

To our customers,

---

## Old Company Name in Catalogs and Other Documents

---

On April 1<sup>st</sup>, 2010, NEC Electronics Corporation merged with Renesas Technology Corporation, and Renesas Electronics Corporation took over all the business of both companies. Therefore, although the old company name remains in this document, it is a valid Renesas Electronics document. We appreciate your understanding.

Renesas Electronics website: <http://www.renesas.com>

April 1<sup>st</sup>, 2010  
Renesas Electronics Corporation

Issued by: Renesas Electronics Corporation (<http://www.renesas.com>)

Send any inquiries to <http://www.renesas.com/inquiry>.

## Notice

1. All information included in this document is current as of the date this document is issued. Such information, however, is subject to change without any prior notice. Before purchasing or using any Renesas Electronics products listed herein, please confirm the latest product information with a Renesas Electronics sales office. Also, please pay regular and careful attention to additional and different information to be disclosed by Renesas Electronics such as that disclosed through our website.
2. Renesas Electronics does not assume any liability for infringement of patents, copyrights, or other intellectual property rights of third parties by or arising from the use of Renesas Electronics products or technical information described in this document. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
3. You should not alter, modify, copy, or otherwise misappropriate any Renesas Electronics product, whether in whole or in part.
4. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation of these circuits, software, and information in the design of your equipment. Renesas Electronics assumes no responsibility for any losses incurred by you or third parties arising from the use of these circuits, software, or information.
5. When exporting the products or technology described in this document, you should comply with the applicable export control laws and regulations and follow the procedures required by such laws and regulations. You should not use Renesas Electronics products or the technology described in this document for any purpose relating to military applications or use by the military, including but not limited to the development of weapons of mass destruction. Renesas Electronics products and technology may not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations.
6. Renesas Electronics has used reasonable care in preparing the information included in this document, but Renesas Electronics does not warrant that such information is error free. Renesas Electronics assumes no liability whatsoever for any damages incurred by you resulting from errors in or omissions from the information included herein.
7. Renesas Electronics products are classified according to the following three quality grades: “Standard”, “High Quality”, and “Specific”. The recommended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below. You must check the quality grade of each Renesas Electronics product before using it in a particular application. You may not use any Renesas Electronics product for any application categorized as “Specific” without the prior written consent of Renesas Electronics. Further, you may not use any Renesas Electronics product for any application for which it is not intended without the prior written consent of Renesas Electronics. Renesas Electronics shall not be in any way liable for any damages or losses incurred by you or third parties arising from the use of any Renesas Electronics product for an application categorized as “Specific” or for which the product is not intended where you have failed to obtain the prior written consent of Renesas Electronics. The quality grade of each Renesas Electronics product is “Standard” unless otherwise expressly specified in a Renesas Electronics data sheets or data books, etc.
  - “Standard”: Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; and industrial robots.
  - “High Quality”: Transportation equipment (automobiles, trains, ships, etc.); traffic control systems; anti-disaster systems; anti-crime systems; safety equipment; and medical equipment not specifically designed for life support.
  - “Specific”: Aircraft; aerospace equipment; submersible repeaters; nuclear reactor control systems; medical equipment or systems for life support (e.g. artificial life support devices or systems), surgical implantations, or healthcare intervention (e.g. excision, etc.), and any other applications or purposes that pose a direct threat to human life.
8. You should use the Renesas Electronics products described in this document within the range specified by Renesas Electronics, especially with respect to the maximum rating, operating supply voltage range, movement power voltage range, heat radiation characteristics, installation and other product characteristics. Renesas Electronics shall have no liability for malfunctions or damages arising out of the use of Renesas Electronics products beyond such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of its products, semiconductor products have specific characteristics such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Further, Renesas Electronics products are not subject to radiation resistance design. Please be sure to implement safety measures to guard them against the possibility of physical injury, and injury or damage caused by fire in the event of the failure of a Renesas Electronics product, such as safety design for hardware and software including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult, please evaluate the safety of the final products or system manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. Please use Renesas Electronics products in compliance with all applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive. Renesas Electronics assumes no liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. This document may not be reproduced or duplicated, in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products, or if you have any other inquiries.

(Note 1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its majority-owned subsidiaries.

(Note 2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.



## **Application Note**

# **NEC V85x - 78k0 - 78k0S**

## **Single-Chip Microcontroller**

### **Standalone NEC LIN-driver for Master and Slave**

---

Global Document No. U18063EE2V2AN00 (2nd edition)

Document No. EACT-AN-5502-V2.2

Date Published April 2003

© NEC Corporation 2003

Printed in Germany

# Table Of Contents

Chapter 1.	Preface .....	6
Chapter 2.	Introduction .....	8
Chapter 3.	LIN Specification Details .....	10
3.1.	Intention .....	10
3.2.	The protocol .....	10
3.2.1.	Overview .....	10
3.2.2.	Frame-dividing .....	10
3.3.	Master Frame Layout .....	11
3.3.1.	SyncHBreak .....	11
3.3.2.	SyncField .....	12
3.3.3.	Identifier .....	12
3.4.	Slave Frame Layout .....	14
3.5.	Protocol Frames .....	14
Chapter 4.	LIN Master-driver: NEC V850 .....	16
4.1.	Introduction .....	16
4.2.	LIN-Master Overview .....	16
4.3.	List of used files .....	16
4.3.1.	Hardware.h .....	17
4.3.2.	UART.h .....	19
4.3.3.	LIN.h .....	21
4.3.4.	M_Master.h .....	23
4.4.	M_Master.c .....	24
4.4.1.	startLin .....	24
4.4.2.	stopLin .....	25
4.4.3.	sendBusToStop .....	26
4.4.4.	initHardware .....	27
4.4.5.	initUART .....	28
4.4.6.	initTimer .....	29
4.4.7.	initScheduleTable .....	30
4.4.8.	initIDLengthTable .....	31
4.4.9.	startTimer .....	33
4.4.10.	stopTimer .....	34
4.4.11.	setUARTForSyncBreak .....	35
4.4.12.	setUARTOnNormalSpeed .....	36
4.4.13.	scheduleSending .....	37
4.4.14.	sendData .....	39
4.4.15.	Interrupt SioTxInt .....	40
4.4.16.	Interrupt SioRxInterrupt .....	41
4.4.17.	TimerCompInterrupt .....	44

# Table Of Contents

<b>Chapter 5.</b>	<b>LIN-Slave Driver 78k0</b>	<b>46</b>
5.1.	Introduction	46
5.2.	LIN-Slave Overview	46
5.2.1.	Receiving non-standard-format SyncHBreak-Field	46
5.3.	List of used files	46
5.3.1.	Hardware.h	47
5.3.2.	UART.h	49
5.3.3.	LIN.h	51
5.3.4.	M_Slave.h	52
5.4.	Functions of the LIN-Slave driver	52
5.4.1.	startLIN	53
5.4.2.	stopLin	54
5.4.3.	sendBusToStop	55
5.4.4.	initHardware	56
5.4.5.	initUART	57
5.4.6.	initTimer	58
5.4.7.	initDataTable	59
5.4.8.	initDataTableLength	60
5.4.9.	calculateChecksum	61
5.4.10.	sendData	62
5.4.11.	startTimer	63
5.4.12.	stopTimer	64
5.4.13.	setUARTOnNormalSpeed	65
5.4.14.	setUARTForSyncBreak	66
5.4.15.	<i>scheduleSending</i>	67
5.4.16.	<b>Interrupt</b> <i>SioTxInterrupt</i>	69
5.4.17.	<b>Interrupt</b> <i>SioRxInterrupt</i>	70
5.4.18.	<b>Interrupt</b> <i>ExternalInterrupt</i>	74
5.4.19.	<i>TimerComplInterrupt</i>	75
<b>Chapter 6.</b>	<b>LIN-Master Driver 78k0</b>	<b>76</b>
6.1.	Intention	76
6.2.	Realisation	76
<b>Chapter 7.</b>	<b>Differences to the LIN-Master-driver using LIN-UART6</b>	<b>78</b>
7.1.	Intention	78
7.2.	LIN-UART6 - short overview	78
7.3.	List of Adaptions	78
7.3.1.	Sending SyncHBreak-Frames	78
7.3.2.	Reacting on WakeUp-Signals	78
7.4.	Use of changes in the NEC-LIN_Master driver	78
7.4.1.	Changes to LIN_m.h	79
7.4.2.	Changes to UART_m.h	79
7.4.3.	Changes to m_Master.c	79

# Table Of Contents

<b>Chapter 8.</b>	<b>Differences to the LIN-Slave Driver using LIN-UART6</b>	<b>80</b>
<b>8.1.</b>	<b>Intention</b>	<b>80</b>
<b>8.2.</b>	<b>LIN-UART6 - short overview</b>	<b>80</b>
<b>8.3.</b>	<b>List of Adaptions</b>	<b>80</b>
<b>8.3.1.</b>	<b>Receiving SynchBreak-Frames</b>	<b>80</b>
<b>8.3.2.</b>	<b>Receiving Sync-Fields</b>	<b>80</b>
<b>8.3.3.</b>	<b>Reacting on WakeUp-Signals</b>	<b>81</b>
<b>8.4.</b>	<b>Use of changes in the NEC-LIN_Slave driver</b>	<b>81</b>
<b>8.4.1.</b>	<b>Changes to LIN.h</b>	<b>81</b>
<b>8.4.2.</b>	<b>Changes to UART.h</b>	<b>81</b>
<b>8.4.3.</b>	<b>Changes to m_slave.c</b>	<b>82</b>
<b>Appendix A</b>	<b>Application for the V850 LIN-Master Driver</b>	<b>84</b>
<b>Appendix B</b>	<b>Application for the LIN-Slave Driver</b>	<b>86</b>
<b>Appendix C</b>	<b>LIN-Emulation using CANoe with option LIN</b>	<b>90</b>
<b>Appendix D</b>	<b>Network Overview</b>	<b>98</b>
<b>Appendix E</b>	<b>Revision History</b>	<b>100</b>
<b>Appendix F</b>	<b>Software included</b>	<b>102</b>
<b>Appendix G</b>	<b>Technical details, ressources, Implementations</b>	<b>104</b>

# List of Figures

1.	Overview of the whole LIN-message-frame .....	10
2.	SyncHBreak-Field .....	11
3.	Sync-Field .....	12
4.	Identifier .....	12
5.	Response by Slave upon valid identifier .....	14
6.	Header-file hardware_m.h .....	17
7.	Header file UART.h .....	19
8.	Header-file LIN.h - Initialization .....	21
9.	Header-file Master_M.h .....	23
10.	Function startLin .....	24
11.	Function stopLin .....	25
12.	Function sendBusToStop .....	26
13.	Function initHardware .....	27
14.	Function initUART .....	28
15.	Function initTimer .....	29
16.	Function initScheduleTable .....	30
17.	Function initIDLengthTable .....	31
18.	Example of an initialized receive-Table .....	32
19.	Function startTimer .....	33
20.	Function stopTimer .....	34
21.	Function setUARTForSyncBreak .....	35
22.	Function setUARTOnNormalSpeed .....	36
23.	Function scheduleSending .....	37
24.	Function sendSyncBreak .....	39
25.	Function interrupt sioRxInterrupt .....	40
26.	SioRxInterrupt re-reading just sent data .....	41
27.	SioRxInterrupt - Response Part 1 .....	41
28.	SioRxInterrupt - inputCt == 3 .....	42
29.	Interrupt-Receive - store received Data .....	43
30.	Function interrupt TimerCompareInterrupt .....	44
31.	Header-file Hardware.h .....	47
32.	Header-file UART.h .....	49
33.	Header-file LIN.h - Definitions .....	51
34.	Header-file Slave.h .....	52
35.	Function startLin .....	53
36.	Function stopLin .....	54
37.	Function sendBusToStop .....	55
38.	Function initHardware .....	56
39.	Function initUART .....	57
40.	Function initTimer .....	58
41.	Function initDataTable .....	59
42.	Function initIDLengthTable .....	60
43.	Function calculateChecksum .....	61
44.	Routine sendData .....	62
45.	Function startTimer .....	63
46.	Function stopTimer .....	64
47.	Function setUARTOnNormalSpeed .....	65
48.	Function <i>setUARTForSyncBreak</i> .....	66
49.	Function scheduleSending .....	67
50.	Function interrupt sioRxInterrupt .....	69

# List of Figures

51.	Reception of a Framing-Error .....	70
52.	Schedule-position WAIT_FOR_SYNC_FIELD .....	71
53.	Reception of a Framing-Error .....	72
54.	External Interrupt-Function .....	74
55.	Function interrupt TimerCompareInterrupt .....	75
56.	Changes to LIN_m.h .....	79
57.	Changes to UART_m.h .....	79
58.	Changes to LIN.h for usage of UART6 .....	81
59.	Changes to UART.h for usage of UART6 .....	81
60.	Changes to m_slave.c - Receive-Routine .....	82
61.	Changes to initHardware following UART6-Functionality .....	82
62.	Changes to interrupt TimerCompInterrupt-Routine .....	83
63.	Modifications for UART6 - initHardware .....	83
64.	Definitions for application-use of LIN-Master-driver .....	84
65.	Initializations of standard LIN-Master-driver routines .....	85
66.	Main-routine of LIN-Master-application .....	85
67.	Slave-driver settings in the Slave LIN-header-file .....	86
68.	External definitions in the Slave-application .....	86
69.	Initialization of variables by settings and initial calls after reset ....	87
70.	Cyclic called LIN-main-routine .....	88
71.	Overview of the LIN-Emulation-environment, Master active .....	90
72.	Variables used by the Master-emulation .....	90
73.	Pre-start-routines .....	91
74.	On Start-routine .....	91
75.	Routine onTimer .....	92
76.	Protocol-overview using the LIN-Slave-driver .....	93
77.	Frame-detail of used LIN-slave-driver .....	93
78.	Definitions of internally used variables .....	94
79.	Lin-Slave-routine onPreStart in emulation .....	94
80.	Emulated Slave-routine on Start .....	95
81.	Protocol-overview using the LIN-Master-driver .....	96
82.	Frame-detail using the LIN-Master-driver .....	96
83.	Bus protocol using the NEC LIN-Master-driver .....	97
84.	Network-outline for test purposes .....	98
85.	Delivered files used by the LIN-Master-driver .....	102
86.	Delivered files used by the LIN-Slave-driver .....	102



## **Chapter 1   Preface**

After two years working with LIN in various applications, NEC joined the LIN-consortium as an associated member. NEC actively contributes its know-how in the field of LIN in that body. As a result of these activities NEC created a complete driver-suite, divided into a Master- and a Slave-thread.

The dominating intention on these LIN-drivers is to give customers a ready-to-use piece of software without dealing with the details of the LIN-protocol itself. Rather the customer can concentrate on the applications on top of the LIN-communication.

[MEMO]

## Chapter 2 Introduction

This document describes the stand-alone LIN-drivers for a *Master-Slave-System* using **NEC** micro-controllers.

The initial version of the LIN-driver was developed on the **V850-SF1** operating as a *LIN-Master* and the 78F9116 operating as a *LIN-Slave*. The development refers to the LIN-specification **version 1.1**. The adaptation to the recent version 1.2 is planned (i.e. including the special-frames-information into the drivers).

The Master-driver is capable to send various frames to LIN-slaves, attached to the LIN-bus. These LIN-messages will cause the slaves to send data back to the Master, or will initiate the Master itself to broadcast some data itself to the slaves, depending on the implemented message-scheduler.

The *LIN-slave-driver* recognizes one identifier for sending data to the *LIN-Master*, and one identifier to receive data sent by the LIN-Master in a broadcast. This can be changed by modifying the internal driver-structure.

All these settings are subject to be changed by header-files for Master and Slave separately. There are several settings which have to be done by the user before running the driver with Master or Slave. The detailed description of these items is located in the middle part of this document starting with Chapter 4 on page 16.

The last starting with Chapter 7 on page 78 will identify the differences between the Master- and the Slave-part using NECs improved LIN-UART6. These differences are bound to a short application that sends several identifiers by the Master while the Slave is only reacting on one identifier to send data, and on another identifier in order to receive data.

The appendix describes the application in detail, the standard driver-procedure-call (the API), the common timing-interface, and non-standard function-calls.

Besides, an example for a LIN-application and pictures of its hardware configuration are shown, along with some information on how the drivers have been tested and what kind of changes are feasible for future enhancements.

[MEMO]

## Chapter 3 LIN Specification Details

### 3.1 Intention

For a better and faster understanding, a short introduction into the usage and possibilities of the LIN-specification is given here.

This part of the application note is based on the LIN specification version 1.2, but the LIN-drivers presented by this application note support only version 1.1. Therefore major changes from version 1.1 to version 1.2 are not pointed out explicitly as there is the Multi-Master-usage and the special LIN-identifiers, which can be used for service-instructions or for non-LIN-standard data-format-delivery.

For the complete LIN-specification and its most recent release, please take a look at <http://www.lin-subbus.org>.

### 3.2 The protocol

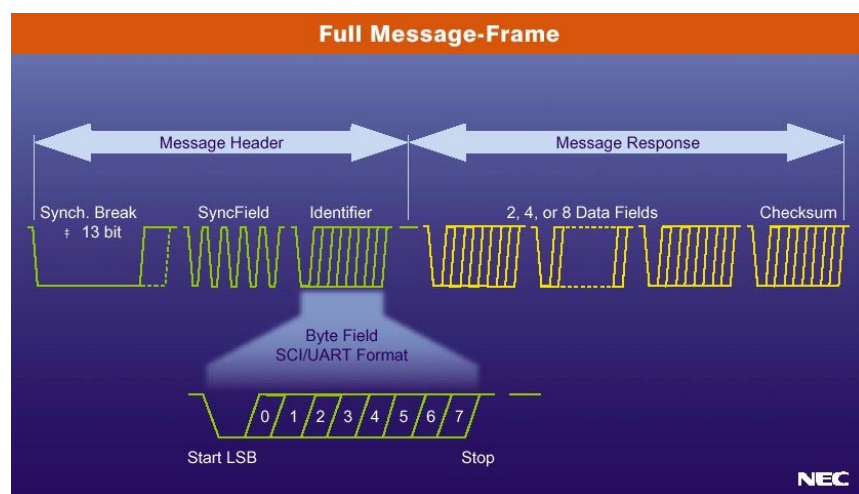
#### 3.2.1 Overview

The LIN-protocol is specified to schedule requests from a LIN-Master in a cyclic way to all attached LIN-Slaves. Such requests for information will cause to broadcast an identifier for the requested data to the bus. Based on this, exactly one Slave-node has to send data back to the Master. All other Slaves can use this data as well and act on the information sent by the particular Slave. In addition, the Master itself can send data to the bus using the implemented Slave-part inside the Master-device. This data can be used for status update, check-data and similar purposes.

To prevent more than one Slave sending data back to the Master responding to the same identifier, the application-coordinator should take care that one identifier is assigned to one Slave-response only. Thus the protocol requires that data is divided into single parts each linked to separate identifiers when different Slaves are affected.

#### 3.2.2 Frame-dividing

The following figure shows the transmission of a whole LIN-frame:



**Figure 3-1: Overview of the whole LIN-message-frame**

Like outlined in the above picture, the LIN-frame is divided into two major parts, the request by the Master and the response by the Slave. Each of them is divided itself into smaller-parts called fields.

### 3.3 Master Frame Layout

The request by the Master is split into three fields that sends parts from the table of the schedule to the bus. These three fields are called the **SyncHBreak**, the **SyncField** and the **Identifier**.

The sum of these three fields is called the LIN-header, which has to be received correctly by all Slaves attached to the bus in order to confirm that all devices will listen and use nearly the same bus-baudrate.

#### 3.3.1 SyncHBreak

The SyncHBreak is a non-standard UART-frame with a data-length of at least 13 Bit-times compared to the Master-Bit-time.

This frame is used as a *hello-world* - message and should set the applications running on the attached Slaves to the mode **Waiting for the SyncField**.

The length of the SyncHBreak has to be at least 13Bit to fit in a LIN-system, where Slaves are running with an RC-oscillator for timebase-generation. With a maximum allowed dis-accuracy of 20%, the standard 8N1-data-format is received as 12-Bit data. To achieve that the received data is recognized as a SyncHBreak-field and not as a standard-format data-field with the maximum allowed deviation, a SyncHBreak consists of at least 13 BitTimes. The maximum value of the SyncHBreak depends on the application and the overall length of the LIN-protocol including the response of a Slave with a maximum of eight bytes of data and one byte of checksum.

Figure 3-2: shows the layout of a SyncHBreak-field:



**Figure 3-2: SyncHBreak-Field**

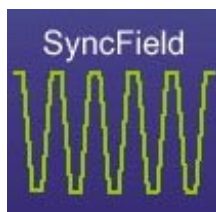
### 3.3.2 SyncField

The SyncField consists of the standard 8N1-data 0x55. This data-toggling will act as a synchronisation-message. If a Slave is running on an RC-oscillator-base, the incoming data is measured between at least one rising and one falling edge.

With the knowledge of the overall bus-speed, the Slave can calculate the new internal settings to fit the bus-speed for the actual transmission.

In the case, the Slave does not run on RC-oscillator but with a quartz or ceramic resonator, only a 0x55 should be received without any further action needed.

Figure 3-3: shows the SyncField:



**Figure 3-3: Sync-Field**

### 3.3.3 Identifier

The identifier ends the transmission of the request by the Master. Valid identifiers are listed in the Appendix C of the actual LIN-specification.

