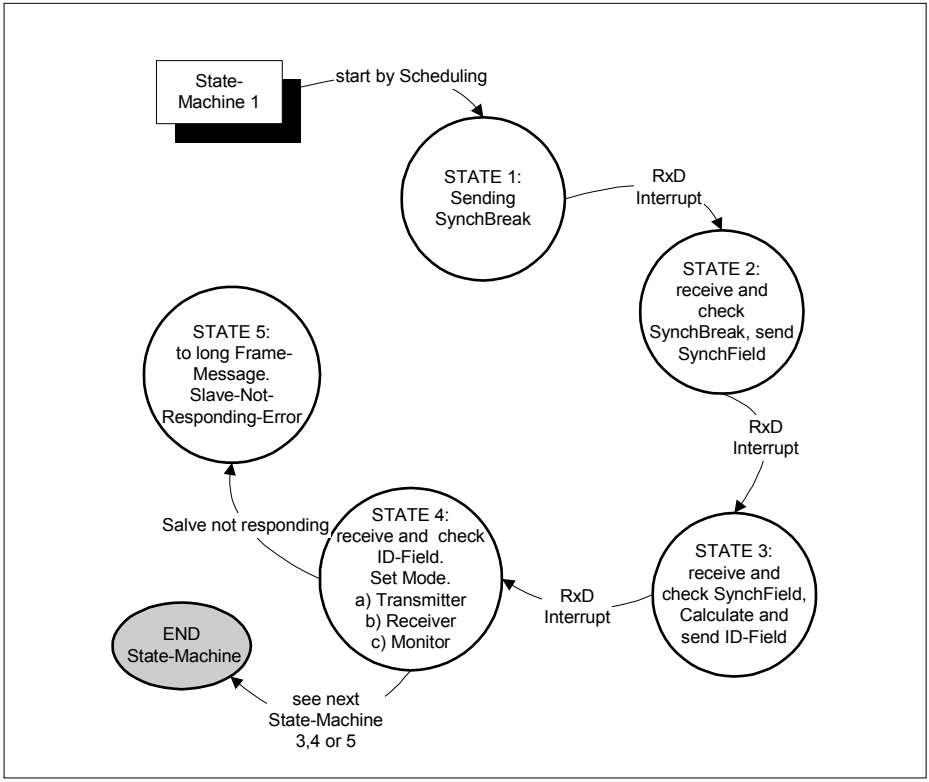

Figure 16 Overview of state-machine 1: Sending Header. This state-machine has 5 substates.

Table 5 STATE and the responsible Transceiver-Status

State	Respective Function
STATE 1: **TS1**	in `LIN_vSendSynch0x00()`-function
STATE 2: **TS2**	in `LIN_vRxd_Interrupt()`-function → `LIN_vReceiveSynch0x00(…)`
STATE 3: **TS4**	in `LIN_vRxd_Interrupt()`-function → `LIN_vReceiveSynchField(…)`
STATE 4: **TS5**	in `LIN_vRxd_Interrupt()`-function → `LIN_vReceiveIdField(…)`
STATE 5: **TIME_FrameMax**	in `LIN_vCapCom_Interrupt()`-function

8.1.2 State-Machine 2: Receive Header

Figure 17 shows an overview of the state-machine, which will be used when receiving a header from the Master-Task. For more details please have a look at the Appendix.

