LIN controller

Author: Martin Patak

Year
2010

Abstract

An increasing number of electronic devices connected via LIN, CAN, and FlexRay buses in passenger cars leads to the problem of testing these complex networks. Tools for testing one or two buses have been available for years, but these personal computer based tools usually use a standard controller, and therefore it is not possible to test non-standard states and a real-time response with them.

This thesis presents a system capable of real-time and non-standard testing of the LIN network. The solution is based on three main components (a controller, a trigger and an injector) designed in the form of IP functions. These components are integrated in an FPGA chip together with a μController. A test is then executed in the following way. The μController receives commands via the RS232 interface in a specified format, sets the system to execute the test, and sends off the result.

This system is used on a network testing with a single LIN bus, but the concept of this system can be used for testing of many LIN, CAN and FlexRay buses.

Contents

1 LIN controller **1**

1.1 Overview 1

1.2 Interface 2

1.2.1 Overview 2

1.2.2 Writing data to registers 2

1.2.3 Reading data from registers 2

1.2.4 LIN database connection 2

1.3 Register description 6

1.3.1 LIN controller general settings 6

1.3.1.1 CONTROL1 6

1.3.1.2 CONTROL2 7

1.3.1.3 STATUS1 7

1.3.1.4 ERRORS 8

1.3.1.5 ID_MASK 9

1.3.1.6 ID_FILT 9

1.3.1.7 CLK_DIV 9

1.3.2 LIN controller Tx registers 9

1.3.2.1 TX_BUF 11

1.3.2.2 TX_FREE 11

1.3.2.3 TX_ID 11

1.3.2.4 TX_TRG 11

1.3.2.5 TX_BYTE 12

1.3.2.6 TIMEST 12

1.3.3 LIN controller Rx registers 12

1.3.3.1 MSG_CNT 14

1.3.3.2 RX_ID 14

1.3.3.3 RX_TIMEST_NB 14
1.3.3.4 RX_BYTE 14
1.3.3.5 RX_TIMEST 14
1.3.3.6 RX_READ 14
1.4 Controller architecture 14
1.4.1 Checksum gen 15
1.4.2 Parity gen 15
1.4.3 Divider 15
1.4.4 Majority sampler 15
1.4.5 Receiver 17
1.4.6 Transmitter 17
1.4.7 Configuration registers 17
1.4.8 Core state machine 17
1.4.9 Message controller 20
1.5 Conclusion 23

2 LIN trigger 24
2.1 Overview 24
2.2 Interface 25
2.3 Register description 27
2.3.1 CONTROL1 27
2.3.2 INTSRC 27
2.3.3 CLK_DIV 30
2.3.4 DATA/TMST 30
2.3.5 LENGTH 30
2.3.6 MASK 30
2.4 Reset states 31
2.5 Data comparison 31
2.5.1 Example 31
2.6 Output 32
2.7 Controller architecture 33
2.7.1 Configuration registers 34
2.7.2 Receiver 34
2.7.3 Core state machine 35
2.8 Conclusion 35

3 **LIN injector** 36
3.1 Overview 36
3.2 Interface 37
3.3 Register description 39
3.3.1 CLK_DIV 39
3.3.2 LENGTH 39
3.3.3 CONFIG 39
3.3.4 DATAx 39
3.4 Reset values 40
3.5 Injector architecture 40
3.5.1 Configuration registers 40
3.5.2 Clock divider 41
3.5.3 Transmitter 41
3.6 Conclusion 42

4 **LIN database** 43
4.1 Overview 43
4.2 Interface 45
4.3 Registers 47
4.3.1 IDx 47
4.3.2 CLK_DIV 47
4.4 Reset values 48
4.5 Database architecture 48
4.5.1 Core state machine 49
4.5.2 Configuration registers 50
4.6 Conclusion 50

5 **System conception** 52
5.1 Overview 52
5.2 Implemented system 53
5.2.1 Block connection 54
5.2.2 μController Nios II 55
5.2.2.1 Configuration description 55
5.2.2.2 Program inside the μController 56
5.2.2.3 Compiling and downloading the software 58

5.2.3 Time stamp generator 59
5.2.4 D Flip-Flop 59
5.2.5 System input/output pins 59
5.2.6 LIN transceiver TJA 1020 60
5.2.7 Configuring the FPGA 61
5.3 Communication protocol 62
5.3.1 Protocol types 63
5.4 Conclusion 64

Bibliography **66**

List of Figures

1.1 Interface 3
1.2 Writing to registers 5
1.3 Reading data from registers 5
1.4 Trigger system 12
1.5 Rx buffer 13
1.6 Block diagram 16
1.7 RAM map 18

2.1 LIN trigger 25
2.2 Trigger on a gap between messages 29
2.3 Trigger on a gap between bytes 29
2.4 A message is longer than N 29
2.5 Data comparison schematics 32
2.6 Interrupt output 33
2.7 Schematics of the LIN trigger 34

3.1 Interface 37
3.2 Architecture 40

4.1 Signals from LIN database 44
4.2 Block connection with controller and triggers 44
4.3 Interface 45
4.4 Choosing default or table values 48
4.5 Schematics 49
4.6 Memory map 51

5.1 Structure of the implemented system 53
5.2 Transceiver connection 61
5.3 The JTAG configuration scheme 62

List of Tables

1.1 Interface description 4
1.2 General registers 6
1.3 Reset values 9
1.4 Tx registers 10
1.5 Rx registers 13
1.6 Implementation in the FPGA 23

2.1 Interface description 26
2.2 Registers 28
2.3 Reset values 31
2.4 Compilation results 35

3.1 Interface description 38
3.2 Registers 38
3.3 Reset values 40
3.4 Compilation results 42

4.1 Interface description 46
4.2 Address bits 46
4.3 Registers 47
4.4 Compilation results 51

5.1 Nios II system, configuration 56
5.2 Input / output pins 60
5.3 Communication protocol 63
5.4 Messages type 64
5.5 Compilation results 65

Chapter 1

LIN controller

1.1 Overview

LIN is very similar to UART but it contains synchronization break which is 13 bits long and therefore it cannot be implemented by a classic UART unit, which is a part of every μController. Some controllers support the LIN frame functionality, but the interface is the same as for an UART.

The LIN frame consists of a synchronization break, byte 0x55, ID, up to eight data bytes, and a checksum. Implementing this in the μController takes a lot of computing time, because each byte must be sent and read separately. Most of IPs designed for LIN functionality are designed this way.

The LIN controller, designed in here, offers much more flexibility, and has been designed as a full messages interface with many extra extensions. This controller is based on the XILINX LIN controller [1].

The LIN controller, the core written in VHDL, offers adjustable number of Rx buffers (buffers for the full message including the time stamp for each byte), filter of IDs, and an adjustable number of Tx buffers with extern or time stamp triggering. This controller can therefore be used as a master, a slave, or just for listening to all messages on the bus.