According to the identifier, one LIN-Slave has to send actual data to the Master, others may act on this identifier sent by the Master or on the data sent by the Slave.

The sending Slave can be a dedicated Slave providing data to the Master, or a Slave-task contained inside the Master, by which a broadcast of data from the Master to all attached Slaves can be done.

Figure 3-4: shows one valid identifier:



**Figure 3-4: Identifier**

As most important item the identifier includes the length of the awaited data sent by the Slave. Each identifier has its fixed length. With a theoretically maximum of 64 identifiers provided by the LIN-specification, there are 32 identifiers with a length of two Byte, 16 identifiers with a length of four Byte and 16 identifier with a length of eight Byte possible.

The relation between identifier value and its length is fixed. The length can be calculated out of the identifier as shown in Table 3-1:

ID 4	ID 5	number of data-fields
0	0	2 Byte
1	0	2 Byte
0	1	4 Byte
1	1	8 Byte

**Table 3-1:** *Relation between identifier value and length of the response*



### 3.4 Slave Frame Layout

The layout of frames for the LIN-slaves just contains the **Respons-Data** and a **Checksum** referenced to the request by Master.

Following the transmitted identifier, the activated Slave delivers the data byte by byte to the bus. In addition, a Checksum is calculated, depending on the data, which will be added after the data is sent to the bus. The checksum ends the communication.

The following figure shows the possible amount of a response by the Slave:



**Figure 3-5: Response by Slave upon valid identifier**

### 3.5 Protocol Frames

There are several more frames related to the standardized protocol. The description of these frames can be obtained from the LIN-specification. Some of them are:

- LIN-Stop
- WakeUp

[MEMO]

## Chapter 4 LIN Master-driver: NEC V850

### 4.1 Introduction

Using our knowledge build up in the past years developing systems and solutions for application for our customers, NEC discovered the need to develop LIN-drivers for NEC-devices without forcing the customers to use third-party tools for code generation or bus-debugging.

The main requirement for the implementation of this driver was the spare usage of resources .

Following this target, a set of small stand-alone drivers was implemented, which can be used without charge by our customers inside their applications.

### 4.2 LIN-Master Overview

The drivers for LIN-Master and LIN-Slave are both written in C. The main feature is the full access by the user. This facilitates to adapt the driver to special needs of the application by changing dedicated parameters inside the driver files.

These changes can affect the LIN-protocol definitions like the values of identifiers to be sent by the master, the amount of identifiers within the identifier-list, timings between the single requests by the Master (LIN-header) and the time until the response of the Slave awaited after the LIN-header finished. Other items with respect to particular UART macros and their settings for registers (i.e. baud rate setup) have to be done by the customer if other devices than those mentioned in the description are used. All these changes and some more minor changes in addition will be explained later on.

### 4.3 List of used files

The files generated for the LIN-Master are the following:

- m\_master.c
- m\_master.h
- LIN\_m.h
- hardware\_m.h
- UART\_m.h

The header-files contain the specific user-defined settings, the M\_Master.c-file contains all functions used to run the procedures needed by a LIN-Master in a LIN-bus system.

The application itself needs at least one more file, which calls the functions defined in M\_Master.c and the main-application-routine.

- V85\_main\_LINMaster.c

The header-files for the used device (df????h,...) are application- and hardware-specific and no subject of description in this document.

As a first step, the header-files with the used and needed configuration will be described, followed by the layout of the driver-file itself. Finally, a sample application will be explained including directions of how to use the necessary LIN-functions of the driver.

### 4.3.1 Hardware.h

The file hardware.h is used to define all hardware-related settings, which are not included in the LIN.h and the UART.h section.

The following picture shows the outline of the hardware.h-file:

```
// Timer-settings
// This timer is used for setting up a Master-counter loop as fixed timebase. All
// values described here and in the other config-files are related to this fixed time
// and should not be changed besides a special device is used not being caught by
// this file.

// user-defined settings
// register-setting to start cap/comp
#define TIMER_START          TMC0 = 0x0C
// mode the timer uses
#define TIMER_STOP           TMC0 = 0x00
// value which will be compared to
#define TIMER_COUNT_VAL      0x0640
// value of reg PRM00
#define TIMER_PRESCALAR_VAL_0 0xFF
// value of reg PRM01
#define TIMER_PRESCALAR_VAL_1 0x00
#define TIMER_CAPTURE_MODE    0x00

// standard-settings
#define TIMER_COUNT_REGISTER   TM0
#define TIMER_CAPTURE_COMPARE_REGISTER CR00
#define TIMER_MODE_CONTROL_REGISTER TMC0
#define TIMER_OUTPUT_CONTROL_REGISTER TOC0
#define PRESCALAR_MODE_REGISTER_0 PRM00
#define PRESCALAR_MODE_REGISTER_1 PRM01
#define PORT_MODE_REGISTER     PM2
#define CAPTURE_COMPARE_CONTROL_REGISTER CRC0

// interrupt-vector-settings
#define ASYNCHRONOUS_SERIAL_RECEIVE_INTERRUPT_VECTOR 0x00000210
#define ASYNCHRONOUS_SERIAL_TRANSMIT_INTERRUPT_VECTOR 0x00000220
#define TIMER_ZERO_COMPARE_HIT_INTERRUPT_VECTOR 0x00000150
```

**Figure 4-1: Header-file hardware\_m.h**

This file contains the definitions for the timer-related settings.

The registers of the timer are defined by macros to standard names, which are used within the LIN-driver. When the names of the registers change on a new device, only the related definitions inside the header-file have to be changed.

In a next step, the values of the register are defined. If other settings are needed, these definitions provide to change them quickly without touching the code of the driver.

All these settings have to be re-defined by the customer if another device shall be used or if the application requires different settings for the timer.

Please be aware of the fact, that the timer generates the tick, which is used by both, driver and application. **Changing the settings for the timer-base may have effect in that way that the driver will not work correctly anymore! In some cases, the settings for the timer have to be adjusted to fit the device-specified needs.**

The timer is set up to generate one tick once every millisecond. This will not put too much work load on the device but guarantees a good resolution. In addition, the used timer can be taken by the customers-application to generate the 1msec-tick. Therefore, the LIN-driver provides the flag **linFlag-Field.FLAG\_TIMER\_USE**, which can be scanned by the application.

**Please be aware to clear this timer-Flag (assign value 'FALSE') inside the polling routine testing the flag, when it is used inside the application, in order to guarantee the driver keeps running!**

Statement	Comment	Definition	To be adjusted by customer depending on the device used
TIMER_START	Value to the TM-ctr-register which starts counting	0x0C	
TIMER_STOP	Value to the TM-ctr-register which stops counting	0x00	
TIMER_CAPTURE_MODE	Sets the Timer into capture-mode	0x00	
TIMER_COUNT_REGISTER	Macro-definition for the TM-register	TM0	
TIMER_CAPTURE_COMPARE_REGISTER	Macro-definition for the CR-register	CR00	
TIMER_MODE_CONTROL_REGISTER	Macro-definition for the TMC-register	TMC0	
TIMER_OUTPUT_CONTROL_REGISTER	Macro-definition for the TO-register	TOC0	
PRESCALAR_MODE_REGISTER_0	Macro-definition for the PRM0-register	PRM00	
PRESCALAR_MODE_REGISTER_1	Macro-definition for the PRM1-register	PRM01	
PORT_MODE_REGISTER	Macro-definition for the PM-register	PM2	
CAPTURE_COMPARE_CONTROL_REGISTER	Macro-definition for the CRC-register	CRC0	
TIMER_COUNT_VAL	Value @ which an interrupt is generated (1 msec)	0x0640	
TIMER_PRESCALAR_VAL_0	Prescalar-value for PRM0	0xFF	
TIMER_PRESCALAR_VAL_1	Prescalar-value for PRM1	0x00	
ASYNCHRONOUS_SERIAL_RECEIVE_INTERRUPT_VECTOR	Address, where INT for Ser-receive is vectored	0x00000210	
ASYNCHRONOUS_SERIAL_TRANSMIT_INTERRUPT_VECTOR	Address, where INT for Ser-transmit is vectored	0x00000220	
TIMER_ZERO_COMPARE_HIT_INTERRUPT_VECTOR	Address, where INT for TIM-compare is vectored	0x00000150	

### 4.3.2 UART.h

The UART.h defines all settings, which are needed to use the device 'UART', excluding those definitions used for the LIN-specification.

Following figure shows a short overview of the used definitions:

```
// hardware-related settings

// user-defined settings
// sets Bit3 to 1
#define PM_Rx_set      PM1 | 0x08
// sets Bit4 to 1
#define PM_Tx_set      PM1 | 0x10
// sets Bit3 to 0
#define PM_Rx_resetPM1 & 0xF7
// sets Bit4 to 0
#define PM_Tx_resetPM1 & 0xE0
// sets Bit3 to 1
#define P_Rx_set       P1 | 0x08
// sets Bit4 to 1
#define P_Tx_set       P1 | 0x10
// sets Bit3 to 0
#define P_Rx_resetP1 & 0xF7
// sets Bit4 to 0
#define P_Tx_resetP1 & 0xE0
// #define TRUE      1
// #define FALSE     0
// sets ASIM0 transmit/receive
#define START_UART     UART_MODE_REGISTER = 0xC8
// sets to stop transmit/receive
#define STOP_UART      UART_MODE_REGISTER = 0x08
// sets to 2 Stop-Bit for SHB
#define START_UART_SHBUART_MODE_REGISTER = 0xCC
// switch whether CSIM-Register is present or not
#define CSIM_REGISTER_PRESENT 0
#define CSIM_VALUE 0x00

// standard-settings
#define BAUD_RATE_CONTROL_REGISTER BRGC0
#define BAUD_RATE_MODE_CONTROL_REGISTER_0 BRGMC00
#define BAUD_RATE_MODE_CONTROL_REGISTER_1 BRGMC01
#define UART_MODE_REGISTER ASIM0
#define UART_ERROR_REGISTER ASIS0
#define TRANSMIT_SHIFT_REGISTER TXS0
#define RECEIVE_BUFFER_REGISTER RXB0
#define PARITY_ERROR 0x04
#define FRAMING_ERROR 0x02
#define OVERRUN_ERROR 0x01
#define CSIM_REGISTER_PRESENT
#ifdef
#define SERIAL_OPERATION_MODE_REGISTER CSIM0
#endif // _UART_M_H
```

**Figure 4-2: Header file UART.h**

At first, the port and the port mode calculations for setting and clearing the UART-ports are defined. This part is utilized for reception and transmission of data during initializing. Next, some macros for start and stop are defined in order to provide concise access to the UART.

The re-definitions of names for the UART-registers are done to achieve full portability to all NEC-devices.

At last, the definitions for the UART-errors are done. These errors are used inside the driver to monitor the correct reception of incoming data. Related to these errors, the receive-routine is able to distinguish between data received inside the Sync-Break field from standard 8N1-data.

All these settings have to be adjusted by the customer when the driver is used with other devices.

Statement	Comment	Definition	To be adjusted by customer depending on the device used
PM_Rx_set	Definition to set the PM-register of the Rx-pin	PM1   0x08	
PM_Tx_set	Definition to set the PM-register of the Tx-pin	PM1   0x10	
PM_Rx_reset	Definition to reset the PM-register of the Rx-pin	PM1 & 0xF7	
PM_Tx_reset	Definition to reset the PM-register of the Tx-pin	PM1 & 0xE0	
P_Rx_set	Definition to set the P-register of the Rx-pin	P1   0x08	
P_Tx_set	Definition to set the P-register of the Tx-pin	P1   0x10	
P_Rx_reset	Definition to reset the P-register of the Rx-pin	P1 & ~xF7	
P_Tx_reset	Definition to reset the P-register of the Tx-pin	P1 & ~xE0	
BAUD_RATE_MODE_CONTROL_REGISTER_0	Macro-definition for the BRGMC0-register	BRGMC00	
BAUD_RATE_MODE_CONTROL_REGISTER_1	Macro-definition for the BRGMC1-register	BrRGMC01	
START_UART_SHB	Sets the UART into mode to send SHB-field	0xCC	
START_UART	Sets the UART into mode to start normal sending	0xC8	
STOP_UART	Stops the UART from sending	0x08	
CSIM_Register_Present	Defines whether CSIM-register is present or not	FALSE	
CSIM_Value	Sets the CSIM-register-value if present	0x00	
BAUD_RATE_CONTROL_REGISTER	Macro-definition for the BRGC-register	BRGC0	
UART_MODE_REGISTER	Macro-definition for the UART-mode-register	ASIM0	
UART_ERROR_REGISTER	Macro-definition for the UART-error-register	ASIS0	
TRANSMIT_SHIFT_REGISTER	Macro-definition for the TXS-register	TXS0	
RECEIVE_BUFFER_REGISTER	Macro-definition for the RXB-register	RXB0	
Parity_Error	Definition for the Parity-Error inside the ASIS-reg.	0x04	
Framing_Error	Definition for the Framing-Error in the ASIS-reg.	0x02	
Overrun_Error	Definition for the Overrun-Error in the ASIS-reg.	0x01	
SERIAL_OPERATION_MODE_REGISTER	Macro-definition for the CSIM-register	CSIM0	

The switch CSIM\_Register\_Present selects if a CSIM-register is attached inside the chip. This register is not present on all devices. The switch needs to be declared with respect to the particular device.

### 4.3.3 LIN.h

The LIN.h header-file contains the settings and definitions for all LIN-related settings. Most of these definitions are subject to be changed by the customer to meet the specific requirements of the application.

Here is a figure of the LIN.h-header-file:

```
// User-defined
// LIN-specific setting

#define BRGC_SETTING_19200_0 0xD0
#define BRGC_SETTING_19200_1 0x02
#define BRGC_SETTING_19200_2 0x00

#define BRGC_SETTING_9600_0 0xD0
#define BRGC_SETTING_9600_1 0x03
#define BRGC_SETTING_9600_2 0x00

// settings for pointers and receive-buffers

unsigned char *p_IDENTIFIER_TABLE;
unsigned int *p_ID_DELAY_TABLE;
unsigned char *p_ID_LENGTH_TABLE;
unsigned char RECEIVE_TABLE[30];
unsigned char *p_RECEIVE_TABLE;
unsigned char TEMP_RECEIVE_TABLE[15];
unsigned char *p_TEMP_RECEIVE_TABLE;

/* Definition of identifiers */

/* set up here the real count of scheduleTableEntries!! */
unsigned char SCHEDULE_TABLE_LENGTH = 5;
// defines the id-entrys for scheduling; ID[0..5]={0x04,0x06,0x20,0x2F,0x36}!!!!
unsigned char IDENTIFIER_TABLE[5] = {0xC4,0x06,0x20,0x6F,0x76};
//unsigned char IDENTIFIER_TABLE[5] = {0xC4,0xc4,0xc4,0xc4,0xc4};
//defines the id-entrys for internal Slave-routine
unsigned char INTERNAL_IDENTIFIER_TABLE[1] = {0xC4};
// set this to zero to en/disable
unsigned char INT_ID_TBL_LENGTH = 1;
unsigned char *p_INTERNAL_IDENTIFIER_TABLE;

// defines the array for application-data-storage
unsigned char APPLICATION_DATA_ARRAY[3] = {0x01,0x01,0xFD};
unsigned char *p_APPLICATION_DATA_ARRAY;
unsigned char APP_DATA_ARRAY_LENGTH = 3;
unsigned int ID_DELAY_TABLE[5] = {0x004D,0x004D,0x004D,0x004D,0x004D};
unsigned char ID_LENGTH_TABLE[5];

// settings for Time-outs
#define TIMEOUT_FRAME = 2000 // timeout no frame received
#define TIMEOUT_MASTER = 500 // timeout no Master-frame received
#define TIMEOUT_RESPONSE = 5000 // timeout no Slave responded
#define TIMEOUT_SYNCHBREAK = 100 // timeout no SynchBreak received
#define TIMEOUT_SYNCFIELD = 50 // timeout no SyncField
#define TIMEOUT_IDENTIFIER = 50 // timeout identifier but no SyncField

// settings for Frame-scheduling
#define TIME_BETWEEN_FRAMES 500 // initial value
#define TIME_AFTER_SYNC_BREAK 4
#define TIME_AFTER_SYNC_FIELD 2

// standard-settings
#define VERSION = 1.0_NEC // just a simple version-control-variable
#define SPEED_LOW = 2400 // speed-settings for the LIN-bus-protocol
#define SPEED_MEDIUM = 9600
#define SPEED_HIGH = 19200
#define WRITE_ACTIVE 0x01
#define WRITE_PASSIVE 0x00
#define MASTER = 1 // set this to one if Master is used
#define SLAVE = 0 // set this to one if Slave is used
#define YES = 1
#define NO = 0
#define READY = 1
#define NOT_READY = 0
// #define TRUE = 1
// #define FALSE = 0

#define MESSAGE_COUNT = 12 // number of used messages
#define FLAG_HEADER TRUE // Flag for Rx-Routine Master to receive data correctly
#define SYNC_BREAK 0x00
#define SYNC_FIELD 0x55
#define REQUEST_SLEEP 0x80
#define WAKEUP 0x80
```

**Figure 4-3: Header-file LIN.h - Initialization**



Here all definitions are given that are necessary for the LIN-specific hardware-layout of the LIN-drivers. Some of the definitions has to be changed by the customer, the rest of them are self-adapting macros

Statement	Comment	Definition	To be adjusted by customer depending on the device used
BRGC_SETTING_19200_0	Setting of BRGC0-register for 19200 baud	0xD0	
BRGC_SETTING_19200_1	Setting of BRGC1-register for 19200 baud	0xD2	
BRGC_SETTING_19200_2	Setting of BRGC2-register for 19200 baud	0x00	
BRGC_SETTING_9600_0	Setting of BRGC0-register for 9600 baud	0xD0	
BRGC_SETTING_9600_1	Setting of BRGC1-register for 9600 baud	0x03	
BRGC_SETTING_9600_2	Setting of BRGC2-register for 9600 baud	0x00	
SCHEDULE_TABLE_LENGTH	Count of identifiers used inside the sched-table	5	
IDENTIFIER_TABLE[5]	Table containing the identifiers sent by the master	s. header-file	
INTERNAL_IDENTIFIER_TABLE[1]	Identifier(s) causing the master to sent data itself	s. header-file	
INT_ID_TBL_LENGTH	Count of identifiers used inside the sched-table	1	
APPLICATION_DATA_ARRAY[3]	Array containing the data of application-hardware	here as example	
APP_DATA_ARRAY_LENGTH	Length of this array	3	
ID_DELAY_TABLE[5]	Table containing the length describing the pause between two frames	s. header-file	
ID_LENGTH_TABLE[5]	constant array	s. header-file	

#### 4.3.4 M\_Master.h

This header-file contains statements, which will be used in the routine for Master-driver. These definitions are neither device-specific nor hardware-related.

```
// used for declarartion of external/non-external VARs

#ifdef DECLARE_VARS
#define _NEC_
#else
#define _NEC_ extern
#endif

#define MAX_CHECKSUM 0x00FF

enum RUNMODI {HALT=1, STOP=2, BROADCAST_SLEEP=3, RETURN_TO_TEST_MONITOR = 4};

_NEC_struct {
    unsigned int ERROR_NO_ERROR           : 1;
    unsigned int ERROR_BIT_ERROR          : 1;
    unsigned int ERROR_CHECKSUM           : 1;
    unsigned int ERROR_SLAVE_NOT_RESPONDING : 1;
}linError;

_NEC_struct {
    unsigned int FLAG_RECEIVE             : 1;
    unsigned int FLAG_TIMER_RUN           : 1;
    unsigned int FLAG_HEADER_ACTIVE       : 1;
    unsigned int FLAG_ERROR_OCCURENCE     : 1;
    unsigned int FLAG_LIN_ACTIVE          : 1;
    unsigned int FLAG_TIMER_USE           : 1;
    unsigned int FLAG_SCHEDULE_DATA_SEND  : 1;
// for Slave-rt in Master-drv
    unsigned int NORMAL_SEND              : 1;
    unsigned int INT_ID_TBL_USED           : 1;
} linFlagField;

#ifdef _NEC_
#undef _NEC_
#endif
```

**Figure 4-4: Header-file Master\_M.h**

There are two structures, which contain the flags for error handling and for all regular operations of the program.

In addition, the MAX\_CHECKSUM used inside the receive-routine for checksum-testing is defined. Further, some operational modes are available. They distinguish between 'LIN-bus is stopped' or 'device is set into power-saving-mode'.

#### 4.4 M\_Master.c

The M\_Master.c contains all routines to engage the LIN-driver onto the Master-device.

There are several routines, which will be described later on. Most of them are called internally for initialisation, interrupt-request handling, data-recognition and data-transmission.

Only two routines have to be called from the main application, startLin and startScheduling.

##### 4.4.1 startLin

Function Prototype	input Variables	output Variables	calls Function:
startLin	-----	-----:	initHardware

The function startLin calls the function initHardware for further initialisations.

```
void startLin (void) {
/*=====*/
/* FunctionName: startLin                               */
/* IN/OUT      : -/-                                     */
/* Description : The application will call this routine to enable all  */
/*               LIN-related hardware and driver-settings.           */
/*               After calling this function, the scheduleMessages    */
/*               routine has to be called to start cyclic send of messages. */
/*=====*/
// starts initialization of all LIN- and Hardware-related items
initHardware ();
linFlagField.FLAG_LIN_ACTIVE      = TRUE;
linFlagField.FLAG_HEADER_ACTIVE  = TRUE;
linFlagField.FLAG_ERROR_OCCURENCE = FALSE;
linError.ERROR_BIT_ERROR         = FALSE;
linError.ERROR_CHECKSUM          = FALSE;
linError.ERROR_SLAVE_NOT_RESPONDING = FALSE;
WakeUp = TRUE;
}
```

**Figure 4-5: Function startLin**

Following the call of *initHardware()* several flags are initialized, which are used in this state and in the called function within the initialization.