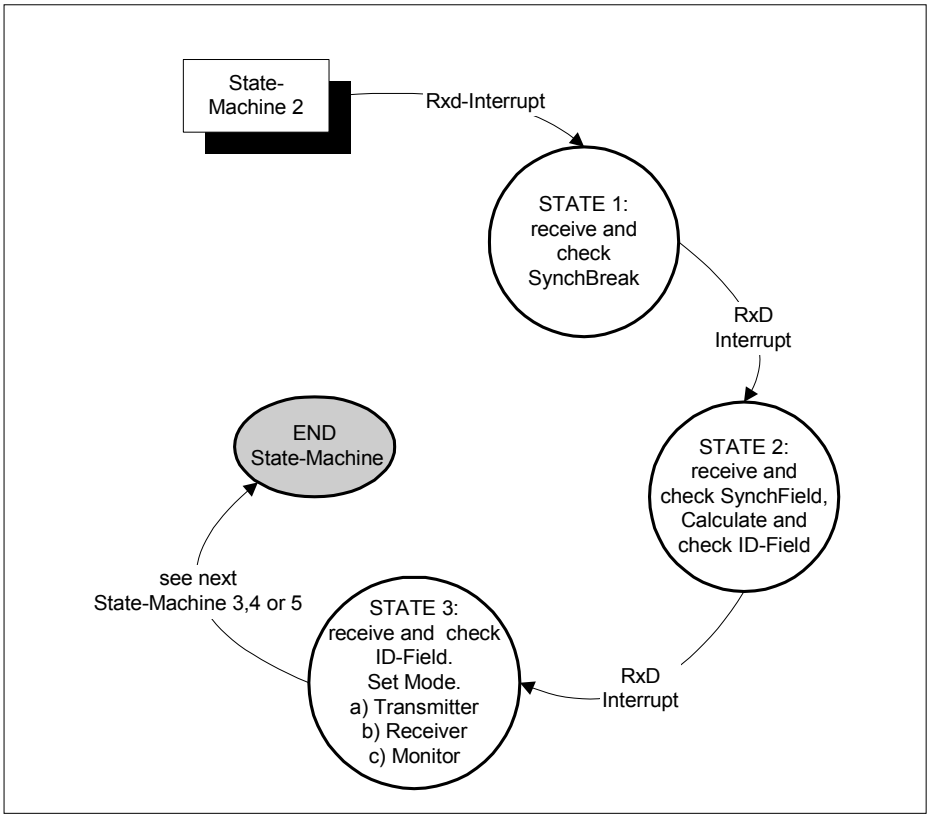

Figure 17 Overview of state-machine 2: Receiving Header. This state-machine has 4 substates.

Table 6 STATE and the responsible Transceiver-Status

State	Respective Function
STATE 1: **TS1**	in `LIN_vRxd_Interrupt()`-function → `LIN_vReceiveSynchBreak(…)`
STATE 2: **TS4**	in `LIN_vRxd_Interrupt()-function` → `LIN_vReceiveSynchField(…)`
STATE 3: **TS5**	in `LIN_vRxd_Interrupt()-function` → `LIN_vReceiveIdField(…)`

8.1.3 State-Machine 3: Transmit databytes

Figure 18 shows an overview of the state-machine, which will be used when sending the databytes on the bus if the ID was defined as SEND-ID. For more details please have a look at the Appendix.

