To our customers,

Old Company Name in Catalogs and Other Documents

On April 1st, 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 1st, 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.

RENESAS

Application Note

NEC V85x - 78K0 - 78K0S

Standalone NEC LIN-driver for Master and Slave

Single-Chip Microcontroller

Document No. U17137EE1V0AN00
Date Published April 2004

NOTES FOR CMOS DEVICES

① PRECAUTION AGAINST ESD FOR SEMICONDUCTORS

Note:

Strong electric field, when exposed to a MOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop generation of static electricity as much as possible, and quickly dissipate it once, when it has occurred. Environmental control must be adequate. When it is dry, humidifier should be used. It is recommended to avoid using insulators that easily build static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work bench and floor should be grounded. The operator should be grounded using wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions need to be taken for PW boards with semiconductor devices on it.

② HANDLING OF UNUSED INPUT PINS FOR CMOS

Note:

No connection for CMOS device inputs can be cause of malfunction. If no connection is provided to the input pins, it is possible that an internal input level may be generated due to noise, etc., hence causing malfunction. CMOS devices behave differently than Bipolar or NMOS devices. Input levels of CMOS devices must be fixed high or low by using a pull-up or pull-down circuitry. Each unused pin should be connected to V_{DD} or GND with a resistor, if it is considered to have a possibility of being an output pin. All handling related to the unused pins must be judged device by device and related specifications governing the devices.

③ STATUS BEFORE INITIALIZATION OF MOS DEVICES

Note:

Power-on does not necessarily define initial status of MOS device. Production process of MOS does not define the initial operation status of the device. Immediately after the power source is turned ON, the devices with reset function have not yet been initialized. Hence, power-on does not guarantee out-pin levels, I/O settings or contents of registers. Device is not initialized until the reset signal is received. Reset operation must be executed immediately after power-on for devices having reset function.

- The information in this document is current as of 20.04, 2004. The information is subject to change without notice. For actual design-in, refer to the latest publications of NEC Electronics data sheets or data books, etc., for the most up-to-date specifications of NEC Electronics products. Not all products and/or types are available in every country. Please check with an NEC sales representative for availability and additional information.

- No part of this document may be copied or reproduced in any form or by any means without prior written consent of NEC Electronics. NEC Electronics assumes no responsibility for any errors that may appear in this document.

- NEC 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 NEC Electronics products listed in this document or any other liability arising from the use of such NEC Electronics products. No license, express, implied or otherwise, is granted under any patents, copyrights or other intellectual property rights of NEC Electronics or others.

- Descriptions of circuits, software and other related information in this document are provided for illustrative purposes in semiconductor product operation and application examples. The incorporation of these circuits, software and information in the design of customer's equipment shall be done under the full responsibility of customer. NEC Electronics no responsibility for any losses incurred by customers or third parties arising from the use of these circuits, software and information.

- While NEC Electronics endeavors to enhance the quality, reliability and safety of NEC Electronics products, customers agree and acknowledge that the possibility of defects thereof cannot be eliminated entirely. To minimize risks of damage to property or injury (including death) to persons arising from defects in NEC Electronics products, customers must incorporate sufficient safety measures in their design, such as redundancy, fire-containment and anti-failure features.

- NEC Electronics products are classified into the following three quality grades: “Standard”, “Special” and “Specific”.

The "Specific" quality grade applies only to NEC Electronics products developed based on a customer-designated “quality assurance program” for a specific application. The recommended applications of NEC Electronics product depend on its quality grade, as indicated below. Customers must check the quality grade of each NEC Electronics product before using it in a particular application.

"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.

"Special": 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, life support systems and medical equipment for life support, etc.

The quality grade of NEC Electronics products is “Standard” unless otherwise expressly specified in NEC Electronics data sheets or data books, etc. If customers wish to use NEC Electronics products in applications not intended by NEC Electronics, they must contact NEC Electronics sales representative in advance to determine NEC Electronics 's willingness to support a given application.

Notes: **1.** " NEC Electronics" as used in this statement means NEC Electronics Corporation and also includes its majority-owned subsidiaries.

2. " NEC Electronics products" means any product developed or manufactured by or for NEC Electronics (as defined above).

M8E 02.10

Regional Information

Some information contained in this document may vary from country to country. Before using any NEC product in your application, please contact the NEC office in your country to obtain a list of authorized representatives and distributors. They will verify:

- Device availability
- Ordering information
- Product release schedule
- Availability of related technical literature
- Development environment specifications (for example, specifications for third-party tools and components, host computers, power plugs, AC supply voltages, and so forth)
- Network requirements

In addition, trademarks, registered trademarks, export restrictions, and other legal issues may also vary from country to country.

NEC Electronics America Inc.
Santa Clara, California
Tel: 408-588-6000
800-366-9782
Fax: 408-588-6130
800-729-9288

NEC Electronics (Europe) GmbH
Duesseldorf, Germany
Tel: 0211-65 03 1101
Fax: 0211-65 03 1327

Sucursal en España
Madrid, Spain
Tel: 091- 504 27 87
Fax: 091- 504 28 60

Succursale Française
Vélizy-Villacoublay, France
Tel: 01-30-67 58 00
Fax: 01-30-67 58 99

Filiale Italiana
Milano, Italy
Tel: 02-66 75 41
Fax: 02-66 75 42 99

Branch The Netherlands
Eindhoven, The Netherlands
Tel: 040-244 58 45
Fax: 040-244 45 80

Branch Sweden
Taeby, Sweden
Tel: 08-63 80 820
Fax: 08-63 80 388

United Kingdom Branch
Milton Keynes, UK
Tel: 01908-691-133
Fax: 01908-670-290

NEC Electronics Hong Kong Ltd.
Hong Kong
Tel: 2886-9318
Fax: 2886-9022/9044

NEC Electronics Hong Kong Ltd.
Seoul Branch
Seoul, Korea
Tel: 02-528-0303
Fax: 02-528-4411

NEC Electronics Singapore Pte. Ltd.
Singapore
Tel: 65-6253-8311
Fax: 65-6250-3583

NEC Electronics Taiwan Ltd.
Taipei, Taiwan
Tel: 02-2719-2377
Fax: 02-2719-5951

Table of Contents

Chapter 1 Preface 11

Chapter 2 Introduction. 12

Chapter 3 LIN Specification Details 13
3.1 Intention 13
3.2 The Protocol 13
3.2.1 Overview 13
3.2.2 Frame-dividing 14
3.3 Master Frame Layout 15
3.3.1 SyncHBreak 15
3.3.2 SyncField 16
3.3.3 Identifier 16
3.4 Slave Frame Layout 17
3.5 Protocol Frames. 17

