LIN (LOCAL INTERCONNECT NETWORK) SOLUTIONS

by Microcontroller Division Applications

INTRODUCTION

Many mechanical components in the automotive sector have been replaced or are now being replaced by intelligent mechatronical systems. A lot of wires are needed to connect these components. To reduce the amount of wires and to handle communications between these systems, many car manufacturers have created different bus systems that are incompatible with each other.

In order to have a standard sub-bus, car manufacturers in Europe have formed a consortium to define a new communications standard for the automotive sector. The new bus, called LIN bus, was invented to be used in simple switching applications like car seats, door locks, sun roofs, rain sensors, mirrors and so on.

The LIN bus is a sub-bus system based on a serial communications protocol. The bus is a single master / multiple slave bus that uses a single wire to transmit data.

To reduce costs, components can be driven without crystal or ceramic resonators. Time synchronization permits the correct transmission and reception of data. The system is based on a UART / SCI hardware interface that is common to most microcontrollers.

The bus detects defective nodes in the network. Data checksum and parity check guarantee safety and error detection.

As a long-standing partner to the automotive industry, STMicroelectronics offers a complete range of LIN silicon products: slave and master LIN microcontrollers covering the protocol handler part and LIN transceivers for the physical line interface. For a quick start with LIN, STMicroelectronics supports you with LIN software enabling you to rapidly set up your first LIN communication and focus on your specific application requirements.
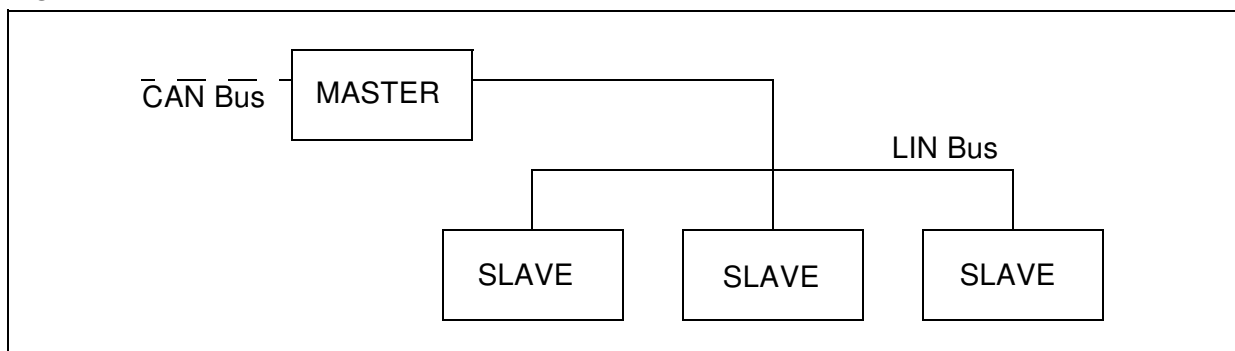
Figure 1. LIN Network Overview

Table of Contents

Table of Contents

1 LIN PROTOCOL

The aim of this chapter is to give an overview of the LIN protocol and concept. For detailed and up-to-date information please refer to the official LIN homepage: www.lin-subbus.org where you can register for the LIN specification package.

The LIN specification package consists of three parts:

■ The LIN protocol specification

■ The LIN configuration language description

■ The LIN API

The first part describes the LIN physical and data link layers. The second part describes the LIN configuration language. The LIN configuration language enables the user LIN network to be described in a file (how many nodes, how many frames, frame description, baudrate etc.). The goal of this specification is to ease communications between the parties involved in the development of a LIN network like car manufacturers and their suppliers. The third and last part is about the software implementation of the LIN protocol and specifies some points on how the software implementation has to be done.

The LIN concept uses a single master / multiple slave model. Only the master is able to initiate a communication. A LIN frame consists of a header and a response part. To initiate a communication with a slave the master sends the header part. If the master wants to send data to the slave it goes on sending the response part. If the master requests data from the slave the slave sends the response part.

Figure 2. Basics of LIN communication
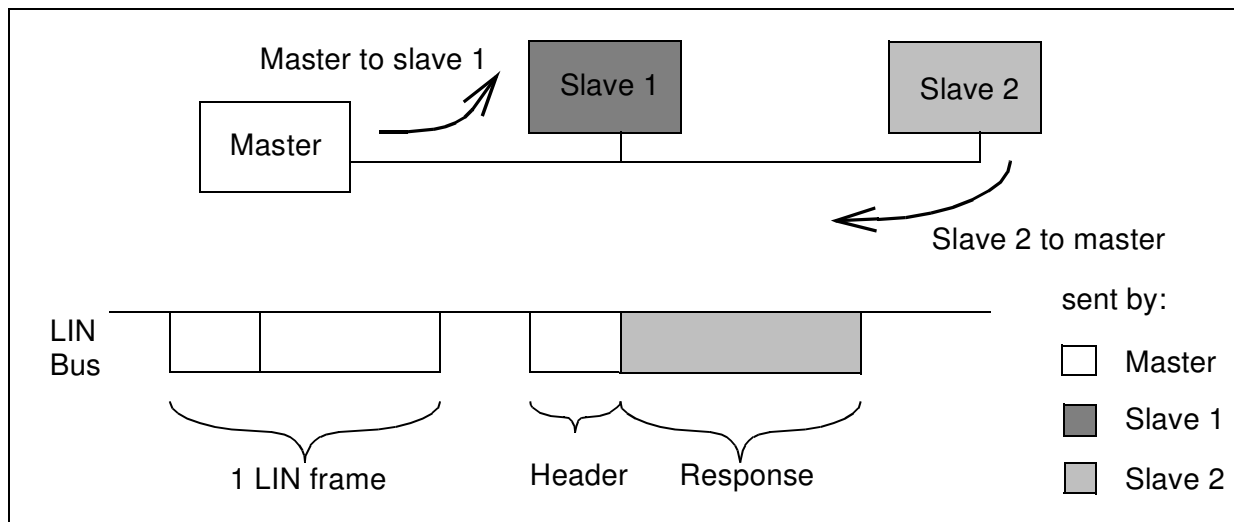
Direct communication between slaves is not possible. But as all nodes always listen to the bus, a master request can be used to handle slave-to-slave communications.

The LIN protocol is object-oriented and not address-oriented. The header contains the identifier which identifies the LIN frame and the data it contains. Different nodes may receive the same frame data.

The response part consists mainly of data of selectable length (1 to 8 bytes). The data are secured by an 8 bit checksum.

The LIN protocol is time-trigger oriented. The master periodically sends the same sequence of LIN frames. Each sequence, the master and the slaves update the data they send and receive. The sequence sent by the master may change depending on application events.

Example: The slave is a sensor measuring a analog value which is communicated to the master via LIN. The slave continuously measures its analog input independently from the LIN communication. In response to a master request (periodical) the slave sends the up-to-date/ last measured value of the analog input.

Figure 3. Time-triggered protocol
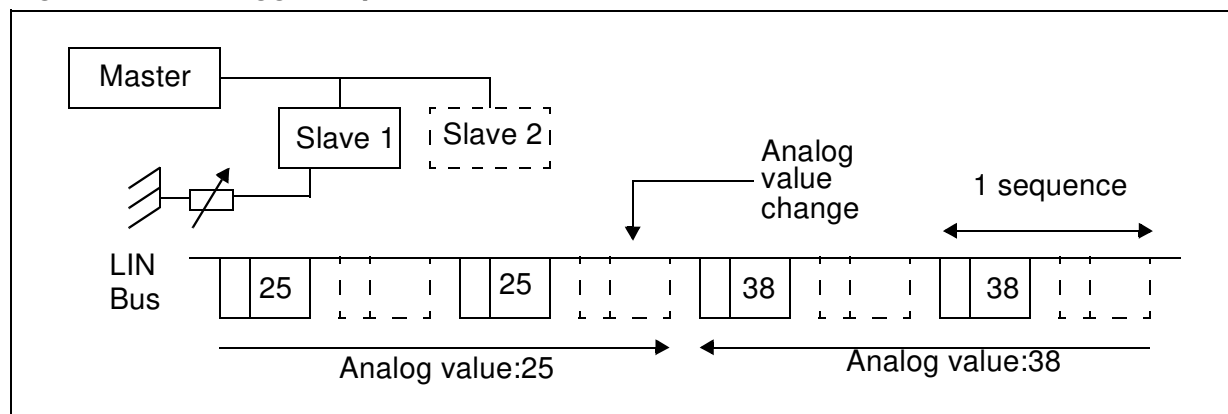
In order to achieve a good level of security, different mechanisms exist like parity bits on the identifier or checksum on data bytes.

One important feature of the protocol is to enable the slave MCUs to run with low cost oscillators such as an integrated RC oscillator provided that the accuracy is better than +/-15%. For this purpose the header contains a sync field byte consisting of the constant 0x55. This byte enables each slave to measure the master bit time and to synchronize its clock accordingly.

In order to obtain very low power consumption, the master is able to send a sleep frame. Any node can go into low power mode. To wake up the network, any node can send a so-called wake-up signal.

Figure 4. Sleep mode - wake-up
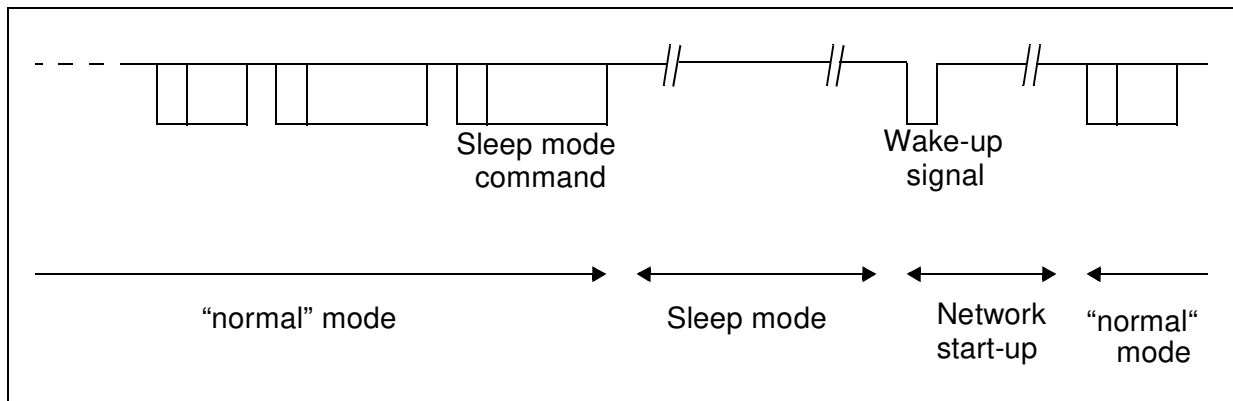
2 LIN PRODUCTS

A typical LIN node consists of a microcontroller for handling the LIN protocol and a LIN transceiver for interfacing the digital part and the physical line (see Figure 5 . LIN bus topology). STMicroelectronics offers both kind of products.

Figure 5. LIN bus topology
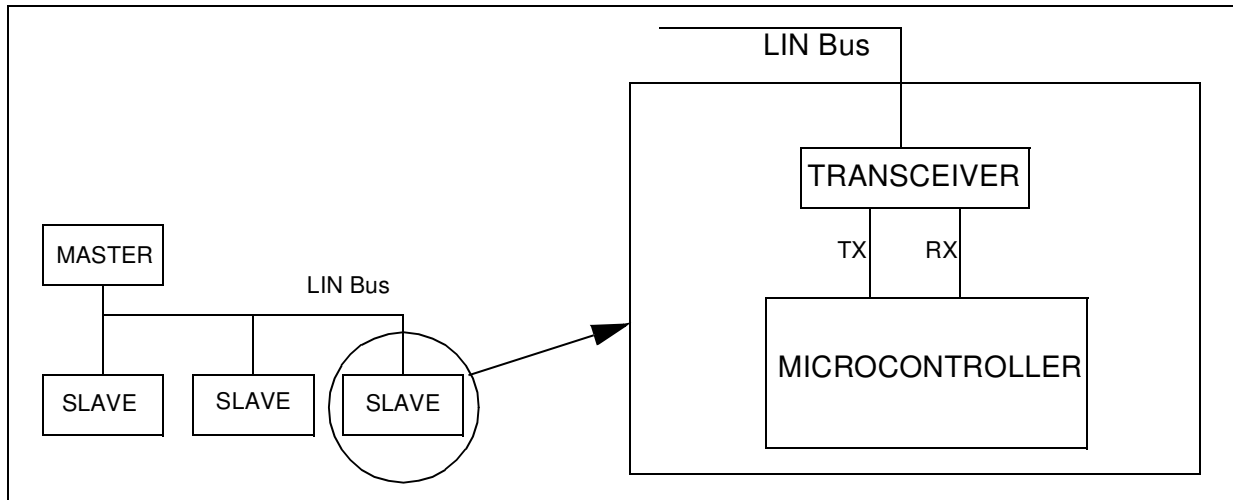
2.1 LIN MICROCONTROLLERS

STMicroelectronics offers a wide range of microcontrollers suitable for master and/or slave nodes.