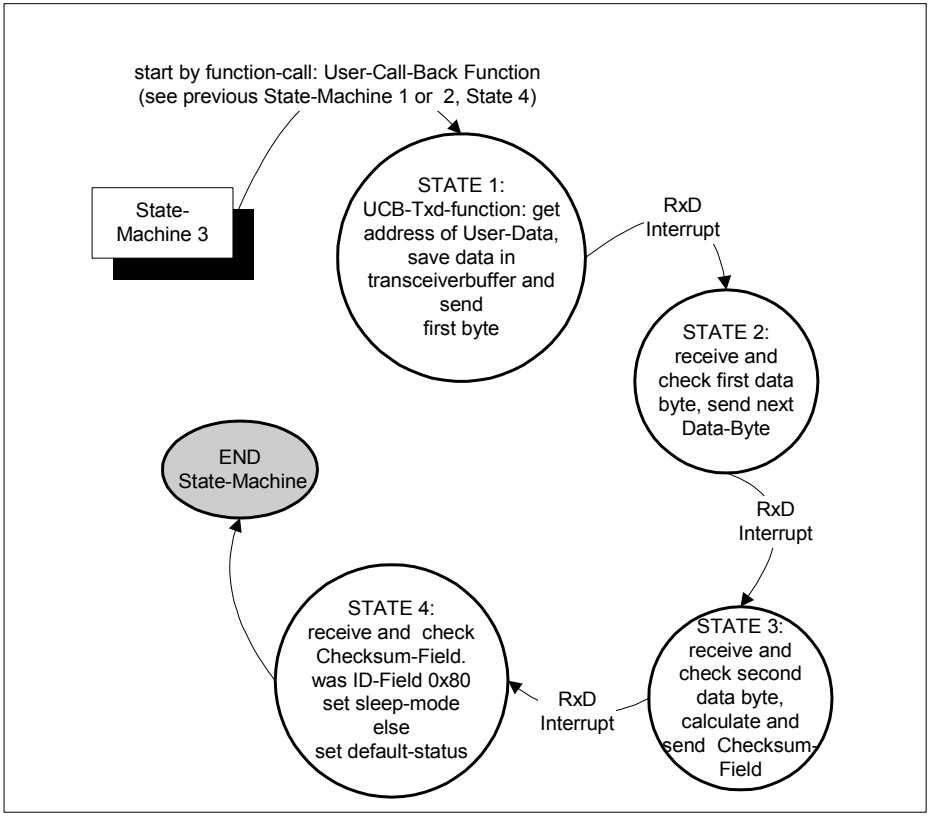

Figure 18 Overview of state-machine 3: Transmit databytes. This state-machine has 4 substates.

Table 7 STATE and the responsible Transceiver-Status

State	Respective Function
STATE 1: **TS6**	in `LIN_vRxd_Interrupt()`-function (function call: `UCB-Txd`)
STATE 2: **TS6**	in `LIN_vRxd_Interrupt()`-function
STATE 3: **TS6**	in `LIN_vRxd_Interrupt()`-function
STATE 3: **TS9**	in `LIN_vRxd_Interrupt()`-function

8.1.4 State-Machine 4: Receive databytes

Figure 19 shows an overview of the state-machine, which will be used when receiving the databytes from the bus if the ID was defined as REC-ID. For more details please have a look at the Appendix.