It significantly reduces the processor computation time. For example, it can be set to transmit four messages according to the time stamp or an outer signal (or combination of both), and then cause interrupt (either because all messages have been transferred, or an error occurred). Due to this functionality, the controller is suitable for being a part of an automotive bus testing system.

1.2 Interface

1.2.1 Overview

The LIN controller IP is a core with a WISHBONE slave interface [2]. The interface is compliant with the WISHBONE Slave Rev. B.3 interface with a separated 8, 16, and 32-bit input/output data buses[1]. The interface signals are shown in the figure 1.1 and their description can be found in the table 1.1.

1.2.2 Writing data to registers

Writing data to registers is shown in the picture 1.2. The value X means how many rising edges of the CLK must be executed to receive the ACK signal (and to store the data). The 8-bit interface does not need any another cycle (ignoring the dash line, shown in the figure), 16-bit interface needs two extra rising edges (four in total), and 32- bit interface needs four (six in total) more rising edges.

1.2.3 Reading data from registers

Reading data from registers is shown in the picture 1.3. The interface does not support burst reading[2]. The value X means how many rising edges of the CLK must be executed to receive the valid data at the output. The 8-bit interface does not need any another cycle (ignoring the dash line, shown in the figure), 16-bit interface needs two extra rising edges (four in total), and 32- bit interface needs four (six in total) more rising edges.

1.2.4 LIN database connection

A LIN database sets information about the message length and the type of checksum after the ID is transmitted to the bus. The LIN controller configured to LIN 2.0 or above needs to get information from the LIN database. After the ID was sent to the bus, the

[1] The design offers two interfaces. The first one, described in this document, is wishbone compatible. The other one has the same signals as the first one, except for the DAT_I and DAT_O. These two signals are replaced with DAT_IO. If WE_I = 1 or chip is not selected then DAT_IO is in the state of high impedance.

[2] Between two readings of the registers, there must be at least one rising edge of CLK_I when either CYC_I or STB_I is equal to zero.

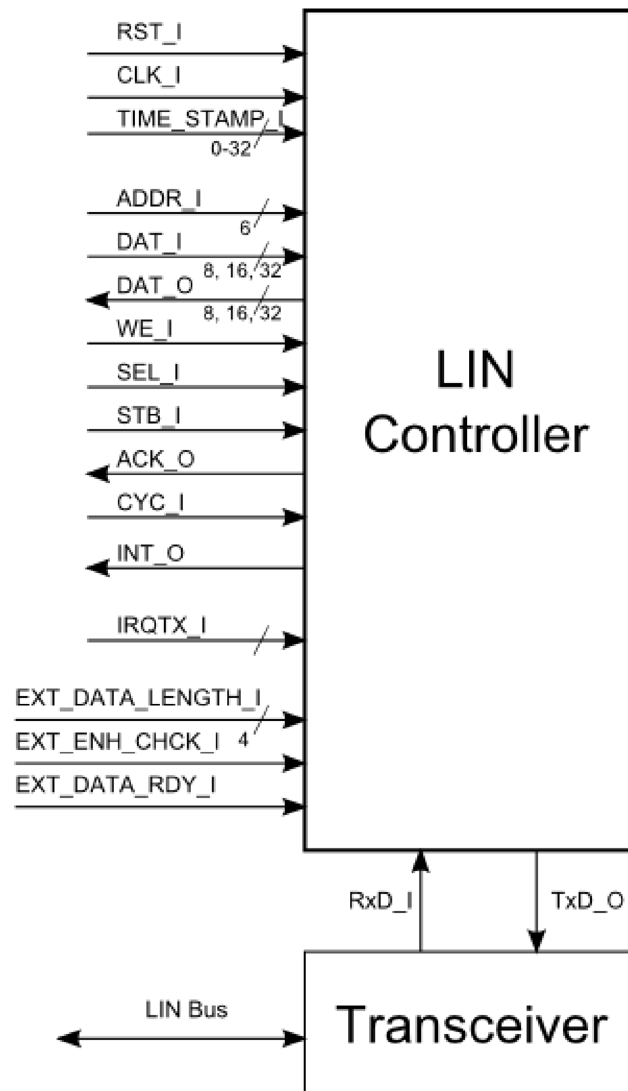

Figure 1.1: Interface

Pin	Activity	Description
RST_I	HIGH	Reset signal
CLK_I	-	Clock input
TIME_STAMP_I[x_1:0]	-	Time stamp input (x_1 : 0 - 32 bit)
ADDR_I[5 : 2,1,0]	-	Address of the register (for the 32, 16 and 8-bit interface, respectively)
DAT_I[31,15,7:0]	-	Data input (32, 16, or 8 bits)
DAT_O[31,15,7:0]	-	Data output (32, 16, or 8 bits), DAT_O is equal to DAT_I when the device is not selected
WE_I	HIGH	Bus access signal : HIGH for the write transfer, LOW for the read transfer
SEL_I	HIGH	Device select bit
STB_I	HIGH	Strobe input indicated a valid data transfer cycle
ACK_O	HIGH	Acknowledge output indicates the termination of a normal bus cycle
CYC_I	HIGH	"Valid bus cycle in progress" bit
INT_O	HIGH	"Interrupt output" bit
IRQTX_I[x_2:0]	HIGH	Signal for transmitting a Tx message ($x_2 + 1 =$ TX_BUFFERS constant can be set in the source code)
EXT_DATA_LENGTH_I[3:0]		Data length from an external LIN database
EXT_ENH_CHCK_I	HIGH	Data checksum type signal from an external database
EXT_DATA_RDY_I	HIGH	Signal from an external database that data are ready
RxD_I	-	Receive data from the LIN transceiver
TxD_O	-	Transmit data to the LIN transceiver