2.1.1 LIN Slave MCUs

Table 1. Very low cost LIN slave MCUs - full software solution - Flash/ROM MCUs

Features	ST72104G1	ST72104G2	ST72216G1	ST72215G2	ST72254G1	ST72254G2
Program memory - bytes	4k	8k	4k	8k	4k	8k
RAM (stack) - bytes	256 (128)					
Peripherals	Watchdog timer, One 16-bit timer, SPI		Watchdog timer, One 16-bit timer, SPI, ADC	Watchdog timer, Two16-bit timers, SPI, ADC	Watchdog timer, Two 16-bit timers, SPI, I2C, ADC	
Operating Supply	3.2V to 5.5V					
CPU Frequency	Up to 8MHz (with oscillator up to 16 MHz)					
Operating Temperature	-40°C to +85°C (-40°C to +105/125°C optional)					
Packages	SO28 / SDIP32					

Table 2. Low cost LIN slave MCUs with hardware SCI - Flash/ROM MCUs

Features	ST72124J2	ST72314J2	ST72314J4	ST72314N2	ST72314N4	ST72334J2	ST72334J4	ST72334N2	ST72334N4
Program memory - bytes	8k	8k	16k	8k	16k	8k	16k	8k	16k
RAM (stack) - bytes	384 (256)	384 (256)	512 (256)	384 (256)	512 (256)	384 (256)	512 (256)	384 (256)	512 (256)
Peripherals	Watchdog timer, Two 16-bit timers, SPI, SCI								
	-	ADC							
Operating Supply	3.0V to 5.5V								
CPU Frequency	Up to 8MHz (with up to 16MHz oscillator)								
Operating Temperature	-40°C to +85°C (-40°C to +105/125°C optional)								
Packages	TQFP44/SDIP42		TQF64/SDIP56		TQF44/SDIP42		TQF64/SDIP56		

2.1.2 LIN Master MCUs

Table 3. EPROM/OTP/ROM MCUs

Features	ST72511R9	ST72511R7	ST72511R6
Program memory - bytes	60k	48k	32k
RAM (stack) - bytes	2048 (256)	1536 (256)	1024 (256)
Peripherals	Watchdog timer, Two 16-bit timers, 8-bit PWM ART, SPI, ADC		
Operating Supply	3.0V to 5.5V		
CPU Frequency	Up to 8MHz (with oscillator up to 16 MHz)		
Operating Temperature	-40°C to +85°C (-40°C to +105/125°C optional)		
Packages	TQFP64		

Table 4. Flash, ROM MCUs (ST7 core)

Features	ST72521R/M9	ST72521R/M7	ST72521R/M6	ST72521R5	ST72521R4
Program memory - bytes	60k	48k	32k	24k	16k
RAM (stack) - bytes	2048 (256)	1536 (256)	1024 (256)	768 (256)	512 (256)
Peripherals	Watchdog timer, 16-bit timers, SPI, SCI, 10-bit ADC, CAN				
	8-bit PWM ART, I2C		8-bit PWM ART		
Operating Supply	2.7V to 5.5V				
CPU Frequency	16 to 50kHz (with 32 to 100kHz oscillator), 500 to 8 MHz (with 1 to 16 MHz oscillator), 2 to 8 MHz (with 2 to 4 MHz oscillator and PLL)				
Operating Temperature	0°C to 70°C/-40°C to +85°C/-40°C to +105°C/-40°C to +125°C/				
Packages	TQFP80(M), TQFP64 (R)				

Table 5. Flash, ROM MCUs (ST9 core)

Features	ST92F150JD	ST92F150JC	ST92F124J	ST92F124
FLASH - bytes	128K	60/128K	60/128K	60/128K
RAM - bytes	6K	2/4K	2/4K	2/4K
EEPROM - bytes	1K	1K	1K	1K
Timers	2MFT, 2 EFT, STIM, WD	2MFT, 0/2 EFT, STIM, WD		
Serial Interface	2 SCI, SPI, I²C	1/2 SCI, SPI, I²C		
ADC	16 x 10 bits	8/16 x 10 bits		
Network Interface	2 CAN, J1850	CAN, J1850	J1850	-
Temp. Range	-40°C to 125°C or -40°C to 85°C			
Package	P/TQFP100	P/TQFP100 and TQFP64		

Note: The master MCUs listed above have all an on-chip CAN peripheral. This corresponds to the initial LIN concept: the LIN network as sub-network of CAN. However any other MCUs (listed above as slave for example) can be used to implement a master node.

2.2 LIN TRANSCEIVERS

To ensure the physical behaviour of the LIN bus STMicroelectronics also offers K-Line drivers and a dedicated LIN Bus Transceiver.

2.2.1 L9637 K-Line Transceiver

The L9637 K-Line transceiver is a monolithic integrated circuit containing standard ISO 9141 compatible interface functions. Its features are listed below.

- Operating power supply voltage range $4.5V \leq V_S \leq 36V$ (40V for transients)

- Reverse supply battery protected down to $V_S \geq -24V$

- Stand-by mode with very low current consumption IS_{SB} 1μA @Vcc 0.5V

- Low quiescent current in OFF condition IS_{OFF} = 120μA

- TTL compatible TX input

- Bidirectional K-I/O pin with supply voltage dependent input threshold

- Overtemperature shut down function selective to K-I/O pin

- Wide input and output voltage range $-24V \leq V_K \leq V_S$

- K output current limitation, typical I_K = 60mA

- Defined OFF output status in under voltage condition and V_S or GND interruption

- Controlled output slope for low EMI

- High input impedance for open V_S or GND connection

- Defined output on status of LO or RX for open LI or K inputs

- Defined K output off for TX input open

- Integrated pull up resistors for TX, RX and LO

- EMI robustness optimized

2.2.2 L9638 LIN Transceiver

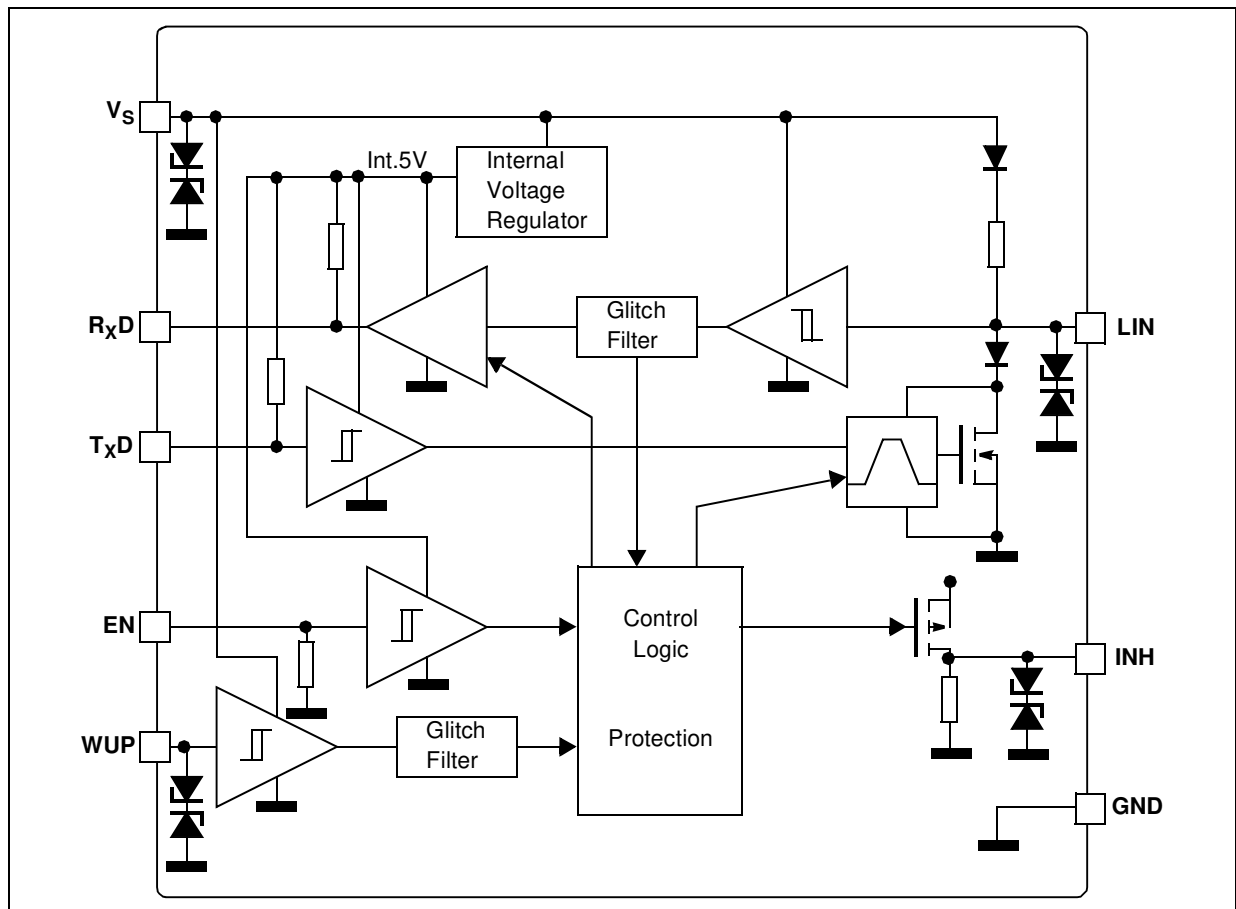The L9638 LIN transceiver is a monolithic integrated circuit fulfilling the LIN specification.

Its features are listed below.

- Wake up capability by:
  - LIN bus
  - External signal (edge triggered)

- System wake up functions:
  - Inhibit output
  - RxD output

- Quiescent current less than 25µA

- Fail safe functions implemented

- Pin compatible to L9637

Figure 6. L9638 block diagram

The following figure (Figure 7) shows a typical application of the L9638 LIN transceiver to-gether with the ST72124J microcontroller and a voltage regulator.

Figure 7. Application of L9638 with ST72124J Microcontroller
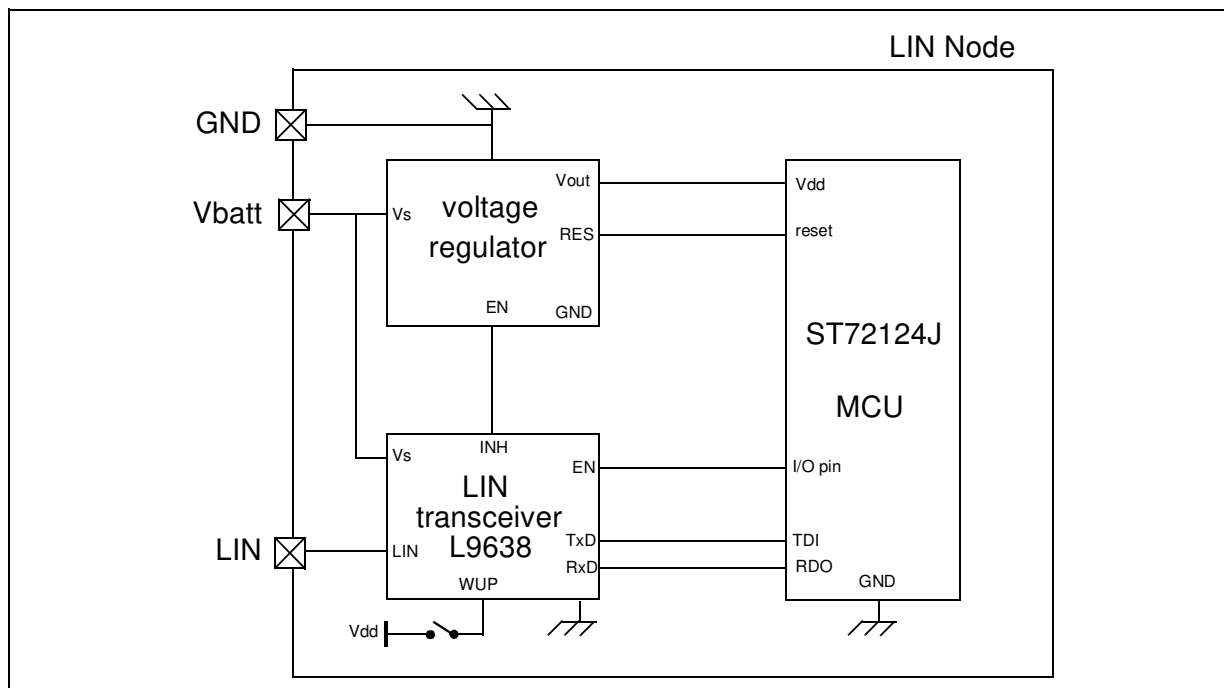
The voltage regulator supplies the application and generates the MCU reset signal. The LIN transceiver is the physical line interface between the SCI (Serial Communication Interface) TDI and RDO pins of the microcontroller and the LIN bus line. The microcontroller handles the LIN protocol and the application functions.