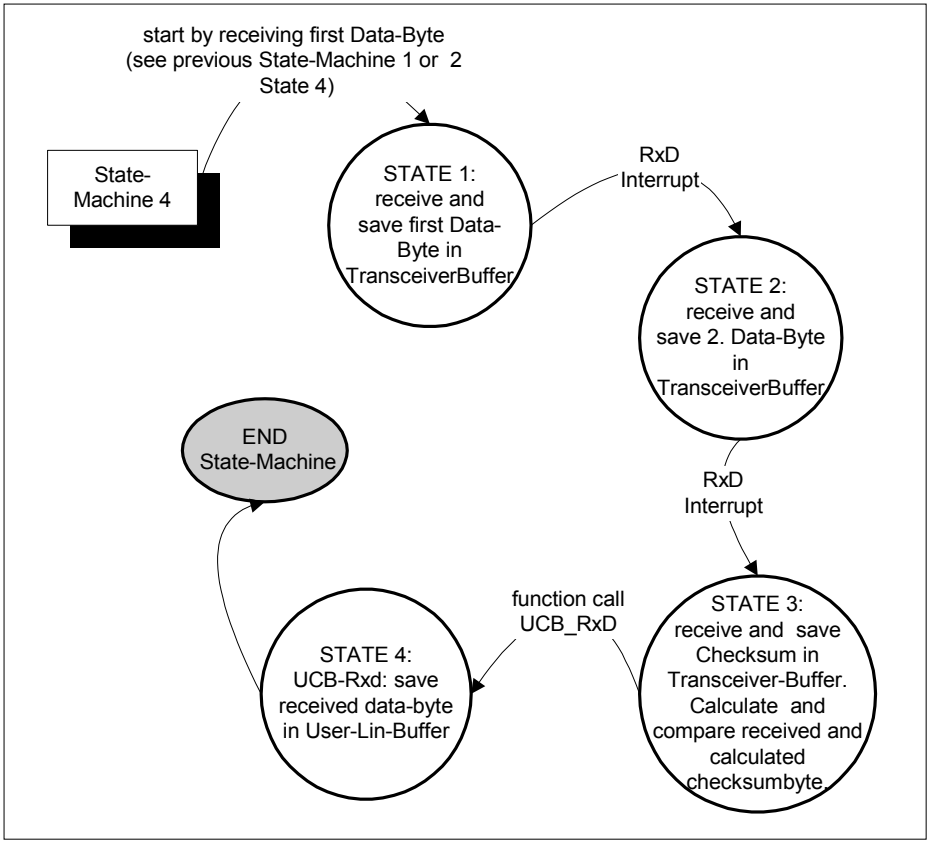

Figure 19 Overview of state-machine 4: Receive databytes. This state-machine has 4 substates.

8.2 Description: STATE and the responsible Transceiver-Status

Table 8 STATE and the responsible Transceiver-Status

State	Respective Function
STATE 1: **TS7**	in `LIN_vRxd_Interrupt()`-function
STATE 2: **TS7**	in `LIN_vRxd_Interrupt()`-function
STATE 3: **TS7**	in `LIN_vRxd_Interrupt()`-function
STATE 3: **TS7**	in `LIN_vRxd_Interrupt()`-function (function call: `UCB-Rxd`)

8.2.1 State-Machine 5: Monitor

Figure 20 shows an overview of the state-machine, which will be used when receiving the databytes from the bus if the ID is not defined in the slave node or if the ID is defined as MON-ID in the master node.

a) SLAVE NODE: If the received ID was not defined then no error-detection is needed, and the databytes don't have to be saved in the user-buffer, but the LIN driver will receive (or monitor) the databyte because: If the LIN driver would stop the state-machine after receiving the ID-Field then the node would go in the default state. After the ID-Field appears, the first databyte on the bus and the Node in the default state would try to receive this databyte as a synchbreak-byte and an unnecessary error would occur.
b) MASTER_NODE: The master node has to check each message. Therefore, the Slave task (in the master node) will calculate the Checksum and check if the transfer was correct.

For more details please have a look at the Appendix.