Chapter 4 LIN Master-Driver: NEC V850 19
4.1 Introduction 19
4.2 LIN-Master Overview 19
4.3 List of Used Files 19
4.3.1 Hardware.h 20
4.3.2 UART.h 22
4.3.3 LIN.h 24
4.3.4 M_Master.h 27
4.4 M_Master.c 28
4.4.1 startLin 28
4.4.2 stopLin 29
4.4.3 sendBusToStop 30
4.4.4 initHardware 31
4.4.5 initUART 32
4.4.6 initTimer 33
4.4.7 initScheduleTable 34
4.4.8 initIDLengthTable 36
4.4.9 startTimer 38
4.4.10 stopTimer 39
4.4.11 setUARTForSyncBreak 40
4.4.12 setUARTOnNormalSpeed 41
4.4.13 scheduleSending 41
4.4.14 sendData 44
4.4.15 Interrupt SioTxInt 44
4.4.16 Interrupt SioRxInterrupt 46
4.4.17 TimerCompInterrupt 49

Chapter 5 LIN-Slave Driver 78K0 51
5.1 Introduction 51
5.2 LIN-Slave Overview 51
5.2.1 Receiving non-standard-format SyncHBreak-Field 51
5.3 List of Used Files 52
5.3.1 Hardware.h 53
5.3.2 UART.h 56
5.3.3 LIN.h 58
5.3.4 M_Slave.h 60
5.4 Functions of the LIN-Slave Driver 61
5.4.1 startLIN 61
5.4.2 stopLin 62

5.4.3 sendBusToStop . . . 63
5.4.4 initHardware . . . 64
5.4.5 initUART . . . 65
5.4.6 initTimer . . . 66
5.4.7 initDataTable . . . 67
5.4.8 initDataTableLength . . . 68
5.4.9 calculateChecksum . . . 69
5.4.10 sendData . . . 70
5.4.11 startTimer . . . 71
5.4.12 stopTimer . . . 72
5.4.13 setUARTOnNormalSpeed . . . 73
5.4.14 setUARTForSyncBreak . . . 74
5.4.15 scheduleSending . . . 74
5.4.16 Interrupt SioTxInterrupt . . . 76
5.4.17 Interrupt SioRxInterrupt . . . 77
5.4.18 Interrupt ExternalInterrupt . . . 82
5.4.19 TimerCompInterrupt . . . 83

Chapter 6 LIN-Master Driver 78K0 . . . 85
6.1 Intention . . . 85
6.2 Realisation . . . 85

Chapter 7 Differences to the LIN-Master-Driver Using LIN-UART6 . . . 87
7.1 Intention . . . 87
7.2 LIN-UART6 - short overview . . . 87
7.3 List of Adaptions . . . 87
7.3.1 Sending SyncHBreak-Frames . . . 87
7.3.2 Reacting on WakeUp-SIgnals . . . 87
7.4 Use of changes in the NEC-LIN_Master driver . . . 88
7.4.1 Changes to LIN_m.h . . . 88
7.4.2 Changes to UART_m.h . . . 88
7.4.3 Changes to m_Master.c . . . 88

Chapter 8 Differences to the LIN-Slave Driver using LIN-UART6 . . . 89
8.1 Intention . . . 89
8.2 LIN-UART6 - short overview . . . 89
8.3 List of Adaptions . . . 89
8.3.1 Receiving SyncHBreak-Frames . . . 89
8.3.2 Receiving Sync-Fields . . . 90
8.3.3 Reacting on Wake-Up-Signals . . . 90
8.4 Use of changes in the NEC-LIN_Slave driver . . . 91
8.4.1 Changes to LIN.h . . . 91
8.4.2 Changes to UART.h . . . 91
8.4.3 Changes to m_slave.c . . . 92

Appendix A Application for the V850 LIN-Master Driver . . . 97

Appendix B Application for the LIN-Slave Driver . . . 99

Appendix C LIN-Emulation Using CANoe with Option LIN . . . 103

Appendix D Network Overview . . . 111

Appendix E Software Included . . . 113

Appendix F Technical Details, Resources, Implementations . . . 115

List of Figures

Figure 3-1: Overview of the whole LIN-Message-Frame 14
Figure 3-2: SyncHBreak-Field 15
Figure 3-3: Sync-Field 16
Figure 3-4: Identifier 16
Figure 3-5: Response by Slave upon Valid Identifier 17
Figure 4-1: Header-file Hardware_m.h 20
Figure 4-2: Header-file UART.h 22
Figure 4-3: Header-file LIN.h - Initialization 25
Figure 4-4: Header-file Master_M.h 27
Figure 4-5: Function startLin 28
Figure 4-6: Function stopLin 29
Figure 4-7: Function sendBusToStop 30
Figure 4-8: Function initHardware 31
Figure 4-9: Function initUART 32
Figure 4-10: Function initTimer 33
Figure 4-11: Function initScheduleTable 35
Figure 4-12: Function initIDLengthTable 37
Figure 4-13: Example of an Initialized Receive-Table 38
Figure 4-14: Function startTimer 38
Figure 4-15: Function stopTimer 39
Figure 4-16: Function setUARTForSyncBreak 40
Figure 4-17: Function setUARTOnNormalSpeed 41
Figure 4-18: Function scheduleSending 42
Figure 4-19: Function sendSyncBreak 44
Figure 4-20: Function interrupt sioRxInterrupt 45
Figure 4-21: SioRxInterrupt re-reading just sent data 46
Figure 4-22: SioRxInterrupt - Response Part 1 46
Figure 4-23: SioRxInterrupt - inputCt == 3 47
Figure 4-24: Interrupt-Receive - store received Data 48
Figure 4-25: Function interrupt TimerCompareInterrupt 49
Figure 5-1: Header-file Hardware.h 53
Figure 5-2: Header-file UART.h 56
Figure 5-3: Header-file LIN.h - Definitions 58
Figure 5-4: Header-file Slave.h 60
Figure 5-5: Function startLin 61
Figure 5-6: Function stopLin 62
Figure 5-7: Function sendBusToStop 63
Figure 5-8: Function initHardware 64
Figure 5-9: Function initUART 65
Figure 5-10: Function initTimer 66
Figure 5-11: Function initDataTable 67
Figure 5-12: Function initIDLengthTable 68
Figure 5-13: Function calculateChecksum 69
Figure 5-14: Routine sendData 70
Figure 5-15: Function startTimer 71
Figure 5-16: Function stopTimer 72
Figure 5-17: Function setUARTOnNormalSpeed 73
Figure 5-18: Function setUARTForSyncBreak 74
Figure 5-19: Function scheduleSending 75
Figure 5-20: Function interrupt sioRxInterrupt 76
Figure 5-21: Reception of a Framing-Error 77
Figure 5-22: Schedule-position WAIT_FOR_SYNCH_FIELD 78
Figure 5-23: Reception of a Framing-Error 79
Figure 5-24: External Interrupt-Function 82
Figure 5-25: Function Interrupt TimerCompareInterrupt 83