In order to lower power consumption the microcontroller is able to switch off the LIN trans-ceiver via the L9638 "EN" input. The transceiver is then able to switch off the voltage regulator by connecting its "INH" output to the "EN" input of the voltage regulator. In this state any ac-tivity on the LIN bus will cause the L9638 to wake the voltage regulator up via the "INH" pin. Another wake-up source is the "WUP" pin of the L9638 that can be used for contact sensing. Any edge on this pin will also wake up the regulator.

3 LIN SOFTWARE

Table 6. Software Overview

Version	2.0
Supported nodes	slave
Supported MCUs	all ST7 MCUs
LIN protocol specification revision	rev 1.2

The LIN standard includes the specification of the communication protocol but also the use of associated tools.

STMicroelectronics supports the development of your LIN application by providing ready-to-use LIN software. This software only handles the communication protocol part. For a complete software development tool solution you can contact LIN specialist third party tools manufacturers like VCT (http://www.vct.se) or Vector (http://www.vector-informatik.de).

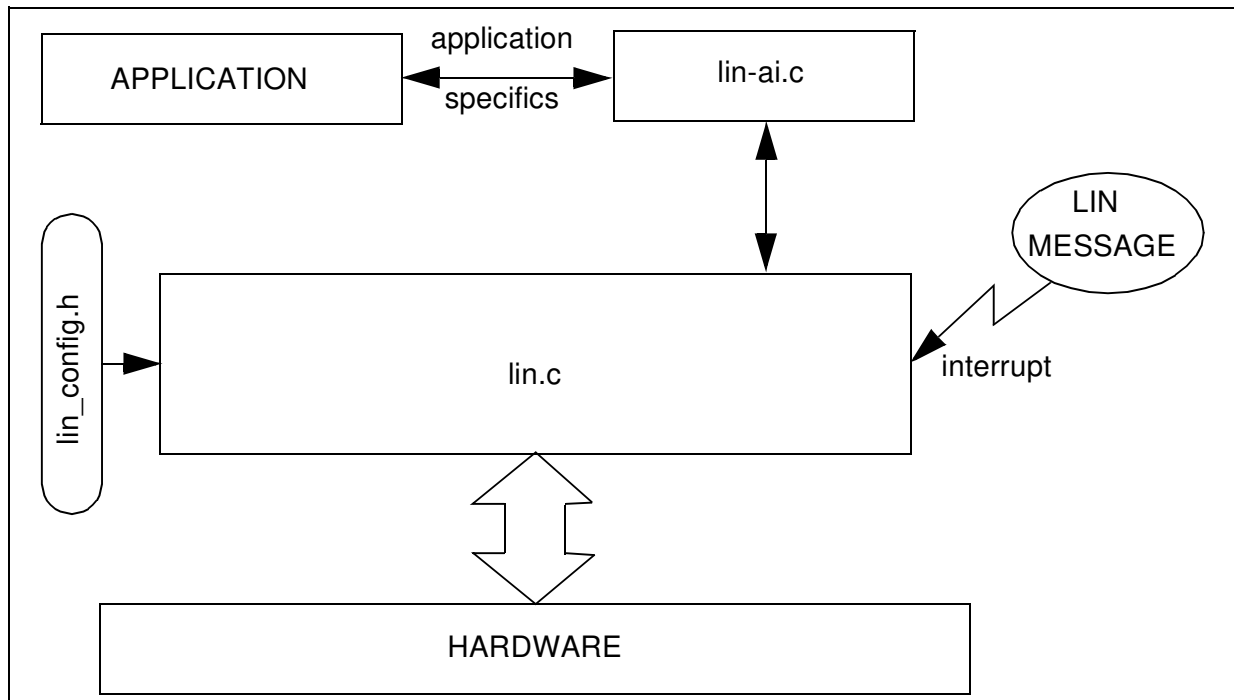The software supports LIN slave nodes. It consists of 4 files:

– lin.c/h: protocol handler code

– lin_config.c: LIN parameter configuration file

– lin_ai.c: application interface

A fifth additional file is delivered:

– lib.h: library file (macros, types definition)

The software supports the COSMIC C compiler.

Figure 8. Software Architecture

The software is interrupt driven. As soon as a message begins on the bus, an interrupt is generated and automatically handles the protocol. This means that the protocol handler is totally autonomous. It runs in the background. When the received frame is decoded by the software this is notified to the application in the lin_ai.c. In this file the user is able to customize the behaviour of the application upon reception of a frame for example.

3.1 TYPES AND MACRO DEFINITIONS: LIB.H

3.1.1 Debug settings

see 4.4.1 Timing considerations

3.1.2 Types

The software uses predefined types for 1-byte and 2-byte variables.

The name used for the one-byte type is "uByte".

The name used the two-byte type is "uWord".

Beside these, a third type is used to define two-byte variables that can also be accessed high byte or low byte only.

```
typedef union {
    unsigned int w_form;
    struct {
        unsigned char high, low;
    } b_form;
 } TwoBytes;
```

These types are defined in the lib.h file.

3.1.3 Macros

Three macros for register bit access are defined:
– SetBit(var,pos): Set bit "pos" of "var" variable
– ClrBit(var,pos): Clear bit "pos" of "var" variable
– ValBit(var,pos): Test bit "pos" of "var" variable. Return "0" if reset another value otherwise.

3.2 PROTOCOL HANDLER: LIN.P/H

The lin.p file contains the protocol handler. The user has no access to this file. It should simply be linked to the rest of the application. The lin.p file is encrypted and can therefore not be read but must be compiled and linked to the rest of the application.

The lin.h contains the definition of new types and the prototypes of the functions defined in lin.c.

3.2.1 Type definition

t_error

```
typedef enum {NO_ERROR,BIT_ERROR,ID_PARITY_ERROR,CHECKSUM_ERROR,
              NO_ID_MATCH,TIMEOUT_ERROR,DATA_RECEIVED,DATA_REQUEST,
              WAKE_UP,UART_ERROR,SYNCH_BREAK_ERROR}t_error;
```

defines the different error code values that the software functions are able to return.

t_message_direction

```
typedef enum {ID_DATAREQUEST,ID_DATASENT}t_message_direction;
```

This type is used internally in lin.c and in lin_drv.c.

t_id_list

```
typedef struct{
      uByte id;
      t_message_direction dir;
      uByte length;} t_id_list;
```

defines the type of an identifier list.

t_one_databyte_output

```
typedef struct{
      t_error error_code;
      uByte data_byte;} t_one_databyte_output;
```

defines a type of function return value consisting of an error code made up of error type and a data byte.

t_header

```
typedef struct
{
      uByte identifier;
      uByte length;} t_header;
```

A LIN frame consists of a header and a response. This typedef defines the header part type.

t_response

```
typedef struct
{
      uByte data[8];
      uByte checksum;} t_response;
```

A LIN frame consists of a header and a response. This typedef defines the response part type.

3.2.2 User interface functions

To integrate this software into your application software you have to link 3 or 4 functions defined in lin.c to your project.

Here are the prototypes of these functions:

Function	void LIN_Init(void)
Parameters	none
Return value	none
Description	LIN communication initialization Has to be called after reset

Function	void LIN_SendWakeUpSignal(void)
Parameters	none
Return value	none
Description	When the LIN network is in sleep mode and the application wants to wake it up, this function has to be called to send a wakeup frame.

Function	void LIN_Interrupt(void)
Parameters	none
Return value	none
Description	LIN interrupt service routine. You should link this function to the corresponding interrupt vector: timer or SCI interrupt according to the SCI hardware used (see 3.3 LIN Configuration File: lin_config.h).

Function	void LINTimeOut_Interrupt(void)
Parameters	none
Return value	none
Description	LIN timeout interrupt service routine (exists only in some cases). It only exists if SCI is defined or if the timer defined for timeouts is not the one used for the SCI emulation You should link this function to the corresponding timer interrupt vector (see 3.2.3 Timeout handling and 3.3 LIN Configuration File: lin_config.h).

3.2.3 Timeout handling

The LIN communication timeout handling is done using one output compare (OC) feature of the 16 bit timer. If the SCI communication is emulated using a 16 bit timer (uses one OC and one IC) the user can and should define the same timer for both the SCI communication and the timeout handling. In this case the timer is fully used for the LIN communication and the LIN software takes full control of the selected timer. If the user decides to configure 2 different timers for the SCI communication and the timeout handling or if the SCI is not emulated by a 16 bit timer; timeouts are handled using only one OC of a separate timer. In order to leave the unused OC/IC features free for the application, the application software is responsible for:

3.2.3.1 Initializing the timer

The application can initialize the resources that are not used by the LIN software as needed but is also responsible for initializing the OC defined for the timeouts handling in this way:

OC interrupt

The OCIE flag in Control Register 1 must be set. A write access must be made to the high byte of the defined Output Compare Register to disable the corresponding interrupt separately. The LIN software will enable it when needed. If the application is not using the other Output Compare, a write access must be done to the high byte of the corresponding output compare register to disable the feature.

Timer Clock Initialization

The timer clock/prescaler has to be defined to fulfil the LIN software requirements. The reason is that the timeouts to be handled must be smaller than a timer period otherwise the output compare cannot work properly. The LIN software takes the faster clock that respects this condition. The "prescaler" to be set can be calculated using the following equations. The smallest value (2,4, or 8) that fulfils the equation is the prescaler value.

Table 7. Equation for prescaler value calculation

SCI emulation	(FCPU/prescaler/BAUDRATE)*163 < 65536
SCI emulation & resynchr.	1,15*(FCPU/prescaler/BAUDRATE)*163 < 65536
On-chip SCI used	(FCPU/prescaler/BAUDRATE)*165 < 65536

Timer Interrupt Initialization

The LIN software will use the define OC feature in such a way that an OC interrupt will be generated if a LIN communication timeout occurs. The corresponding timer interrupt has to be defined by the application and the "LINTimeOut_Interrupt" function described above has to be inserted in it. The function checks if the OCxF flag is set and resets it after completion.

3.3 LIN CONFIGURATION FILE: LIN_CONFIG.H

The lin_config.h file allows the user to configure the LIN communication. The following symbols should be set:

#define FCPU

#define	FCPU
Description	Speed in MHz of the MCU internal frequency

#define UART_TIMERA

#define UART_TIMERB

#define SCI

#define	UART_TIMERA, UART_TIMERB or SCI
Description	Define <u>only one</u> of these options. Define UART_TIMERA or UART_TIMERB if you use a microcontroller without SCI and depending on the timer you want to use. The serial communication will be emulated by software with a timer. If you use a microcontroller with an on-chip SCI, define SCI. The hardware SCI peripheral will be used.

#define LIN_PORT_ADD

#define LIN_RX_PINNB

#define	LIN_PORT_ADD
Description	Needs to be defined only if UART_TIMERA or UART_TIMERB is defined. Set which pin is the LIN RX pin. This symbol should be set to the address of the data register of the I/O port linked to the LIN RX pin (PC3>set address of PCDR register).

#define	LIN_RX_PINNB
Description	Needs to be defined only if UART_TIMERA or UART_TIMERB is defined. Set which pin is the LIN RX pin. This symbol should be set to the number of the pin linked to the LIN RX pin (PC3>set 3).

Example: ST72-104/215/216/254 MCU family

	SCI emulated by Timer A		SCI emulated by Timer B	
	LIN RX on IC1	**LIN RX on IC2**	**LIN RX on IC1**	**LIN RX on IC2**
LIN RX is pin:	PB0 (Port B pin 0)	PB2 (Port B pin 2)	PC0 (Port C pin 0)	PC3 (Port C pin 3)
LIN_PORT_ADD	0x04	0x04	0x00	0x00
LIN_RXC_PINNB	0	2	0	3

#define UART_IC1

#define UART_IC2

#define	UART_IC1, UART_IC2
Description	Needs to be defined only if UART_TIMERA or UART_TIMERB is defined. Define <u>only one</u> of these options. Define UART_IC1 if the LIN RX pin is connected to the Input Capture 1 pin. Define UART_IC2 if the LIN RX pin is connected to the Input Capture 2 pin.