At the end of the initialization-routine, a wake-Up-frame is sent to the bus to set all attached nodes in the wake-up-mode followed by the first frame set-up taken from the table carrying the schedule.

#### 4.4.2 stopLin

Function Prototype	input Variables	output Variables	calls Function:
stopLin	_stopMode	-----:	sendBusToStop

The function stopLin will be called to stop all working on LIN-specific functions. The flag *linFlagField.FLAG\_LIN\_ACTIVE* is set to FALSE by this function. This enables the function *startLin()* to issue an initialization when it is called again.

```
void stopLin (unsigned int _stopMode) {
/*=====*/
/* FunctionName: stopLin */
/* IN/OUT      : -/- */
/* Description : The application will call this routine to end all */
/*               services regarding LIN. */
/*               The scheduling is stopped and all values are re-set into */
/*               init-values */
/*=====*/

linFlagField.FLAG_LIN_ACTIVE = FALSE;
sendBusToStop (_stopMode );
}
```

**Figure 4-6: Function stopLin**

Calling the function *sendBusToStop()* sets the node into the desired stop-Mode depending on the parameter *\_stopMode*. The flag *linFlagField.FLAG\_LIN\_ACTIVE* will be set to FALSE to indicate the stop mode to the application.

## 4.4.3 sendBusToStop

Function Prototype	input Variables	output Variables	calls Function:
sendBusToStop	_runModeState	-----;	-----

When the system is set into stop mode, this function will be called to stop all work on LIN and set the device into the mode which is desired by the calling function.

```
void sendBusToStop (unsigned int _runModeState) {
/*=====*/
/* FunctionName: sendBusToStop */
/* IN/OUT      : -/- */
/* Description : This function is called when the Bus has to be set into Stop- */
/*               Mode. All LIN-related actions are stopped, the init-values are */
/*               re-written and the device is set into Stop-mode */
/*=====*/

    switch (_runModeState){
        case NOCHANGE:
            break;

        case HALT:
            // sets device into HALT-Mode
            _HALT;
            break;

        case STOP:
            // sets device into STOP-Mode
            _STOP;
            break;

        case BROADCAST_SLEEP:
            // sets device into STOP after broadc. Sleep
            sendData (WAKEUP);
            break;

        case RETURN_TO_TEST_MONITOR:
            // put the stuff here in for related tests...
            break;

        default:
            break;
    }
}
```

**Figure 4-7: Function sendBusToStop**

There are several modes available:

- NOCHANGE: The system runs without any change (needed, if the Master has a functionality, which causes him to run without changes)
- HALT: The system is set into HALT-mode
- STOP: The system goes to STOP-mode
- BROADCAST\_SLEEP: A sleep-request is sent to the LIN-bus
- RETURN\_TO\_TEST\_MONITOR: Function is needed for internal test purposes.

## 4.4.4 initHardware

Function Prototype	input Variables	output Variables	calls Function:
initHardware	-----	-----	initUART, initTimer, setUARTOnNormalSpeed, initScheduleTable, init IDLengthTable

The initHardware calls all routines that will initialize the used hardware macros and software modules.

```

void initHardware (void) {
/*=====*/
/* FunctionName: initHardware */
/* IN/OUT      : -/- */
/* Description : This function calls all other related init- and set- */
/*              functions for UART, timer, ScheduleTable. */
/*              */
/*              */
/*=====*/

// sets the UART-registers to init-values
initUART ();
// sets the UART-speed to standard bus-speed
setUARTOnNormalSpeed ();
// sets the Timer-registers to init-values
initTimer ();
// sets the sched-table to the first valid input
initScheduleTable ();
// inits the buffer for received messages
initIDLengthTable ();
linFlagField.FLAG_SCHEDULE_DATA_SEND = TRUE;
stateMode = 0;
}

```

**Figure 4-8: Function initHardware**

Additionally, the flag *linFlagField.FLAG\_SCHEDULE\_DATA\_SEND* is set to TRUE, and the flag *stateMode* is cleared.

## 4.4.5 initUART

Function Prototype	input Variables	output Variables	calls Function:
initUART	-----	-----;	-----

This routine sets the UART in the state to send and receive data via the UART-Macro.  
The UART is started at this time.

```
void initUART (void) {
    /*=====*/
    /* FunctionName: init UART                                     */
    /* IN/OUT      : -/-                                          */
    /* Description : This routine inits all values recent for the UART- */
    /*               macro.                                         */
    /*               It is called by the initHardware-function      */
    /*=====*/

    BAUD_RATE_CONTROL_REGISTER      =BRGC_SETTING_19200_0;
    BAUD_RATE_MODE_CONTROL_REGISTER_0 =BRGC_SETTING_19200_1;
    BAUD_RATE_MODE_CONTROL_REGISTER_1 =BRGC_SETTING_19200_2;
    SERICO = 0;
    STICO  = 0;
    START_UART;
    // set Port-Mode to 1 for receive
    PM_Rx_set;
    // set Port-Mode to 0 for transmit
    PM_Tx_reset;
    // set Port to 0 for output
    P_Tx_reset;
    PM1 = 0x2F;
    P1  = 0x2E;
}
```

**Figure 4-9: Function initUART**

The UART-mode is started by setting the baudrate-register to the desired busspeed, clearing the receive- and transmit interrupt-registers, and writing the value START\_UART into the UART-control-register. Afterwards, the input- and output-modes are defined and the ports are configured for the direction needed.

## 4.4.6 initTimer

Function Prototype	input Variables	output Variables	calls Function:
initTimer	-----	-----:	stopTimer

This routine sets the timer to a free-running mode. The time-base should be set to an equivalent of 256  $\mu$ sec, in order to derive a resolution that is suitable to generate the timer-tick for application and driver-calls.

In the initial state, the driver is set to 256  $\mu$ sec.

```

void initTimer (void) {
    /*=====*/
    /* FunctionName: initTimer */
    /* IN/OUT      : -/- */
    /* Description : This routine inits all values recent for the UART- */
    /*               macro. */
    /*               It is called by the initHardware-function */
    /*               */
    /*=====*/

    stopTimer ();
    // sets timer-base to 1 msec for all application-needs
    TIMER_CAPTURE_COMPARE_REGISTER = TIMER_COUNT_VAL;
    PRESCALAR_MODE_REGISTER_0      = TIMER_PRESCALAR_VAL_0;
    PRESCALAR_MODE_REGISTER_1      = TIMER_PRESCALAR_VAL_1;
    // sets Timer into Compare-Mode
    CAPTURE_COMPARE_CONTROL_REGISTER= TIMER_CAPTURE_MODE;
    // enable output for Timer
    TIMER_OUTPUT_CONTROL_REGISTER = TIMER_CONTROL_VALUE;
}

```

**Figure 4-10: Function initTimer**

At first the timer is stopped in order to prevent misbehaviour while accessing the timer-macro. Then the timer is set into compare-mode, the output is enabled, and the combination of prescalar/capture-compare-register is set to fit the above described needs for the internal time-base.

The routine does not provide return values.



## 4.4.7 initScheduleTable

Function Prototype	input Variables	output Variables	calls Function:
initScheduletable	-----	-----;	-----

The function `initScheduleTable` sets several pointers needed for data-calculation and reception with standard values. These values are set like in the following:

- The **`p_id_delay_table`** is set to the beginning of the **`identifier-delay-table`**
- the **`p_identifier_table`** is set to the beginning of the **`identifier_table`**
- **`p_id_length_table`** points to the first position of the **`id_length_table`**
- **`p_receive_table`** points to the start of the **`receive-table`**
- **`p_temp_receive_table`** is set to the starting-address of **`temp_receive_table`**

After setting these values, the **`receiveTablePosition`**, **`receiveTableOverallPosition`** and the **`schedulePosition`** are set to zero. At this time all values needed for the scheduling are initialized.

```
void initScheduleTable (void) {
/*=====*/
/* FunctionName: initScheduleTable */
/* IN/OUT      : -/- */
/* Description : This function is called by the initHardware-routine */
/*              and will init the Schedule-Table by choosing the right */
/*              table and setting the pointer to the valid first */
/*              input. */
/*=====*/

// sets Pointer to start of array
p_ID_DELAY_TABLE = &ID_DELAY_TABLE[0];
// sets Pointer to start of array
p_IDENTIFIER_TABLE = &IDENTIFIER_TABLE[0];
// sets Pointer to start of array
p_ID_LENGTH_TABLE = &ID_LENGTH_TABLE[0];
// sets Pointer to start of array
p_RECEIVE_TABLE = &RECEIVE_TABLE[0];
// sets Pointer to start of array
p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
// inits receiveTablePosition
receiveTablePosition = 0;
// inits receiveTableOverallPosition
receiveTableOverallPosition = 0;
// inits schedulePosition
schedulePosition = 0;

inputCt = 0;

if ((INT_ID_TBL_LENGTH) == 0){

    linFlagField.INT_ID_TBL_USED = FALSE;
} else {
    linFlagField.INT_ID_TBL_USED = TRUE;
}
p_INTERNAL_IDENTIFIER_TABLE = &INTERNAL_IDENTIFIER_TABLE[0];
p_APPLICATION_DATA_ARRAY = &APPLICATION_DATA_ARRAY[0];
linFlagField.NORMAL_SEND = TRUE;
receiveBufferPositionCounter = 0;
}
```

**Figure 4-11: Function `initScheduleTable`**

At the end, the variable `inputCt` is set to zero, and depending on `INT_ID_TBL_LENGTH`, some more flags used by this routine are initialized.

## 4.4.8 initIDLengthTable

Function Prototype	input Variables	output Variables	calls Function:
initIDLengthTable	-----	-----;	-----

The ID-Length-Table will be initialized with this function.

At first, the global variable *receiveTableLength* is set to zero. For the length of the identifier-table (calculated by the variable *scheduleTableLength*) the identifiers are ANDed with **0x30**. The result defines one out of four possible values for the length of the response by the Slave:

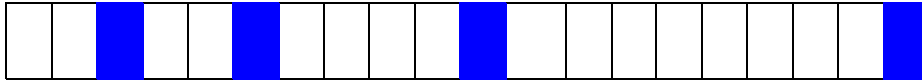
- 0x00: The response has a length of *two* Bytes
- 0x10: The response has a length of *two* Bytes
- 0x20: The response has a length of *four* Bytes
- 0x30: The response has a length of *Eight* Bytes

```
void initIDLengthTable ( void ) {
/*=====*/
/* FunctionName: initIDLengthTable */
/* IN/OUT      : -/- */
/* Description : This function is called by the initHardware-routine */
/*              and will init the Length- and Receive-Table by chosing*/
/*              and setting the pointer to the valid first */
/*              input. */
/*=====*/
    unsigned int length_input = 0;
    unsigned int i;
    receiveTableLength = 0;
    for ( i=0; i<=SCHEDULE_TABLE_LENGTH-1;i++){
        switch( ((*p_IDENTIFIER_TABLE) & (0x30))){
            case 0x00:
            case 0x10:
                length_input = 2;
                break;
            case 0x20:
                length_input = 4;
                break;
            case 0x30:
                length_input = 8;
                break;
            default:
                length_input = 0;
                break;
        } // end of switch
        //inc one for checksum
        ID_LENGTH_TABLE[i] = (length_input + 1);
        //adds length + checksum
        receiveTableLength += ++length_input;
        // increments array-length for write-active/passive-flag
        receiveTableLength++;
        // sets the buffer-Active-Flag to non-readable
        (*(p_RECEIVE_TABLE+ID_LENGTH_TABLE[i])) = 0x01;
        (p_RECEIVE_TABLE += (ID_LENGTH_TABLE[i]));
        // sets pointer to start-Bit of next data
        p_RECEIVE_TABLE++;
        // increments the pointer to next identifier in array
        p_IDENTIFIER_TABLE++;
    } // end of for
    // sets the pointer back again to the beginning of array
    p_IDENTIFIER_TABLE = &IDENTIFIER_TABLE[0];
    // sets the pointer back again to the beginning of array
    p_RECEIVE_TABLE = &RECEIVE_TABLE[0];
} // end of void
```

**Figure 4-12: Function initIDLengthTable**

The result of each calculated value will be added to the array *id\_length\_table[i]*. The value will be summarized in the variable *receiveTableLength* to get the result as an overall sum. Additionally, each summarized count will add one to the result, because the position of the write-active/passive-flag has to be taken into account as well.

This flag provides the status of reading/writing to the application. If the driver writes data into the receive-array, the flag is set (means 0x01) and the application has to poll this flag to ensure that only valid data will be read.



**Figure 4-13: Example of an initialized receive-Table**

In the initial state all the flags are set to 0x01 to ensure that the application knows that no valid data is inside the array.

After the full-initialization, the pointer *p\_receive\_table* points again to the beginning of the receive-table.

## 4.4.9 startTimer

Function Prototype	input Variables	output Variables	calls Function:
startTimer	-----	-----:	-----

This function is called to start the timer.

The flag *linFlagField.FLAG\_TIMER\_RUN* is set to signal that the timer is running. Then the **TIMER\_MODE\_CONTROL\_REGISTER** is written with a respective value to start the timer.

```
void startTimer (void) {
/*=====*/
/* FunctionName: startTimer */
/* IN/OUT      : -/- */
/* Description : This function is started when the scheduleMessages- */
/*               routine is called out of the application-main-file. */
/*               All init-settings fit the actual hardware at the mo- */
/*               ment and the timer can run with 1msec turnaround. */
/*=====*/

linFlagField.FLAG_TIMER_RUN = TRUE;
TIMER_START;
}
```

**Figure 4-14: Function startTimer**

## 4.4.10 stopTimer

Function Prototype	input Variables	output Variables	calls Function:
stopTimer	-----	-----:	-----

If the timer shall be stopped, this function needs to be called. Here, the flag *linFlagField.FLAG\_TIMER\_RUN* is set to *FALSE* to signal that the timer is not running anymore. After this, the ***TIMER\_MODE\_CONTROL\_REGISTER*** is set into stop-mode.

```
void stopTimer (void) {
/*=====*/
/* FunctionName: stopTimer                               */
/* IN/OUT      : -/ -                                     */
/* Description : The stopTimer function is called to halt all Timer-de-*/
/*               pendend actions while settings to the timer-control- */
/*               registers will be done.                          */
/*               After that, the timer has to be re-started.        */
/*=====*/
linFlagField.FLAG_TIMER_RUN = FALSE;
TIMER_STOP;
}
```

**Figure 4-15: Function stopTimer**

## 4.4.11 setUARTForSyncBreak

Function Prototype	input Variables	output Variables	calls Function:
setUARTForSyncBreak	-----	-----:	-----

In order to send the SyncBreak, the **BAUD\_RATE\_CONTROL\_REGISTER** needs to be set to at least half of baud rate of the bus. Thus, the normal the length of a data byte in relation to the bus speed enforces a SyncBreak with at least 18 bit to be received by the Slaves.

```
void setUARTForSyncBreak (void) {
    /*=====*/
    /* FunctionName: setUARTForSyncBreak */
    /* IN/OUT      : -/- */
    /* Description : This function will change the baudrate of the LIN- */
    /*               Master so the SyncHBreak-field is sent with half-speed*/
    /*               of the attached LIN-bus, so the restrictions for the */
    /*               Break-field can be hit. */
    /*=====*/

    STOP_UART;
    BAUD_RATE_CONTROL_REGISTER=BRGC_SETTING_9600_0;
    BAUD_RATE_MODE_CONTROL_REGISTER_0=BRGC_SETTING_9600_1;
    BAUD_RATE_MODE_CONTROL_REGISTER_1=BRGC_SETTING_9600_2;
    START_UART_SHB;
}
```

**Figure 4-16: Function setUARTForSyncBreak**

## 4.4.12 setUARTOnNormalSpeed

Function Prototype	input Variables	output Variables	calls Function:
setUARTOnNormalSpeed	-----	-----;	-----

This function is used to set the UART back to the previous baud rate if the SyncBreak was sent successfully. Here, too, the UART-macro has to be stopped before the baud rate is changed and started again afterwards.

```
void setUARTOnNormalSpeed (void) {
/*=====*/
/* FunctionName: setUARTOnNormalSpeed */
/* IN/OUT      : -/_actualBaudRate */
/* Description : This function is used to init the UART to the bus- */
/*               speed used by the LIN-system. */
/*               Later-on, the original bus-speed will be re-stored by */
/*               calling this function */
/*=====*/

STOP_UART;
BAUD_RATE_CONTROL_REGISTER =BRGC_SETTING_19200_0;
BAUD_RATE_MODE_CONTROL_REGISTER_0 =BRGC_SETTING_19200_1;
BAUD_RATE_MODE_CONTROL_REGISTER_1 =BRGC_SETTING_19200_2;
START_UART;
}
```

**Figure 4-17: Function setUARTOnNormalSpeed**

## 4.4.13 scheduleSending

Function Prototype	input Variables	output Variables	calls Function:
scheduleSending	-----	-----	startTimer, sendData, setUARTForSyncBreak,

The function *scheduleSending()* is the main procedure to be called by the application. With this call an internal statemachine is started, which then calls the functions for SyncBreak, SyncField and identifier.

```

void scheduleSending ( void ) {
/*=====*/
/* FunctionName: scheduleSending */
/* IN/OUT : -/- */
/* Description : This function is the routine called by the applica- */
/* tion. At the first beginning, a SyncHBreak is send */
/* to init the bus and the related in-driver settings */
/* =====*/
#ifdef shortSendRoutine
// if timer is not running yet
if ( !linFlagField.FLAG_TIMER_RUN ) {

    startTimer ();

} // end of if ( !Flag_TimerRun )
if ( WakeUp == TRUE ){
    dataToWrite = WAKEUP;
    sendData ( dataToWrite );
    WakeUp = FALSE;
    stateMode = 1;
    inFramePosition = 1;
    schedNextLINMessage = TIME_BETWEEN_FRAMES;
    inputCt = 0;
} else {
} // wakeup == true

switch(stateMode){
    case 1:
// sets UART on half speed
    setUARTForSyncBreak ();
    dataToWrite = SYNC_BREAK;
    linFlagField.FLAG_SCHEDULE_DATA_SEND = FALSE;
    inFramePosition = 2;
    inputCt = 0;
    if ( p_RECEIVE_TABLE != (&RECEIVE_TABLE[0] + receiveBufferPositionCounter) ){
        p_RECEIVE_TABLE = (&RECEIVE_TABLE[0] + receiveBufferPositionCounter);
// Slave has not responded
        linError.ERROR_SLAVE_NOT_RESPONDING = TRUE;
        p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
        p_ID_LENGTH_TABLE++;
// if the buffer overflows...
        if ( p_RECEIVE_TABLE >= ((&RECEIVE_TABLE[0]) + receiveTableLength ) ){
            p_RECEIVE_TABLE = &RECEIVE_TABLE[0];
            p_ID_LENGTH_TABLE = &ID_LENGTH_TABLE[0];
            receiveBufferPositionCounter = 0;
        } else {
        }
    } else {
// correct answer
        linError.ERROR_SLAVE_NOT_RESPONDING = FALSE;
//right position in queue, proceed
    }
    sendData ( dataToWrite );
    break;
default:
    break;
}
}
#endif // from shortSendRoutine

```

Figure 4-18: Function *scheduleSending*



For compatibility, there are two routines supporting scheduling provided, but only the **shortSendRoutine** should be used. This routine will send all data in one concatenated stream without being disturbed by application routines to long. Thus, the frames sent by the Master will be sent correctly.

A **#define** in the beginning of the file declares which of the two routines is selected.

The internal flow is divided by sending a protocol WakeUp-Frame and sending the Master header-Message.

In a first action, the timer-status is tested on RUN. Here the timer is started when it is not running yet. The time base of the scheduler is fixed to 1 msec.

Afterwards, the Wake-Up-Frame is sent if this is demanded by protocol and bus-status. The following Switch-statement is used by the Message-header-send-routine:

- state-Mode: Sending SyncBreak

In this state, the SyncBreak-field will be sent by the LIN-Master to the Slaves. The UART will be set to half-speed, so the non-standard SyncBreak-field can be sent. Depending on the answer received by the Master, the following action will be fulfilled inside the Transmit-Interrupt-Routine:

If the routine detects, that the Slave was not answering, the depending error-Flag is set, and the transmission for the next frames SyncHBreak is activated.

Otherwise, the state-counter is incremented and all other data (SyncField, Identifier, Data-Bytes and Parity) are sent using the Transmission-Routine sendData.

## 4.4.14 sendData

Function Prototype	input Variables	output Variables	calls Function:
sendData	_dataTableEntry	-----	-----

The function *sendData()* is called out of the scheduling-task for all Master-frame actions (i.e. sending the SyncBreak-Field, the Sync-Field, and the identifier) and out of the Transmit-Interrupt following the same action.