Figure 7-1: Changes to LIN_m.h 88
Figure 7-2: Changes to UART_m.h 88
Figure 8-1: Changes to LIN.h for usage of UART6 91
Figure 8-2: Changes to UART.h for usage of UART6 91
Figure 8-3: Changes to m_slave.c - Receive-Routine 92
Figure 8-4: Changes to initHardware following UART6-Functionality 93
Figure 8-5: Changes to interrupt TimerCompInterrupt-Routine 94
Figure 8-6: Modifications for UART6 - initHardware 95
Figure A-1: Definitions for application-use of LIN-Master-driver 97
Figure A-2: Initializations of standard LIN-Master-driver routines 97
Figure A-3: Main-routine of LIN-Master-Application 98
Figure B-1: Slave-Driver Settings in the Slave LIN-header-file 99
Figure B-2: External Definitions in the Slave-Application 99
Figure B-3: Initialization of Variables by Settings and Initial Calls after Reset 100
Figure B-4: Cyclic Called LIN-main-routine 101
Figure C-1: Overview of the LIN-Emulation-environment, Master active 103
Figure C-2: Variables used by the Master-Emulation 104
Figure C-3: Pre-start-routines 104
Figure C-4: On Start-routine 105
Figure C-5: Routine onTimer 105
Figure C-6: Protocol-overview using the LIN-Slave-driver 106
Figure C-7: Frame-detail of used LIN-slave-driver 106
Figure C-8: Definitions of Internally Used Variables 107
Figure C-9: Lin-Slave-routine onPreStart in Emulation 107
Figure C-10: Emulated Slave-routine on Start 108
Figure C-11: Protocol-overview using the LIN-Master-driver 109
Figure C-12: Frame-detail using the LIN-Master-driver 109
Figure C-13: Bus protocol using the NEC LIN-Master-driver 110
Figure D-1: Network-outline for Test Purposes 111
Figure E-1: Delivered Files Used by the LIN-Master-driver 113
Figure E-2: Delivered Files Used by the LIN-Slave-driver 113

List of Tables

Table 3-1: Relation between Identifier Value and Length of the Response 17
Table 4-1: Hardware.h Related Settings 21
Table 4-2: UART.h Related Settings 23
Table 4-3: LIN.h Initialization Related Settings 26
Table 5-1: Hardware.h Related Settings 54
Table 5-2: UART.h Related Settings 57
Table 5-3: LI.h Related Settings 59

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.

Chapter 2 Introduction

This document describes the stand-alone LIN-drivers for a *Master-Slave-System* using **NEC** microcontrollers.
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 “LIN Master-Driver: NEC V850” on page 19.

The last starting with Chapter 7 “Differences to the LIN-Master-Driver Using LIN-UART6” on page 87 will identify the differences between the Master- and the Slave-part using NEC’s 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.

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

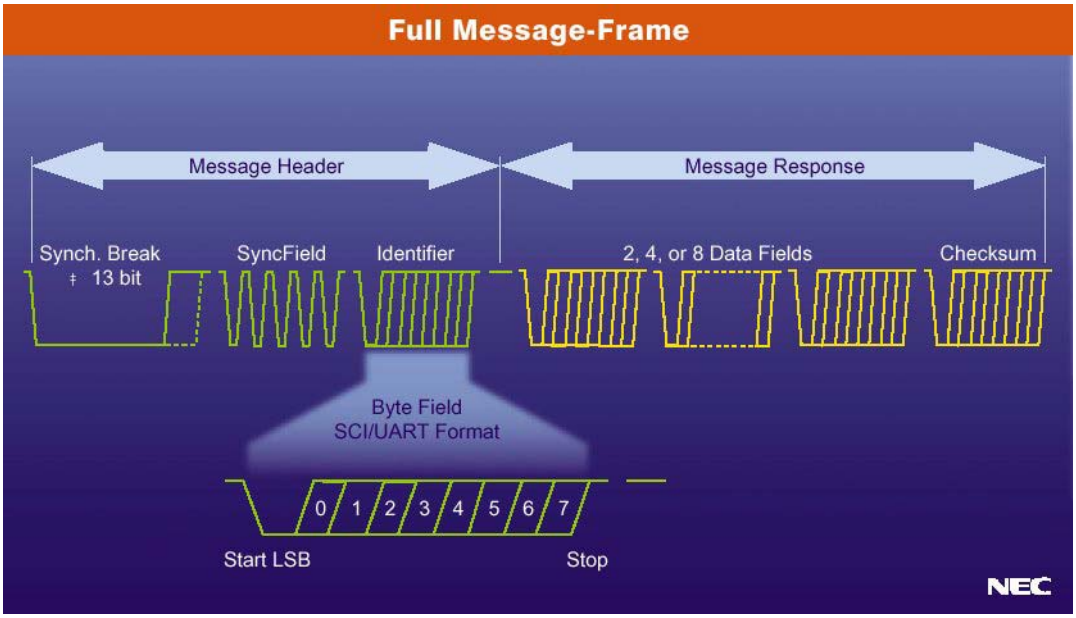
3.2.1 Overview

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 attached ECU's rather than sending a specific address 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

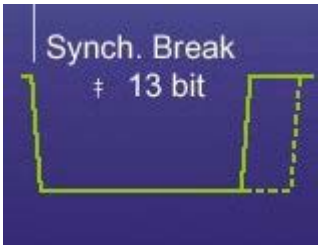
The request by the Master is split into three fields that sends parts from the table of the schedule to the bus. These three fields are called the ***SyncHBreak***, the ***SyncField*** and the ***Identifier***.
The sum of these three fields is called the LIN-header, which has to be received correctly 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 13 Bit 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 Bit Times. 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:

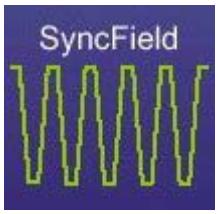
Figure 3-2: SyncHBreak-Field



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

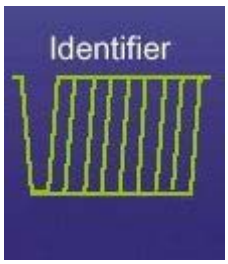


3.3.3 Identifier

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, “Relation between Identifier Value and Length of the Response,” on page 17.