#define UART_OC1

#define UART_OC2

#define	UART_OC1, UART_OC2
Description	Needs to be defined only if UART_TIMERA or UART_TIMERB is defined. Define <u>only one</u> of these options. Define UART_OC1 if the LIN RX pin is connected to the Output Compare 1 pin. Define UART_OC2 if the LIN RX pin is connected to the Output Compare 2 pin.

#define TIMEOUT_TIMERA

#define TIMEOUT_TIMERB

#define	TIMEOUT_TIMERA, TIMEOUT_TIMERB
Description	Define <u>only one</u> of these options. Define which timer is used for the timeout handling. If you already defined a timer for the SCI emulation set the same timer for the timeout handling. This will optimize the use of resources: A timer is fully used for the LIN software and the second is free for the application.

#define TIMEOUT_OC1

#define TIMEOUT_OC2

#define	TIMEOUT_OC1, TIMEOUT_OC2
Description	Define <u>only one</u> of these options. Define which output compare of the previously defined timer is used for the timeout handling. **Warning**: If the same timer is used for both the SCI emulation and the timeouts handling, do not define the same output compare for both features.

#define BAUDRATE

#define	BAUDRATE
Description	LIN communication speed in bit/s

#define BRR

#define ExPR

#define	BRR, ExPR
Description	Needs to be defined only if SCI is defined. Set both registers to obtain the baudrate previously defined. BRR is the "baudrate register" and ExPR will set the same value for both the "extended receive prescaler division register" and the "extended transmit prescaler division register". Refer to the datasheet of the MCU you are using ("Serial Communication Interface" chapter).

Note: Here is a list of baudrates and corresponding settings of (BRR,ExPR) depending on the CPU frequency. Note that there are some small differences in the SCI prescaler between for example the ST72324/321/521 and ST72314/334 derivatives. As a result the value of BRR and ExPR may also depend on the MCU derivative.

Table 8. Example of (BRR,ExPR) values versus baudrate and f_{CPU}

	ST72x314/334/124			ST72x324/321/521		
f_{CPU}(*)	8MHz	4MHz	2MHz	8MHz	4MHz	2MHz
bit rate(bit/s) 4,8k	(0xD2,0x00)	(0xC9,0x00)	(0xC0,0x00)	(0xDB,0x00)	(0xD2,0x00)	(0xC9,0x00)
9,6k	(0xC9,0x00)	(0xC0,0x00)	(0x00,0x0D)	(0xD2,0x00)	(0xC9,0x00)	(0xC0,0x00)
19,2k	(0xC0,0x00)	(0x00,0x0D)	high quantification error	(0xC9,0x00)	(0xC0,0x00)	high quantification error
20k	(0x00,0x19)	high quantification error	high quantification error	(0x00,0x19)	high quantification error	high quantification error

(*): $f_{CPU}=f_{OSC}/2$ if PLL is not used

#define ID_TABLE_SIZE

#define	ID_TABLE_SIZE
description	number of LIN frames to be handled by the application. See Section 3.4

#define RESYNCH

#define	RESYNCH
description	Can only be defined if UART_TIMERA or UART_TIMERB has been defined.
	If the application is working with an inaccurate clock the LIN software is able to resynchronize to the master clock (refer to the LIN protocol). Define this symbol to activate this feature.

3.4 APPLICATION INTERFACE: LIN_AI.C

This file is the application interface and should be filled by the user. In this way the user can define the LIN communication of his application.

The lin_ai.c file consists of:

The ID_Table variable

You fill this variable to define the identifiers of the LIN frames that the application has to handle.

Each member of this list corresponds to a LIN frame and its corresponding identifier. Each member is of type *t_id_list* (See 3.2.1 Type definition) and has to be defined in the following way:

{*identifier, direction, data length*}

identifier represents the whole identifier byte including the parity bits.

direction represents the data flow direction, is of type *t_message_direction* and should therefore be set to *ID_DATAREQUEST* for data being requested by the master and to be sent by the application/slave or to *ID_DATASENT* for data being sent by the master to the application/slave.

data length represents the number of data bytes of the corresponding frame. It can be set between 0 to 8.

Note: The LIN protocol specification gives some advice concerning the coding of the data length through the ID5 and ID4 bits in the identifier byte. But this coding is no longer mandatory since revision 1.2 of the specification.

The number of members has to be entered in the lin_config.h file (See 3.3 LIN Configuration File: lin_config.h).

3 Notification functions

The LIN software is interrupt driven which means you do not have to poll any variables to handle LIN communication. When activity appears on the LIN bus, the LIN interrupt service routine is entered and starts decoding the LIN frame. Once the LIN software is able to notify an

event to the application, one of the 3 notification functions is called. These functions are delivered empty and by expanding them the user is able to fully define his application LIN communication. The 3 functions correspond to the 3 kinds of events:

Function	uByte * DataRequest_Notification(@tiny t_header *header)
Parameters	pointer to a variable of type t_header
Return value	pointer to an array
Description	This function is called on reception of a LIN header which requests data i.e. which is defined in the ID_Table with the qualifier "ID_DATAREQUEST". The function passes the received header as a pointer. The user has to complete this function and return a pointer to the array containing the data to be sent. Note: The data bytes are buffered by the LIN software just after this function call so that the user does not have to handle data sharing between the application and the LIN software.

| Function	void DataReceived_Notification(@tiny t_header *header, @tiny t_response *response)
Parameters	pointer to a variable of type t_header, pointer to a variable of type t_response
Return value	none
Description	This function is called upon the reception of a LIN header and response which was sent by the master to the application i.e. which is defined in the ID_Table with the qualifier"ID_DATASENT". The function passes the received header and response as a pointer. The user has to complete this function to handle the received data.

Function	void Error_Notification(t_error error_code)
Parameters	variable of type t_error
Return value	none
Description	This function is called upon the detection of an error. The function passes the type of error (See t_error type definition in 3.2.1 Type definition). The user has to complete this function if special action has to be taken in case of errors.

4 EXAMPLES

The purpose of this chapter is to give you an example, describing step by step how to use the LIN software. This example was defined to demonstrate the LIN software and not to show a typical LIN application.

The network consists of the master node and 2 slave nodes.

Figure 9. LIN network example
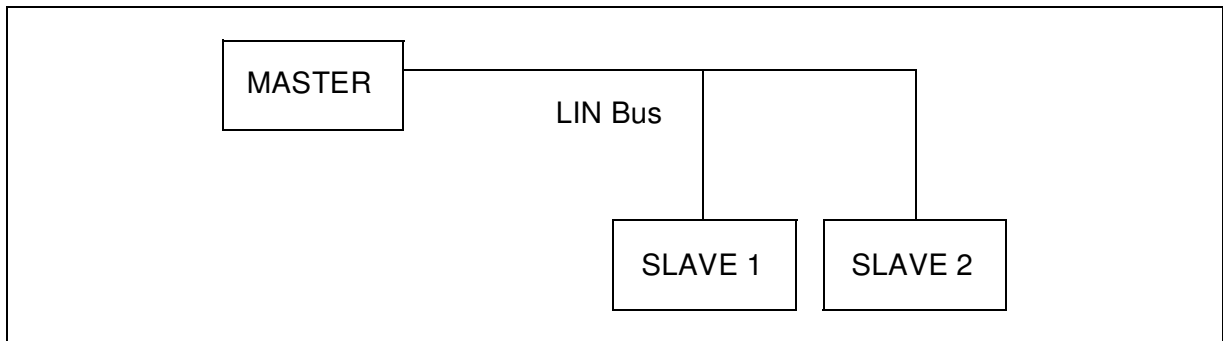
The application example is slave1.

The LIN communication consists of 4 LIN frames.

Frame name	Identifier ID[7..0]	Message Length	Data Direction
DataToSlave1	0x03	2	master to slave1
RequestToSlave1	0x20	4	slave1 to master
RequestToSlave2	0x76	2	slave2 to master
MasterReq (sleep mode command)	0x80(*)	2	master to all slaves

(*): The example is given for the LIN specification package 1.1 as many tools still don't support the 1.2 package.

Figure 10. LIN communication example
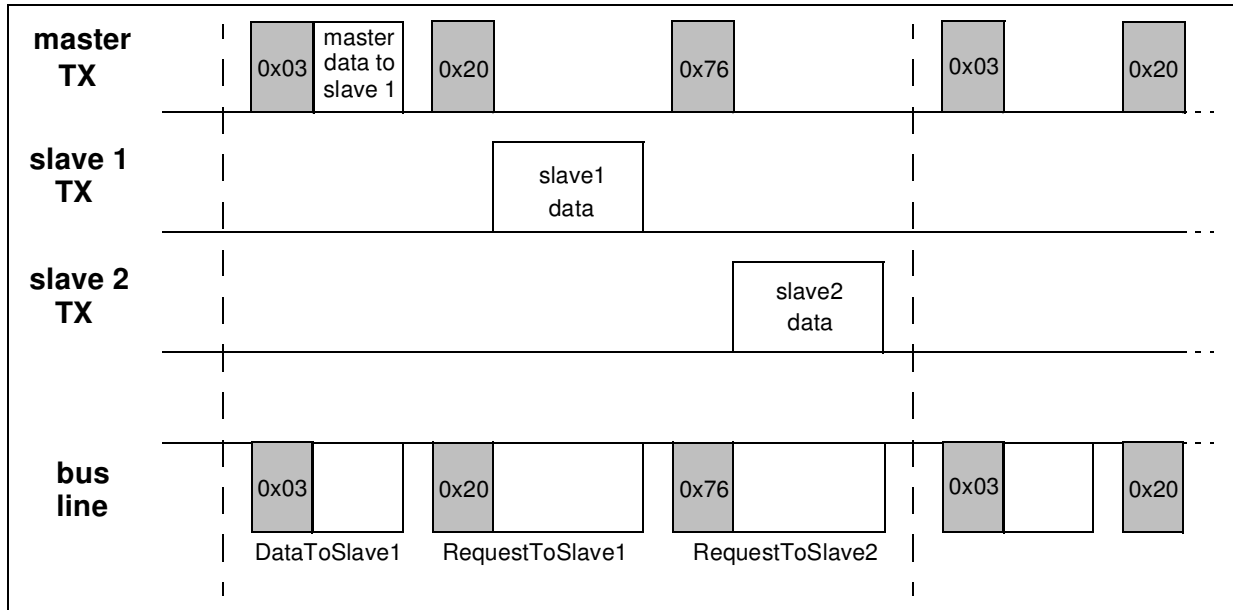
The figure above represents the LIN communication in "running" mode. The "SleepMo-deCOmmand" frame is sent by the master to set the LIN network in sleep mode and is therefore not part of the normal communication.

The baud rate is 9600 baud.

This communication corresponds to the following LIN description file:

Figure 11. LIN description file of the example

```
LIN_description_file ;
LIN_protocol_version = 1.1;
LIN_language_version = 1.1;
LIN_speed = 9.6 kbps;
Nodes {
Master: master, 1 ms, 0.1 ms;
Slaves: slave1, slave2;
}
Signals {
MasterDataToSlave1:16,5,master;
Slave1DataA:16,0,slave1;
Slave1DataB:8,0,slave1;
Slave1DataC:8,0,slave1;
Slave2DataA:8,0,slave2;
Slave2DataB:8,0,slave2;
}
Frames {
DataToSlave1:03,master {
MasterDataToSlave1,0;
}
RequestToSlave1:32,slave1 {
Slave1DataA,0;
Slave1DataB,16;
Slave1DataC,24;
}
RequestToSlave2:54,slave2 {
Slave2DataA,0;
Slave2DataB,8;
}
}
Schedule_tables {
Test_Schedule {
DataToSlave1 delay 10 ms;
RequestToSlave1 delay 15 ms;
RequestToSlave2 delay 15 ms;
    }
}
```

The rest of the example is divided into 2 main parts. The described example is first implemented on the ST72254G2 MCU which has no SCI peripheral. This part demonstrates the capability of the LIN software to emulate the whole LIN protocol using the embedded 16-bit timer. The second part describes the implementation on the ST72334N4 which has an SCI peripheral.