Table 1.1: Interface description

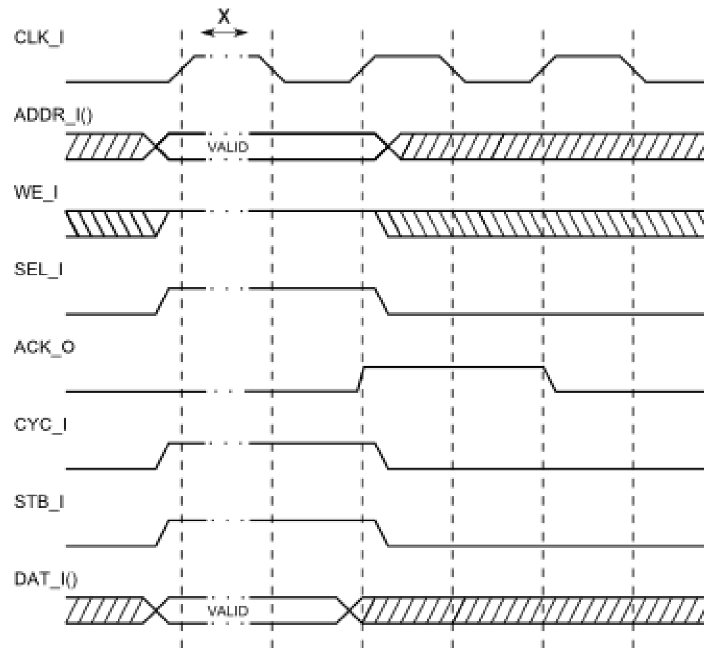

Figure 1.2: Writing to registers

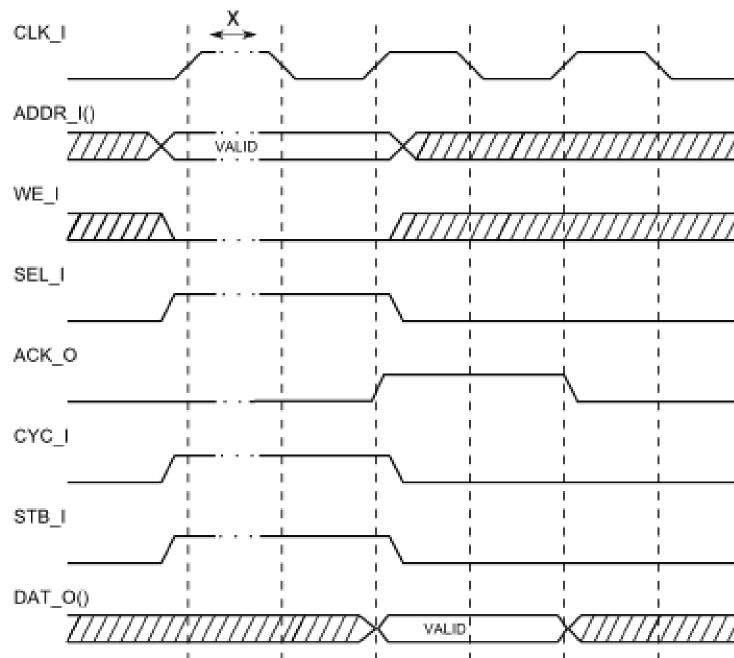

Figure 1.3: Reading data from registers

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	CONTROL1	-	-	-	TMSGPR	LPBCK	LIN13	ABAU	MASL
0x01	CONTROL2	-	-	-	IWAKE	IERR	-	ITX	IRX
0x02	STATUS1	-	-	-	WAKE	ERR	IDL	TDRE	RDRE
0x03	ERRORS	-	ESH	OVRL	XMIT	FRAM	PAR	CKS	ORUN
0x04	ID_MASK	-	-	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x05	ID_FILT	-	-	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x06	CLK_DIV1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x07	CLK_DIV2	-	-	-	-	BIT 11	BIT 10	BIT 9	BIT 8

Table 1.2: General registers

controller waits until the signal EXT_DATA_RDY_I (data from the database ready) is set. Then it reads the length and the data checksum type, and continues sending/receiving the message. This process is shown in the figure 4.1.

1.3 Register description

1.3.1 LIN controller general settings

General settings described in the table 1.2 set behaviour and indicate the basic statuses of the LIN controller. Values after a reset can be found in the table 1.3.

1.3.1.1 CONTROL1

This read/write register contains LIN controller settings

MASL 0 - Controller is master
1 - Controller is slave

ABAU 0 - auto baud rate disabled
1 - auto baud rate enabled (slave controller only) [3]

[3]CLK_DIV register must be set to the default value, or the bit rate must be smaller than the smallest bit rate on the bus

LIN13 0 - the message length and the checksum according to the LIN 1.3 specification
1 - the message length and the checksum according to an external table - it must be connected to the LIN database

LPBCK 0 - LIN driver is receiving all messages including its own messages
1 - loop back disabled[4] (receives messages which are transmitted by someone else)

TMSGPR 0 - time stamp has priority
1 - outer signal has priority
This is valid only if the controller is master and both events occur when the device is not transmitting.

1.3.1.2 CONTROL2

This read/write register contains the LIN driver interrupt settings

IRX interrupt when the Rx register is not empty
0 - interrupt disabled
1 - interrupt enabled

ITX interrupt when the Tx register is empty
0 - interrupt disabled
1 - interrupt enabled

IERR interrupt when an error occurred
0 - interrupt disabled
1 - interrupt enabled

IWAKE interrupt when a wake up signal occurred
0 - interrupt disabled
1 - interrupt enabled

1.3.1.3 STATUS1

This read/write register contains information about the state of the LIN controller

[4] There must be a free space in the Rx buffer even if the ID is not going to be saved. First of all, the ID is saved to the Rx buffer. Then it is determined whether the message will be saved or not. Therefore, full Rx buffers can cause ORUN error even if the ID is not going to be saved.

RDRE 0 - Rx buffer is empty
1 - Rx buffer is not empty

TDRE 0 - All Tx registers are full
1 - Not all Tx registers are full

IDL 0 - The controller is active
1 - The controller is idle

ERR 0 - No error flags set
1 - An error flag is set
Cleared when ERRORS register is read

WAKE 0 - No break was generated on the bus
1 - a break or wake up signal was generated
Cleared when STATUS1 register is read [5]

1.3.1.4 ERRORS

This read-only register contains the LIN bus error flags. It is cleared to the reset state when read.

ORUN 1 - Message was lost. The Rx buffer was full.

CKS 1 - Checksum error in the received frame.

PAR 1- Parity error was detected in the ID.

FRAM 1 - Framing error was encountered.

XMIT 1 - Bit error during transmission.

OVRL 1 - No slave response within the time-out.

ESH 1 - Short frame was received.

[5] It is generated when dominant state appeared on the bus for longer than 11 bits. It is generated even if master initializes the start.

CONTROL1	0x0
CONTROL2	0x9
STATUS1	0x6
ERRORS	0x0
ID_MASK	0x0
ID_FILT	0x0
CLK_DIV1	0x2
CLK_DIV2	0x0

Table 1.3: Reset values

1.3.1.5 ID_MASK

This read/write register sets the mask of the ID.
0 - bit must not match
1 - bits must match

1.3.1.6 ID_FILT

This read/write register sets the ID.

1.3.1.7 CLK_DIV

This read/write registers set the bit rate.

$$CLK_DIV = \frac{clk}{16 \cdot bit_rate} \tag{1.1}$$

Divisors 1 or 0 are invalid.

1.3.2 LIN controller Tx registers

The number of Tx registers is adjustable (in the source code for this controller). Each buffer contains a flag if it is empty, an ID, a time stamp for triggering, and data.

If the controller is a slave, then it replies to the master's ID either if some buffer contains a message with the same ID and time stamp, or if an outer signal occurred.

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x08	TX_BUF	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x09	TX_FREE	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0C	TX_ID	IDONL	-	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0D	TX_TRG	BUSY	TXNW	-	-	-	-	TIMEST	OUTER
0x10	TX_BYTE_0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x11	TX_BYTE_1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x12	TX_BYTE_2	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x13	TX_BYTE_3	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x14	TX_BYTE_4	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x15	TX_BYTE_5	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x16	TX_BYTE_6	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x17	TX_BYTE_7	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x18	TIMEST_0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x19	TIMEST_1	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
0x1A	TIMEST_2	BIT 23	BIT 22	BIT 21	BIT 20	BIT 19	BIT 18	BIT 17	BIT 16
0x1B	TIMEST_3	BIT 31	BIT 30	BIT 29	BIT 28	BIT 27	BIT 26	BIT 25	BIT 24

Table 1.4: Tx registers

1.3.2.1 TX_BUF

This read/write register selects the TX buffer [6]. This value is incremented by one by writing 1 to the BUSY bit in TX_TRG[7].