Table 3-1: Relation between Identifier Value and Length of the Response

ID4	ID5	Number of data-fields
0	0	2 Byte
1	0	2 Byte
0	1	4 Byte
1	1	8 Byte

3.4 Slave Frame Layout

The layout of frames for the LIN-slaves just contains the ***Response-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
- Wake-Up

[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:

Figure 4-1: Header-file Hardware_m.h

```
// 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 catched 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 ASYNCHRONUS_SERIAL_RECEIVE_INTERRUPT_VECTOR    0x00000210
 #define ASYNCHRONUS_SERIAL_TRANSMIT_INTERRUPT_VECTOR   0x00000220
 #define TIMER_ZERO_COMPARE_HIT_INTERRUPT_VECTOR        0x00000150
```

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.

Caution: 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.

Caution: 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!

Table 4-1: Hardware.h Related Settings

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_PRESCALER_VAL_0	Prescaler-value for PRM0	0xFF	√
TIMER_PRESCALER_VAL_1	Prescaler-value for PRM1	0x00	√
ASYNCHRONUS_SERIAL_RECEIVE_INTERRUPT_VECTOR	Address, where INT for Serial receive is vectored	0x00000210	√
ASYNCHRONUS_SERIAL_TRANSMIT_INTERRUPT_VECTOR	Address, where INT for Serial 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:

Figure 4-2: Header-file UART.h

```
// 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_1BRGMC01
      #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
      #ifdef         CSIM_REGISTER_PRESENT
      #define        SERIAL_OPERATION_MODE_REGISTER CSIM0
      #endif //_UART_M_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 portabiltiy 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.

Table 4-2: UART.h Related Settings (1/2)

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-register	0x04	√

Table 4-2: UART.h Related Settings (2/2)

Statement	Comment	Definition	To be adjusted by customer depending on the device used
Framing_Error	Definition for the Framing-Error in the ASIS-register	0x02	√
Overrun_Error	Definition for the Overrun-Error in the ASIS-register	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.

A figure of the LIN.h-header-file is shown at the next page:

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

```
// 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 scheduleTabelEntrys!! */
      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
```

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.

Table 4-3: LIN.h Initialization Related Settings

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 schedule 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 schedule 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 definition are neither device-specific nor hardware-related.

Figure 4-4: Header-file Master_M.h

```
// used for declaration 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
```

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.

Figure 4-5: Function startLin

```
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 initalization 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;
}
```

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 wakeup-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.

Figure 4-6: Function stopLin

```
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);
}
```

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.

Figure 4-7: Function sendBusToStop

```
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;
   }
}
```

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.

Figure 4-8: Function initHardware

```
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;
}
```

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.

Figure 4-9: Function initUART

```
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;
 SERIC0 = 0;
 STIC0 = 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;
}
```

The UART-mode is started by setting the baudrate-register to the desired bus speed, 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 µsec, 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 µsec.

Figure 4-10: Function initTimer

```
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;


}
```

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 prescaler/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.

Figure 4-11: Function initScheduleTable

```
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;
}
```

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

Figure 4-12: Function initIDLengthTable

```
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 choosing */
 /*               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
```

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.

Figure 4-14: Function startTimer

```
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 1 msec turnaround.      */
 /*=========================================================================*/

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

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.

Figure 4-15: Function stopTimer

```
void stopTimer (void) {
 /*====================================================================*/
 /* FunctionName: stopTimer                                            */
 /* IN/OUT    : -/ -                                                   */
 /* Description: The stopTimer function is called to halt all Timer-    */
 /*              dependend 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;
}
```

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.

Figure 4-16: Function setUARTForSyncBreak

```
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;
}
```

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.

Figure 4-17: Function setUARTOnNormalSpeed

```
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;
}
```

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 state machine is started, which then calls the functions for SyncBreak, SyncField and identifier.

Figure 4-18: Function scheduleSending

```
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
```

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 ms.
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.

Figure 4-19: Function sendSyncBreak

```
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;

}
```

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.

Figure 4-20: Function interrupt sioRxInterrupt

```
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

}
```

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)

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

```
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
```

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.:

Figure 4-22: SioRxInterrupt - Response Part 1

```
 } 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;
```

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.

Figure 4-23: SioRxInterrupt - inputCt == 3

```
{
 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 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 Figure 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.

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

```
 } 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-adress
          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
}
```

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 coasted receive-data and the formerly calculated checksum are used to detect a checksum-overrun. One count is added to the result if an overrun 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.

Figure 4-25: Function interrupt TimerCompareInterrupt

```
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
}
```

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-filed.

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-bus system.
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:

Figure 5-1: Header-file Hardware.h

```
// 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        INTSR20_vect
  #define INTTX_VECT        INTST20_vect
  #define EXT_INT_VECT      INTP0_vect
```

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.

Caution: 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.

Table 5-1: Hardware.h Related Settings (1/2)

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 1 ms-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-Definition 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	√

Table 5-1: Hardware.h Related Settings (2/2)

Statement	Comment	Definition	To be adjusted by customer depending on the device used
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:

Figure 5-2: Header-file UART.h

```
// 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
```

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 portabiltiy 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.

Table 5-2: UART.h Related Settings

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 existent 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:

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

```
// 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];
```

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.

Table 5-3: LI.h Related Settings

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.

Figure 5-4: Header-file Slave.h

```
  #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_SYNCH_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_IDENTFIER_PARITY    ;
    extern bit ERROR_INCONSISTENT_SYNCH  ;
    extern bit ERROR_FIRST_TIME          ;
    extern bit ERROR_RECEIVE             ;