The TxD-Register is set with the parameter, which is given by the calling function..

```
void sendData (unsigned char _dataTableEntry){
    /*=====*/
    /* FunctionName: sendData */
    /* IN/OUT      : _dataTableEntry/- */
    /* Description : The sendData-Function is used to generate the */
    /*               data that will be the actual data. This data */
    /*               is put into the out-register and the function will */
    /*               wait until the data is sent */
    /*=====*/
    // sets the TXS-register to the actual data-value
    TRANSMIT_SHIFT_REGISTER = _dataTableEntry;
}
```

**Figure 4-19: Function sendSyncBreak**

After writing the data into the transmit register, the function returns.

## 4.4.15 Interrupt SioTxInt

Function Prototype	input Variables	output Variables	calls Function:
SioTxInterrupt	-----	-----	-----

This function is called when the transmission of one byte ended successfully. Dependent on the value of *inFramePosition*, the UART is set back again to normal speed to provide the following data with a speed defined by the application. If the whole frame is sent, the *inFramePosition* is set back again to init-Mode.

```
void SioTxInt( void )
#pragma ghs interrupt
{
    /*=====*/
    /* FunctionName: interrupt SioTxInterrupt */
    /* IN/OUT      : -/interrupt */
    /* Description : This interrupt-function is started when the Tx-ready- */
    /*               Interrupt is received. The wait-Flag is set to ready */
    /*               so all attached functions will know when the data is */
    /*               sent. */
    /*=====*/
    if (linFlagField.NORMAL_SEND == TRUE){
        switch (inFramePosition){
            case 2:
                inFramePosition = 3;
                linFlagField.FLAG_SCHEDULE_DATA_SEND = FALSE;
                setUARTOnNormalSpeed ();
                sendData ( SYNC_FIELD );
                break;          // for next SyncBreak!

            case 3:
                dataToWrite = *( p_IDENTIFIER_TABLE + schedulePosition );
                linFlagField.FLAG_SCHEDULE_DATA_SEND = FALSE;
                inFramePosition = 1;
                schedNextLINMessage = *( p_ID_DELAY_TABLE + schedulePosition ); // variable Var-array for sched
                if (schedulePosition <= (SCHEDULE_TABLE_LENGTH - 2) ){
                    schedulePosition += 1;
                }else{
                    schedulePosition = 0;
                }
                sendData ( dataToWrite );
                break;

            case 4:
                //scheduleSending();
                break;

            case 5:
                // inFramePosition = 1;
                break;
            default:
                break;
        } // end of switch

    } else { // NORMAL_SEND
        p_APPLICATION_DATA_ARRAY++;
        if (p_APPLICATION_DATA_ARRAY > ((&APPLICATION_DATA_ARRAY[0])+APP_DATA_ARRAY_LENGTH-1) ){
            p_APPLICATION_DATA_ARRAY = &APPLICATION_DATA_ARRAY[0];
            linFlagField.NORMAL_SEND = TRUE;
        } else {
            dataToWrite = *p_APPLICATION_DATA_ARRAY;
            sendData (dataToWrite);
        }
    } // NORMAL_SEND
}
}
```

**Figure 4-20: Function interrupt sioRxInterrupt**

In addition, the routine will distinguish between an identifier, which causes external Slaves to send data to the Master, and identifiers, on which the Master itself will start delivering data.

## 4.4.16 Interrupt SioRxInterrupt

Function Prototype	input Variables	output Variables	calls Function:
SioRxInterrupt	-----	-----	-----

This receive-interrupt is the routine, which offers the many possible cases. Therefore, the declaration is split into logical parts as follows:

There are different causes that generate an interrupt:

- Re-reading of data just sent
- Reception of data sent by an attached LIN-Slave
- Reception of data sent by the own Slave-routine (of the Master)

```
void SioRxInt ( void )
#pragma ghs interrupt
{
    /*=====*/
    /* FunctionName: interrupt SioRxInterrupt */
    /* IN/OUT      : -/interrupt */
    /* Description : This interrupt-function is started when the Rx-ready- */
    /*               Interrupt is received. The receiveFlag is set to ready */
    /*               so all attached functions will know when the data is */
    /*               received. */
    /*=====*/
    unsigned char i;
    unsigned char j;

    linFlagField.FLAG_RECEIVE = TRUE;
    // stores received data for error-confirmation
    receiveData = RECEIVE_BUFFER_REGISTER;
    inputCt++;
    if (inputCt <= 3){
        if (inputCt == 3){
            receiveBufferPositionCounter += ((*p_ID_LENGTH_TABLE) + 1);
            if ( receiveData == dataToWrite ){ // transmitted=received

```

**Figure 4-21: SioRxInterrupt re-reading just sent data**

The reception-routine will count the incoming amount of data by increasing the *inputCt*. In case *inputCT* is lower or equal to three (number of Header-Frame-contents), the following code will be executed.:

```
    } else {
        // a temporarily buffer is used to store the data sent by the slave
        // data + checksum are stored already
        if ( inputCt > ( (*p_ID_LENGTH_TABLE)+2)){ //the received data is the checksum
            testChecksum = 0xFF - testChecksum;

```

**Figure 4-22: SioRxInterrupt - Response Part 1**

Otherwise, the driver will step to the next piece of code to store the incoming data and calculate the checksum in flow of the ongoing reception.

A new differentiation is done, if the *inputCt* is equal to three or less than this. Upon the received data, the routine checks whether the received value exists in the table containing the identifiers or if the LIN-Master has to act upon or not.

```
{
    receiveBufferPositionCounter += ((*p_ID_LENGTH_TABLE) + 1);
    if ( receiveData == dataToWrite ){ // transmitted=received
// nothing happens, send and receive are equal, no error-routine
    linFlagField.FLAG_ERROR_OCCURENCE = FALSE;
    if (linFlagField.INT_ID_TBL_USED == TRUE){
        for (j=1;j<=INT_ID_TBL_LENGTH;j++){
            if (*p_INTERNAL_IDENTIFIER_TABLE == receiveData){
                linFlagField.NORMAL_SEND = FALSE;
                dataToWrite = *p_APPLICATION_DATA_ARRAY;
                sendData (dataToWrite);

                break;

            } else { // if = receiveData
                linFlagField.NORMAL_SEND = TRUE;
                p_INTERNAL_IDENTIFIER_TABLE++;
            } // if = receiveData
        } // for ...
        p_INTERNAL_IDENTIFIER_TABLE = &INTERNAL_IDENTIFIER_TABLE[0];
    } // if INT_ID_TBL_LENGTH
    } else { //transmitted != received
// a data-inconsistence-Error has occurred
        linFlagField.FLAG_ERROR_OCCURENCE = TRUE;
        linError.ERROR_BIT_ERROR = TRUE;
    } // transmitted != received
}
```

**Figure 4-23: SioRxInterrupt - inputCt == 3**

Figure 4-23 shows the next step in software; the routine tests the data-integrity by comparing the received value with the value originally sent. If both values did not match, two bits, one describing a general occurrence of an error, and another with detailed error-information (here: bit error) are set. These two error bits are provided to the application.

If both values match, the error flag is cleared, and the incoming data is compared with the contents of the table *INT\_ID\_TBL* for the length of the implemented identifiers until a match is found between an entry and the received data.

If the whole Master-Frame is broadcasted to the bus and received by the LIN-Master without any error, the next part, the reception of data returned by the LIN-slaves is checked.

The first test concerns the checksum (see fig. 4-24). If the internal state points to verification of the checksum-field, the received data is compared to the internally calculated value. If both match, the checksum will be stored into the temporary buffer set-up for the currently received data. Then the whole response-frame is copied into the application-memory. If the verification of the checksum fails, all flags and pointers are cleared to their initial values needed for receiving the next data frame. In that case there will be no data copied to the application-memory, which preserves the last correctly received data as valid data for the application.

```

    } else {
// a temporarily buffer is used to store the data sent by the slave
// data + checksum are stored already
    if ( inputCt > ( (*p_ID_LENGTH_TABLE)+2) ){ //the received data is the checksum
        testChecksum = 0xFF - testChecksum;
// if checksum calculated and send are consistent...
        if (receiveData == testChecksum ){ // received data matches checksum calc.
// set Read-enable-Bit to disabled
            *p_TEMP_RECEIVE_TABLE = receiveData; // store checksum in Temp-buffer
// sets Buffer to non-valid
// del. for testing only*(p_RECEIVE_TABLE + ((*p_ID_LENGTH_TABLE))) = 0x00;
            p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
// copy old temp buffer to real buffer
            for ( i = 1; i <= ((*p_ID_LENGTH_TABLE));i++){
//store Temp-data in receive-Buffer
                *p_RECEIVE_TABLE = *p_TEMP_RECEIVE_TABLE;
// increase pointer
                p_RECEIVE_TABLE++;
                p_TEMP_RECEIVE_TABLE++;
            } // for
            *p_RECEIVE_TABLE = 0x01;
        } else {
// don't copy data
// checksum doesn't match ...
// re-set Temp-Receive-table to starting-address
            p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
            p_RECEIVE_TABLE += *p_ID_LENGTH_TABLE;
// linFlagField.FLAG_HEADER_ACTIVE = FALSE;
        }
        tempChecksum = 0;
        testChecksum = 0;
        p_RECEIVE_TABLE++;
        p_ID_LENGTH_TABLE++;
        p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
        if (p_RECEIVE_TABLE >= (&RECEIVE_TABLE[0] + receiveTableLength ) ){
            p_RECEIVE_TABLE = &RECEIVE_TABLE[0];
            p_ID_LENGTH_TABLE = &ID_LENGTH_TABLE[0];
            receiveBufferPositionCounter = 0;
        } else { // p_receive_Table
        } // p_receive_Table
    } else { // inputCt > 2
// increments VAR for correct array-handling
// stores data in Temp-buffer
        *p_TEMP_RECEIVE_TABLE = receiveData;
        p_TEMP_RECEIVE_TABLE++;
//calculating the checksum
        tempChecksum = testChecksum;
        testChecksum += receiveData;
        if ( ((unsigned int)(receiveData) + tempChecksum) > MAX_CHECKSUM ){
            testChecksum++;
        } // end if > MAX_CHECKSUM
    } // inputCt > 2
} // inputCt == 3
} //endif
}

```

**Figure 4-24: Interrupt-Receive - store received Data**

The last step concerns to the reception of data from standard response-frames. The data is stored in the temporary reception-buffer, and a temporary value for the checksum is calculated.

The sum of the casted receive-data and the formerly calculated checksum are used to detect a checksum-overflow. One count is added to the result if an overflow has occurred.

At this point the receive-routine ends.

## 4.4.17 TimerCompInterrupt

Function Prototype	input Variables	output Variables	calls Function:
TimerCompInterrupt	-----	-----	-----

The Timer-Compare-Interrupt is executed when a match of the counter of the timer and its compare register is encountered. If the time for scheduling the next LIN-message is reached, the Message-scheduler is reset, and the variable *TimerUse* is set.

```

void TimerCompInt (void)
#pragma ghs interrupt
{
    /*=====*/
    /* FunctionName: interrupt TimerCompareInterrupt */
    /* IN/OUT      : -/interrupt */
    /* Description : This interrupt-function is started when the free-running*/
    /*               Timer-value compares to the pre-set Value in the com- */
    /*               pare-register. Different counter-vars will be set to  */
    /*               schedule different tasks */
    /*=====*/

    LIN_Message_Scheduler += 1;

    if ( ((unsigned int ) (LIN_Message_Scheduler)) == schedNextLINMessage ){

        LIN_Message_Scheduler = 0;
        // the routine is active to send data to slaves and confirm errors
        // may be left activated!! linFlagField.FLAG_HEADER_ACTIVE = TRUE;
        linFlagField.FLAG_TIMER_USE = 1;
        linFlagField.FLAG_SCHEDULE_DATA_SEND = TRUE;
    } // if
}

```

**Figure 4-25: Function interrupt *TimerCompareInterrupt***

The variable *TimerUse* is tested in a cyclic way by the main-application. If both fit, the function *startSchedule* will be called.

The resolution of the timer is 1 millisecond. This value is also suitable for other tasks of the application. Thus, if necessary the timer can be used by the application too.

[MEMO]



## Chapter 5 LIN-Slave Driver 78k0

### 5.1 Introduction

Using our knowledge build up in the past years developing systems and solutions for application for our customers, NEC discovered the need to develop LIN-drivers for NEC-devices without forcing customers to use third-party tools for code generation or bus-debugging.

The main requirement for the implementation of this driver was the spare usage of resources .

Following this target, a set of small stand-alone drivers was implemented, which can be used without charge by our customers inside their applications.

### 5.2 LIN-Slave Overview

The drivers for LIN-Master and LIN-Slave are both written in C. The main feature is the full access by the user. This facilitates to adapt the driver to special needs of the application by changing dedicated parameters inside the driver files.

These changes can affect the LIN-protocol definitions like the values of identifiers to be sent by the master, the amount of identifiers within the list of identifiers, timings between the single requests by the LIN-Master (s. LIN-header), and the time until the response of a LIN-Slave is awaited after the LIN-header finished.

Other items with respect to particular UART macros and their settings for registers (i.e. baud rate setup) have to be done by the customer if other devices than those mentioned in the description are used.

All these changes and some more minor changes in addition will be explained later on.

#### 5.2.1 Receiving non-standard-format SyncHBreak-Field

An external interrupt pin is used in order to establish a standard solution for NEC-devices with a level-triggered UART-macro.

The external interrupt pin is normally already in use in order to implement the power-safe-mode as described in the LIN-specification. Thus, no extra resource has to be used for the LIN-SyncHBreak-reception.

In case that no external interrupt pin is available for the detection of the SyncHBreak-field, there are two more standard implementations provided by the LIN-Slave-driver, which can be included to the software easily by setting **#define-switches** in the respective files of the LIN-driver. One option is the measurement of the SyncField after receiving the first framing error when the LIN-Slave is running at very high-speed. Then the UART needs to run with a speed that is at least 8-times faster than the LIN-bus-speed.

The other option can be used on devices, where the UART is implemented by an edge-triggered macro. In this case, the incoming data is sampled by the first falling edge after the reception of the first framing error inside the SyncHBreak-field.

### 5.3 List of used files

The files generated for the LIN-Slave are the following:

- m\_slave.c
- m\_slave.h
- LIN\_s.h
- hardware\_s.h
- UART\_s.h

The header-files contain the specific user-defined settings, the file **m\_slave\_s.c** contains all functions used to run the procedures needed by a LIN-Slave in a LIN-bussystem. The application itself needs one more file in addition:

- main\_LINSlave\_s.c

The header-files for the particular device (in78000.h, dfabcd.h,...) are application and hardware-specific and no subject of description in this document.

In a first step, the header-files with the applicable settings for the configuration are described followed by the explanation of the code of the driver. Finally, an application will be defined including comments on functions used.

### 5.3.1 Hardware.h

The file hardware.h carries all hardware-related settings, which are not included in the LIN.h and the UART.h-section. The following figure shows the outline of the hardware.h-file:

```
// user-defined settings
#define TIMER_START                0x31
#define TIMER_STOP                 0x00
// standard-settings
#define TIMER_COUNT_REGISTER       TM20
#define TIMER_CAPTURE_COMPARE_REGISTER CR20
#define TIMER_MODE_CONTROL_REGISTER TMC20
#define TIMER_OUTPUT_CONTROL_REGISTER TOC20
#define PRESCALAR_MODE_REGISTER    PRM20
#define PORT_MODE_REGISTER         PM2
#define CAPTURE_COMPARE_CONTROL_REGISTER CRC20
#define TIMER_OUTPUT               0x01
#define PRESCALAR_FX_2_8_MHZ       0x30
#define TIMER_BASE                  0x04D6
// settings for External Interrupt-Usage
#define ERROR_INTERRUPT_FLAG       PIF0
#define MASK_LOW                   MK0
#define MASK_HIGH                  MK1
#define MASK_EXT_INT               PMK0
#define EDGE_EXT_INT               INTM0
#define MASK_ALL                   0xFF
#define USE_INT                    0
#define EXT_INT_RISING_EDGE        0x54
// definition of related stuff
#define FALSE                      0
#define HIGH                       1
#define LOW                        0
// Definitions for Interrupt-Vectors
#define INTTM_VECT                  INTTM20_vect
#define INTRX_VECT                  INTRSR20_vect
#define INTTX_VECT                  INTST20_vect
#define EXT_INT_VECT                INTP0_vect
```

**Figure 5-1: Header-file Hardware.h**

The scope of this file is the definition of the resources for the timer and its interrupts. The registers of the timer are defined by macros with standard names that are used within the LIN-driver. Additionally, some initial values for these registers are defined.

All these settings have to be re-specified by the customer if another device is used or if the timer has to be adapted to fit the needs of the application.

Please be aware of the fact that the timer generates the tick, which is used by both, LIN-driver and application. **Changing the settings for the timer-base may have effect in that way that the driver will not work correctly anymore!**

The timer is setup to generate one tick every 256 µsec. This will not put too much load to the device but it provides a fairly resolution.

Special settings are made to provide the timer with the ability to use an external interrupt as a trigger when detecting the SynchBreak-field.

Statement	Comment	Definition	To be adjusted by customer depending on the device used
TIMER_START	Definition to write start-value into Timer-Register	0x31	
TIMER_STOP	Definition to write stop-value into Timer-Register	0x00	
TIMER_COUNT_REGISTER	Macro-Definition for the TM-Register	TM20	
TIMER_CAPTURE_COMPARE_REGISTER	Macro-Definition for the CR-Register	CR20	
TIMER_MODE_CONTROL_REGISTER	Macro-Definition for the TMC-Register	TMC20	
TIMER_OUTPUT_CONTROL_REGISTER	Macro-Definition for the TOC-Register	TOC20	
PRESCALAR_MODE_REGISTER	Macro-Definition for the PRM-Register	PRM20	
PORT_MODE_REGISTER	Macro-Definition for the PM-Register	PM2	
CAPTURE_COMPARE_CONTROL_REGISTER	Macro-Definition for the CRC-Register	CRC20	
TIMER_OUTPUT	Definition to set timer to output (not necessary)	0x01	
PRESCALAR_FX_2_8_MHZ	Prescaler-Value for Timer	0x30	
TIMER_BASE	Timer-base for 1msec-tic	0x04D6	
ERROR_INTERRUPT_FLAG	Macro-Definition for the Error-Interrupt	PIF0	
MASK_LOW	Macro-Definition for the Mask-register	MK0	
MASK_HIGH	Macro-Definition for the Mask-register	MK1	
MASK_EXT_INT	Macro-Definition to mask the EXT-INT	PMK0	
EDGE_EXT_INT	Macro-Def. to set the EXT-INT to edge-triggered	INTM0	
MASK_ALL	Definition to set Mask-register to mask all	0xFF	
USE_INT	Definition to set EXT-INT to be used	0	
EXT_INT_RISING_EDGE	Definition to set the EXT-INT to rising edge	0x54	
INTTM_VECT	Macro re-Definition for Timer-INT-Vector	INTTM20_vect	
INTRX_VECT	Macro re-Definition for Receive-INT-Vector	INTSR20_vect	
INTTX_VECT	Macro re-Definition for Transmit-INT-Vector	INTST20_vect	
EXT_INT_VECT	Macro re-Definition for EXT-INT-Vector	INTP0_vect	

### 5.3.2 UART.h

The file UART.h defines all settings that are needed to use the 'logical device' UART. The definitions necessary to support operations that are specific to LIN are not part of this file. The following figure shows a short overview of the used definitions:

```
// user-defined settings

#define PM_Rx      PM2.2
#define PM_Tx      PM2.1
#define P_Rx       P2.2
#define P_Tx       P2.1
#define PM_EXT_INT PM2.3
#define P_EXT_INT  P2.3
// sets all values in ASIM0 to transmit and receive data
#define START_UART  UART_MODE_REGISTER = 0xC8
// sets all values to stop transmit/receive
#define STOP_UART   UART_MODE_REGISTER = 0x08
// switch whether CSIM-Register is present or not
#define CSIM_REGISTER_PRESENT 0
#define CSIM_VALUE 0x00
// standard-settings
#define BAUD_RATE_CONTROL_REGISTER BRGC20
#define UART_MODE_REGISTER          ASIM20
#define UART_ERROR_REGISTER         ASIS20
#define TRANSMIT_SHIFT_REGISTER     TXS20
#define RECEIVE_BUFFER_REGISTER     RXB20
// Parity-error in ASIS-register
#define PARITY_ERROR                 0x04
// Framing-error in ASIS-register
#define FRAMING_ERROR                0x02
// Overrun-error in ASIS-register
#define OVERRUN_ERROR                0x01

#ifdef CSIM_REGISTER_PRESENT
#define SERIAL_OPERATION_MODE_REGISTER CSIM0
#endif
```

**Figure 5-2: Header-file UART.h**

At first, the addresses for the port and the port-mode (registers) are defined for setting up the reception and the transmission of data.

The macros for the start and stop of the UART are defined for clear and fast access to the UART. The definitions for the registers of the UART are done to give full portability to all NEC-devices. Further some macros for handling of UART-errors are defined. All these settings have to be adjusted by the customer if any other device like the one provided with the example (78F9116) is used.

The table below lists the items that need to be checked.