1.3.2.2 TX_FREE

This read/write register holds the first free TX buffer. The value 0xFF means that there is no free TX buffer.

1.3.2.3 TX_ID

This read/write register sets the ID settings for the selected buffer.

BIT 0 - 5 Sets the ID

IDONL 0 - Device sends data bytes
1 - Device sends only the ID to the bus. It is active only if the device is a master.

1.3.2.4 TX_TRG

This read/write register determines which signal triggers the transmission.

OUTER 1 - An outer signal is used to trigger the transmission.

TIMEST 1 - The time stamp is used to trigger the transmission. If the time is equal to the time stamp, a message is to be transmitted.

TXNW 1 - The message is sent immediately[8]

BUSY 0 - Buffer is empty
1 - Buffer is full
Writing 1 to this bit sets this buffer to 1 and the TX_BUF register is incremented by one. When the message is transmitted, this bit is set to 0.

The mechanism of triggering is shown in the figure 1.4.

If the device is a slave, receives an ID, finds this ID in registers, the BUSY flag is set, and Send_TX_Buffer outer or timer is set, the message is transmitted and the BUSY flag

[6] The number of buffers is adjustable. The maximal number of Tx registers is 255 (0 to 254)
[7] It is set to 0 when the number of buffers is reached.
[8] The flag 'ready for sending' is set - it has the same priority as if the time stamp event occurred.

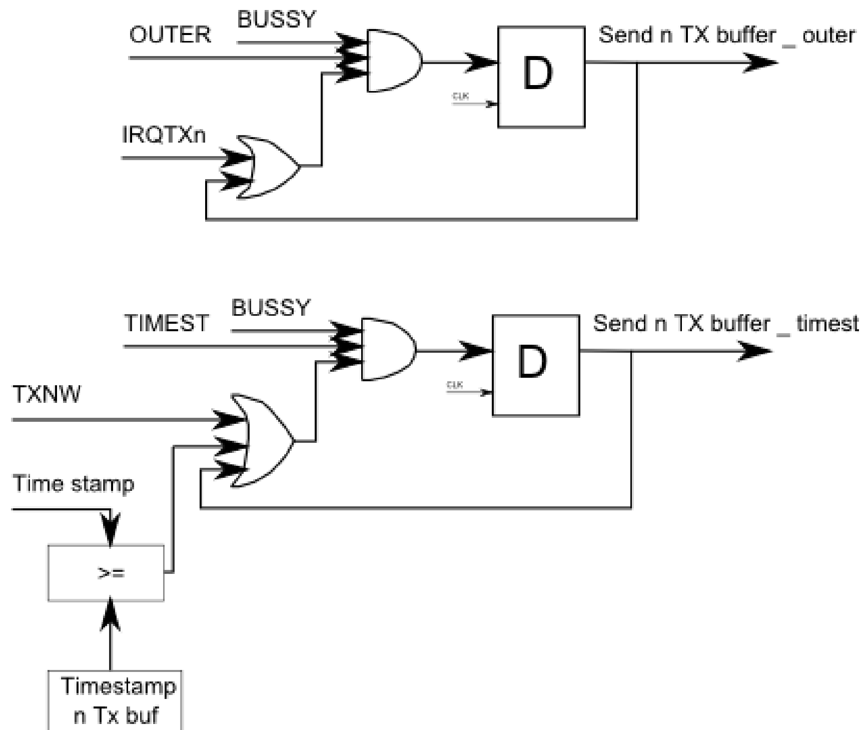

Figure 1.4: Trigger system

is cleared. If the device is a master, the LIN bus is free, and some signal Send_TX_Buffer outer or timer is set, then the master transmits the message according to the priority (time stamp or outer). If no message in the TX buffer has the BUSY flag set to 1 or all Send_TX_Buffer outer or timer flags are zero, then the master stays idle.

1.3.2.5 TX_BYTE

This read/write registers set bytes in the message

1.3.2.6 TIMEST

This read/write registers set the trigger time stamp. If the TIMEST bit in TX_TRG is 0, it is ignored. If time stamp is shorter than 32 bits, not assessed bits return an unspecified value.

1.3.3 LIN controller Rx registers

The LIN Rx registers are shown in the figure 1.5. The depth of the buffer is adjustable (in the source code). It behaves like a FIFO (data are stored in the RAM; only a pointer

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x1C	MSG_CNT	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x1D	RX_ID	-	-	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x1E	TIMEST_NB	-	-	-	-	BIT 3	BIT 2	BIT 1	BIT 0
0x20	RX_BYTE_0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x21	RX_BYTE_1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x22	RX_BYTE_2	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x23	RX_BYTE_3	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x24	RX_BYTE_4	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x25	RX_BYTE_5	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x26	RX_BYTE_6	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x27	RX_BYTE_7	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x28	RX_TIMEST_0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x29	RX_TIMEST_1	BIT 15	BIT 14	BIT 13	BIT 12	BIT 11	BIT 10	BIT 9	BIT 8
0x2A	RX_TIMEST_2	BIT 23	BIT 22	BIT 21	BIT 20	BIT 19	BIT 18	BIT 17	BIT 16
0x2B	RX_TIMEST_3	BIT 31	BIT 30	BIT 29	BIT 28	BIT 27	BIT 26	BIT 25	BIT 24
0x2C	RX_READ	-	-	-	-	-	-	-	-

Table 1.5: Rx registers

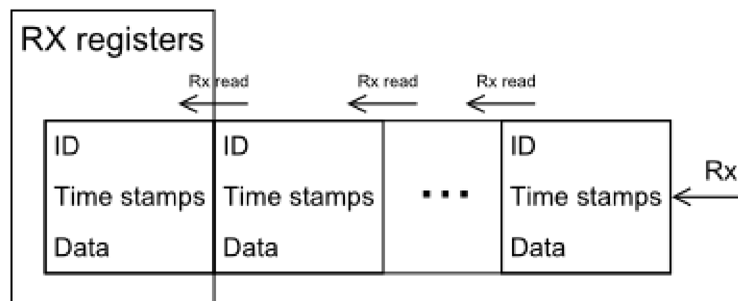

Figure 1.5: Rx buffer

is incremented). It is possible to read the oldest unread message.

1.3.3.1 MSG_CNT

This read-only register holds the number of messages in the Rx buffer.

1.3.3.2 RX_ID

This read-only register holds the ID of the received message.

1.3.3.3 RX_TIMEST_NB

This read/write register sets which time stamp is in RX_TIMEST register. The time stamp is recorded at the end of the stop bit.