4.1 IMPLEMENTATION ON THE ST72254G2 - SOFTWARE EMULATED SCI

4.1.1 Step by Step Configuration

lin_config.h

Setting Group	Comments	Text line in lin_config.h
CPU operating frequency	An external resonator of 16 Mhz is used. As a result the internal CPU frequency is 8MHz. So "FCPU" is set to 8000000.	#define FCPU 8000000
Communication peripheral	The ST72254 has no SCI peripheral on chip. As a result the UART communication has to be emulated by one of the 16bit timer TimerA or TimerB. TimerA is chosen in this example. As a result "UART_TIMERA" is defined and "UART_TIMERB" and "SCI" are commented out.	#define UART_TIMERA //#define UART_TIMERB //#define SCI
LIN RX and LIN TX pin definition	Depending on the constraints of the board layout and according to the ST72254 pinout we select the Input Capture 1 pin and the Output Compare 1 pin for respectively the LIN RX and LIN TX signals. The Input Capture 1 pin of TimerA is linked to the Port B pin 0 (see pin description in the MCU datasheet). As a result LIN_RX_PINNB" is set to 0. Port B data register has the address 0x04. As a result "LIN_PORT_ADD" is defined to 0x04	#define UART_IC1 //#define UART_IC2 #define UART_OC1 //#define UART_OC2 #define LIN_PORT_ADD 0x04 #define LIN_RX_PINNB 0
Timeouts	The UART communication is already using one input capture and one output compare of TimerA. Setting TimerA for the timeout handling will complement the use of the timer. So we define "TIMEOUT_TIMERA". Output Compare 1 is already used by the UART communication (see UART_OC1) so we define the output compare 2 for the timeouts handling.	#define TIMEOUT_TIMERA //#define TIMEOUT_TIMERB //#define TIMEOUT_OC1 #define TIMEOUT_OC2
LIN baudrate	The example LIN baudrate is 9600 Baud.	#define BAUDRATE 9600

Setting Group	Comments	Text line in lin_config.h
SCI peripheral registers	An SCI peripheral is not used. As a result "BRR" and "ExPR" are commented out.	//#define BRR 0xC9 //#define ExPR 0x00
Number of LIN frames to be handled	The network communication consists of 4 LIN frames. The application example handles 3 of them: DataToSlave1, RequestToSlave1 and SleepModeCommand.	#define ID_TABLE_SIZE 3
Resynchronization	The application is working with a accurate clock (<2%). The resynchronisation feature is not needed. "RESYNCH" is commented out.	//#define RESYNCH

lin_ai.c

The first part of the lin_ai.c code is the definition of the LIN frames the application has to handle. Out of the 4 defined for the whole network the application handles the 3 following frames:

Frame name	Identifier ID[7..0]	Message Length	Data Direction
DataToSlave1	0x03	2	master to slave1
RequestToSlave1	0x20	4	slave1 to master
SleepModeCommand	0x80	2	master to all slaves

The corresponding setting of "*ID_Table*" is:

```
const t_id_list ID_Table[]=
                {
                {0x03, ID_DATASENT, 2},
                {0x20, ID_DATAREQUEST, 4},
                {0x80, ID_DATASENT, 8}
                };
```

The second part of the lin_ai.c code consists of 3 notification functions and is the kernel of the LIN communication. Filling the notification functions enables you to define the behaviour of the application upon a master data request (*DataRequest_Notification* function) on the reception of data from the master (*DataReceived_Notification* function) and when errors occur (*Error_Notification* function).

Master data request (*DataRequest_Notification*)

The application handles one data request from the master corresponding to the identifier 0x20 (*ID_Table*[1]). The application has to return a pointer to an array containing the data to be sent. In this example we declare an array "*slave_data[]*". This array will be shared between the

application updating it with the latest data and the LIN communication sending its content on request of the master. The corresponding code for the "*DataRequest_Notification*" function is:

```
extern uByte slave_data[];
uByte * DataRequest_Notification(@tiny t_header *header)
{
 if(header->identifier == ID_Table[1].id){
  return(slave_data);
 }
}
```

Data reception (*DataReceived_Notification*)

The application handles 2 "data" frames from the master corresponding to the identifier 0x03 and 0x80. The first frame is part of the "normal" communication. The second frame is a sleep command frame that can be sent by the master at any time to interrupt normal communication and set all nodes into low power mode.

On reception of the first frame the application saves the received data into the variable "*master_data[]*".

On reception of the sleep command frame the application sets the ST7 in Halt mode. Before setting the ST7 in Halt mode the wake-up sources should be acitivated. An application function, "PORTS_WakeUp_On()", is called. Two pins are configured as interrupt and will wake the ST7 up on a corresponding interrupt request. The first pin is the Laniards pin. As a result any bus activity will wake the application up. The second is an application pin that should be also able to wake up the application.

The corresponding code for the "*DataReceived_Notification*" function is:

```
extern uByte master_data[];
void DataReceived_Notification(@tiny t_header *header, @tiny t_response *re-
sponse)
{
 if(header->identifier == ID_Table[0].id){
  master_data[0]=response->data[0];
  master_data[1]=response->data[1];
 }
else if(header->identifier == ID_Table[2].id){
  PORTS_WakeUp_On();
  _asm("halt\n");
 }
}
```

Any activity on the bus will wake up the ST7 out of Halt mode. As soon as the ST7 is ready to execute the next instruction any incoming frame can be received.

The application can also be woken up by a sensor connected to one pin of the ST7 and then should also wake up the whole LIN network. For this a wake-up frame has to be sent, which is done by the "*LIN_SendWakeUpSignal()*" function. This function is therefore inserted in the application wake-up interrupt:

```
@interrupt void PORTS_0_Interrupt(void)
{
unsigned char i;
 i=250;
 while(i--);
 if(!(PADR&0xFE))
 {
  PAOR&=0xFE;
  LIN_SendWakeUpSignal();
 }
}
```

The software is ready!

4.2 IMPLEMENTATION ON THE ST72334N4 - HARDWARE SCI

4.2.1 Step by Step Configuration

4.2.1.1 lin_config.h

Setting group	Comments	Text line in lin_config.h
CPU operating frequency	An external 16 MHz resonator is used. As a result the internal CPU frequency is 8 MHz. So "FCPU" is set to 8000000.	#define FCPU 8000000
Communication peripheral	The ST72334N4 has an on chip SCI peripheral. As a result "SCI" is defined, "UART_TIMERA" and "UART_TIMERB" are commented out.	//#define UART_TIMERA //#define UART_TIMERB #define SCI
LIN RX and LIN TX pin definitions	These are not needed when using an on-chip SCI peripheral. All symbols are commented out.	//#define UART_IC1 //#define UART_IC2 //#define UART_OC1 //#define UART_OC2 //#define LIN_PORT_ADD 0x04 //#define LIN_RX_PINNB 0

Timeouts	One output compare of one timer is needed for the timeouts handling. We choose the output compare 1 of timer A. So "TIMEOUT_TIMERA" and "TIMEOUT_OC1" are defined.	#define TIMEOUT_TIMERA //#define TIMEOUT_TIMERB #define TIMEOUT_OC1 //#define TIMEOUT_OC2
LIN baudrate	The example LIN baudrate is 9600 Baud.	#define BAUDRATE 9600
SCI peripheral registers	See Table 8. Example of (BRR,ExPR) values versus baudrate and fCPU.	//#define BRR 0xC9 //#define ExPR 0x00
Number of LIN frames to be handled	The network communication consists of 4 LIN frames. The application example handles 3 of them: DataToSlave1, RequestToSlave1 and SleepModeCommand.	#define ID_TABLE_SIZE 3
Resynchronization	Not supported using the on-chip SCI peripheral. "RESYNCH" is commented out.	//#define RESYNCH

4.2.1.2 lin_ai.c

The first part of the lin_ai.c code is the definition of the LIN frames the application has to handle. Out of the 4 defined for the whole network the application handles the 3 following frames:

Frame name	Identifier ID[7..0]	Message Length	Data Direction
DataToSlave1	0x03	2	master to slave1
RequestToSlave1	0x20	4	slave1 to master
SleepModeCommand	0x80	2	master to all slaves

The corresponding setting of "*ID_Table*" is:

```
const t_id_list ID_Table[]=
                {
                {0x03, ID_DATASENT, 2},
                {0x20, ID_DATAREQUEST, 4},
                {0x80, ID_DATASENT, 8}
                };
```

The second part of the lin_ai.c code consists of 3 notification functions and is the kernel of the LIN communication. Filling the notification functions enables you to define the behaviour of the application upon a master data request (*DataRequest_Notification* function) upon the reception of data from the master (*DataReceived_Notification* function) and upon errors (*Error_Notification* function).

4.2.1.3 Master data request (*DataRequest_Notification*)

The application handles one data request from the master corresponding to the identifier 0x20 (*ID_Table*[1]). The application has to return a pointer to an array containing the data to be sent. In this example we declare an array "*slave_data[]*". This array will be shared between the application updating it with the last data and the LIN communication sending its contents on request of the master. The corresponding code for the "*DataRequest_Notification*" function is:

```
extern uByte slave_data[];
uByte * DataRequest_Notification(@tiny t_header *header)
{
 if(header->identifier == ID_Table[1].id){
  return(slave_data);
 }
}
```

4.2.1.4 Data reception (*DataReceived_Notification*)

The application handles 2 "data" frames from the master corresponding to the identifier 0x03 and 0x80. The first frame is part of the "normal" communication. The second frame is a sleep command frame that can be sent by the master at any time to interrupt the normal communication and set all nodes into low power mode.

On reception of the first frame the application saves the received data into the variable "*master_data[]*".

On reception of the sleep command frame the application sets the ST7 in Halt mode. Before setting the ST7 in Halt mode the wake-up sources should be acitivated. An application function, "PORTS_WakeUp_On()", is called. Two pins are configured as interrupt and will wake the ST7 up upon a corresponding interrupt request. The first pin is connected to the LIN_RX pin (the SCI RX pin has no interrupt capability). As a result any bus activity will wake the application up. The second is an application pin that should be also able to wake up the application.

The corresponding code for the "*DataReceived_Notification*" function is:

Any activity on the bus will wake up the ST7 out of Halt mode. As soon as the ST7 is ready to execute the next instruction any incoming frame can be received.

The application can also be woken up by a sensor connected to one pin of the ST7 and then should also wake up the whole LIN network. To do this, a wake-up frame has to be sent, which is done by the "*LIN_SendWakeUpSignal()*" function. This function is therefore inserted in the application wake-up interrupt routine:

The last point to be configured is the timeout handling and the setting of the corresponding timer, which is timer A. The application software is not using timer A for other purposes. As de-

```
extern uByte master_data[];
void DataReceived_Notification(@tiny t_header *header, @tiny t_response *re-
sponse)
{
 if(header->identifier == ID_Table[0].id){
  master_data[0]=response->data[0];
  master_data[1]=response->data[1];
 }
else if(header->identifier == ID_Table[2].id){
  PORTS_WakeUp_On();
  _asm("halt\n");
 }
}
```

```
@interrupt void PORTS_0_Interrupt(void)
{
unsigned char i;
 i=250;
 while(i--);
 if(!(PADR&0xFE))
 {
  PAOR&=0xFE;
  LIN_SendWakeUpSignal();
 }
}
```

scribed in 3.2.2 User interface functions the application is responsible for the initialization and the interrupt routine.

Initialization:

Firstly, the output compare must be configured: The OCIE flag of the TACR1 register must be set and the defined OC disabled by writing the high byte of the OC1 register. As the application is not using OC2 it is disabled also in the same way.

Secondly, the timer prescaler must be calculated using the equations given in Table 7:

on-chip SCI used	(FCPU/**prescaler**/BAUDRATE)*165 < 65536

1st test: prescaler=2

(FCPU/**prescaler**/BAUDRATE)*165= 8000000/2/9600*165=68750 > 65536 doesn't match

1st test: prescaler=4

(FCPU/**prescaler**/BAUDRATE)*165= 8000000/4/9600*165=34375 < 65536 match!

The prescaler has to be set to 4 which corresponds to writing the value (0,0) in the (CC0,CC1) bits in Timer A Control Register 2.