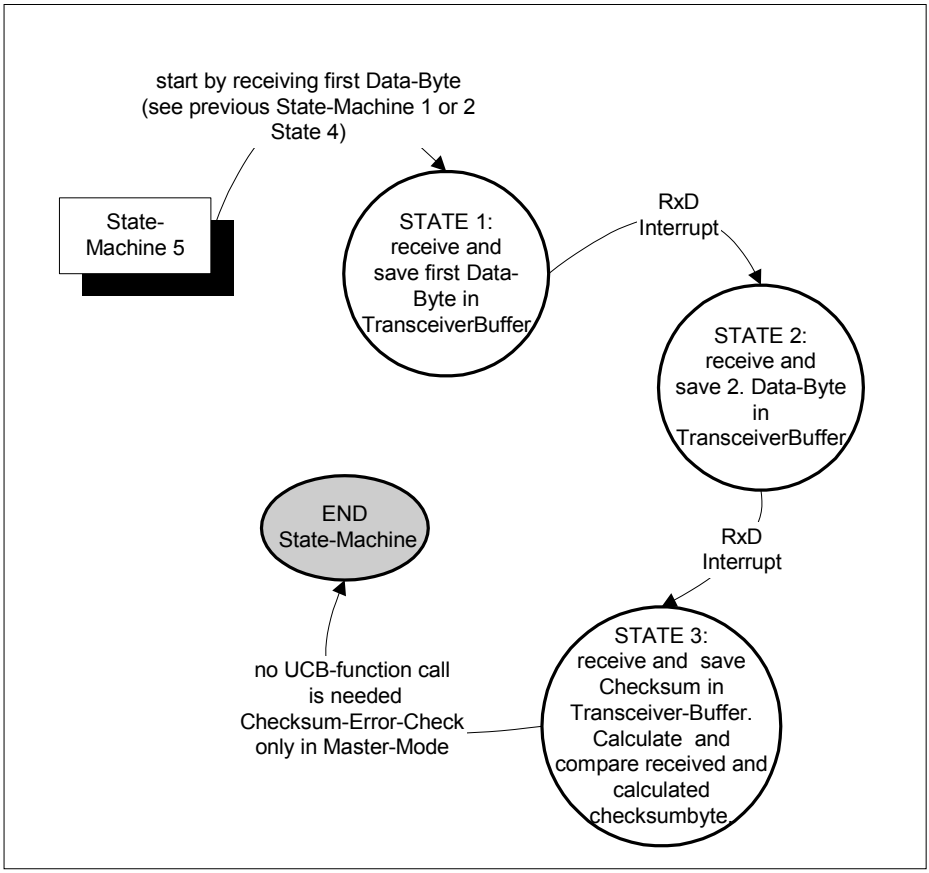

Figure 20 Overview of state-machine 5: Receive databytes. This state-machine has 4 substates.

Table 9 STATE and the responsible Transceiver-Status

State	Respective Function
STATE 1: **TS8**	in `LIN_vRxd_Interrupt()`-function
STATE 2: **TS8**	in `LIN_vRxd_Interrupt()`-function
STATE 3: **TS8**	in `LIN_vRxd_Interrupt()`-function

9 Appendix D: LIN Driver Status Fields

The LIN driver operates by state-machines. To get an overview how the respective state-machines work and which states are needed for the different LIN-Function, a State Transition Table (STT) follows. This STT-description is needed in order to understand the following detailed description of the state-machines.

Table 10 Legend

Symbol	Description
1	Bit is used
0 or blank	Bit is unused
X	Don't care
A	Bit is used, see condition

Table 11 Abbreviations for the states

Symbol	Description
MSx	Main state of the node
NSx	State in the node status field
TSx	State in the transceiver status field
PSx	State in the pending status field

Status Fields

```
Typedef enum
{
    l_sleep                         = 0x01,
    l_ready_for_synchbreak          = 0x02,
    l_not_ready_for_synchbreak      = 0x04,
    l_disconnect                    = 0x08
}NODE_STATUS;
```

```
typedef enum
{
   l_default_status          = 1,
   l_synchbreak_started      = 2,
   l_syndel_started          = 3,
   l_synchbreak_done         = 4,
   l_synchfield_received     = 5,
   l_i_am_transmitter        = 6,
   l_i_am_receiver           = 7,
   l_i_am_tester             = 8,
   l_checksum_ok             = 9
}TRANSCEIVER_STATUS;
typedef enum
{
   l_bit_error                         = 0x01,
   l_checksum_error                    = 0x02,
   l_id_parity_error                   = 0x04,
   l_slave_not_responding_error        = 0x08,
   l_inconsistent_synch_field_error    = 0x10,
   l_no_bus_activity_error             = 0x20
}ERROR_STATUS;
typedef enum
{
   l_default_pending   = 0x01,
   l_wakeup_pending  = 0x02,
   l_sleep_pending     = 0x04
}PENDING_STATUS;
```

Table 12 Main Modes

State	Mode	Description: #define statement
MS1	MASTER	If defined as master, node has a master task and a slave task
MS2	SLAVE	If defined as slave, node has only an slave task

Table 13 Synchbreak Modes

State	Mode	Description: #define statement
SB0	SEND_SYNCH_ MODE:= 0	Generates synchbreak by sending 0x00 with lower baudrate
SB1	SEND_SYNCH_ MODE:= 1	Generates synchbreak with compare function.