- 0 - ID
- 1 - 9 - byte (+ checksum[9]) 1 - 8 respectively

1.3.3.4 RX_BYTE

This read-only register holds the data bytes of the received message.

1.3.3.5 RX_TIMEST

This read-only register holds the time stamp of the event addressed in the RX_TIMEST_NB register.

1.3.3.6 RX_READ

This write-only register sets the flag that this message was read. The next message (if any) is shifted in the RX registers.

1.4 Controller architecture

The LIN controller architecture is based on the Xilinx LIN controller [1] (Xilinx provides the source code in VHDL) which provides basic functions for sending the ID, data

[9] E.g. if the message contains 4 data bytes, the checksum is the 5th byte

bytes and the checksum. The LIN controller has been changed so that it supports the desired functionality. The Xilinx LIN controller registers were changed, so they are not compatible.

The LIN controller consists of nine components. The block diagram[10] is shown in the figure 1.6. The controller contains a new block, the message controller, which is a state machine that controls the transmission and the reception of the whole messages.

The core state machine was upgraded in order to be compatible with the LIN 2.x specification. Due to this, also the receiver block must have been changed to generate auto baud rate. A very significant change was made in the configuration registers. A RAM memory was added to this component, the registers were readdressed, and the interface was changed to be compatible with the WISHBONE specification.

1.4.1 Checksum gen

The checksum is calculated according to the LIN specification. This component contains a parallel data input and output, and a clock reset and clear.

1.4.2 Parity gen

The parity generator has a parallel input (5 bits) and a two-bit output. It generates parity bits according to the LIN specification.

1.4.3 Divider

The clock divider generates a bit rate from the clock input. This bit rate is adjustable with clk_div1 and clk_div2 inputs. It also generates a clock which is 16 times faster than the bit rate.

1.4.4 Majority sampler

The majority sampler uses bit_clkx16 (16 times faster clock than a bit clock), and makes an average of the last 16 samples. It causes the output (SER_IN) to be delayed for 0.5 bit.

[10] However, not all connections are showed.

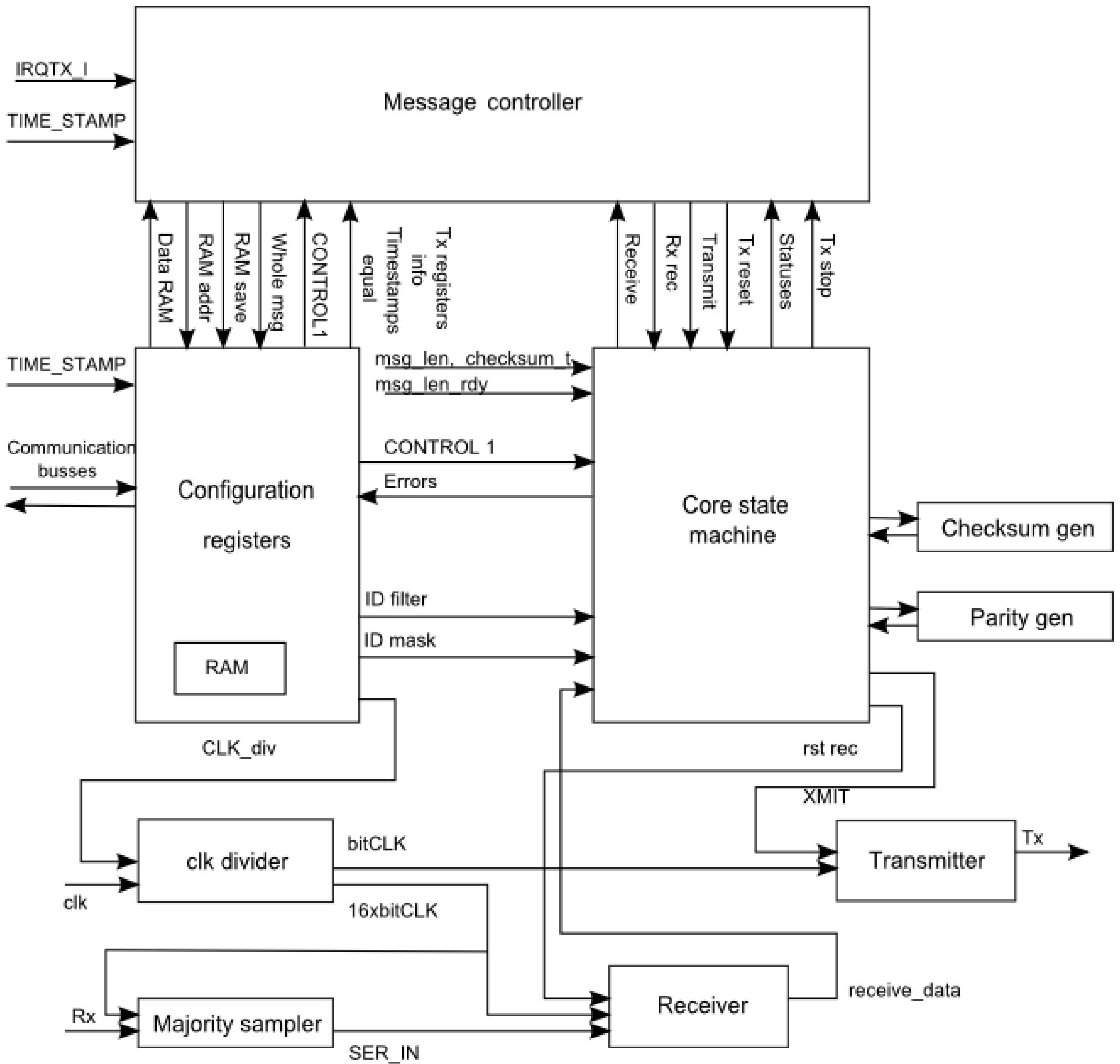

Figure 1.6: Block diagram

1.4.5 Receiver

The receiver is implemented as a shift register. It is synchronized on the start bit and it takes the sample in the middle of the bit. The majority sampler delays the input for 0.5 bit and receiver samples in the middle of the bit. Therefore, the bit reception is delayed for one bit. The receiver also supports calculating the automatic bit rate for which a 19 bit counter is needed[11].

1.4.6 Transmitter

The transmitter receives the whole byte and transmits it to the bus (generating the start and stop bits). After the byte is transmitted, the transmitter sets the DONE flag.

The transmitter is also responsible for checking whether the transmitted data are equal to the received data. If they do not match, the xmit_error flag is set.

1.4.7 Configuration registers

The configuration registers store all settings for the LIN controller. They are accessible via the WISHBONE slave interface. All settings are stored in the registers. TX_BYTEs, TX_IDs, RX_BYTEs, RX_IDs and RX_TIMESTAMPs are stored in the RAM. The map of the RAM is shown in the figure 1.7. The number of Tx and Rx buffers and the time stamp length is adjustable. A true dual port RAM (single clock, active on falling edge of CLK_I) is implemented. It supports reading and writing to the RAM from the μController interface and the message interface at the same time.