Statement	Comment	Definition	To be adjusted by customer depending on the device used
PM_Rx	Adress-Definition for PM-Rx-register	PM2.2	
PM_Tx	Adress-Definition for PM-Tx-register	PM2.1	
P_Rx	Adress-Definition for Receive-Port-Pin	P2.2	
P_Tx	Adress-Definition for Transmit-Port-Pin	P2.1	
PM_EXT_INT	Adress-Definition for Ext-INT-Port	PM2.3	
P_EXT_INT	Adress-Definition for Ext-INT-Port	P2.3	
START_UART	Macro-Definition to Start the UART	0xC8	
STOP_UART	Macro-Definition to Stop the UART	0x08	
CSIM_REGISTER_PRESENT	Switch if the CSIM-register is present or not	0	
CSIM_VALUE	Value of the possible existant CSIM-register	0x00	
BAUD_RATE_CONTROL_REGISTER	Macro-Definition for the BRGC-register	BRGC20	
UART_MODE_REGISTER	Macro-Definition for the ASIM-register	ASIM20	
UART_ERROR_REGISTER	Macro-Definition for the ASIS-register	ASIS20	
TRANSMIT_SHIFT_REGISTER	Macro-Definition for the TxS-register	TXS20	
RECEIVE_BUFFER_REGISTER	Macro-Definition for the Rx-register	RXB20	
PARITY_ERROR	Definition for Parity-Error	0x04	
FRAMING_ERROR	Definition for Framing-Error	0x02	
OVERRUN_ERROR	Definition for Overrun-Error	0x01	
SERIAL_OPERATION_MODE_REGISTER	Macro-Definition for CSIM-register	CSIM0	

### 5.3.3 LIN.h

The LIN.h header-file contains the settings and definitions for all LIN-related settings. Some of these definitions are subject to be changed by the customer for adaptation to the application. The figure below lists the file LIN.h:

```
// LIN-specific setting

//setting for double-speed on UART for SynchBreak-Field
unsigned char BAUDRATE_SYNC_BREAK = 0x10;
// setting for normal bus speed in application, e.g. 19.200
unsigned char BAUDRATE_NORMAL_SPEED = 0x30;

// ID send data to Master on req.
unsigned char IDENTIFIER = 0x6F;
// ID rec. data from Master on req.
unsigned char IDENTIFIER2 = 0x2E;

unsigned char DATA_TABLE_LENGTH= 0x04;
// standard-settings
#define VERSION = 1.0_NEC // just a simple VERSION-control-variable
#define MASTER = 0 // set this to one if Master is used
#define SYNC_BREAK 0x00
#define REQUEST_SLEEP 0x80
unsigned char *p_data_table;
unsigned char *p_read_active_table;
unsigned char *p_data_valid_table;
unsigned char *p_IDENTIFIER_table;
unsigned char *p_data_table_read_position;
unsigned char *p_write_app_table;
unsigned char *p_valid_app_table;
unsigned char *p_read_allowed_app_table;
unsigned int *p_id_delay_table;
unsigned char *p_ID_LENGTH_TABLE;
unsigned char ID_LENGTH_TABLE[1] = {2}; // Data-Bytes
unsigned char TEMP_RECEIVE_TABLE[3];
unsigned char *p_TEMP_RECEIVE_TABLE;
unsigned char data_table[6];
```

**Figure 5-3: Header-file LIN.h - Definitions**

The baudrate for the Synch-Break and the normal baudrate are defined in LIN.h. Additionally, the identifiers, upon which the LIN-Slave has to react with the correspondent ID-length are given here.

The definitions *MASTER*, *SYNC\_BREAK* and *REQUEST\_SLEEP* are used inside the driver for internal LIN-communication. Changing these definition may result in malfunction of the driver..

Statement	Comment	Definition	To be adjusted by customer depending on the device used
BAUDRATE_SYNCHBREAK		0x10	
BAUDRATE_NORMALSPEED		0x30	
IDENTIFIER		0x6F	
IDENTIFIER2		0x2E	
DATATABLELENGTH		0x04	
VERSION		1.0_NEC	§
MASTER		0x01	§
SYNC_BREAK		0x00	§
REQUEST_SLEEP		0x80	§

### 5.3.4 M\_Slave.h

This header-file contains statements for the LIN-Slave, which are not device-specific or hardware-related.

```
#define TRUE 1
#define FALSE 0
enum ON_OFF {OFF = 0, ON = 1};
enum EN_DIS {DISABLE = 0, ENABLE = 1};

enum STATE_MODE {STATE_MODE_INIT =0,
                  STATE_MODE_NORMAL_SEND=1,
                  STATE_MODE_RE_INIT =2};

enum IN_FRAME_POSITION {IN_FRAME_POS_WAIT_FOR_SYNC_BREAK=1,
                        IN_FRAME_POS_WAIT_FOR_SYNC_FIELD =2,
                        IN_FRAME_POS_WAIT_FOR_IDENTIFIER =3};

// bit-field for LIN-status-flags
extern bit FLAG_LIN_ACTIVE ;
extern bit FLAG_FIRST_TIME_SCHED ;
extern bit FLAG_TIMER_USE ;
extern bit FLAG_TIMER_RUN ;
extern bit FLAG_SCHEDULE_DATA_SEND ;
extern bit FLAG_DATA_DELIVERED ;
extern bit FLAG_RECEIVE_DATA ;
extern bit FLAG_ERROR_OCCURED ;
extern bit FLAG_RECEIVE_MASTER_DATA ;

// bit-field for LIN-Error-Flags
extern bit ERROR_NO_ERROR ;
extern bit ERROR_BIT_ERROR ;
extern bit ERROR_CHECKSUM ;
extern bit ERROR_SLAVE_NOT_RESPONDING;
extern bit ERROR_IDENTIFIER_PARITY ;
extern bit ERROR_INCONSISTENT_SYNC ;
extern bit ERROR_FIRST_TIME ;
extern bit ERROR_RECEIVE ;

// additional values
#define RAMP_COMPARE_VALUE 0x0F
#define MAX_CHECKSUM 0x00FF
#define SYNC_FIELD 0x55
```

**Figure 5-4: Header-file Slave.h**

There are several 'enum' statements given, which define standard values for the internal state-machine or frequently used values like ON/OFF. Additionally, bit-fields for operational status or error status of the LIN-Slave are declared here. MAX\_CHECKSUM and RAMP\_COMPARE\_VALUE are used for internal operations of the driver.

## 5.4 Functions of the LIN-Slave driver

The file m\_Slave.c contains all routines to engage the LIN-Slave driver on a device. The driver consists of several functions. Most of them are called internally for initialisation, handling interrupt requests, and reception or transmission of data.

The application interface of the driver (API) is realized by the functions *startLIN*, *scheduleSending* and *calculateChecksum*, which have to be called from the user program.

5.4.1 *startLIN*

Function Prototype	input Variables	output Variables	calls Function:
startLin	-----	-----;	initHardware

```

void startLin (void) {
/*=====*/
/* FunctionName: startLin */
/* IN/OUT      : -/- */
/* Description : The application will call this routine to enable all */
/*               LIN-related hardware and driver-settings. */
/*               After calling this function, the scheduleMessages */
/*               routine has to be called to start cyclic send of messages.*/
/*=====*/

FLAG_FIRST_TIME_SCHED = TRUE;
FLAG_DATA_DELIVERED = TRUE;
FLAG_TIMER_RUN = FALSE;
FLAG_LIN_ACTIVE = TRUE;
// starts init of all LIN- and Hardware-related items
initHardware ();
}

```

**Figure 5-5: Function startLin**

The function *startLIN()* initializes some flags required inside the driver for proper approach before the function *initHardware()* is called. This function will reset and initialize all necessary resources for the LIN-driver.



5.4.2 *stopLin*

Function Prototype	input Variables	output Variables	calls Function:
stopLin	_stopMode	-----	-----

The function *stopLin()* will be called to stop all working on LIN-specific functions. Depending on the parameter *\_stopMode* the desired power-save-mode will be entered. This is done inside the function *sendBusToStop()*.

```
void stopLin (unsigned int _stopMode) {
/*=====*/
/* FunctionName: stopLin                               */
/* IN/OUT      : -/-                                   */
/* Description : The application will call this routine to end all      */
/*               services regarding LIN.                               */
/*               The scheduling is stopped and all values are re-set into */
/*               init-values                                           */
/*=====*/

linFlagField.FLAG_LIN_ACTIVE = FALSE;
}
```

**Figure 5-6: Function *stopLin***

The flag *FLAG\_LIN\_ACTIVE* provides the current status to the driver to the application.

### 5.4.3 sendBusToStop

When the system shall enter a power saving mode, this function is called to stop all operations of LIN-driver. The device can be set into the HALT-mode or STOP-mode via the parameter `_mode`. These modes provide different level of power consumption. In HALT-mode the CPU stops but the peripherals are still provided with a clock. In STOP-mode the peripherals are stopped, too,

```
void sendBusToStop (_mode) {
/*=====*/
/* FunctionName: sendBusToStop                                     */
/* IN/OUT      : -/-                                             */
/* Description : This func is called when Bus has to be set into Stop- */
/*              Mode. All LIN-related actions are stopped, the init-values are */
/*              re-written and the device is set into Stop-mode         */
/*=====*/
switch (_mode){

case 0:
    _HALT ();
    break;

case 1:
    _STOP();
    break;

default:
    break;
}
}
```

**Figure 5-7: Function sendBusToStop**

5.4.4 *initHardware*

Function Prototype	input Variables	output Variables	calls Function:
initHardware	-----	-----:	initUART, initTimer, init IDLengthTable

*InitHardware()* calls several routines that initializes all H/W-resources used by the LIN-driver.

```

void initHardware (void) {
/*=====*/
/* FunctionName: initHardware                                     */
/* IN/OUT      : -/-                                           */
/* Description : This function calls all other related init- and set- */
/*               functions for UART, timer, ScheduleTable.      */
/*=====*/
PMK0 = TRUE;
// sets INTP0 to rising edge
EDGE_EXT_INT = EXT_INT_RISING_EDGE;
PIF0 = FALSE;
// sets the UART-registers to init-values
initUART ();
// standard bus-speed
setUARTOnNormalSpeed (BAUDRATE_NORMAL_SPEED);
// sets the Timer-registers to init-values
initTimer ();
// sets the sched-table to the first valid input
initDataTable ();
ERROR_FIRST_TIME = TRUE;
if (FLAG_FIRST_TIME_SCHED){
    FLAG_RECEIVE_MASTER_DATA = FALSE;
// scheduling of SyncHBreak-field is initiated
inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
*p_data_table_read_position = 0;
FLAG_ERROR_OCCURED = FALSE;
ERROR_FIRST_TIME = TRUE;
// running 1st time data-count
FLAG_TIMER_USE = TRUE;
stateMode = STATE_MODE_INIT;
FLAG_SCHEDULE_DATA_SEND = FALSE;
// setting for 1st schedule...
FLAG_DATA_DELIVERED = TRUE;
ERROR_RECEIVE = FALSE;
// if Timer is not running, start Timer!
if ( !FLAG_TIMER_RUN ) {
    startTimer ();
} // end of if FLAG_TIMER_RUN

    FLAG_FIRST_TIME_SCHED = FALSE;
} else {
}
}
}

```

**Figure 5-8: Function *initHardware***

The first action is to configure the external interrupt with standard settings in order to be able to receive data in case of a SyncBreak. Then, different functions are called, which perform the initialization for different H/W-macros and the (software) state machine of the LIN-driver.

5.4.5 *initUART*

Function Prototype	input Variables	output Variables	calls Function:
<i>initUART</i>	-----	-----;	-----

The function *initUART()* is called by the function *initHardware()* and sets the UART in the state to send and receive data.

```

void initUART (void) {
/*=====*/
/* FunctionName: init UART                                     */
/* IN/OUT      : -/-                                          */
/* Description : This routine inits all values recent for the UART- */
/*              macro.                                         */
/*              It is called by the initHardware-function      */
/*              */
/*=====*/

    STOP_UART;
    BRGC20 = BAUDRATE_NORMAL_SPEED;
    START_UART;
    // set Port-Mode to 1 for receive
    PM_Rx = HIGH;
    // set Port-Mode to 0 for transmit
    PM_Tx = LOW;
    // set Port to 1 for input
    P_Rx = LOW;
    // set Port to 0 for output
    P_Tx = HIGH;

    #if defined ( useExtIntForSHBInSlave )
    // enables ext-Int for sharing with Rx-Port
    PM_EXT_INT = HIGH;
    #endif
}

```

**Figure 5-9: Function *initUART***

Therefore, the baudrate is set to a standard LIN-speed (here, 19.2 kbaud) and the registers for port and port-mode are configured to receive and transmit. Finally the UART is enable andf the communication can start. If the reception of SyncHBreak via external interrupt is used, the corresponding port is set to 'HIGH'.

5.4.6 *initTimer*

Function Prototype	input Variables	output Variables	calls Function:
initTimer	-----	-----;	stopTimer

The routine *initTimer()* sets the timer to a free-running mode. The base of the timer needs be set to an equivalent of ~250  $\mu$ sec. This provides a reasonable good resolution to generate a timer-tick for application and LIN-driver.

```
void initTimer (void) {
/*=====*/
/* FunctionName: initTimer                               */
/* IN/OUT      : -/-                                     */
/* Description : This routine inits all values recent for the UART- */
/*              macro. */
/*              It is called by the initHardware-function          */
/*              */
/*=====*/

    stopTimer ();
    // sets timer-base to 250  $\mu$ sec for all application-needs
    TIMER_CAPTURE_COMPARE_REGISTER = TIMER_BASE;
}
```

**Figure 5-10: Function *initTimer***

## 5.4.7 initDataTable

Function Prototype	input Variables	output Variables	calls Function:
initDataTable	-----	-----:	-----

The function *initDataTable()* resets pointers and arrays to their initial settings.

```

void initDataTable (void) {
    /*=====*/
    /* FunctionName: initScheduleTable */
    /* IN/OUT      : -/- */
    /* Description : This function is called by the initHardware-routine */
    /*               and will init the Schedule-Table by choosing the right */
    /*               table and setting the pointer to the valid first */
    /*               input. */
    /*=====*/
    // sets Pointer to start of array
    p_data_table = &data_table[0];
    p_read_active_table = &(data_table[0]) + DATA_TABLE_LENGTH + 0x02 ;
    // sets Pointer to read-active-flag
    p_data_valid_table = &(data_table[0]) + DATA_TABLE_LENGTH + 0x03 ;
    // sets Pointer to valid-flag
    p_data_table_read_position = &data_table[0];
    p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
    p_ID_LENGTH_TABLE = &ID_LENGTH_TABLE[0];
}

```

**Figure 5-11: Function *initDataTable***

## 5.4.8 initDataTableLength

Function Prototype	input Variables	output Variables	calls Function:
initDataTableLength	-----	-----;	-----

The function *initDataTableLength()* initializes the table containing the identifiers and their length. The length is extracted from the identifier ANDed with 0x30.

```
void initDataTableLength ( void ) {
/*=====*/
/* FunctionName: initTableLength                                     */
/* IN/OUT      : -/-                                              */
/* Description : This function is called by the initHardware-routine */
/*               and will init the Length of the used data-table by */
/*               calculating the length of used identifier          */
/*=====*/

    unsigned int length_input = 0;
    DATA_TABLE_LENGTH = 0;
    switch( ((*p_IDENTIFIER_table) & (0x30))){

    case 0x00:
    case 0x10:
        length_input = 2;
        break;

    case 0x20:
        length_input = 4;
        break;

    case 0x30:
        length_input = 8;
        break;

    default:
        length_input = 0;
        break;
    } // end of switch
    DATA_TABLE_LENGTH += length_input;
} // end of void
```

**Figure 5-12: Function *initIDLengthTable***

In a last action the pointers to the ID-table and to the Receive-table are put to their start values.

5.4.9 *calculateChecksum*

Function Prototype	input Variables	output Variables	calls Function:
calculateChecksum	-----	checksum	ASM: addCarry

To assign a correct checksum to a valid data-array, the function *calculateChecksum()* is called. On return of this function the 'data-valid' flag is set to TRUE.

```

unsigned char calculateChecksum ( void ){
/*=====*/
/* FunctionName: calculateChecksum          */
/* IN/OUT      : -/unsigend char            */
/* Description : This is the function called by the driver which          */
/*              will calculate the checksum regarding to the              */
/*              actual data set by the attached hardware                  */
/*=====*/

unsigned char *temp_p_data_table = p_data_table;
unsigned char checksum_old;
static unsigned int j; // init to 1
unsigned int max = 0x00FF;

checksum = 0;

for ( j=1;j<= DATA_TABLE_LENGTH;j++){
    checksum_old = checksum;
    checksum += *(temp_p_data_table);
//typecast to integer, so value can be 0xFFFF...
    if ( ((unsigned int)(*temp_p_data_table) + checksum_old ) > max ){
        checksum++;
    }
    temp_p_data_table++;
}
checksum = 0xFF -checksum;
return ( checksum );
}

```

**Figure 5-13: Function *calculateChecksum***

The calculation for the checksum adds all values of the valid data-array. After each step of the calculation the intermediate result is compared to the pre-defined *CONST-value 0x00FF*. In case the current sum exceeds this value, the unsigned-char of the current sum is incremented by 1. When all entries of the array have been processed, the checksum is built out of the difference from 0xFF and the accumulated sum over all cycles. An 8-bit value will be returned to the calling routine.



5.4.10 *sendData*

Function Prototype	input Variables	output Variables	calls Function:
sendData	_dataTableEntry	-----	-----

The function *sendData()* will send the data stored in the Transmit-Shift-register upon the call by the application.

```
void sendData (unsigned char _dataTableEntry){
    /*=====*/
    /* FunctionName: sendIdentifier */
    /* IN/OUT      : _idTableEntry/- */
    /* Description : The sendData-Function is used to generate the */
    /*               data that will be the actual data. This data */
    /*               is put into the out-register and the function will */
    /*               wait until the data is sent */
    /*=====*/

    // sets the TXS-register to the actual data-value
    TRANSMIT_SHIFT_REGISTER = _dataTableEntry;
}
```

**Figure 5-14: Routine *sendData***

5.4.11 *startTimer*

Function Prototype	input Variables	output Variables	calls Function:
startTimer	-----	-----:	-----

The routine *startTimer()* is used to start the timer and to set the corresponding flag to signal that the timer is active.

```

void startTimer (void) {
    /*=====*/
    /* FunctionName: startTimer                               */
    /* IN/OUT      : -/-                                       */
    /* Description : This function is started when the scheduleMessages- */
    /*               routine is called out of the application-main-file. */
    /*               All init-settings fit the actual hardware at the mo- */
    /*               ment and the timer can run with 1msec turnaround.    */
    /*=====*/

    FLAG_TIMER_RUN = TRUE;
    TIMER_MODE_CONTROL_REGISTER = TIMER_START;
}

```

**Figure 5-15: Function *startTimer***

## 5.4.12 stopTimer

Function Prototype	input Variables	output Variables	calls Function:
stopTimer	-----	-----;	-----

If the timer shall be stopped by application or driver itself, this function needs to be called. The function sets the flag reporting the timer activity to the application to FALSE. After this, the value forcing the timer to stop is written into the Timer\_Mode\_Control\_Register. The timer stops immediately.

```
void stopTimer (void) {
/*=====*/
/* FunctionName: stopTimer */
/* IN/OUT : -/ - */
/* Description : The stopTimer function is called to halt all Timer-de-*/
/* pendend actions while settings to the timer-control- */
/* registers will be done. */
/* After that, the timer has to be re-started. */
/*=====*/
FLAG_TIMER_RUN = FALSE;
TIMER_MODE_CONTROL_REGISTER = TIMER_STOP;
}
```

**Figure 5-16: Function stopTimer**

## 5.4.13 setUARTOnNormalSpeed

Function Prototype	input Variables	output Variables	calls Function:
setUARTOnNormalSpeed	-----	-----;	-----

This function is used to set the UART back on the actual speed of the LIN-bus when a SyncBreak was received successfully for which the baudrate was changed previously.

```
void setUARTOnNormalSpeed (unsigned char _actualBaudRate) {
    /*=====*/
    /* FunctionName: setUARTOnNormalSpeed */
    /* IN/OUT      : -/_actualBaudRate */
    /* Description : This function is used to init the UART to the bus- */
    /*               speed used by the LIN-system. */
    /*               Later-on, the original bus-speed will be re-stored by */
    /*               calling this function */
    /*=====*/
    STOP_UART;
    BAUD_RATE_CONTROL_REGISTER = _actualBaudRate; //LINSpeed;
    START_UART;
}
```

**Figure 5-17: Function setUARTOnNormalSpeed**



Be sure to follow the above shown syntax to keep the system running after the baudrate has changed. In recent cases - using different UART-macros like the one implemented in the 78F9850 - the above shown approach may be different in some way! Please, refer to the corresponding user manual for more information.

## 5.4.14 setUARTForSyncBreak

Function Prototype	input Variables	Output Variables	calls Function...
<i>setUARTForSyncBreak</i>	Definition to write start-value into Timer-Register	0x31	-----

The function *setUARTForSyncBreak* is used to set the UART to the baudrate fitting best for the reception of an incoming SyncBreak-Field.

```
void setUARTForSyncBreak (void) {
    /*=====*/
    /* FunctionName: setUARTForSyncBreak */
    /* IN/OUT      : -/- */
    /* Description : This function will change the baudrate of the LIN- */
    /*               Master so the SyncBreak-field is sent with half-speed*/
    /*               of the attached LIN-bus, so the restrictions for the */
    /*               Break-field can be hit. */
    /*=====*/
    STOP_UART;
    BAUD_RATE_CONTROL_REGISTER = BAUDRATE_SYNC_BREAK; // set to fastest speed
    START_UART;
}
```

**Figure 5-18: Function *setUARTForSyncBreak***



Depending on the macro used inside the device, the UART is stopped, the baudrate is changed to the needed settings, and the UART is re-enabled again.