// additional values

  #define RAMP_COMPARE_VALUE 0x0F
  #define MAX_CHECKSUM 0x00FF
  #define SYNC_FIELD    0x55
```

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

Figure 5-5: Function startLin

```
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 ();
}
```

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().*

Figure 5-6: Function stopLin

```
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;
}
```

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.

Figure 5-7: Function sendBusToStop

```
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;
  }
}
```

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.

Figure 5-8: Function initHardware

```
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_SYNCH_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 {
  }
}
```

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:
initUART	----------	----------:	----------

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

Figure 5-9: Function initUART

```
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
}
```

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 and 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 µsec. This provides a reasonable good resolution to generate a timer-tick for application and LIN-driver.

Figure 5-10: Function initTimer

```
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 µsec for all application-needs
  TIMER_CAPTURE_COMPARE_REGISTER = TIMER_BASE;
}
```

5.4.7 *initDataTable*

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

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

Figure 5-11: Function initDataTable

```
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];
}
```

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.

Figure 5-12: Function initIDLengthTable

```
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
```

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.

Figure 5-13: Function calculateChecksum

```
unsigned char calculateChecksum (void){
  /*====================================================================*/
  /* FunctionName: calculateChecksum                                    */
  /* IN/OUT      : -/unsigned 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);
}
```

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.

Figure 5-14: Routine sendData

```
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;

}
```

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.

Figure 5-15: Function startTimer

```
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;

}
```

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.

Figure 5-16: Function stopTimer

```
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;

}
```

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.

Figure 5-17: Function setUARTOnNormalSpeed

```
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;
}
```

Caution: 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...
setUARTForSyncBreak	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.

Figure 5-18: Function setUARTForSyncBreak

```
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 = BAUDRATE_SYNC_BREAK; // set to fastest speed
  START_UART;
}
```

Caution: 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:
scheduleSending	----------	----------	sendData

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.

Figure 5-19: Function scheduleSending

```
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 in-driver 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;
      }
  }
}
```

- If the state of the LIN-driver (*stateMode)* equals STATE_MODE_INIT and the data inside the application 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 chosen as parameter. Then, the standard data to be sent including the checksum, which is calculated by an separate 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. Dependent 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.

Figure 5-20: Function interrupt sioRxInterrupt

```
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;

}
```

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 wake-up 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:

Figure 5-21: Reception of a Framing-Error

```
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
     STOP_UART;
   } else {
   }
// correct data was received
```

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 RxD. 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.

Figure 5-22: Schedule-position WAIT_FOR_SYNCH_FIELD

```
 } 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-adress
          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 {
```

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

Figure 5-23: Reception of a Framing-Error

```
// following the iFP, action for received data is caused
   switch (inFramePosition){
// SHBreak-field is considered
    case IN_FRAME_POS_WAIT_FOR_SYNCH_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_SYNCH = TRUE;
// wait for next SHBreak-Field
          inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNCH_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_SYNCH_BREAK;
        } else {
          if (sReceiveData == IDENTIFIER2){
            FLAG_RECEIVE_MASTER_DATA = TRUE;
            stateMode = STATE_MODE_INIT;
        } else {
// wrong data occurred
            FLAG_ERROR_OCCURED = TRUE;
            ERROR_IDENTFIER_PARITY = TRUE;
// wait for next SHBreak-Field
            inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNCH_BREAK;
          }
        }
      break;
// if something funny occurs
      default:
// wait for next SHBreak-Field
        inFramePosition = IN_FRAME_POS_WAIT_FOR_SYNCH_BREAK;
      break;
   }// end of switch
  } // end of else - 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 run-time 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.

Caution: 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.

Figure 5-24: External Interrupt-Function

```
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;

}
```

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.

Figure 5-25: Function Interrupt TimerCompareInterrupt

```
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;
  }
}
```

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

[MEMO]

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 source-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 especailly 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 fullfil 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(SyncBreakTransmitTrigger) in the register ASICL is set to TRUE, so the SyncH-Break-Bit is sent instantaniously. 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.
Additionaly, the settings for the registers BRGC and CKSR are introduced.

Figure 7-1: Changes to LIN_m.h

```
#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
```

7.4.2 Changes to UART_m.h

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

Figure 7-2: Changes to UART_m.h

```
// 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
```

7.4.3 Changes to m_Master.c

The changes to the file m_Master.c addresses how the SyncHBreak-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 SyncHBreak-field and then again reinstalling the original LIN-bus speed afterwards, only the length of the SyncHBreak-field to be sent is specified.
IIf the state to send the SyncHBreak-field is reached, this command is executed instantly, and the SyncHBreak-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 Wake-Up 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 accordantly to fit the real LIN-baudrate
- Reacting on Wake-Up-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 fulfil 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 Wake-Up-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

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

```
// 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;
```

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

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

```
// 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
```

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

Figure 8-3: Changes to m_slave.c - Receive-Routine

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

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.

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

```
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_SYNCH_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 {
    }
}
```

- interrupt *TimerCompInterrupt*

Figure 8-5: Changes to interrupt TimerCompInterrupt-Routine

```
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;
  }
}
```

Due to the Free-Running-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:

Figure 8-6: Modifications for UART6 - initHardware

```
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;
     PU1.4 = 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();
}
```

[MEMO]

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 ms 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.

Figure A-1: Definitions for application-use of 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;
```

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.

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

```
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
```

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.

Figure A-3: Main-routine of LIN-Master-Application

```
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
```

The scheduled main-routine will check two Flags modified by the LIN-driver-software, the linFlag-Field.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:

Figure B-1: Slave-Driver Settings in the Slave LIN-header-file

```
// 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;
```

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.

Figure B-2: External Definitions in the Slave-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;
```

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.

Figure B-3: Initialization of Variables by Settings and Initial Calls after Reset

```
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 ();
```

All required settings for the table 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:

Figure B-4: Cyclic Called LIN-main-routine

```
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
}
```

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.