1.4.8 Core state machine

The core state machine includes three main processes: master_xmit, receive_prc and xmit_prc.

- Master_xmit process sends the break, the synchronization byte 0x55 and the ID to the bus (the bus must be idle and the controller must be a master), and then becomes idle. The following pseudo code describes this behaviour:

[11] 12 bits are used because CLK_DIV can have 12 bits, 4 more bits are needed for dividing CLKx16, and the other 3 bits are for counting 8 data bits

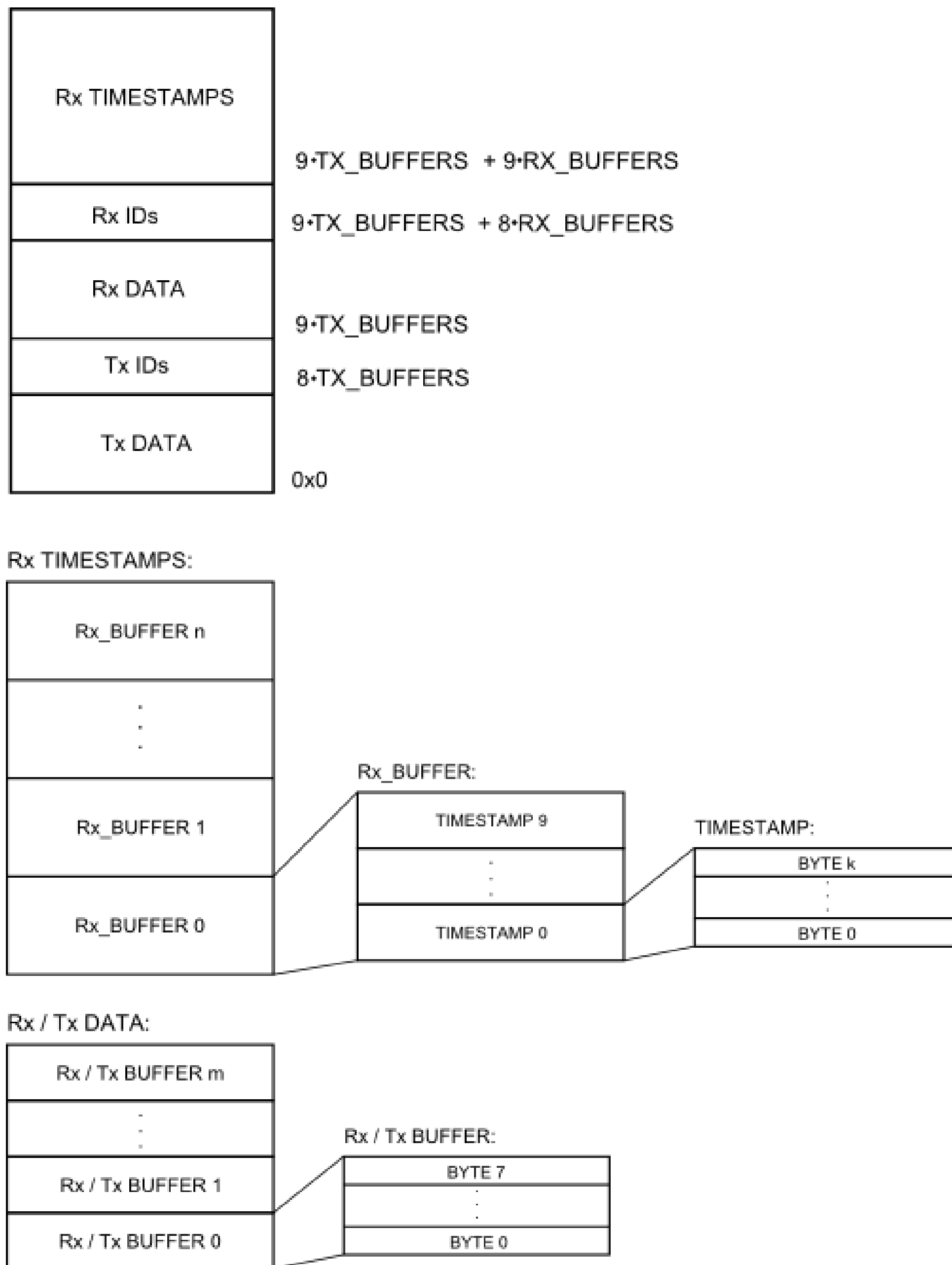

Figure 1.7: RAM map

```
while(1){
          if (Receive_prc = idle && device = master &&
                              transmit_byte = 1){
                    generate_break();
                    generate_1();
                    generate_0x55();
                    send_ID();
}         }
```

- Receive_prc receives a message as a sequence of an ID, data bytes and a checksum, checks its values, and sets errors flags in case an error occurs. In the pseudo code:

```
while(1){
          if (error == 1 || timeout = 1) go idle;
          switch slave_state {
                    idle : if (break_detected())
                              go rec_0x55;
                    rec_0x55 : if (0x55 byte received)
                              go receiveID;
                    receiveID : if (ID received &&
                     ID_filter_and_mask_matched()){
                              if (LIN13 = 1 and LIN database rdy){
                                        set_counters();
                                        go rec_byte;}
                              else {
                                        set counters(according to ID);
                                        go rec_byte; }
                              }
                    rec_byte : if (byte_received())
                                        counter--;
                                        if (counter = 0)
                                                  go rec_checksum;
                    rec_checksum : if (chkck_rec()){
                              if (checksum_matched())
                                        msg_complete;
                                        go idle;
```

```
                } else{
                        wrong checksum;
                        go idle;
}               }
```

- Xmit_prc sends data bytes and the checksum (as the last byte). The byte counter is set by the receiver process. Pseudo code:

```
while(1){
        if(Receive_prc = rec_byte and transmit_byte = 1){
                transmit_byte();
                counter--;
                if (counter = 0)
                        send_checksum();
        }
}
```

1.4.9 Message controller

The message controller is a completely new block, which controls the data flow. This component has an access to the configuration registers and the core state machine. It obtains information about Rx and Tx buffers from the configuration registers and gets statuses of the LIN bus from the core state machine.

This component consists of five main processes:

- Tx_machine is a process which sends data to the bus. The message controller contains registers for the whole Tx message (the RAM_controller_prc copies data before setting the signal for transmitting them). The pseudo code for this component is as follows:

```
while(1){
        if(transmit = 1 and LIN = idle){
                if (device = master) transmit_ID();
                while(not (stop)){
                        transmit_byte();
                        counter++;
}       }       }
```

- Rx_machine receives break, then ID, data bytes and the checksum. The pseudo code is as follows:

```
while(1){
        if(break = 1)go idle;
        switch rec_state :
                case idle :
                        if (rec ID)
                                save_ID();
                                go rec_data;
                case rec_data :
                        if (msg_complete) go idle;
                        if (rec_data()) {
                                save();
                                counter++;
}                       }
```

- Outer_signal_mach is shown in the figure 1.4.
- Main_state_mach controls behaviour of the whole message controller. It can be described by the following pseudo code:

```
while(1){
        if(tx_prc = idle && RAM_controller = idle) {
         if (device = master && LIN_bus = idle) {
                if (OUTER_sig_prior = 1 && outer_sig = 1)
                        copy_outer_Tx_buf();
                else if(OUTER_sig_prior = 0 && timest_sig = 1)
                        copy_timest_buf();
                else if (outer_sig = 1)
                        copy_outer_Tx_buf();
                else if (timest_sig = 1)
                        copy_timest_buf();
         }
         else if (device = slave && received ID){
                find_slave_answer();
}       }}
```

This process generates signals for the RAM_controller_prc.