Please be aware that the approach may be different, depending on the UART-macro used in the particular device. This needs as well to be checked for the routine *setUARTOnNormalSpeed()*. Please, refer to the corresponding user manual for more information.

5.4.15 *scheduleSending*

Function Prototype	input Variables	output Variables	calls Function:
<code>scheduleSending</code>	-----	-----	<code>sendData</code>

The function *startSending()* is the main procedure to be called by the application. It starts an internal state-machine which calls the functions delivering the data to be sent to the requesting LIN-Master as response to his LIN-Frame. Depending on the internal state of the LIN-driver, the previous state is taken into account to ensure the correct data-handling.

```

void scheduleSending ( void ) {
/*=====*/
/* FunctionName: startScheduling                                     */
/* IN/OUT      : -/-                                              */
/* Description : This function is the routine called by the applica- */
/*               tion. At the first beginning, a SynchBreak is send  */
/*               to init the bus and the related LINdriver settings  */
/*=====*/

unsigned char tmp_dtTbLength;
tmp_dtTbLength = DATA_TABLE_LENGTH;
if ( !FLAG_DATA_DELIVERED ){
} else {
    switch ( stateMode ){

        case STATE_MODE_INIT:
            if ( !(*p_valid_app_table) ){
            } else {
                dataToWrite = *p_data_table_read_position;
                FLAG_DATA_DELIVERED = FALSE;
                p_data_table_read_position++;
                sendData ( dataToWrite );
                stateMode= STATE_MODE_NORMAL_SEND;
            }
            break;

        case STATE_MODE_NORMAL_SEND:
            if ((p_data_table_read_position + 1) - &p_data_table[0] ) <= (tmp_dtTbLength + 0x01)){
                FLAG_DATA_DELIVERED = FALSE;
                dataToWrite = *p_data_table_read_position; // inc by if-construct
                sendData (dataToWrite);
                p_data_table_read_position++;
            } else {
                stateMode = STATE_MODE_RE_INIT;
            }
            break;

        case STATE_MODE_RE_INIT:
            p_data_table_read_position = &data_table[0];
            FLAG_DATA_DELIVERED = TRUE;
            FLAG_SCHEDULE_DATA_SEND = FALSE;
            FLAG_ERROR_OCCURED = FALSE;
            stateMode = STATE_MODE_INIT;
            inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNCH_BREAK;
            ERROR_FIRST_TIME = TRUE;
            initUART ();
            break;

        default:
            break;
    }
}
}

```

**Figure 5-19: Function *scheduleSending***

- If the state of the LIN-driver (*stateMode*) equals `STATE_MODE_INIT` and the data inside the appli-

cation table is not valid (indicated by the byte `VALID_APP_TABLE`), no init will be done. When the content of the table is valid, the data to be written to the master is selected, some flags are set, and the data is sent. The function returns after the state was set to `STATE_MODE_NORMAL_SEND`.

- When the function is called the next time, the state `STATE_MODE_NORMAL_SEND` is choosed as parameter. Then, the standard data to be sent including the checksum, which is calculated by an seperate function, is delivered to the LIN-Master. Each time the function is called with this parameter, the pointer accessing the array containing the Slave's data, is incremented. When the checksum is reached, the state `STATE_MODE_RE_INIT` is assigned to the variable *stateMode*.
- In the state `STATE_MODE_RE_INIT` all flags used while delivering the data are forced to their original status, the pointer to the data-array is set to its start address, and potentially occurred errors are cleared.  
Then the variable *stateMode* is set again to `STATE_MODE_INIT` and the counter representing the progress for receiving the standard LIN-data is set to wait for the SynchBreak-field.

5.4.16 Interrupt *SioTxInterrupt*

Function Prototype	input Variables	output Variables	calls Function:
SioTxInterrupt	-----	-----	-----

This function is called when the transmission of one byte is ended successfully. Dependant on the next entry in the scheduler, the value for the timer *schedNextLinMessage* will be set to a new value in the routine for the scheduler.

If the scheduler is in the state to send the data, the current data from the application table is sent. After the last byte of data is sent, the checksum follows.

```

interrupt [INTTX_VECT] void SioTxInterrupt( void ) {
/*=====*/
/* FunctionName: interrupt SioTxInterrupt */
/* IN/OUT      : -/interrupt */
/* Description : This interrupt-function is started when the Tx-ready- */
/*               Interrupt is received. The wait-Flag is set to ready */
/*               so all attached functions will know when the data is */
/*               sent. */
/*=====*/

FLAG_DATA_DELIVERED = TRUE;
}

```

**Figure 5-20: Function interrupt *sioRxInterrupt***



5.4.17 Interrupt *SioRxInterrupt*

Function Prototype	input Variables	output Variables	calls Function:
SioRxInterrupt	-----	-----	a) ( Interrupt ExternalInterrupt )

**(a) Using External Interrupt Pin**

The external interrupt can be used by the wakeup from Stop-Mode. The pin carries a shared function while the device is in normal-mode. It is the reception of the SynchBreak-Field.

After the first Framing-Error occurred, all interrupts are disabled. Only the external interrupt, which has to be shared with the receive-pin RxD, remains active.

The trigger of the interrupt is configured to react on the next appearing rising edge. This next edge is the beginning of the Stop-Bit sent by the LIN-Master. The following figure shows how the implementation is done in detail:

```

interrupt [INTRX_VECT] void SioRxInterrupt ( void ) {
    /*=====*/
    /* FunctionName: interrupt SioRxInterrupt */
    /* IN/OUT      : -/interrupt */
    /* Description : This interrupt-function is started when the Rx-ready- */
    /*               Interrupt is received. The receiveFlag is set to ready */
    /*               so all attached functions will know when the data is */
    /*               received. */
    /*=====*/

    // LOCAL VARs
    unsigned char temp_dtTbLength;
    unsigned char errorType;
    unsigned char errorReceive;

    #if defined ( useExtIntForSHBInSlave )

        temp_dtTbLength = DATA_TABLE_LENGTH;

        errorType = ASIS20;
        errorReceive = RXB20;
        sReceiveData = RECEIVE_BUFFER_REGISTER; // stores received data for error-
                                                //confirmation

        // if Framing-Error has occurred
        if ( ( (errorType & 0x02) == FRAMING_ERROR ) ) {

            if (ERROR_FIRST_TIME == TRUE ){

                // store Mask-register-settings
                ERROR_RECEIVE = TRUE;
                save_MK0 = MK0;
                save_MK1 = MK1;
                // set new Mask for use of external Interrupt
                ERROR_INTERRUPT_FLAG = FALSE; // deletes a possible occurred interrupt-flag
                MASK_LOW = MASK_ALL; // enable external Interrupt only
                MASK_HIGH = MASK_ALL; // disables all related interrupts
                MASK_EXT_INT = USE_INT; // sets the ext-int INTP0 to enabled
            } else {
            }
        }
        // correct data was received
    }
}

```

**Figure 5-21: Reception of a Framing-Error**

After the registers for error and reception information are read-out, the error status is evaluated. If a Framing-Error occurred and the LIN-Slave is waiting for the SynchBreak-field, all interrupts are disabled, the mask-flags are stored, and the external interrupt is activated. Then, the UART is stopped.

The next interrupt received is the external interrupt attached to Rx.D. The UART operates with normal settings. Thus, the driver will receive the next incoming data in as a standard UART-frame. The incoming data can be divided into two cases:

- **IN\_FRAME\_POSITION\_WAIT\_FOR\_SYNC\_FIELD:**  
The received data is the Sync-Field, the respective flags are set to their initial values and the variable *stateMode* is set to *WAIT\_FOR\_IDENTIFIER*.
- **IN\_FRAME\_POSITION\_WAIT\_FOR\_IDENTIFIER**  
The received data is compared to the internally stored identifier on which the LIN-Slave has to respond. If the result is TRUE, the transmission of application data is started by setting the flag *FLAG\_SCHEDULE\_DATA\_SEND*, and *stateMode* is assigned *WAIT\_FOR\_SYNC\_BREAK*.

```

    } else {
if (FLAG_RECEIVE_MASTER_DATA){
    { // this part is called when the slave-response is awaited
//the received data is the checksum
    if ( inputCt >= ( (*p_ID_LENGTH_TABLE))){
        testChecksum = 0xFF - testChecksum;
        inputCt = 0;
// enables RETURN to normal proc.
        FLAG_RECEIVE_MASTER_DATA = FALSE;
// if checksum calculated and send are consistent...
// received data matches checksum calc.
        if (sReceiveData == testChecksum ){
// store checksum in Temp-buffer
            *p_TEMP_RECEIVE_TABLE = sReceiveData;

            p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
        } else {
// don't copy data
// checksum doesn't match ...
// re-set Temp-Receive-table to starting-address
            p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
        }
        tempChecksum = 0;
        testChecksum = 0;

        p_TEMP_RECEIVE_TABLE = &TEMP_RECEIVE_TABLE[0];
    } else {
// increments VAR for correct array-handling
// stores data in Temp-buffer
        inputCt++;
        *p_TEMP_RECEIVE_TABLE = sReceiveData;
        p_TEMP_RECEIVE_TABLE++;
//calculating the checksum
        tempChecksum = testChecksum;
        testChecksum += sReceiveData;
        if ( ((unsigned int)(sReceiveData) + tempChecksum) > MAX_CHECKSUM ){
            testChecksum++;
        } // end if > MAX_CHECKSUM
    }
}
    } else {

```

**Figure 5-22: Schedule-position WAIT\_FOR\_SYNC\_FIELD**

The following cases (b) and (c) are not implemented in the current LIN-driver. The standard method is case (a) (external interrupt).

#### **(b) Using higher speed inside Slave**

The routine *SioRxInterrupt()* is divided into two main parts. The first part is the reception of a Framing-Error, the second is the reception of valid data.

- Receiving a Framing-Error

```
// following the iFP, action for received data is caused
switch (inFramePosition){
// SHBreak-field is considered
case IN_FRAME_POS_WAIT_FOR_SYNC_BREAK:
    if (ERROR_RECEIVE) {
        inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_FIELD;
        ERROR_RECEIVE = FALSE;
    } else {
        break;
    }
// SyncField is considered
case IN_FRAME_POS_WAIT_FOR_SYNC_FIELD:
// correct data
    if ( sReceiveData == SYNC_FIELD ){
// counter is set to identifier
        inFramePosition = IN_FRAME_POS_WAIT_FOR_IDENTIFIER;
// wrong data occurred
    } else {
        FLAG_ERROR_OCCURED = TRUE;
        ERROR_INCONSISTENT_SYNC = TRUE;
// wait for next SHBreak-Field
        inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
    }
    break;
// identifier is considered
case IN_FRAME_POS_WAIT_FOR_IDENTIFIER:
// correct data
    if ( sReceiveData == IDENTIFIER ){
        FLAG_SCHEDULE_DATA_SEND = TRUE;
        stateMode = STATE_MODE_INIT;
        inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
    } else {
        if (sReceiveData == IDENTIFIER2 ){
            FLAG_RECEIVE_MASTER_DATA = TRUE;
            stateMode = STATE_MODE_INIT;
        } else {
// wrong data occurred
            FLAG_ERROR_OCCURED = TRUE;
            ERROR_IDENTIFIER_PARITY = TRUE;
// wait for next SHBreak-Field
            inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
        }
    }
    break;
// if something funny occurs
default:
// wait for next SHBreak-Field
    inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
    break;
} // end of switch
} // end of else - error
}
```

**Figure 5-23: Reception of a Framing-Error**

After entering the interrupt, the register ASIS is read and compared to the value 0x02 (equals to Framing-Error). If a Framing-Error has occurred, it is checked whether the error interrupt is the first in the row. If it is the first occurrence of an error, the UART will be set to the high-speed mode.

If a Framing-Error was detected beforehand within this reception-cycle, the port will be checked if it toggled to high level in the meantime. I.e it is checked if the Stop-bit was received during the runtime of the interrupt routine.

The baudrate is set back again to normal speed, if the port has toggled in the meantime, otherwise the UART remains in high-speed mode.

- Receiving Standard Data

The reception of standard data is divided into three parts. Depending on the position inside the scheduler one of them is executed. These part reflect the following states of the LIN-bus:

- waiting for SyncBreak-Field
- waiting for SyncField
- waiting for identifier

#### Waiting for SyncBreak-field

If the first correct data is received, the scheduler is in the position for receiving the SyncBreak-field. According to the fact that framing-errors have occurred before the correct data was received, the error-flags are cleared (assigned FALSE).

The baudrate is set back to the standard defined LIN-bus-speed and the state of the scheduler is incremented to WAIT\_FOR\_SYNC\_FIELD.

#### Waiting for SyncField

The next data received by the LIN-Slave is the Sync-Field. No re-definition of the UART must be done by measuring the time of the SyncField-bit, because most devices are using a quartz-oscillator. The driver just needs to test if the data received is correct (i.e 0x55 is received) or if the incoming data is something different.

If the received data is equal to the awaited SyncField, the state of the scheduler is incremented to WAIT\_FOR\_IDENTIFIER. Otherwise, the error flag *INCONSISTENT\_SYNCH* is set according to the actual LIN-specification. Then, the scheduler return into the state WAIT\_FOR\_SYNCH\_BREAK.

#### Waiting for Identifier

In the last state the scheduler waits for a valid identifier. Actually, the LIN-Slave is configured to act on one identifier per application, but it is possible to increase this amount of identifiers if it becomes necessary.

The incoming data is compared to the identifier set-up in the appropriate header-file. If both are equal, the scheduling of application-data from the LIN-Slave will be enabled and the variable *stateMode* is set to STATE\_MODE\_INIT.

When the data received does not match with the awaited identifier, an error bit is set and the state of the scheduler is re-initialized to WAIT\_FOR\_SYNCH\_BREAK.

#### (c) Using a device with an edge-triggered UART

Some of the devices used as a LIN-Slave may have an UART-macro, which follows an edge-triggered directive to ensure the reception of data on the serial bus.

One device using an edge-triggered macro is the MiniCAN (μPD78K(F)9850). The MiniCAN can be easily configured to use the RX-interrupt caused by the first detected falling edge. Of course the method of using the external interrupt is supported by the device as well.

If the LIN-Master is sending the SynchBreak-field, a Framing-Error will occur after nine bit times. The Framing-Error will be serviced by the reception routine. The UART-macro is enabled again and it will wait for the next falling edge on the Rx-pin. Then the reception of the data is started again.

If the Framing-Error has occurred, the error register is cleared to ensure the reception of the next incoming data.

Next, the mask for the interrupts are modified such that only the external interrupt is enabled. If there are other settings necessary for the application while the SynchBreak is executed, the driver needs to be adapted accordingly.



Be sure not to touch the settings with leaving the driver unable to execute the SynchBreak-detection!

The LIN-driver is in a state that waits for the rising edge on the external interrupt.

5.4.18 Interrupt *ExternalInterrupt*

Function Prototype	input Variables	output Variables	calls Function:
ExternalInterrupt	-----	-----	a) ( Interrupt ExternalInterrupt )

The function *ExternalInterrupt()* is used only when the definition in the heading of the file *m\_slave.c* is set to TRUE and the other definitions are undefined or commented.

After the first framing error has occurred while waiting on the SyncHBreak-field, the external interrupt is enabled. Now the device will react on the first rising edge detected on the RxD-pin.

```

interrupt [EXT_INT_VECT] void ExternalInterrupt ( void ){
    /*=====*/
    /* FunctionName: interrupt SioTxInterrupt                                     */
    /* IN/OUT      : -/interrupt                                                 */
    /* Description : This interrupt-function is started when the Tx-ready-      */
    /*              Interrupt is received. The wait-Flag is set to ready        */
    /*              so all attached functions will know when the data is       */
    /*              sent.                                                         */
    /*=====*/
    // sets the state to wait for Sync_field
    inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_FIELD;

    // re-sets error-condition for receive of Framing-Error
    FLAG_ERROR_OCCURED = FALSE;

    // sets back the Mask-type to standard
    MK0 = save_MK0;
    MK1 = save_MK1;

    //re-starts the UART
    START_UART;
}

```

**Figure 5-24: External Interrupt-Function**

The state-machine is set to the next state 'WAIT\_FOR\_SYNC\_FIELD'. The old masks for the interrupts are restored, the UART is started (it was stopped beforehand to prevent that the UART is running while waiting on the rising edge of the SyncHBreak-field) and the external interrupt is left.

When the protocol activity is monitored by the LIN-driver, it will be detected as a SyncField. Of course data needs to read 0x55 as well.

5.4.19 *TimerCompInterrupt*

Function Prototype	input Variables	output Variables	calls Function:
TimerCompInterrupt	-----	-----	-----

The function *TimerCompInterrupt()* will be called at a match of the actual count of the timer and the contents of the compare-register. If the time for scheduling the next LIN-message is reached (given by the value `RAMP_COMPARE_VALUE`), the scheduler is reset and the variable *TimerUse* is set.

```
interrupt [INTTM_VECT] void TimerCompInterrupt (void) {
/*=====*/
/* FunctionName: interrupt TimerCompareInterrupt */
/* IN/OUT      : -/interrupt */
/* Description : This interrupt-function is started when the free-running*/
/*               Timer-value compares to the pre-set Value in the com- */
/*               pare-register. Different counter-vars will be set to */
/*               schedule different tasks */
/*=====*/
LIN_Message_Scheduler += 1;
// add timer-value because of free-running TM20
CR20 = TM20 + TIMER_BASE;
if ( LIN_Message_Scheduler == RAMP_COMPARE_VALUE ){
    LIN_Message_Scheduler = 0;
    FLAG_TIMER_USE = TRUE;
}
}
```

**Figure 5-25: Function interrupt *TimerCompareInterrupt***

This will indicate the state of the timer to other routines (i.e. application tasks). Please, refer to file `Main.c` for examples.

## **Chapter 6 LIN-Master Driver 78k0**

### **6.1 Intention**

To follow the general demand for small devices acting as a LIN-Master, the driver was ported from the V85x to the 78k0. This gives the customers the possibility to use less expensive and small hardware even for the LIN-Master.

The changes to the given V85x-Master depend on the development-environment (GreenHills Multi for the V85x or IAR-EW for the 78k0). Additionally the LIN-driver is especially tuned to the needs of the 78k0-core in terms of speed.

### **6.2 Realisation**

The pilot-implementation of the LIN-Master driver for the 78k0 was created using the already existing environment for the V850.

Some changes became necessary due to the different, limited resources of the hardware given by the devices. Other changes are done following different sourcing-routines lead by IAR-Electronic Workbench and GHS Multi-IDE.

A detailed description of the differences has been renounced in this document because the issues are self-explaining. Please, refer to the code for further information.

[MEMO]



## Chapter 7 Differences to the LIN-Master-driver using LIN-UART6

### 7.1 Intention

The new LIN-capable macro, LIN-UART6, is especially designed to use LIN with minimum effort. As a consequence some differences to the details of the driver described in the chapters before apply. These changes of the LIN-driver are explained in the following chapters.

### 7.2 LIN-UART6 - short overview

The UART6 has some special features, that can be used to minimize the software overhead, which is used to send and receive the special data as required by the LIN-protocol.

The basic format of the NEC-LIN-driver remains like described before. Only the internal recognition of receiving and sending special LIN-format data like the SynchBreak, the WakeUp from Sleep and the detection of the baudrate were subject to change.

The settings for these special functions are done by a newly added LIN-register, called **ASICL**. Here, settings used by both, the LIN-Master and the LIN-Slave are realised.

### 7.3 List of Adaptions

The main-adaptions to the files of the LIN-Master are the following:

- Sending a SynchBreak-Message with a variable length of 13 to 20 bit without considering the non-standard format
- Reacting on WakeUp-Signals sent by any Slave of the LIN-bus

#### 7.3.1 Sending SynchBreak-Frames

The Synch-Break-field in the LIN-protocol is used as a 'HELLO-WORLD' message. This frame should have a length of at least 13 bit times of the actual LIN busspeed in order to fulfill the special needs assigned to this command-frame. Other than standard UART-macros that can not perform this special requirement, the NEC LIN-UART is capable to be set to a variable length of 13 to 20 bit.

First, the needed SynchBreak-length has to be set with the three bits reserved for this command-frame. After this, the SBTT-Bit(SynchBreakTransmitTrigger) in the register ASICL is set to TRUE, so the SynchBreak-Bit is sent instantaneously. Then the transmission finishes with an interrupt( transmit ready), which is traced by the driver in order to promote state-machine into its next state.

#### 7.3.2 Reacting on WakeUp-Signals

While the node is in Power-Save-mode (HALT or STOP), the RX-pin forces the UART into the receiving state when data is incoming. This is also performed in case the macro was disabled beforehand. The internal state-machine compares the data received to its awaited state and the device will resume to run the LIN-protocol respectively.

### 7.4 Use of changes in the NEC-LIN\_Master driver

The main difference is the change of the schedule for sending SynchBreak-fields to the LIN-bus attached.

### 7.4.1 Changes to LIN\_m.h

The changes to the LIN\_m.h add the newly implemented register ASICL with its various settings into the files of the driver.