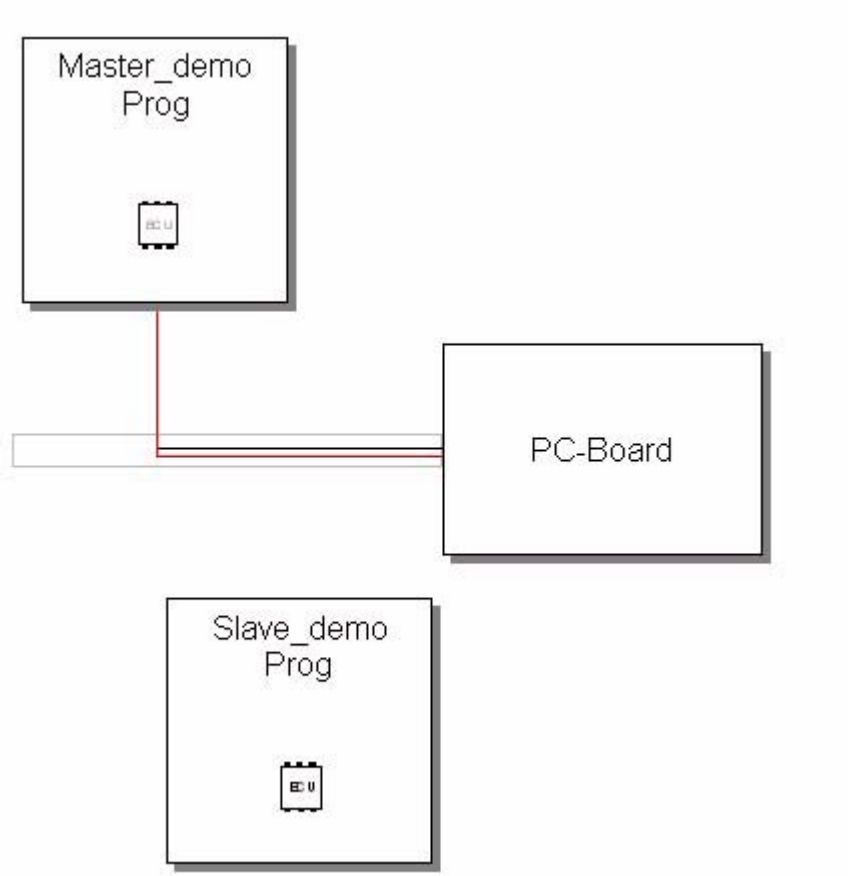
If the Slave receives the request by the LIN-Master, only data that has a “valid” stamp will be sent onto the LIN-bus.

[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



C.1 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:

Figure C-2: Variables used by the Master-Emulation

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

}
```

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

Figure C-3: Pre-start-routines

```
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();

}
```

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:

Figure C-4: On Start-routine

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

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

The routine for the timer itself contains the following code:

Figure C-5: Routine onTimer

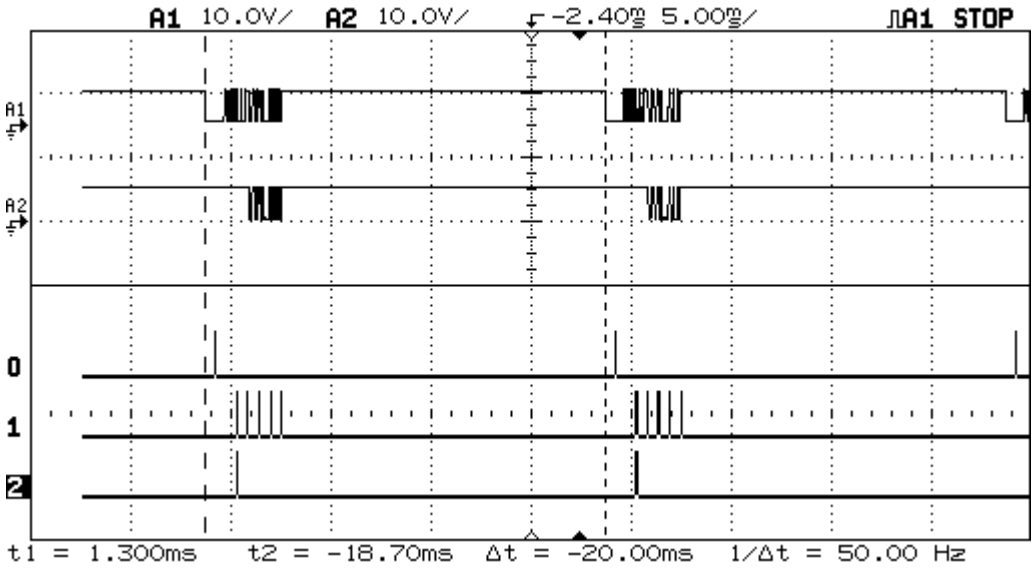
```
on timer changeSchedMode
{
  count++;


  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);

}
```

Using this example, two modes of the scheduler can be used. The different schedules can be invoked with the variable *schedulerMode*.
The new mode of the scheduler is set, the timer is set to a new cycle of 20 ms, 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.

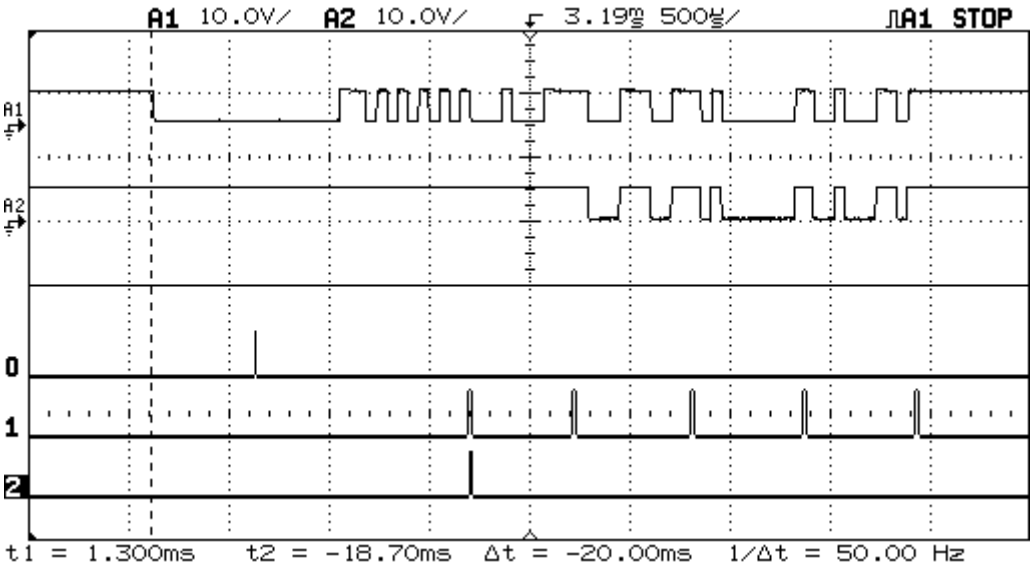
- Bus protocol as an overview

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



- Frame details

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



C.2 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

Figure C-8: Definitions of Internally Used Variables

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

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

Figure C-9: Lin-Slave-routine onPreStart in Emulation