- RAM_controller_prc is the most robust process in this component. It controls reading and saving data (message bytes and time stamps) to RAM and because IDs of Tx buffer are stored in RAM memory it is responsible for searching the answer for a slave response. The pseudo code for this component :

```
void save_timestamp(){
        for(i = 0; i < timestamp.length(); i++){
                address_RAM();
                save_byte();
}       }



void save_data(){
        if(data = ID) address_ID();
        else address_msg_byte();
        save_byte();
}

//method copy data(id + 8 bytes) from RAM to registers
void copy_buffer(in buffer_number){
        address_ID(buffer_number);
        ID = load_data_from_RAM();
        for(i = 0; i < 8; i++){
                address_byte(buffer_number, i);
                byte(i) = load_data_from_RAM();
        }
        set_tx_buff_empty();
        transmit_data();
}

void find_slave_answ(){
        for i in 0 to tx_buffers - 1 {
                if (id_matches() &&
                 (outer_sig || timest_sig)) {
```

total logic elements	total registers	total memory bits	max frequency [MHz]
1171	612	1856	69.2

Table 1.6: Implementation in the FPGA

```
                    copy_buffer(i);
                    break;
}       }           }


while(1){
        if (save_timestamp = 1) save_timestamp();
        if (save_data = 1) save_data;
        if (copy_outer_Tx_buf = 1){
                copy_buffer(min(buf_with_outer_sig()));
        }else if (copy_timest_buf = 1){
                copy_buffer(min(buf_with_timestamp_sig()));
        }
        if (find_slave_answer = 1) {
                find_slave_answ();
}       }
```

1.5 Conclusion

The LIN controller has been tested on the Cyclone II EP2C35F672C6 with four Tx and Rx buffers and 32-bit time stamp and it meets all specifications mentioned in this documentation. The compilation was made with the respect to the smallest area on the chip in the Quartus II version 9.1. The result can be found in the table 1.6.

The maximal frequency is 69 MHz, which is more than sufficient to support the maximum LIN bit rate of 19200 bps.

Chapter 2

LIN trigger

2.1 Overview

A trigger is a very important component, which is used in many devices. In general, a trigger recognizes some pattern in the input signal, and if some event (e.g. a rising or falling edge) occurs, the trigger recognizes the start of this event and sets some signal.

The LIN trigger is a trigger which is adapted for a LIN frame. The trigger recognizes much more patterns than just the falling or rising edge of the signal. It supports triggering on three basic types of input: incorrect frame, data pattern, and specific time events.

The incorrect data frame can be caused by the following errors: a slave does not respond to the master's ID, the frame is too short (not all of the data bytes were sent to the bus before the timeout exceeded), a parity error, or a wrong checksum.

The data pattern triggering means that the trigger reads input bits, and if a desired sequence of bits appears in the input signal, the trigger sets the output signal.

The measuring of specific time events is very useful for capturing bus errors which are not possible to be captured with the previous options. This option offers capturing events when the message frame, the time between two bytes, two messages, or a break are longer than some specific time. It also offers capturing break signals that are shorter than some specific time (for example some invalid character in the input signal).

This LIN trigger offers all kinds of triggering mentioned above, and therefore it is one of the most important component in the car testing system. The type of the event, data, and the time[1] settings are adjustable via the parallel interface.

[1]Time is measured in time stamp units.

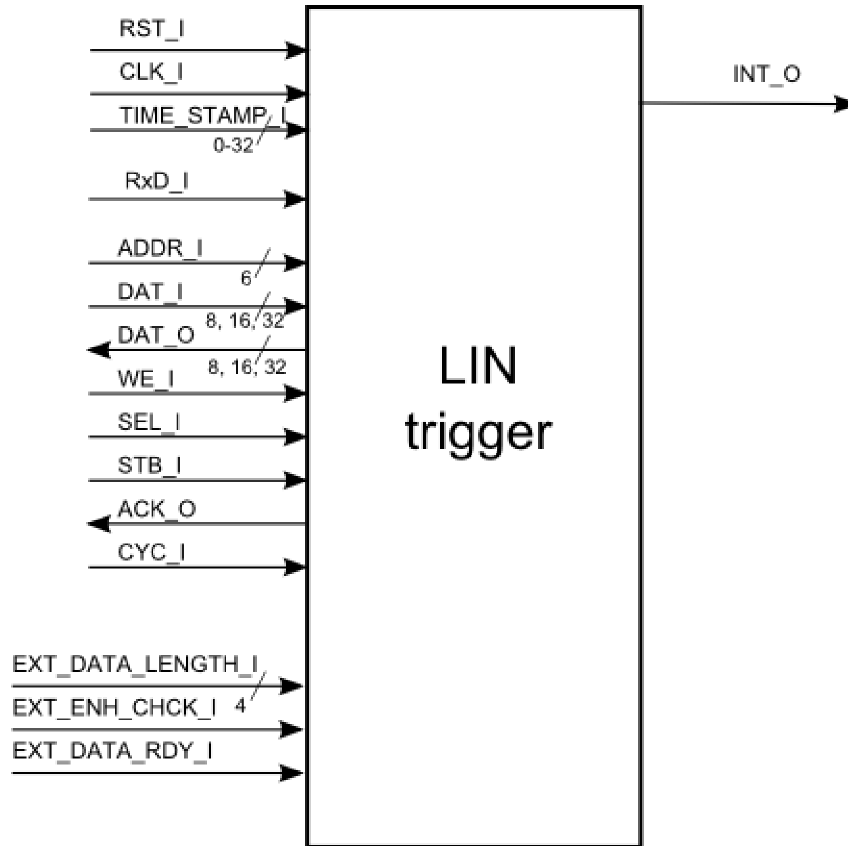

Figure 2.1: LIN trigger

2.2 Interface

The LIN trigger IP is a core with a WISHBONE slave interface [2]. The interface is compliant with the WISHBONE Slave Rev. B.3 interface with separated 32, 16, or 8-bit input/output data buses[2] (data width are adjusted in the source code for this component). The interface signals are shown in the figure 2.1 and their description can be found in the table 2.1.

The communication with the LIN trigger is shown in the figure 1.2 (writing to registers, ignoring the dash line) and 1.3 (reading from registers, ignoring the dash line).