Additionally, the settings for the registers BRGC and CKSR are introduced.

```
#define CKSR_SETTING_19200 CKSR0 = 0x00
#define BRGC_SETTING_19200 BRGC0 = 0x82
#define SEND_SHB_13_BIT ASICL0 = 0x34
#define RECEIVE_SHB_13_BIT ASICL0 = 0x54
```

**Figure 7-1: Changes to LIN\_m.h**

### 7.4.2 Changes to UART\_m.h

The LIN-Master driver needs some additional definitions for the register ASICL as given below.

```
// sets SBF-length and starts scheduling of SBF-send
#define LIN_SEND_SHB_13 LIN_CONTROL_REGISTER = 0x34
#define LIN_SEND_SHB_14 LIN_CONTROL_REGISTER = 0x38
#define LIN_SEND_SHB_15 LIN_CONTROL_REGISTER = 0x3C
#define LIN_SEND_SHB_16 LIN_CONTROL_REGISTER = 0x20
#define LIN_SEND_SHB_17 LIN_CONTROL_REGISTER = 0x24
#define LIN_SEND_SHB_18 LIN_CONTROL_REGISTER = 0x28
#define LIN_SEND_SHB_19 LIN_CONTROL_REGISTER = 0x2C
#define LIN_SEND_SHB_20 LIN_CONTROL_REGISTER = 0x30
```

**Figure 7-2: Changes to UART\_m.h**

### 7.4.3 Changes to m\_Master.c

The changes to the file m\_Master.c addresses how the SynchronBreak-field is sent.

Using the UART6 this task has become much more easier. Instead of changing the baudrate to the half of the speed, sending the SynchronBreak-field and then again reinstalling the original LIN-bus speed afterwards, only the length of the SynchronBreak-field to be sent is specified.

If the state to send the SynchronBreak-field is reached, this command is executed instantly, and the SynchronBreak-field is broadcasted to the LIN-bus.

## Chapter 8 Differences to the LIN-Slave Driver using LIN-UART6

### 8.1 Intention

The new LIN-capable macro, LIN-UART6, is especially designed to use LIN with minimum effort. As a consequence some differences to the details of the driver described in the chapters before apply. These changes of the LIN-driver are explained in the following chapters.

### 8.2 LIN-UART6 - short overview

The UART6 has some special features, that can be used to minimize the software overhead, which is used to send and receive the special data as required by the LIN-protocol.

The basic format of the NEC-LIN-driver remains like described before. Only the internal recognition of receiving and sending special LIN-format data like the SynchBreak, the WakeUp from Sleep and the detection of the baudrate were subject to change.

The settings for these special functions are done by a newly added LIN-register, called **ASICL**. Here, settings used by both, the LIN-Master and the LIN-Slave are realised.

### 8.3 List of Adaptions

The main adaptions to the files of the LIN-Slave are the following:

- Receiving a SynchBreak-Message with a minimum length without considering the non-standard format
- Receiving SyncFields with the possibility to measure the incoming data and changing the internal UART-baudrate accordingly to fit the real LIN-baudrate
- Reacting on WakeUp-Signals sent by any Slave of the LIN-bus

#### 8.3.1 Receiving SynchBreak-Frames

When the UART shall be set into a mode to receive a SynchBreak-field, there is a fast method available using the ASICL-register. Just the bit SBRT (SyncBreakReceiveTrigger) needs to be set, and the UART-macro is put in the mode to receive the SynchBreak sent by the LIN-Master.

While the UART is in this mode, any message on the LIN-bus will stop it from waiting. The received data has to be at least 10.5 bit times long to fulfill the minimum requirement of a SynchBreak-field, while the Slave-node may be in not-synchronized mode.

If the received data is recognized as a valid SynchBreak-field, a receive-Interrupt is generated and the LIN-driver enters the respective state.

Otherwise, the device UART will return to the *waitForSynchBreak()* again, and an error is generated.

#### 8.3.2 Receiving Sync-Fields

The second LIN-option realized in the new UART6 is the reception of the command-frame Sync-Field. To receive this with minimum effort, a port shared with an internal 16-bit timer can be attached to the Rx-pin of the LIN-UART.

If data is received on the Rx-pin, the timer starts counting depending on the settings made by the customer and the application. The application has to stop the timer and needs to calculate the result measured by the timer attached to the UART. The result will enable the software to reinstall internal baudrate to the baudrate, which is actually present on the LIN-bus.

### 8.3.3 Reacting on WakeUp-Signals

While the node is in Power-Save-mode (HALT or STOP), the RX-pin will set the UART into the receiving state when data is incoming. This is also performed in case the macro was disabled beforehand. The internal state-machine compares the data received to its awaited state and the device will resume to run the LIN-protocol respectively.

## 8.4 Use of changes in the NEC-LIN\_Slave driver

There are various changes needed to be attached to the LIN-files. Most of them depend on the different hardware of the UART6. The necessary changes applied to the new LIN-UART will be described in the following:

### 8.4.1 Changes to LIN.h

```
// User-defined
// LIN-specific setting

//setting for double-speed on UART for SynchBreak-Field
unsigned char BAUDRATE_SYNC_BREAK = 0x10;
// setting for normal bus speed in application, e.g. 19.200
unsigned char BAUDRATE_NORMAL_SPEED_CKSR6 = 0;
unsigned char BAUDRATE_NORMAL_SPEED_BRGC6 = 0x80;
// Set to RECEIVE SHB, 13 BIT, MSB-first, Normal Output
unsigned char LIN_RECEIVE_SHB_13BIT = 0x56;
```

**Figure 8-1: Changes to LIN.h for usage of UART6**

The changes made to LIN.h are marginal. Only the settings for the baudrate via the registers CKSR and BRGC are switched, and a setting for register ASCIL is implemented that defines when the reception of a SynchBreak-field should start.

### 8.4.2 Changes to UART.h

```
// standard-settings

#define BAUD_RATE_CONTROL_REGISTER BRGC6
#define BAUD_RATE_CLOCK_SELECT_REGISTER CKSR6
#define UART_MODE_REGISTER ASIM6
#define UART_ERROR_REGISTER ASIS6
#define UART_LIN_MODE_REGISTER ASICL6
#define TRANSMIT_SHIFT_REGISTER TXB6
#define RECEIVE_BUFFER_REGISTER RXB6
```

**Figure 8-2: Changes to UART.h for usage of UART6**

The UART.h will be broadened by some new definitions. They will refer to the new method of setting the baudrate (BRGC- and CKSR-register) and to the new LIN-register ASICL.

### 8.4.3 Changes to m\_slave.c

To provide the LIN-driver with the full functionality, there are some changes needed to various system-functions, which will use the new LIN-functionality:

- `interrupt [INTRX_VECT] void SioRxInterrupt ( void )`

The automatical reception of the SyncHBreak-field necessitates to modify the receive-routine. The first interrupt the state-machine encounters signals that the SyncHBreak-Field is ready.

```
// wait for next SHBreak-Field
    inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
// set UART6 to wait for SHBreak
    UART_LIN_MODE_REGISTER = LIN_RECEIVE_SHB_13BIT;
```

**Figure 8-3: Changes to m\_slave.c - Receive-Routine**

In case of an occurring error, even if the data is sent from the Slave to the Master completely or vice-versa, the UART will be re-set into the state of receiving the SyncHBreak-field again.

- `initHardware`

The routine `initHardware()` needs some additional settings to bring the UART into the state of 'Waiting for SyncHBreak-field' right after the software started.

```
void initHardware (void) {
    /*=====*/
    /* FunctionName: initHardware */
    /* IN/OUT : -/- */
    /* Description : This function calls all other related init- and set- */
    /*                functions for UART, timer, ScheduleTable. */
    /* */
    /* */
    /* ===== */
    initUART ();

    // sets the Timer-registers to init-values
    initTimer ();
    // sets the sched-table to the first valid input
    initDataTable ();
    ERROR_FIRST_TIME = TRUE;
    if (FLAG_FIRST_TIME_SCHED){
        FLAG_RECEIVE_MASTER_DATA = FALSE;
    }
    // scheduling of SyncHBreak-field is initiated
    inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNC_BREAK;
    // set UART6 to wait for SHBreak for the first time
    SBRT6 = 1;
    *p_data_table_read_position = 0;
    FLAG_ERROR_OCCURED = FALSE;
    ERROR_FIRST_TIME = TRUE;
    // running 1st time data-count
    FLAG_TIMER_USE = TRUE;
    stateMode = STATE_MODE_INIT;
    FLAG_SCHEDULE_DATA_SEND = FALSE;
    // setting for 1st schedule...
    FLAG_DATA_DELIVERED = TRUE;
    ERROR_RECEIVE = FALSE;
    // if Timer is not running, start Timer!
    if ( !FLAG_TIMER_RUN ) {
        startTimer ();
    } // end of if FLAG_TIMER_RUN

    FLAG_FIRST_TIME_SCHED = FALSE;
} else {
}
}
```

**Figure 8-4: Changes to initHardware following UART6-Functionality**

- interrupt TimerCompInterrupt

```
interrupt [INTTM_VECT] void TimerCompInterrupt (void) {
/*=====*/
/* FunctionName: interrupt TimerCompareInterrupt          */
/* IN/OUT      : -/interrupt                             */
/* Description : This interrupt-function is started when the free-running*/
/*              Timer-value compares to the pre-set Value in the com- */
/*              pare-register. Different counter-vars will be set to  */
/*              schedule different tasks                          */
/*=====*/
LIN_Message_Scheduler += 1;
// add timer-value because of free-running TM20
CR000 = TM00 + TIMER_BASE;
if ( LIN_Message_Scheduler == RAMP_COMPARE_VALUE ){
    LIN_Message_Scheduler = 0;
    FLAG_TIMER_USE = TRUE;
}
}
```

**Figure 8-5: Changes to interrupt TimerCompInterrupt-Routine**

Due to the Free-Runing-Timer of the used 78F0103 - KB1, the real count has to be incremented by the current Timer-Base to ensure, that the period of counting is measured correctly.

- initUART

The initUART-routine needs some modifications, too, regarding the new LIN-UART-functionality:

```
void initUART (void) {
/*=====*/
/* FunctionName: init UART                                */
/* IN/OUT      : -/-                                       */
/* Description : This routine inits all values recent for the UART- */
/*              macro.                                         */
/*              It is called by the initHardware-function      */
/*=====*/

    STOP_UART;
    CKSR6 = BAUDRATE_NORMAL_SPEED_CKSR6;
    BRGC6 = BAUDRATE_NORMAL_SPEED_BRGC6;
    // set Port-Mode to 1 for receive
    PM_Rx = HIGH;
    // set Port-Mode to 0 for transmit
    PM_Tx = LOW;
    PUL4 = 1;
    // set Port to 0 for input
    // set Port to 1 for output
    // start for ASICL6-register
    DIR6 = 1;
    TXDLV6 = 0;
    // end for ASICL6-register
    POWER6 = 1;
    TXE6 = 1;
    RXE6 = 1;
    PS61 = 0;
    PS60 = 0;
    CL6 = 1;
    SL6 = 0;
    ISRM6 = 1;
    // start for ASICL6-register
    SBL60 = 1;
    SBL61 = 0;
    SBL62 = 1;

    P_Tx = HIGH;
    _EI();
}
```

**Figure 8-6: Modifications for UART6 - initHardware**

## Appendix A Application for the V850 LIN-Master Driver

As a first method to test the settings implemented inside the LIN-Master-driver, a short application using the previously described LIN-Master-driver is attached.

It is a rudimentary implementation that only sends data using a fixed schedule emulating an application, which may get information by attached higher-level tasks and/or some randomly changing data by actuators or similar devices attached to the LIN-Master.

In this example, a table for the schedule with five identifiers and a delay-time of 50 msec. is used. Besides the identifiers, the length of that table has to be defined as the correct operation of the driver depends on this information.

The application needs to declare some standard definitions and it has to include driver-routines in order to be able to run in conjunction with the LIN-Master-driver.

```
#include "in78000.h"
#include "Df9850.h"
#include "M_Master.h"

extern void startLin ( void );
extern void stopLin ( void );
extern void sendBusToStop ( void );
extern void scheduleSending ( void );

extern bit FLAG_TIMER_USE;
extern bit FLAG_SCHEDULE_DATA_SEND;
```

**Figure A-1: Definitions for application-use of LIN-Master-driver**

When these definitions are done, some routines have to be called once in order to initialize the attached hardware, the LIN-driver, and all used variables. For proper functionality, an initial scheduling has to be run.

```
void main(void)
{
    unsigned char temp_count_test= 0;
    PCC  = 0x00;
    CSS  = 0x00;
    OSTS = 0x00;
    MK1  = 0x9E; // sets Mask to SerTrans(, SerRec, SerEr, Timers) to enable

    _DI();

    startLin ( ); // calls init-routine in M_Master.c setting Hardware and LIN

    _EI ( );

    scheduleSending ( ); // initial scheduling
}
```

**Figure A-2: Initializations of standard LIN-Master-driver routines**

The main-routine will collect the data, steered by the implemented application-timer. The data sent by the attached ( and simulated ) LIN-slaves will be stored in an reserved array. Depending on this, further application-routines, which are not implemented here, could act on.

```
while (-1)
{
    if (FLAG_TIMER_USE){

        if ( FLAG_SCHEDULE_DATA_SEND ){

            scheduleSending ();
            FLAG_TIMER_USE = 0;
        } // FLAG_SCHEDULE_DATA_SEND

    } // FLAG_TIMER_USE

} // while (-1)

} // main
```

**Figure A-3: Main-routine of LIN-Master-application**

The scheduled main-routine will check two Flags modified by the LIN-driver-software, the linFlagField.FLAG\_TIMER\_USE and the linFlagField.FLAG\_SCHEDULE\_DATA\_SEND, if they're TRUE or FALSE.

In case of one of them being FALSE, nothing will happen. Otherwise, the FLAG\_TIMER\_USE is re-set to FALSE and the routine scheduleSending will be executed, which as result causes the driver to send the Header-data and wait for the according Slave-data.



## Appendix B Application for the LIN-Slave Driver

To get a running network, a small application using the LIN-slave-driver has been implemented to show the capabilities of the driver and to check out standard LIN-applications.

The definitions for the LIN-slave-driver are slightly different from the option of the LIN-Master-driver:

```
// ID send data to Master on req.
unsigned char IDENTIFIER = 0x6F;
// ID rec. data from Master on req.
unsigned char IDENTIFIER2 = 0x2E;
unsigned char DATA_TABLE_LENGTH= 0x04;
```

**Figure B-1: Slave-driver settings in the Slave LIN-header-file**

Like shown above, there is one identifier for reception and another one for transmission defined to be recognized by the LIN-slave. The length of the attached data-table depends on the identifiers (there is one byte for the checksum and one for the 'data-valid' flag in addition!).

The application has to include several LIN-driver variables and statements as external definitions to ensure operation and data exchange of the LIN-driver with the application.

```
// INCLUDE-FILES
#include "in78000.h"
#include "DF9116A.h"
#include "M_Slave.h"

// FUNCTION-PROTOTYPES

extern void startLin ( void );
extern void stopLin ( void );
extern void sendBusToStop ( void );
extern void startScheduling ( void );
extern void scheduleSending ( void );

// EXTERNAL included VARs

extern unsigned char *p_read_active_table;
extern unsigned char *p_data_valid_table;
extern unsigned char *p_write_app_table;
extern unsigned char *p_valid_app_table;
extern unsigned char *p_read_allowed_app_table;
extern unsigned char data_table[6];
extern unsigned char DATA_TABLE_LENGTH;
extern unsigned char calculateChecksum ( void );
extern unsigned char *p_data_table_read_position;
```

**Figure B-2: External definitions in the Slave-application**

After the declarations are done, some routines have to be called once in order to set variables into their initial state and set the LIN-Slave into running mode. The values *tempi* (i= 1 - 4) are examples for test purposes. Normally they are generated by a real application out of attached sensoric/actoric.

```
void main(void) {

// LOCAL VARs for testing with pre-set values
unsigned char temp1 = 0x12;
unsigned char temp2 = 0x9C;
unsigned char temp3 = 0xF1;
unsigned char temp4 = 0xAB;
// set hardware-related registers
PCC = 0x00;
MK0 = 0x4F;
// sets Mask to SerTrans(, SerRec, SerEr, Timers) to enable
MK1 = 0xFF;

_DI();

p_write_app_table = &data_table[0];
p_valid_app_table = &data_table[0] + DATA_TABLE_LENGTH + 0x02;
p_read_allowed_app_table = &data_table[0] + DATA_TABLE_LENGTH + 0x03;
startLin ( ); // calls init-routine in M_Master.c setting Hardware and LIN

_EI();
```

**Figure B-3: Initialization of variables by settings and initial calls after reset**

All required settings for the tabel defined by the application have to be prepared here. Then the routine *startLin()* is called, which will set the related hard- and software to the values needed to run a LIN session.

The following code has to be implemented as cyclically execution in the application:

```
while (-1)
{
    if (FLAG_TIMER_USE){

        *p_valid_app_table = 0x00;

        *p_write_app_table = temp1;
        p_write_app_table += 1;

        *(p_write_app_table) = temp2;
        p_write_app_table += 1;

        *(p_write_app_table) = temp3;
        p_write_app_table += 1;

        *(p_write_app_table) = temp4;
        p_write_app_table += 1;

        *(p_write_app_table) = calculateChecksum ();
        p_write_app_table = &data_table[0];

        *p_valid_app_table = 0x01;

        FLAG_TIMER_USE = 0;
    } else { // timerUse
        if (FLAG_SCHEDULE_DATA_SEND){
            if ( FLAG_DATA_DELIVERED){
                scheduleSending ();
            } else {
            }else {
            }
        }
    } // end of else timerUse
} // end of while
}
```

**Figure B-4: Cyclic called LIN-main-routine**

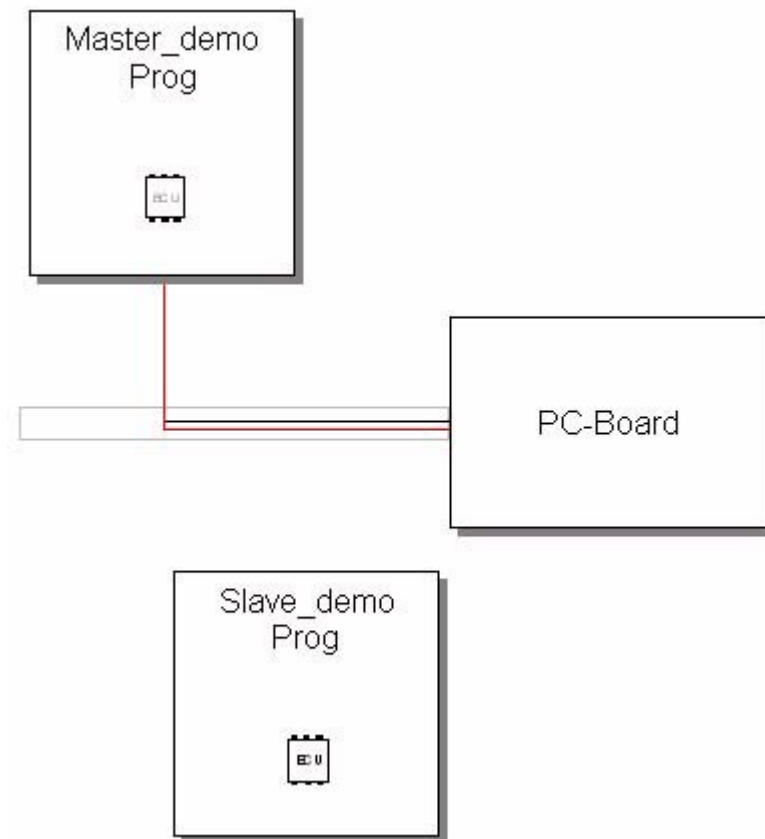
The interrupt of the timer implemented inside the LIN-driver is used to generate random data in this case. In a real environment, some data retrieved by AD-converter or ports are requested here. This data is written into the table of the application. The contents of that table is set to 'not-valid' before writing data and to 'valid' after all data, including the checksum, has been written into it.

If the Slave receives the request by the LIN-Master, only data that has a 'valid' stamp will be sent onto the LIN-bus.

**[MEMO]**

## Appendix C LIN-Emulation using CANoe with option LIN

To ensure that both, the LIN-Master- and the LIN-Slave-driver are running properly, an emulated LIN-environment was implemented. It uses the routines for a Master- and a Slave-driver that normally run on the device.



**Figure C-1: Overview of the LIN-Emulation-environment, Master active**

- **Emulation of LIN-Master**

The LIN-Master, which causes the NEC LIN-driver to run, provides a schedule for five identifiers. One of these identifiers is recognized by the LIN-slave.

At first some internal data is defined:

```
variables {  
    int schedulerMode;  
    msTimer changeSchedMode;  
    int count,errorCount = 0;  
}
```

**Figure C-2: Variables used by the Master-emulation**

In the next step the routines for the Pre-Start follow:

```
on preStart
{
    byte modeFlags1[1] = { 1 }; // only mode 1
    byte modeFlags12[1] = { 3 }; // in mode 1 and mode 2 (0x...11)
    byte modeFlags3[1] = { 4 };
    byte modeFlags4[1] = { 8 };
    byte modeFlags5[1] = { 0x10 };
    byte responseData [2] = {0x23, 0x41};
    byte responseData2[2] = {0x55, 0x54};
    // set LINDa into INIT-state
    LINInitBegin();
    // adjusting of SHB-length and Stop-Bit-Length
    LINMrSchedSetSyncT ( 0x000D, 1 );
    // setting LIN-baudrate
    LINInitSetBaseBaud ( 19200 );
    // sets Master to ON
    LINInitSetMaster(1,1);

    // schedule-table with 20msec cycle and one state
    LINMrSchedSetGlobal(200, 3);
    // ID for Slave, data to send
    LINMrSchedSetRqId(0x0D, 100, modeFlags1);
    LINMrSchedSetRqId(0x08, 100, modeFlags12);
    LINSetResponseData (0x08, 2, responseData2);

    // end of LIN-init
    LINInitEnd();
}
```

**Figure C-3: Pre-start-routines**

Here, the LINDa is configured to send a SHBreak-field with a length of 0x10 bytes, the according Stop-Bit-length is one byte.