Table 14 STT: Node Status Byte

State	Description: enum-statement	0x01 sleep	0x02 ready_for_synchbreak	0x04 not_ready_for_synchbreak	0x08 disconnect
NS1	Node in sleepmode	1	0	0	0
NS2	Sending a new message is only possible if this bit is set.	0	1	0	0
NS3	Node not ready for new synchbreak	0	0	1	0
NS4	Node is not connected to the LIN bus	0	0	0	1

Table 15 Pending Status Byte

State	Description: enum-statement	0x01 default_pending	0x02 wakeup_pending	0x04 sleep_pending
PS1	No pending, default-status	1	0	0
PS2	Node is responsible for network wakeup procedure	0	1	0
PS3	Only available in master node: the sleep-frame will be sent by the next opportunity	0	0	1

Table 16 STT: Transceiver Status Byte

State	Main Mode: MS1 and SB0 are set. Description: • States needed for „Send Message“ • Master • Synchbreak will be generated by sending 0x00 with lower baudrate	1 default_status	2 synchbreak_started	3 syndel_started	4 synchbreak_done	5 synchfield_received	6 i_am_transmitter	7 i_am_receiver	8 i_am_tester	9 checksum_ok
TS1	Node in default state	1								
TS2	The synchbreak is currently being sent		1							
TS3	The synchDelimiter is currently being sent			0						
TS4	The synchbreak phase has been finished				1					
TS5	The synchfield has been received					1				

TS6	The node is destined for sending the databytes						1			
TS7	The node is destined for receiving the databytes							1		
TS8	The node has only monitor function								1	
TS9	The checksum-field has been received correct									1

Table 17 STT: Transceiver Status Byte

State	**Main Mode: MS1 and SB1 are set.** Description: • States needed for „Send Message“ • Master • Synchbreak will be generated by compare function	**1 default_status**	**2 synchbreak_started**	**3 syndel_started**	**4 synchbreak_done**	**5 synchfield_received**	**6 i_am_transmitter**	**7 i_am_receiver**	**8 i_am_tester**	**9 checksum_ok**
TS1	Node in default state	1								
TS2	The synchbreak is currently being sent		1							
TS3	The synchDelimiter is currently being sent			1						
TS4	The synchbreak phase has been finished				1					
TS5	The synchfield has been received					1				
TS6	The node is destined for sending the databytes						1			
TS7	The node is destined for receiving the databytes							1		
TS8	The node has only monitor function								1	
TS9	The checksum-field has been received correctly									1

Table 18 STT: Transceiver Status Byte

State	**Main Mode: MS2 and SB0 is set** **Node Status: NS3 is set** Description: All these states are handled by the slave to receiving a header and: • to send the data bytes • to receive the data bytes • to monitor the data bytes	1 default_status	2 synchbreak_started	3 syndel_started	4 synchbreak_done	5 synchfield_received	6 i_am_transmitter	7 i_am_receiver	8 i_am_tester	9 checksum_ok
TS1	Node in default state	1								
TS2	The synchbreak is currently being sent	0	0	0	0	0	0	0	0	0
TS3	The synchDelimiter is currently being sent	0	0	0	0	0	0	0	0	0
TS4	The synchbreak phase has been finished				1					
TS5	The synchfield has been received					1				
TS6	The node is destined for sending the databytes						A			
TS7	The node is destined for receiving the databytes							B		
TS8	The node has only monitor function								C	
TS9	The checksum-field has been received correct									A

Table 19 STT: Transceiver Status Byte

State	**Main Mode: MS2 and SB1 are set** **Node Status: NS3 is set** Description: All these states are handled by the slave to receiving a header and: • to send the data bytes • to receive the data bytes • to monitor the data bytes	**1 default_status**	**2 synchbreak_started**	**3 syndel_started**	**4 synchbreak_done**	**5 synchfield_received**	**6 i_am_transmitter**	**7 i_am_receiver**	**8 i_am_tester**	**9 checksum_ok**
TS1	Node in default state	1								
TS2	The synchbreak is currently being sent		1							
TS3	The synchDelimiter is currently being sent	0	0	0	0	0	0	0	0	0
TS4	The synchbreak phase has been finished				1					
TS5	The synchfield has been received					1				
TS6	The node is destined for sending the databytes						A			
TS7	The node is destined for receiving the databytes							B		
TS8	The node has only monitor function								C	
TS9	The checksum-field has been received correctly									A

http://www.infineon.com

Published by Infineon Technologies AG