The corresponding initialization code is:

```
   /*---------------------------------------------------------------------
   ROUTINE NAME : TIMA_Init
   INPUT/OUTPUT : None

   DESCRIPTION  : Configure the Timer A

   COMMENTS     :
   ---------------------------------------------------------------------*/
   void TIMA_Init(void)
   {
      TACR1=0x40;    /* -output compare interrupt enabled */

      TACR2=0x00;    /* prescaler = 4 */

      TASR;          /* erase OC2F flag and */
      TAOC2_L;       /* and disable it      */
      TAOC2_H=0x55;

      TASR;          /* erase OC1F flag and */
      TAOC1_L;       /* and disable it      */
      TAOC1_H=0x55;
   }
```

Interrupt:

The timer is only used for the timeout handling so the "LINTimeOut_Interrupt" function just needs to be called in the Timer A interrupt service routine defined by the application which corresponds to the following code:

```
   void LINTimeOut_Interrupt(void);
   /*---------------------------------------------------------------------
   ROUTINE NAME : TIMA_Interrupt
   INPUT/OUTPUT : None

   DESCRIPTION  : timer Interrupt Service Routine

   COMMENTS     :
   ---------------------------------------------------------------------*/
   @interrupt void TIMA_Interrupt(void)
   {
     LINTimeOut_Interrupt();
   }
```

The software is ready!

4.3 STMICROELECTRONICS LIN PACKAGE - EXAMPLE INSTALLATION

4.3.1 LIN package

The LIN software is delivered in a package including the software itself, this application note and the above examples for the ST72254 and the ST72334 MCUs.

Figure 12. STMicroelectronics LIN package
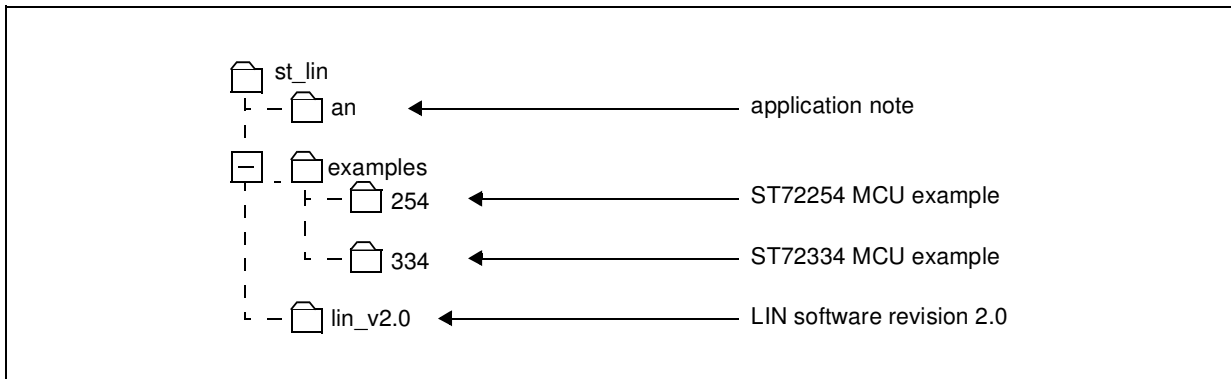
You will receive a "zip" file called st_lin.zip that will generate the above directory tree. Create a new folder we will call the working directory and extract the files into it.

The directory tree and architecture are the same for both example.

Figure 13. Example directory tree
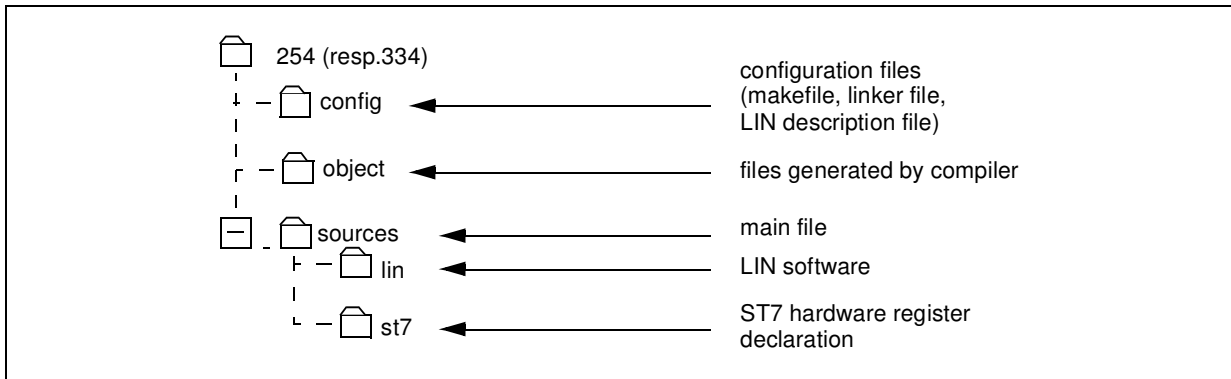
4.3.2 Quick start with STVD7 and Cosmic C Compiler

In order to make the examples run you need the following software to be installed:

-STVD7: STMicroelectronics visual debugger for the ST7 microcontroller family with inte-grated editing and environment features. This software is free of charge. You can download it by accessing the STMicroelectronics MCU homepage: mcu.st.com

– **Cosmic C Compiler:** Cosmic C compiler for the ST7 target. For further information or contacts go to: http://www.cosmic-software.com/

In order to be able to compile and start a debug session and even flash an ST7 MCU with the example code you need to create a new STVD7 workspace and configure your COSMIC tools installation directory.

– Create a new STVD7 project:

start the STVD7 software.

First check that the Cosmic compiler installation directory is configured: "Project>Toolchains Path..." Under "Cosmic Builder Path" enter (if not already configured) the compiler path (where cxst7.exe is located).

We are ready to create a new project. Select File>New Workspace. Under "Workspace filename" enter "334env" (or "254env"). Under "Workspace location" enter the examples directories: <your working directory>\st_lin\examples\334 (or <your working directory>\st_lin\examples\254). Click on "Next". Fill in the next dialog box as follows:

Figure 14. Project configuration
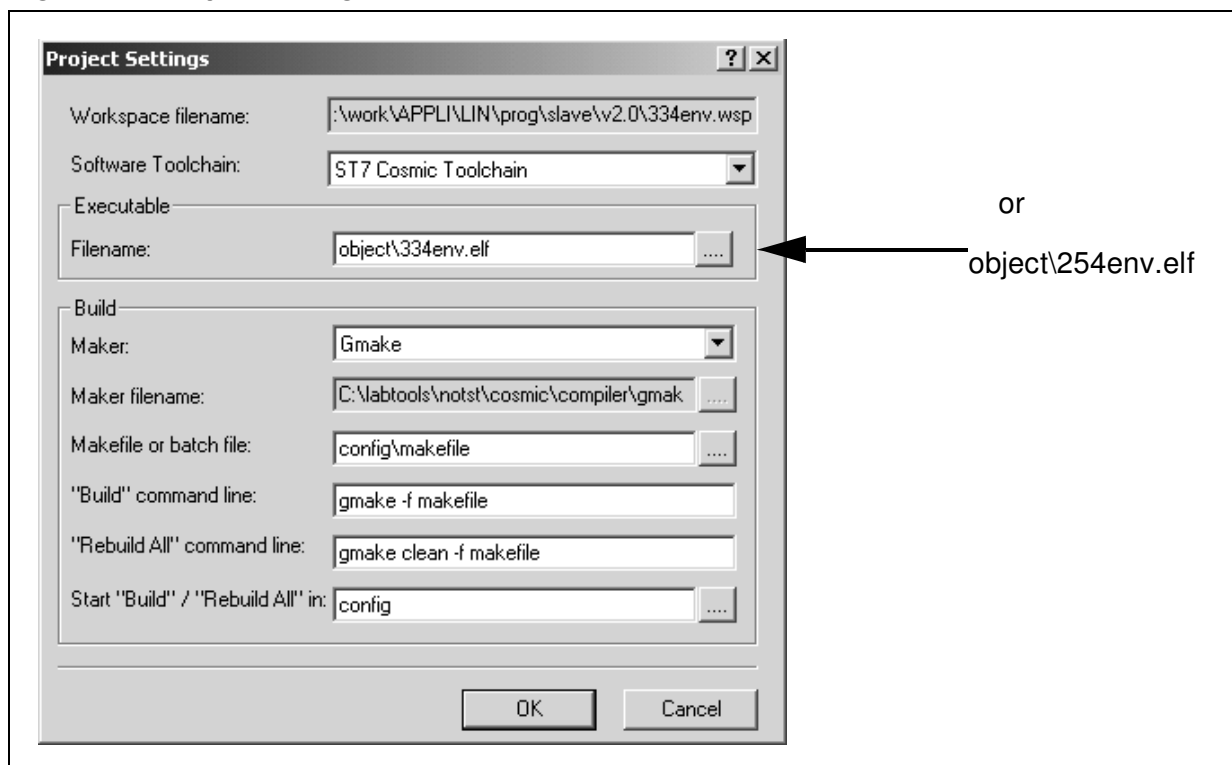
Click "OK".

To have easy access to your source files, configure the source file directories in the workspace window. The example has 3 folders containing the source files:

<your working directory>\st_lin\examples\334\sources\ (or -\254\sources)

main file location (application code location)

<your working directory>\st_lin\examples\334\sources\lin (or -\254\sources\lin)

LIN software location

<your working directory>\st_lin\examples\334\sources\st7 (or -\254\sources\st7)

hardware register declaration

The workspace window should look like this:

Figure 15. Workspace window configuration
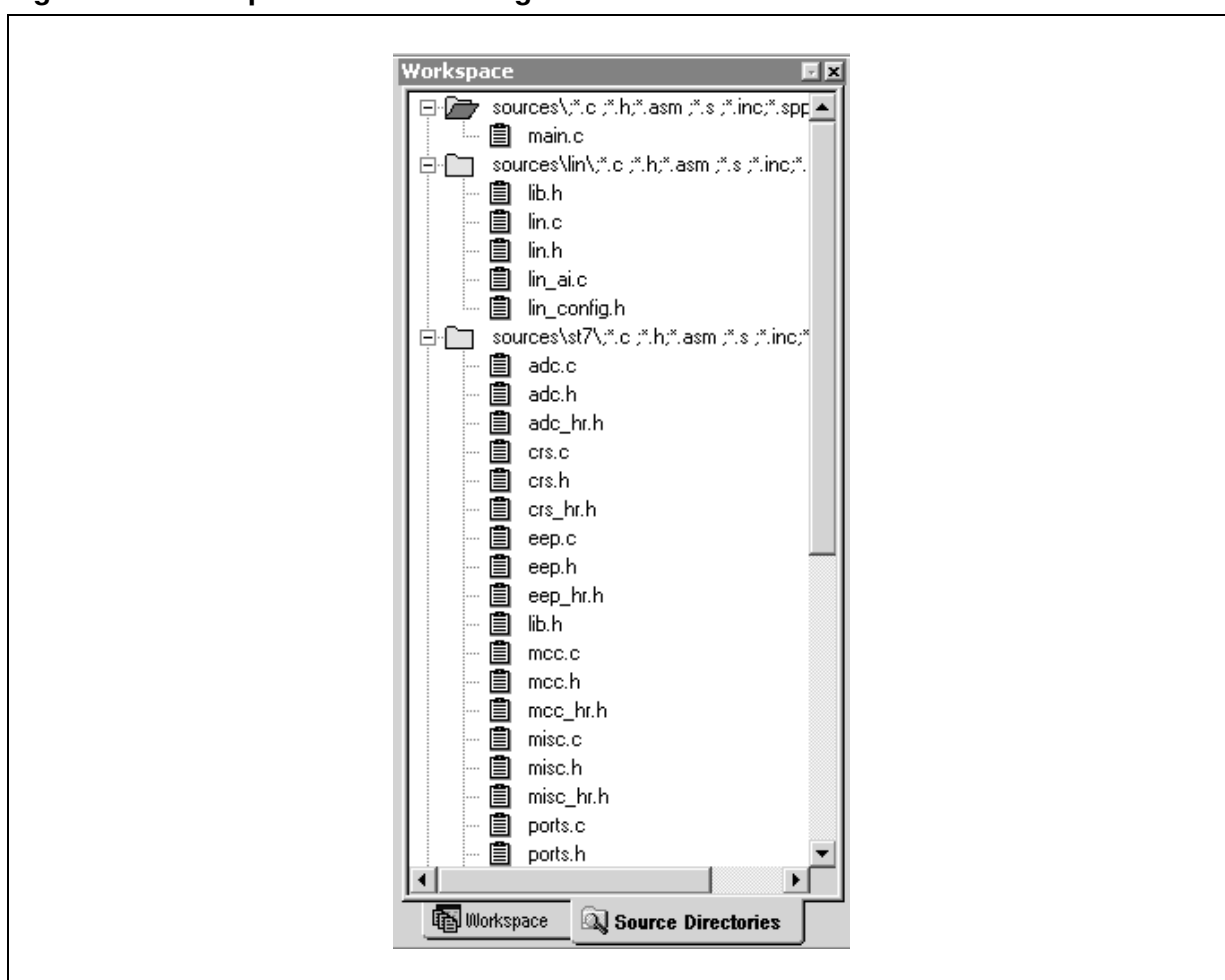
The workspace is ready. Save it in the working directory: File>Save Workspace

-**Cosmic installation path**

The examples are delivered with all necessary configuration files like makefiles and linker files. As far as possible paths are given that are relative to the working directory so you don't

need to update them when you move the example to another directory/PC. One file contains absolute paths: The linker file located in <your working directory>\st_lin\examples\334\config (or -\254\config) and named 334env.lkf (resp. 254env.lkf). You need to enter 3 paths. Edit the file. See Figure 16:

Figure 16. COSMIC linker file configuration

```
...
+seg .share -a UZPAGE -is -sRAM

<ENTER COSMIC INSTALL PATH>\lib\crtsx.st7 # startup routine
..\object\main.o # application program
..\object\adc.o
..\object\crs.o
..\object\eep.o
..\object\mcc.o
..\object\misc.o
..\object\ports.o
..\object\sci.o
..\object\spi.o
..\object\tima.o
..\object\timb.o
..\object\trap.o
..\object\lin.o
..\object\lin_ai.o
<ENTER COSMIC INSTALL PATH>\lib\libm.st7
<ENTER COSMIC INSTALL PATH>\lib\libims.st7

+seg .const -b 0xFFE0 -k # vectors start address
..\object\vector.o # interrupt vectors
...
```

In place of "<ENTER COSMIC INSTALL PATH>" (3 times) enter your Cosmic compiler installation path (path of "cxst7.exe"). For example replace "<ENTER COSMIC INSTALL PATH>\lib\crtsx.st7" by "c:\cosmic\\lib\crtsx.st7". Save the file.