```
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();

  LINSlSimulate (0x06);
  LINSlSimulate (0x20);
  LINSlSimulate (0x2F);
  LINSlSimulate (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();

}
```

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”:

Figure C-10: Emulated Slave-routine on Start

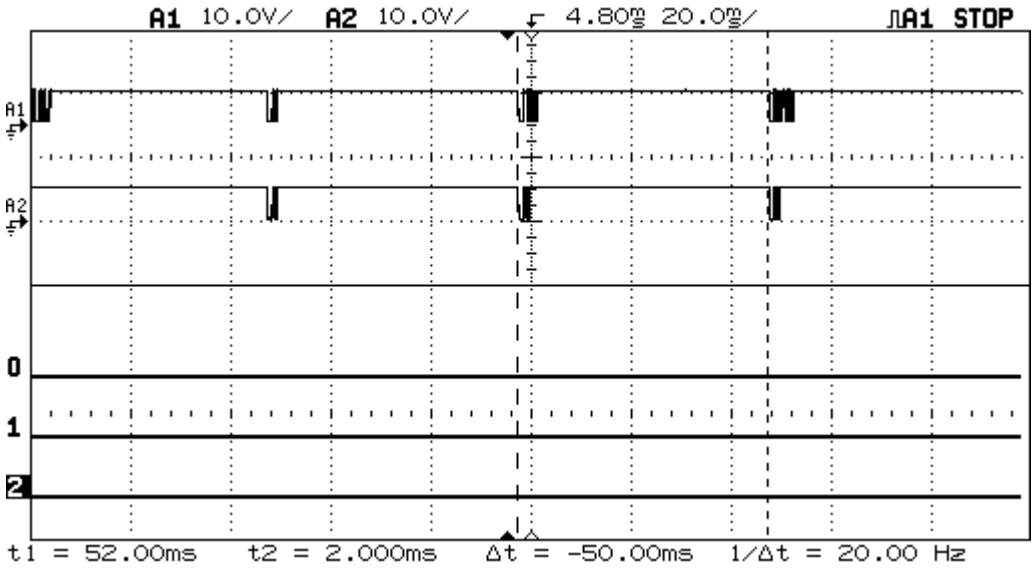
```
on start
{
  write("Use key >1< to select scheduler mode 1");
  write("Use key >2< to select scheduler mode 2");
}
```

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:

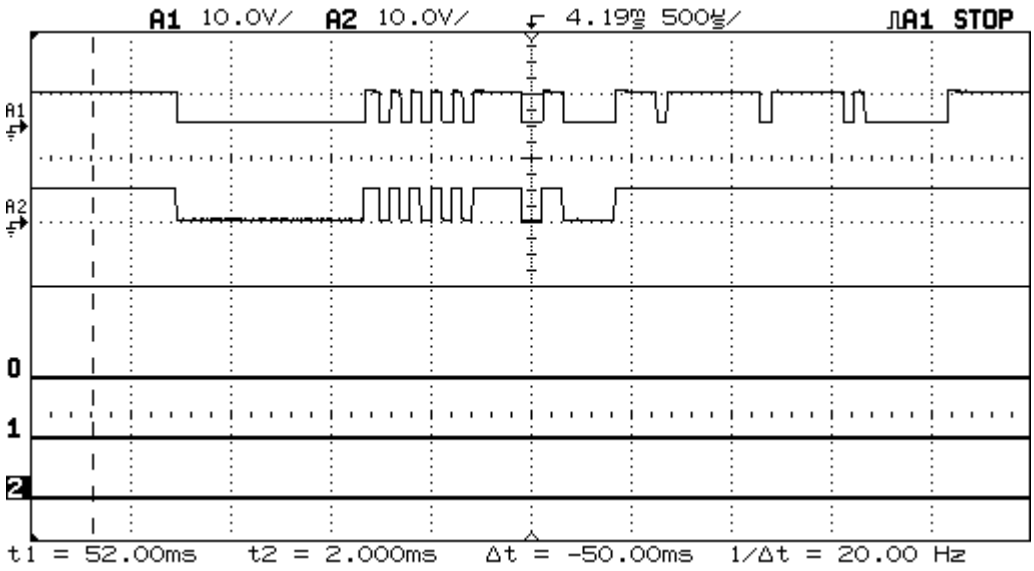
- Bus protocol as an overview

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



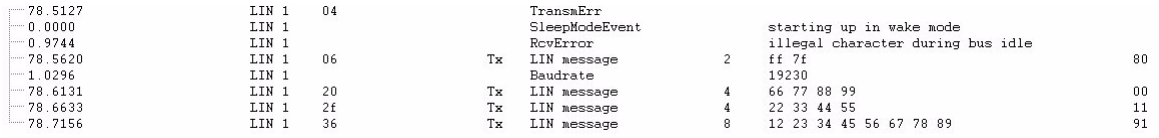
- Frame details

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



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

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



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

Appendix D Network Overview

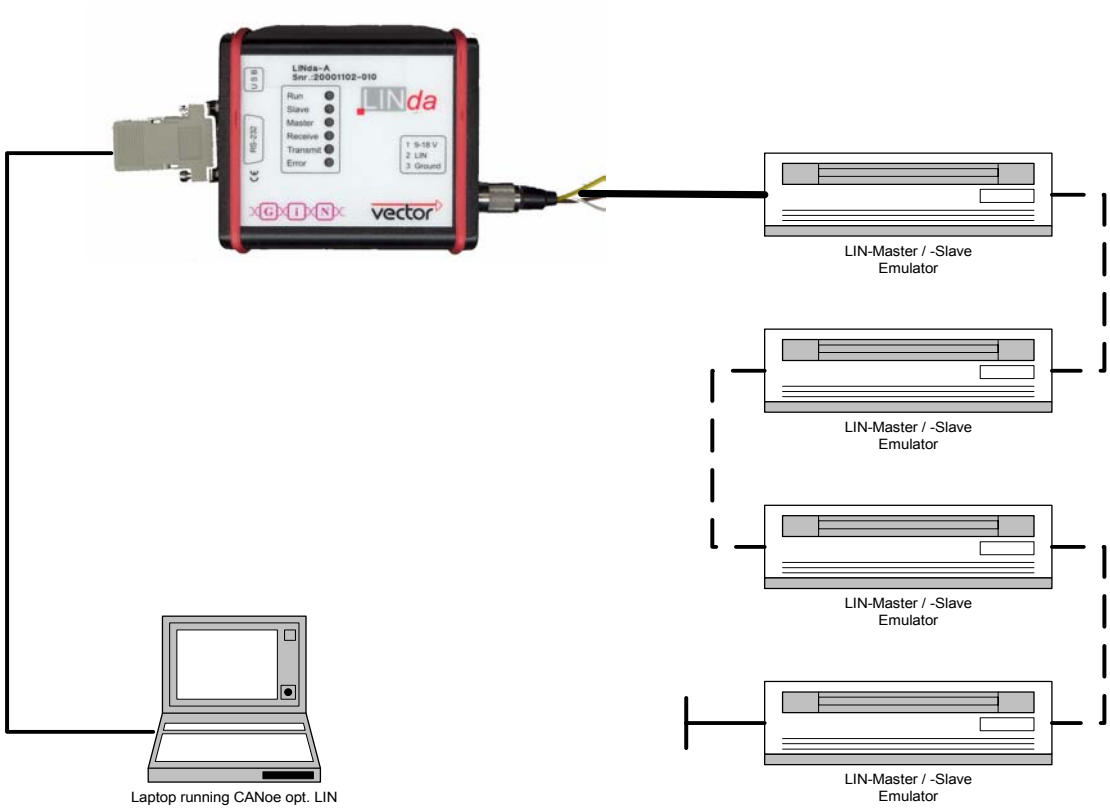
To ensure that both, the LIN-Master- and the LIN-Slave-driver are running properly, an emulated LIN-environment was implemented, which makes use of routines for a Master- and a Slave-driver.

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 D-1: Network-outline for Test Purposes



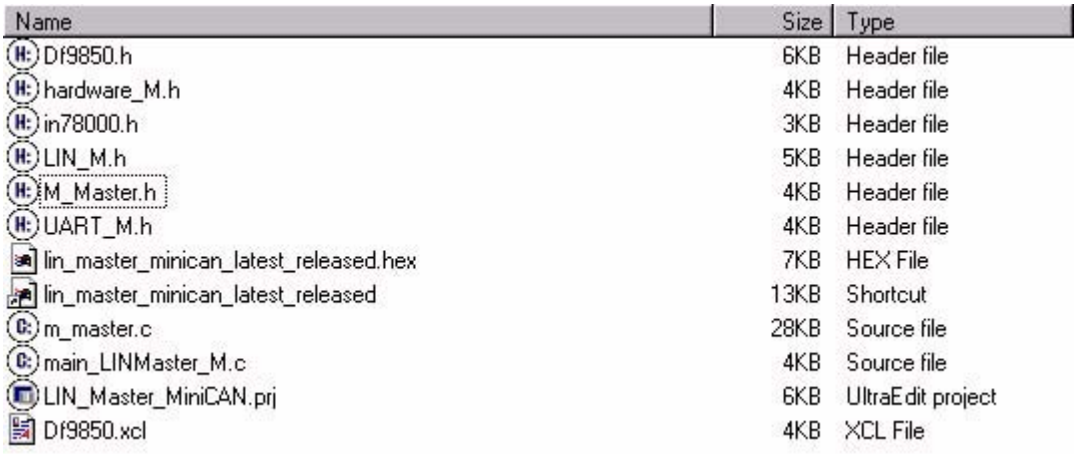
[MEMO]

Appendix E 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

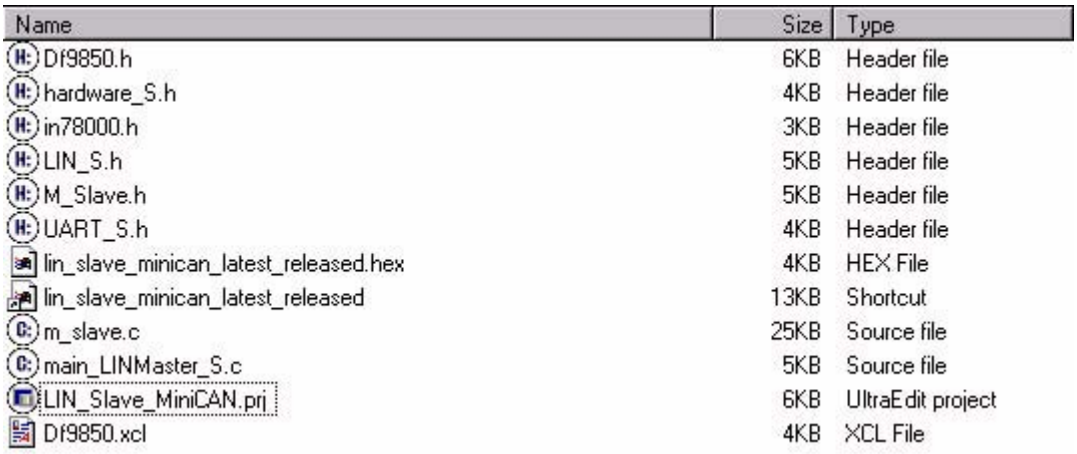
Figure E-1: Delivered Files Used by the LIN-Master-driver



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

- LIN-Slave

Figure E-2: Delivered Files Used by the LIN-Slave-driver



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

[MEMO]

Appendix F Technical Details, Resources, Implementations

- LIN-Master-driver used resources

The resources 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 resources

The resources 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 implemented optimizations enabled;
stack-size: 40 Byte - no optimizations and tests are done until now

- LIN-Driver with UART6-support:

The resources used by the UART6-LIN-drivers are smaller than their pendants. The stack-size will be the same, the code-size is shrunk 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]

NEC

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

Europe
NEC Electronics (Europe) GmbH
Market Communication Dept.
Fax: +49(0)-211-6503-1344

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

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

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

Asian Nations except Philippines
NEC Electronics Singapore Pte. Ltd.
Fax: +65-6250-3583

Japan
NEC Semiconductor Technical Hotline
Fax: +81- 44-435-9608

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	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐
Organization	☐	☐	☐	☐

[MEMO]