The LIN-baudrate is set to 19.2 kbaud and the table for the schedule is setup to send the identifier 0x04.

When LINDa starts working, the routine onStart is executed:

```
on start
{
    // initial setting of setTimer to 200msec.
    setTimer ( changeSchedMode, 200 );
}
```

**Figure C-4: On Start-routine**

Herein, the used internal Timer is set to a initial value of 200 msec.

The routine for the timer itself contains the following code:

```
on timer changeSchedMode
{
    count++;

    switch (schedulerMode) {
        // setting for usage of only one identifier
        case 0: schedulerMode = 1; break;
        case 1: schedulerMode = 0; break;
    }
    // set new scheduler-mode
    LINMrSchedSetMode(schedulerMode);
    // set Timer to new cycle
    setTimer ( changeSchedMode, 1000);
}
```

**Figure C-5: Routine onTimer**

Using this example, two modes of the scheduler can be used. The different schedules can be evoked with the variable *schedulerMode*.

The new mode of the scheduler is set, the timer is set to a new cycle of 20 msec, and status information is put to the out-window.

The following two pictures show the running system with an emulated LIN-Master and the NEC-LiN-Slave-driver.

•Bus protocol as an overview

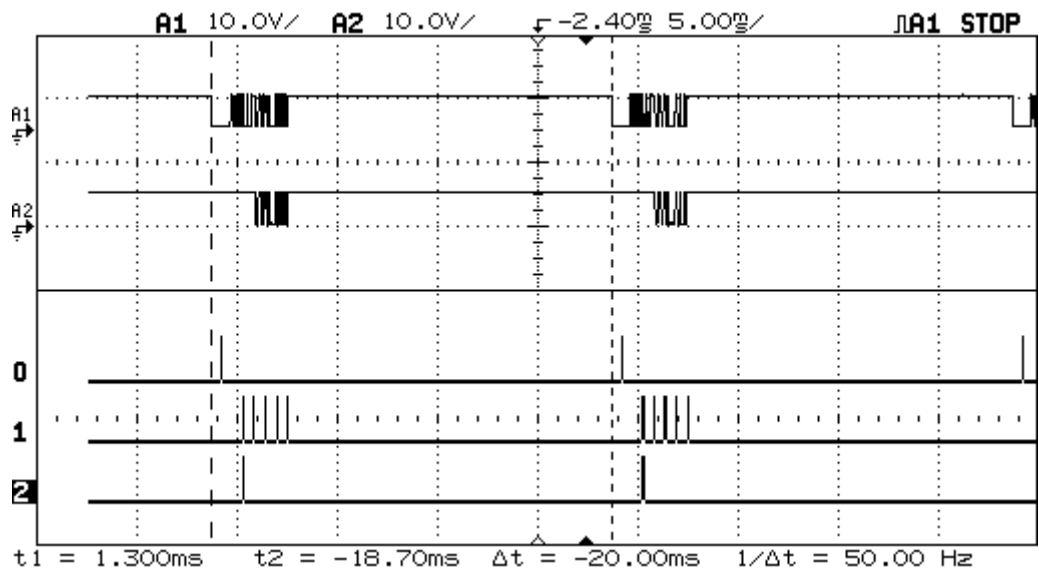


Figure C-6: Protocol-overview using the LIN-Slave-driver

•Frame details

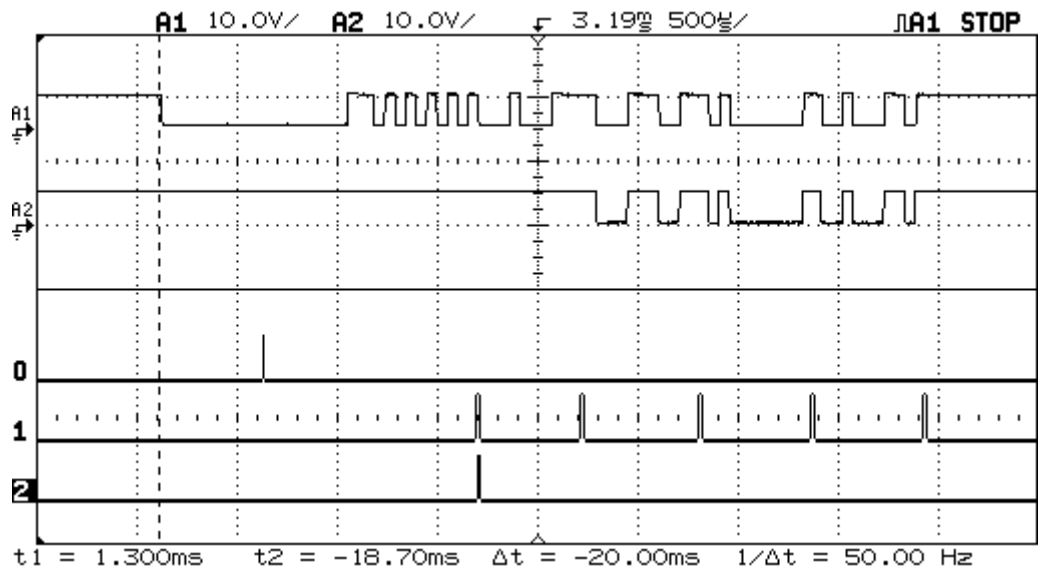


Figure C-7: Frame-detail of used LIN-slave-driver



- **Emulation of LIN-slave**

For using the LIN-Slave, a slightly different version of the Vector LIN-emulation is taken.

First, some variables are defined

```
variables {
    int sID = 4;
    int schedulerMode = 1;
    message 0x04 sLINResp;
}
```

**Figure C-8: Definitions of internally used variables**

At the 'onPrestart' condition, the initializations for the LIN-slave are executed:

```
on preStart
{
    byte responseData2[2] = { 0xFF, 0x7F };
    byte responseData3[4] = { 0x66, 0x77, 0x88, 0x99 };
    byte responseData4[4] = { 0x22, 0x33, 0x44, 0x66 };
    byte responseData5[8] = { 0x12, 0x23, 0x34, 0x45, 0x56, 0x67, 0x78, 0x89 };

    // LINDa-Init
    LINInitBegin();

    LINS1Simulate (0x06);
    LINS1Simulate (0x20);
    LINS1Simulate (0x2F);
    LINS1Simulate (0x36);
    // set LIN-baudrate
    LINInitSetBaseBaud ( 19200 );
    // set Length of SHBreak-field
    LINMrSchedSetSyncT ( 20, 1 );
    // set identifier and according data-length
    LINSetDlc ( 0x06, 0x02 );
    LINSetDlc ( 0x20, 0x04 );
    LINSetDlc ( 0x2F, 0x04 );
    LINSetDlc ( 0x36, 0x08 );
    LinSetResponseData(0x06, 0x02, responseData2);
    LinSetResponseData(0x20, 0x04, responseData3);
    LinSetResponseData(0x2F, 0x04, responseData4);
    LinSetResponseData(0x36, 0x08, responseData5);
    LINInitEnd();
}
```

**Figure C-9: Lin-Slave-routine onPreStart in emulation**

The LIN-baudrate is set to 19.2 kbaud and the awaited SHBreak-length is set to 18 bit times with a Stop-Bit-length of two bit times.

The identifier where the Slave is acting on is 0x04, the according DLC is set to two byte.

After this initializations, the Slave is set into Run-mode using the routines in the routine 'onStart':

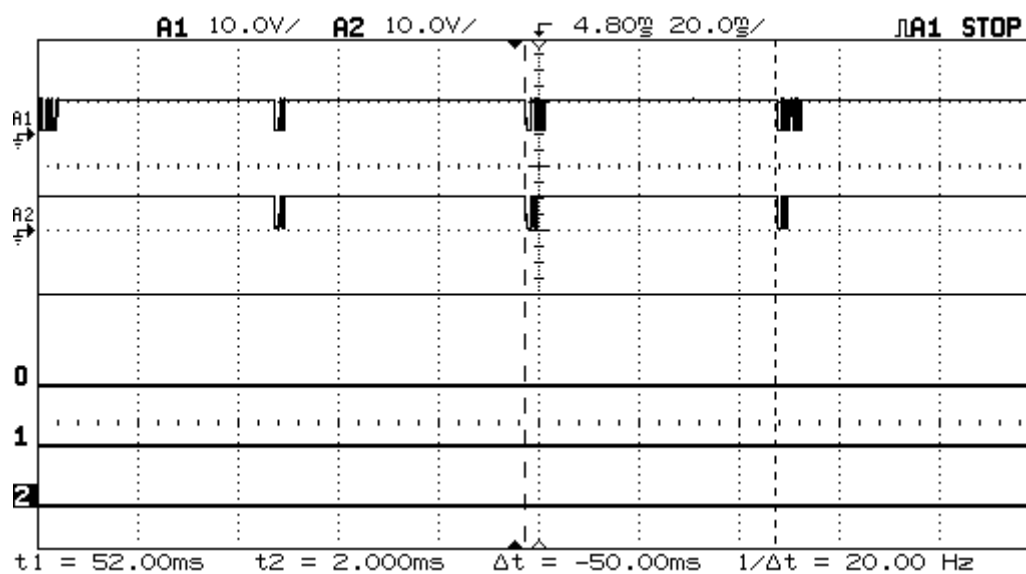
```
on start
{
  write( "Use key >1< to select scheduler mode 1");
  write( "Use key >2< to select scheduler mode 2");
}
```

**Figure C-10: Emulated Slave-routine on Start**

The example above defines key-strokes. The driver will run with the pre-defined settings and the mode of the scheduler can be changed by striking the key '1' or key '2'.

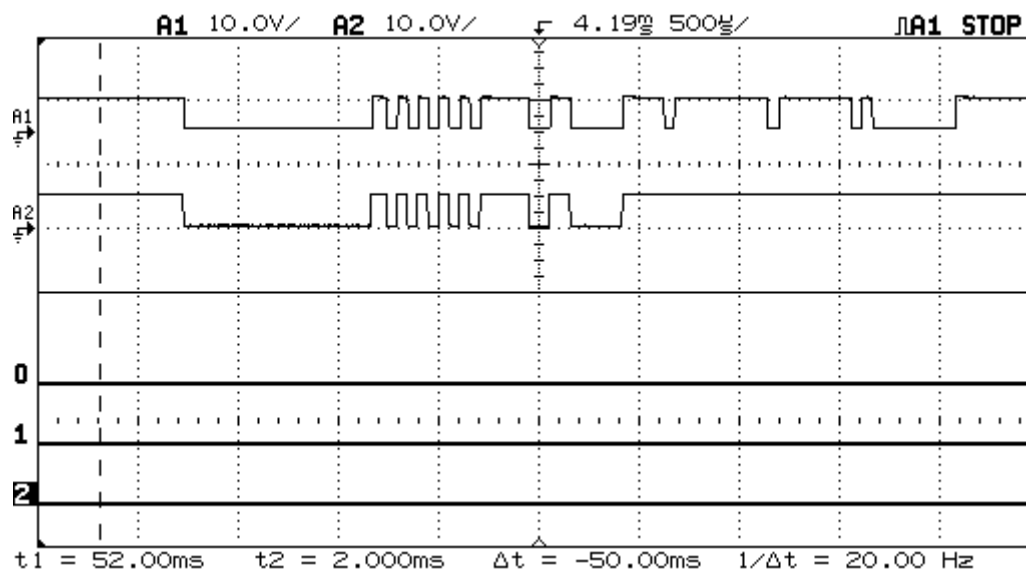
The following two pictures show the running system with an emulated LIN-Slave and the NEC-LIN-Master-driver:

- **Bus protocol as an overview**



**Figure C-11: Protocol-overview using the LIN-Master-driver**

- **Frame details**



**Figure C-12: Frame-detail using the LIN-Master-driver**

The bus protocol can be monitored in the output-window of CANoe:

78.5127	LIN 1	04	TransmErr					
0.0000	LIN 1		SleepModeEvent			starting up in wake mode		
0.9744	LIN 1		RcvError			illegal character during bus idle		
78.5620	LIN 1	06	Tx	LIN message	2	ff 7f		80
1.0296	LIN 1			Baudrate		19230		
78.6131	LIN 1	20	Tx	LIN message	4	66 77 88 99		00
78.6633	LIN 1	2f	Tx	LIN message	4	22 33 44 55		11
78.7156	LIN 1	36	Tx	LIN message	8	12 23 34 45 56 67 78 89		91

**Figure C-13: Bus protocol using the NEC LIN-Master-driver**

## Appendix D Network Overview

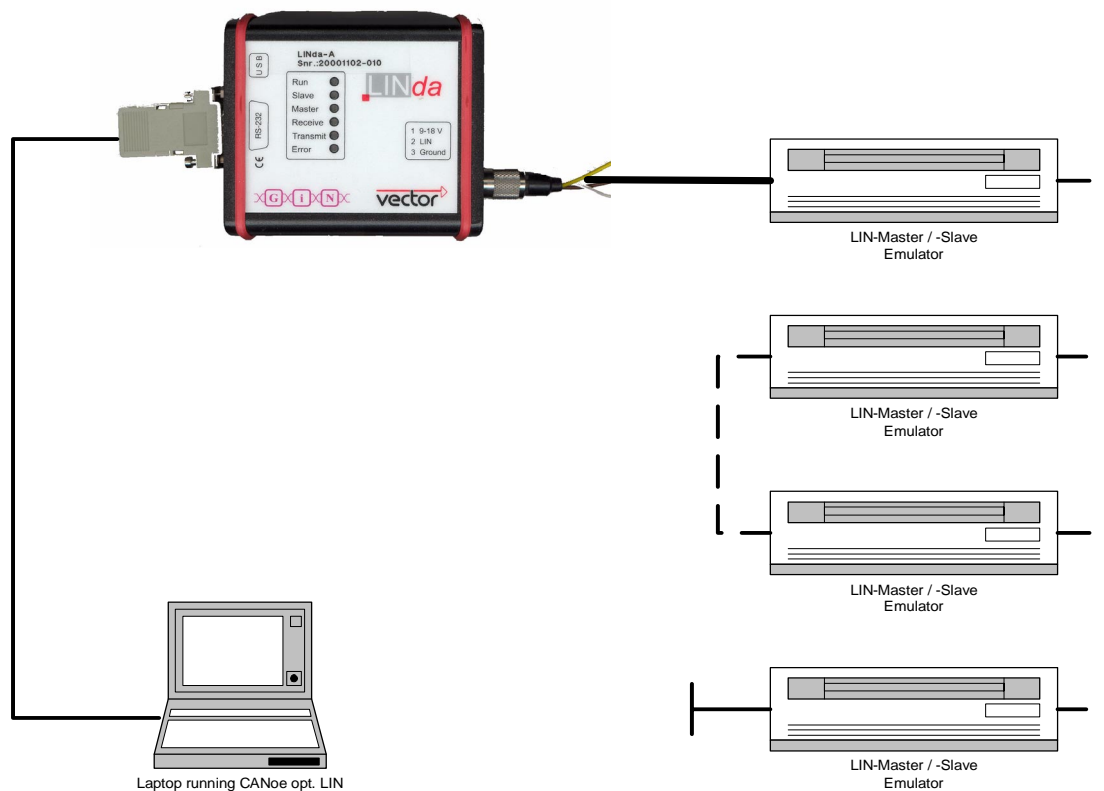
To ensure that both, the LIN-Master- and the LIN-Slave-driver are running properly, an emulated LIN-environment was implemented, which makes use of routines for a Master- and a Slave-driver.

The used hardware consists of:

- A PC/Laptop to drive the Vector-Informatik Environment of CANoe-LIN
- The LINda as connection between CANoe option LIN and the used transceiver
- A board with an implemented LIN-transceiver, which is connected on the one side to the LINda as link to CANoe option LIN and on the other side to the application, e.g. an emulator or real device.
- Target hardware, where the real driver runs on

The components above already form a complete network. The complexity of the network depends on the amount of nodes attached. They can be designed by real devices or emulators.

The following photo shows how such a network designed for testing will look like:



**Figure H-7: Network-outline for test purposes**

**[MEMO]**

## Appendix E Revision History

- Documentation

Version	Description
V 2.0	Initial Documentation
V2.1	Added LIN - UART6-Features
V2.2	Enhanced description

- LIN-Master-driver

Version	Description
V 2.0	Initial Driver-release for LIN-Master
V 2.1	Added Driver with Functionality for LIN - UART6

- LIN-Slave-driver

Version	Description
V 2.0	Initial Driver-release for LIN-Slave
V 2.1	Added Driver with Functionality for LIN - UART6

**[MEMO]**



## Appendix F Software included

The following files are necessary to use the described LIN-features, where the DF\*.\*-files, the .hex, the .lnk, the .xcl and the .prj are project- and device-dependant:

- LIN-Master

Name	Size	Type
Df9850.h	6KB	Header file
hardware_M.h	4KB	Header file
in78000.h	3KB	Header file
LIN_M.h	5KB	Header file
M_Master.h	4KB	Header file
UART_M.h	4KB	Header file
lin_master_minican_latest_released.hex	7KB	HEX File
lin_master_minican_latest_released	13KB	Shortcut
m_master.c	28KB	Source file
main_LINMaster_M.c	4KB	Source file
LIN_Master_MiniCAN.prj	6KB	UltraEdit project
Df9850.xcl	4KB	XCL File

**Figure F-1: Delivered files used by the LIN-Master-driver**

- LIN-Slave

Name	Size	Type
Df9850.h	6KB	Header file
hardware_S.h	4KB	Header file
in78000.h	3KB	Header file
LIN_S.h	5KB	Header file
M_Slave.h	5KB	Header file
UART_S.h	4KB	Header file
lin_slave_minican_latest_released.hex	4KB	HEX File
lin_slave_minican_latest_released	13KB	Shortcut
m_slave.c	25KB	Source file
main_LINMaster_S.c	5KB	Source file
LIN_Slave_MiniCAN.prj	6KB	UltraEdit project
Df9850.xcl	4KB	XCL File

**Figure F-2: Delivered files used by the LIN-Slave-driver**

**[MEMO]**

## Appendix G Technical details, ressources, Implementations

- LIN-Master-driver used ressources

The ressources used by the LIN-Master-driver are subject to be changed when adapting the LIN-driver to the final internal implementation, version 1.1. Currently, the following resources are used:

code-size: 2000 Byte - with no optimizations enabled; the result may differ at ~700 Byte

stack-size: 60 Byte - no optimizations and tests are done until now

- LIN-Slave-driver used ressources

The ressources used by the LIN-Slave-driver are subject to be changed when adapting the LIN-driver to the final internal implementation, version 1.1. Currently, the following resources are used:

code-size: 1001 Byte - with full, manually implemeted optimizations enabled;

stack-size: 40 Byte - no optimizations and tests are done until now

- LIN-Driver with UART6-support:

The ressources used by the UART6-LIN-drivers are smaller than their pendants. The stack-size will be the same, the code-size is shrinked with a minimum of ~100 Bytes with no optimizations.

- Relation to LIN-specification-version

The LIN-driver will be numbered with the same scheme concerning the main numbers as the official LIN-specification. The actual LIN-driver uses the standard described in the specification V1.1.

If there are changes to the used LIN-driver-version without changing implementations defined in higher-numbered LIN-specifications, the version of the NEC LIN-drivers will be changed to e.g. 1.11.

**[MEMO]**

## Facsimile Message

From:

Name

Company

Tel.

FAX

Address

Although NEC has taken all possible steps to ensure that the documentation supplied to our customers is complete, bug free and up-to-date, we readily accept that errors may occur. Despite all the care and precautions we've taken, you may encounter problems in the documentation. Please complete this form whenever you'd like to report errors or suggest improvements to us.

*Thank you for your kind support.*

### North America

NEC Electronics Inc.  
Corporate Communications Dept.  
Fax: 1-800-729-9288  
1-408-588-6130

### Hong Kong, Philippines, Oceania

NEC Electronics Hong Kong Ltd.  
Fax: +852-2886-9022/9044

### Asian Nations except Philippines

NEC Electronics Singapore Pte. Ltd.  
Fax: +65-250-3583

### Europe

NEC Electronics (Europe) GmbH  
Technical Documentation Dept.  
Fax: +49-211-6503-274

### Korea

NEC Electronics Hong Kong Ltd.  
Seoul Branch  
Fax: 02-528-4411

### Japan

NEC Semiconductor Technical Hotline  
Fax: 044-548-7900

### South America

NEC do Brasil S.A.  
Fax: +55-11-6465-6829

### Taiwan

NEC Electronics Taiwan Ltd.  
Fax: 02-2719-5951

I would like to report the following error/make the following suggestion:

Document title: \_\_\_\_\_

Document number: \_\_\_\_\_ Page number: \_\_\_\_\_

If possible, please fax the referenced page or drawing.

Document Rating	Excellent	Good	Acceptable	Poor
Clarity	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Technical Accuracy	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