The example is ready. You can build the example (F7). Build will generate a 334env.elf (or 254env.elf) for debugging and a 334env.s19 (254env.s19) for flashing an MCU. Refer to the STVD7 documentation for details using of the STVD7 editor and debugger.

4.4 PERFORMANCE

The above examples correspond to the 2 main kinds of software configuration, the SCI communication emulated by software or supported by the hardware SCI peripheral. The corre-

sponding software performance is different. Table 9 and Table 10 give performance summaries for both examples.

Table 9. ST72254 Example performance summary

Compiler version	v4.3a	
Memory model	+modm - memory short	
Compiler options	+debug	
Code size	whole project	2.1 kbyte
	lin.o	1.8 kbyte
	lin_ai.o	44 byte
LIN interrupt - CPU load	17%	
Max LIN baudrate @16MHz	16kbaud	

Table 10. ST72334 Example performance summary

Compiler version	v4.3a	
Memory model	+modms - memory small	
Compiler options	+debug	
Code size	whole project	1.4 kbyte
	lin.o	1.0 kbyte
	lin_ai.o	44 byte
LIN interrupt - CPU load	2%	
Max LIN baudrate @16MHz	no limit-20kbaud	

4.4.1 Timing considerations

The runtime performance of the software depends on many parameters like the memory model, the compiler options and the compiler version and also on the application (lin_ai.c is part of the LIN interrupt). As a result it is impossible to give generally applicable software runtime performance data.

A first timing consideration is the maximum reachable speed running the software without any application software. In this case only the software emulating the SCI communication (ST72254 example) has speed limitations. In the above ST72254 example the maximum speed reachable was 16kbaud. The software using the on-chip SCI peripheral has no LIN speed limitation.

Then if you add some application code in the lin_ai.c file, you make the LIN interrupt service routine longer and therefore decrease the software performance. That's why in order to achieve better performance you should keep actions done in the lin_ai.c file as short as possible. Nevertheless the software that uses the on-chip SCI peripheral should not be speed limited under 20kbaud.

The final timing consideration is when the application software needs to interrupt the LIN communication. The LIN software is interrupt-driven and between two interrupts the application software can run some other code. As long as the application software is interruptable, the LIN software will interrupt it when needed. Problems can occur if the application software is not in-

terruptable for a long time so that the LIN interrupt is called too late and an event is lost (bit/byte). As too many parameters are involved it's impossible to give generally-applicable performance data. Therefore we implemented a timing analysis feature (DEBUG_MODE) you can activate when debugging your software.

This feature can be activated in the lib.h file using 5 symbols:

#define	DEBUG_MODE
Description	When defined, this activates the timing debug feature. This feature configure an pin as output and uses it to indicate when the LIN interrupt is entered (set) and when it is left (reset).

#define	DEBUG_PxDRADD
	DEBUG_PxDDRADD
	DEBUG_PxORADD
Description	To configure the used I/O pin, first define these symbols to the address of the corresponding data register, data direction register and option register.

#define	DEBUG_LIN_IT_PIN
Description	Set this symbol to the pin number used (ex: 5 when using PC5)

The final piece of information you need to do the timing analysis is how much can the application delay the occurrence of the LIN interrupt? For this we need to go into more detail to analyze the way the software is handling each bit/byte. Once again, depending on whether the SCI communication is emulated or not the software is works in very different ways:

4.4.2 Using the Emulated SCI

Figure 17. SCI emulation: LIN Reception
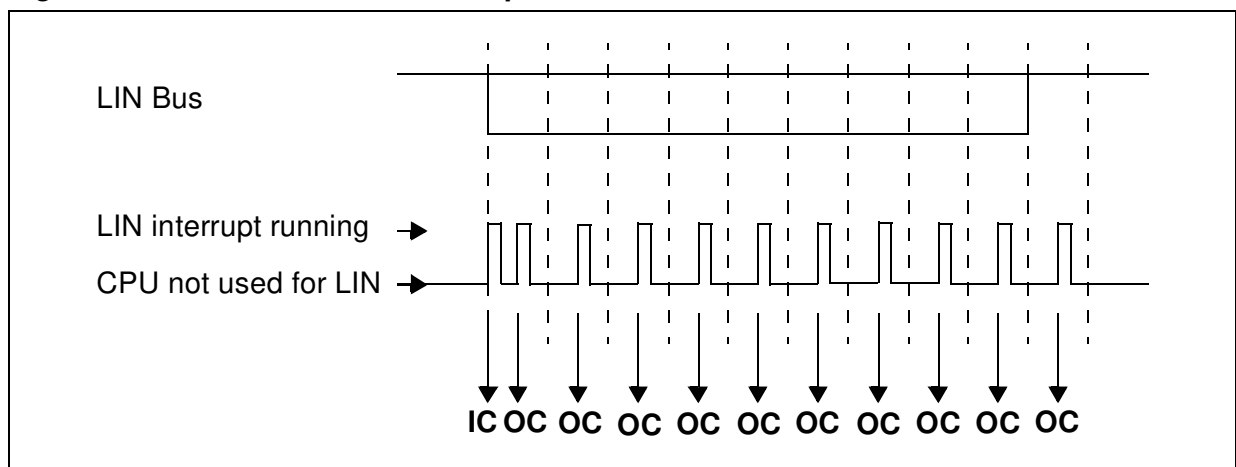
4.4.2.1 Reception

The SCI is emulated by software using the input capture and output compare of the on-chip 16-bit timer. When the bus is idle, the software waits for a negative edge: The input capture interrupt is activated and calls the LIN interrupt when a negative edge occurs. The input capture time is used to generate an output compare in the middle of this first bit. The LIN interrupt routine returns to the calling program. When the output compare event occurs the LIN interrupt is called again. The bus level is checked and a new output compare is set to occur in the middle of the next bit. This last process is repeated until the stop bit.

As a result the application software should not:

– delay the occurrence of the IC interrupt too much. Specifically: A problem occurs if the first output compare is set after the expected occurrence of the output compare event, which is the middle of the bit. So as long as the IC interrupt ends before the middle of the bit, the delay is acceptable. See Figure 18.

Figure 18. IC Interrupt Handing in Reception Mode

– Delay the occurrence of each OC interrupt too much. Specifically: A problem occurs if the sample time defined at the beginning of the interrupt is delayed so that it occurs after the end of the bit. See Figure 19.

Figure 19. OC interrupt Handling in Reception Mode
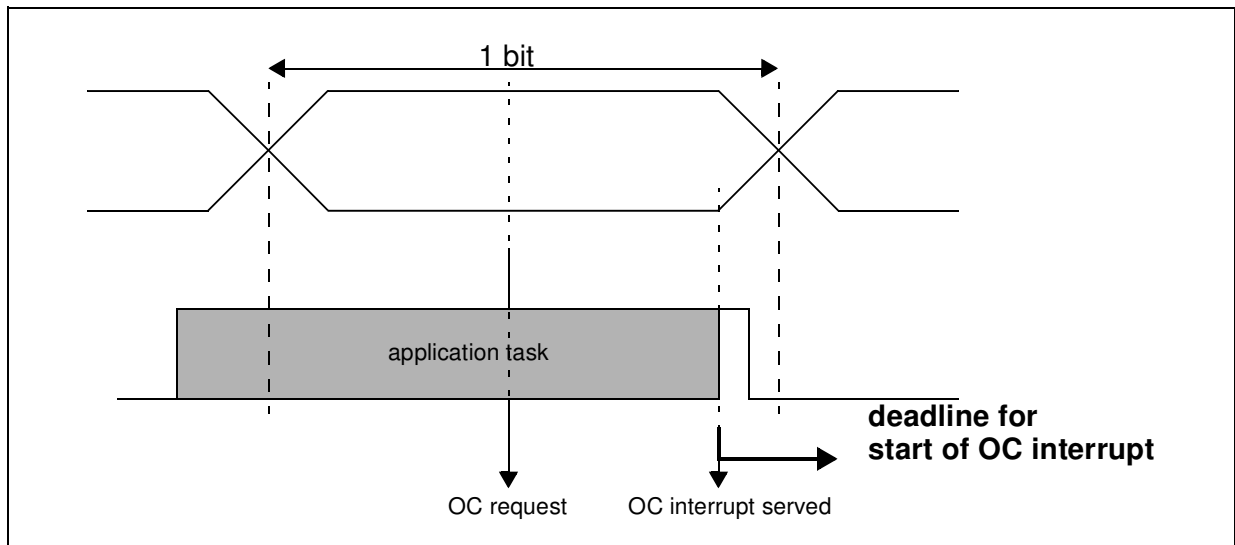
4.4.2.2 Transmission

For transmission the software only uses the Output Compare feature of the 16-bit timer. For each bit, two OC interrupts are generated. The first one is generated to output the value of the new bit. The second is used to read back the bus and check whether the output value is actually being sent, in other words to check for a bit error. See Figure 20.

Figure 20. SCI emulation: LIN Transmission
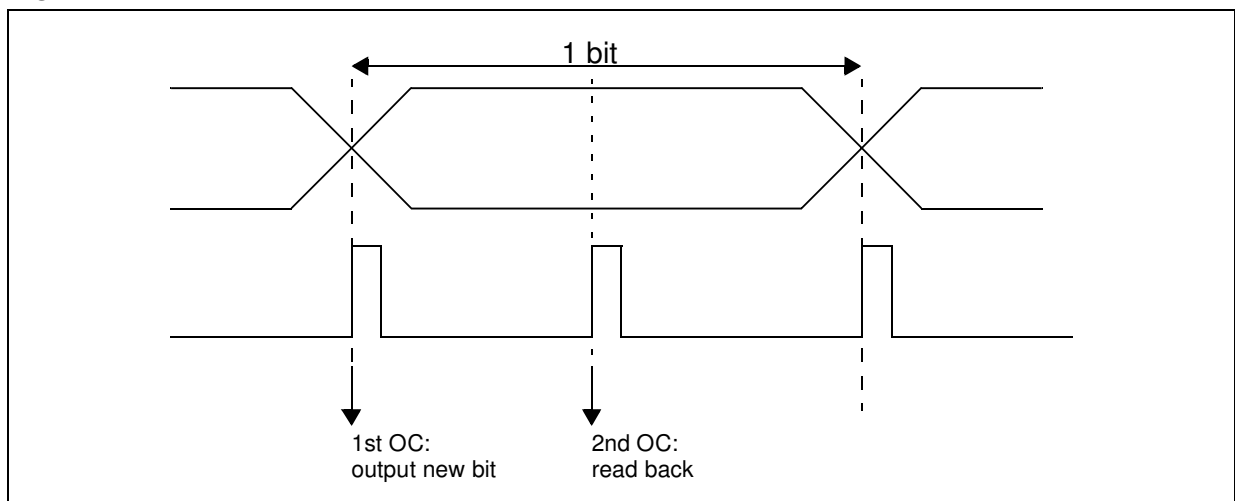
The bit transition time is precise because the timer output compare itself outputs the new level when an output compare event occurs. Software delays do not influence the bit transitions. The first OC sets a new OC for the middle of the bit. The second reads the LIN bus level and prepares the next OC to output the next bit.

As a result the application software should not:

– Delay the occurrence of the first OC interrupt too much. Specifically: A problem occurs if the second output compare is set after the expected time, which is the middle of the bit. So as long as the first OC interrupt ends before the middle of the bit, the delay is acceptable. See Figure 21.

Figure 21. Handling the first OC Interrupt in transmission mode
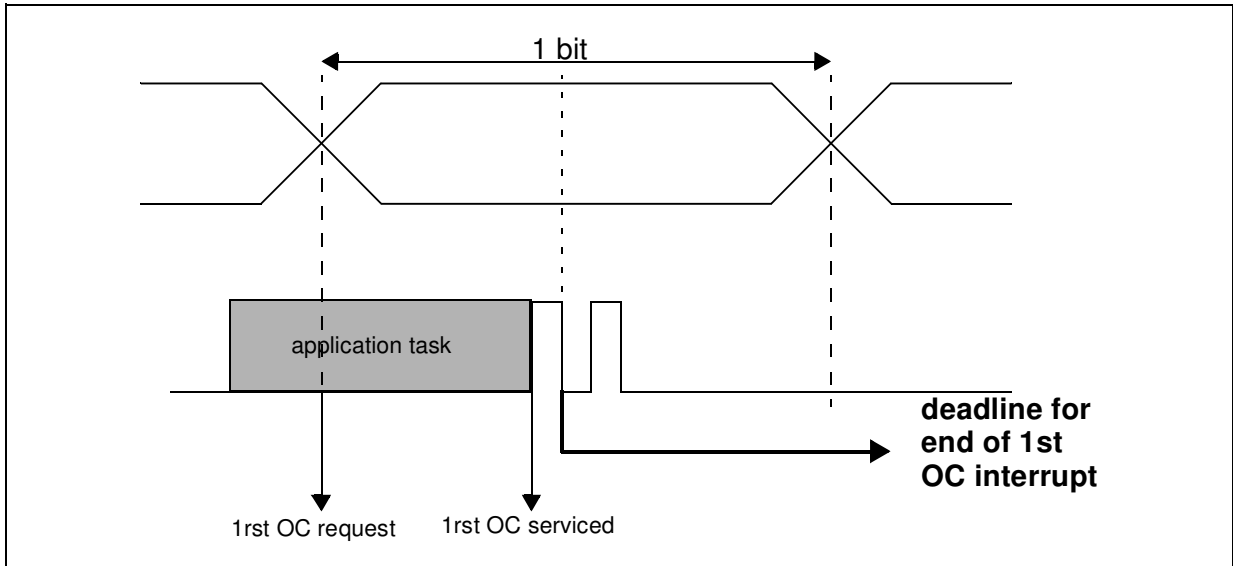
– Delay the occurrence of the second OC interrupt too much. Specifically: A problem occurs if the sample time defined at the beginning of the interrupt is delayed outside the bit. See Figure 22

Figure 22. Handling the second OC Interrupt in transmission mode
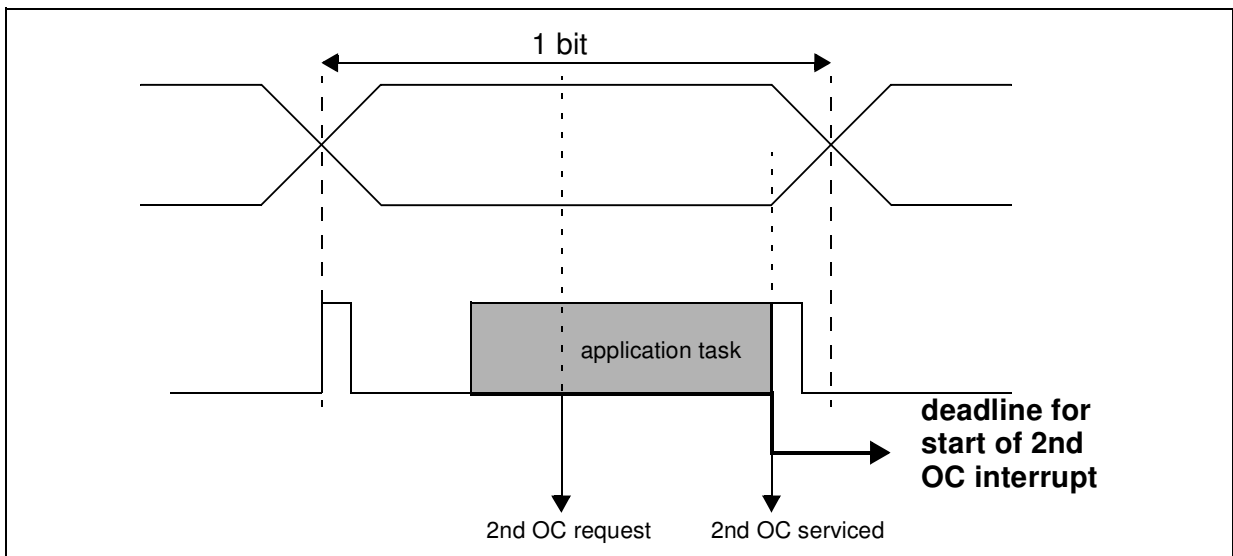
4.4.3 Using the on-chip SCI

Using the on-chip SCI the flexibility is considerably increased. The CPU load is very low and the capability of the application software to run at the same time as the LIN software is high.

Firstly a LIN interrupt occurs every byte instead of every bit (even every half bit in transmission) in the timer solution. And secondly the received byte is buffered in hardware which allows a lot of flexibility.

Figure 23. CPU load with on-chip SCI peripheral
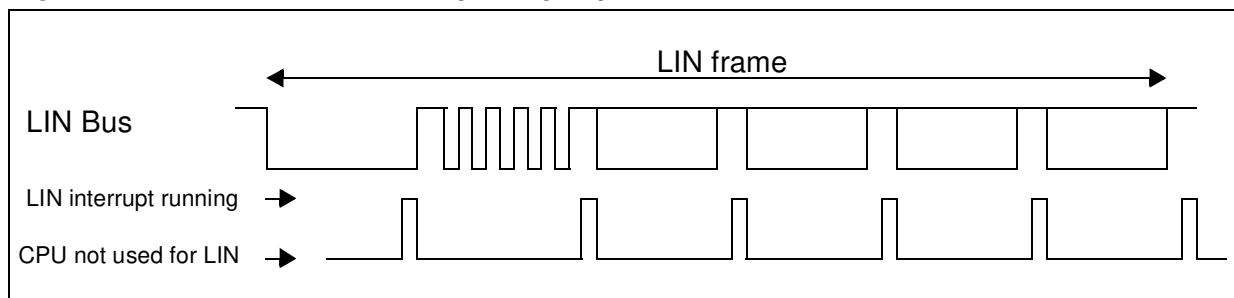
4.4.3.1 Reception

The on-chip peripheral is handling the SCI communication and notifies a reception at the end of each byte. A LIN interrupt is then generated and handles the received data according to the LIN protocol.

Figure 24. Reception Notification Interrupt
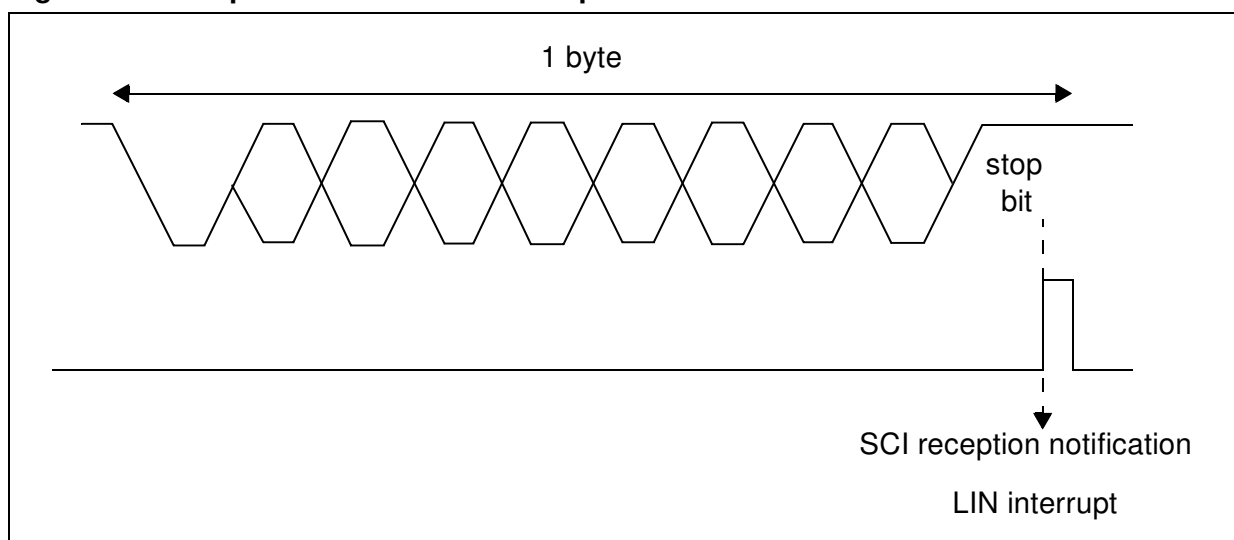
When the SCI notifies the reception it copies at the same time the received value into a buffer. As a result the peripheral is ready for the next reception and the software can still hold the received data until the end of the new reception. Afterwards an overrun condition occurs.

As a result the application software should not delay the occurrence of the SCI reception notification interrupt so much that an overrun condition occurs. Specifically: The end the LIN interrupt should end before the stop bit of the next byte.

Figure 25. Reception Interrupt Handling
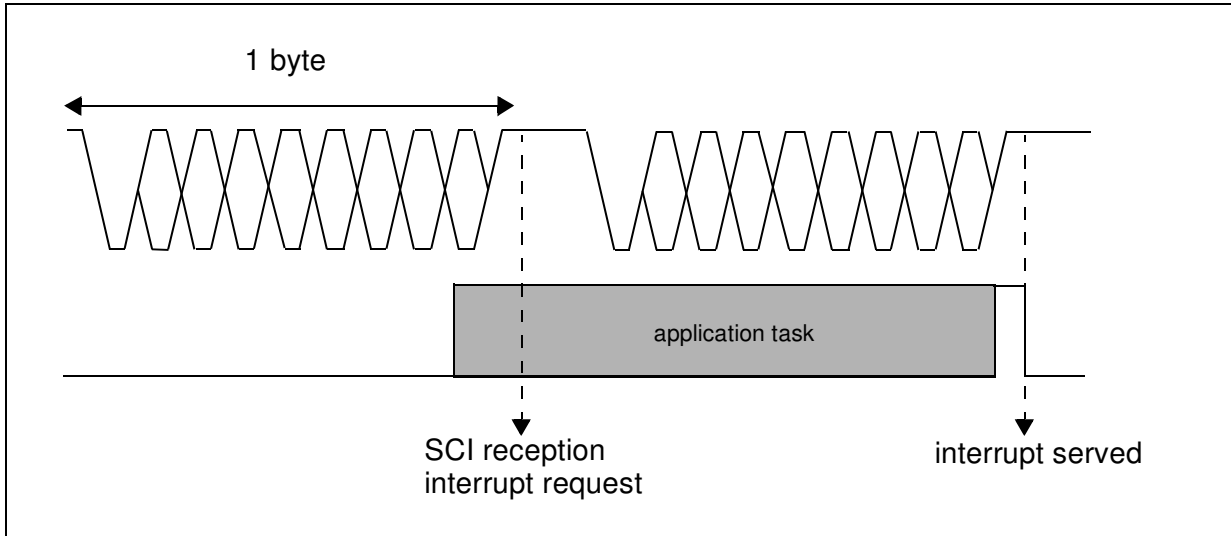
4.4.3.2 Transmission

In transmission delays coming from the application cannot disturb the proper working of the software. This will delay the transmission and the issue is more a timeout issue on the current transmitted frame: If the interrupt time is very long the transmitted frame may exceed the maximum allowed frame time. The SCI interrupt in transmission occurs also at the stop bit. If the occurrence of the interrupt is delayed by the application the interbyte time will increase.

5 SUMMARY OF CHANGES

Revision	Main changes	Date
1.0	First version	August 2001
1.1	Remove section "HOW TO GET THE LIN SOFTWARE?"	April 2002

LIN (LOCAL INTERCONNECT NETWORK) SOLUTIONS