[2]The design offers two interfaces. The first one, described in this document, is wishbone compatible. The other one has the same signals as the first one, except for the DAT_I and DAT_O. These two signals are replaced with DAT_IO. If WE_I = 1 or chip is not selected, then DAT_IO is in the state of high impedance.

Pin	Activity	Description
RST_I	HIGH	Reset signal
CLK_I	-	Clock input
TIME_STAMP_I[x_1:0]	-	Time stamp input (x_1 : 0 - 32 bits)
ADDR_I[5 : 2,1,0]	-	Address of the register (for the 32, 16 and 8-bit interface, respectively. Described in the table 4.2)
DAT_I[31,15,7:0]	-	Data input (32, 16, or 8 bits)
DAT_O[31,15,7:0]	-	Data output (32, 16, or 8 bits), DAT_O is equal to DAT_I when the device is not selected
WE_I	HIGH	Bus access signal: HIGH for the write transfer, LOW for the read transfer
SEL_I	HIGH	Device select bit
STB_I	HIGH	Strobe input indicated a valid data transfer cycle
ACK_O	HIGH	"Acknowledge output" indicates the termination of a normal bus cycle
CYC_I	HIGH	"Valid bus cycle in progress" bit
INT_O	HIGH	"Interrupt output" bit
EXT_DATA_LENGTH_I[3:0]		Data length from an external LIN database
EXT_ENH_CHCK_I	HIGH	Data checksum type signal from an external database
EXT_DATA_RDY_I	HIGH	Signal from an external database that data are ready
RxD_I	-	Receive data from the LIN transceiver

Table 2.1: Interface description

2.3 Register description

2.3.1 CONTROL1

This read/write register sets the LIN trigger.

CHCK Checksum error
- 0 - interrupt disabled
- 1 - interrupt enabled

PAR Wrong parity
- 0 - interrupt disabled
- 1 - interrupt enabled

FRAME Framing error
- 0 - interrupt disabled
- 1 - interrupt enabled

NOSL No slave response[3]
- 0 - interrupt disabled
- 1 - interrupt enabled
- 0 - interrupt disabled
- 1 - interrupt enabled

SHFR Short frame
- 0 - interrupt disabled
- 1 - interrupt enabled

Any interrupt source (even more than one) can be selected.

2.3.2 INTSRC

This read/write register selects the interrupt source. No more than one source can be selected.

If BRKL, BRKS, BTWM, BTWB, or MSGL bit is set, then the BYTE/TMST registers set the value N.

[3]It also indicates whether the message is shorter. For example in case 5 bytes (4 data + checksum) were expected but only 4 or less bytes were received during the timeout.

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	CONTROL1	-	-	-	SHFR	NOSL	FRAME	PAR	CHCK
0x01	INTSRC	-	-	DEF	MSGL	BTWB	BTWM	BRKS	BRKL
0x02	CLK_DIV1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x03	CLK_DIV2	ABAU	LIN13	-	-	BIT 11	BIT 10	BIT 9	BIT 8
0x04	DATA0/TMST0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x05	DATA1/TMST1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x06	DATA2/TMST2	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x07	DATA3/TMST3	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x08	DATA4	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x09	DATA5	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0A	DATA6	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0B	DATA7	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0C	DATA8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0D	DATA9	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x0E	LENGTH	RSTMS	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x10	MASK0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x11	MASK1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x12	MASK2	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x13	MASK3	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x14	MASK4	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x15	MASK5	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x16	MASK6	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x17	MASK7	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x18	MASK8	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x19	MASK9	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0

Table 2.2: Registers

Figure 2.2: Trigger on a gap between messages

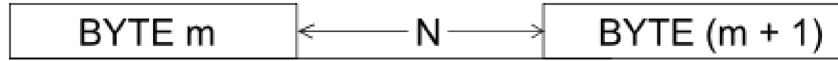

Figure 2.3: Trigger on a gap between bytes

BRKL The break is longer than N
0 - interrupt disabled
1 - interrupt enabled

BRKS The break is shorter than N
0 - interrupt disabled
1 - interrupt enabled

BTWM The time between two messages (figure 2.2)
0 - interrupt disabled
1 - interrupt enabled

BTWB The time between two bytes (figure 2.3)
0 - interrupt disabled
1 - interrupt enabled

MSGL The length of a message (figure 2.4)
0 - interrupt disabled
1 - interrupt enabled

DEF Defined sequence (defined by the DATA and MASK registers)
0 - interrupt disabled
1 - interrupt enabled

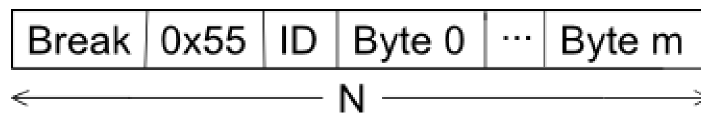

Figure 2.4: A message is longer than N

2.3.3 CLK_DIV

ABAU 0 - auto baud rate disabled
1 - auto baud rate enabled (slave controller only) [4]

LIN13 0 - the message length and the checksum according to the LIN 1.3 specification
1 - the message length and the checksum according to an external table - it must be connected to the LIN database

This read/write registers set the bit rate

$$CLK_DIV = \frac{clk}{16 \cdot bit_rate} \tag{2.1}$$

Divisors 1 or 0 are invalid.

2.3.4 DATA/TMST

This read/write registers set the time event or data comparison, depending on the INTSRC register setting.

2.3.5 LENGTH

This read/write register sets the length of the message in the data comparison. If the mode defined sequence is not used, this register is ignored. If it is set to "0000000", then the data length does not matter.

RSTMS If this bit is set, then the mask is reset to the reset value.

2.3.6 MASK

This read/write registers set the mask.
0 - bit is relevant
1 - bit is not relevant
All MASKx registers can be set to 0xFF by setting the RSTMS bit in the LENGTH register to 1.

[4] CLK_DIV register must be set to the default value, or the bit rate must be smaller than the smallest bit rate on the bus.

CONTROL1	0x0
INTSRC	0x0
CLK_DIV1	0x2
CLK_DIV2	0x0
TMSTx	0x0
LENGTH	0x0
MASKx	0xFF

Table 2.3: Reset values

2.4 Reset states

Default values after reset are specified in the table 2.3.

2.5 Data comparison

Data comparison mode is "enable" if the DEF bit in the INTSRC register is set. Data and mask registers are 10 bytes long (ID + 8 data bytes + checksum). Data comparison works as described in the figure 2.5. This figure also shows the data alignment. Data comparison is bit sensitive. For example, it can trigger to 13 bits. After the 13 bits are received, they are evaluated in order to find out whether they match the data and mask bits.

The interrupt is set when the equation 2.2 holds and the number of received bits is equalto the value of the LENGTH register or the LENGTH is equal to zero.

$$(shift\ register == data) \mid mask \tag{2.2}$$

According to the equation 2.2, if the LENGTH is set for example to 5, then the MASK bits 0 to 74 must be set to 1.

2.5.1 Example

Triggering on the sequence

```
ID = 1000 0000
```

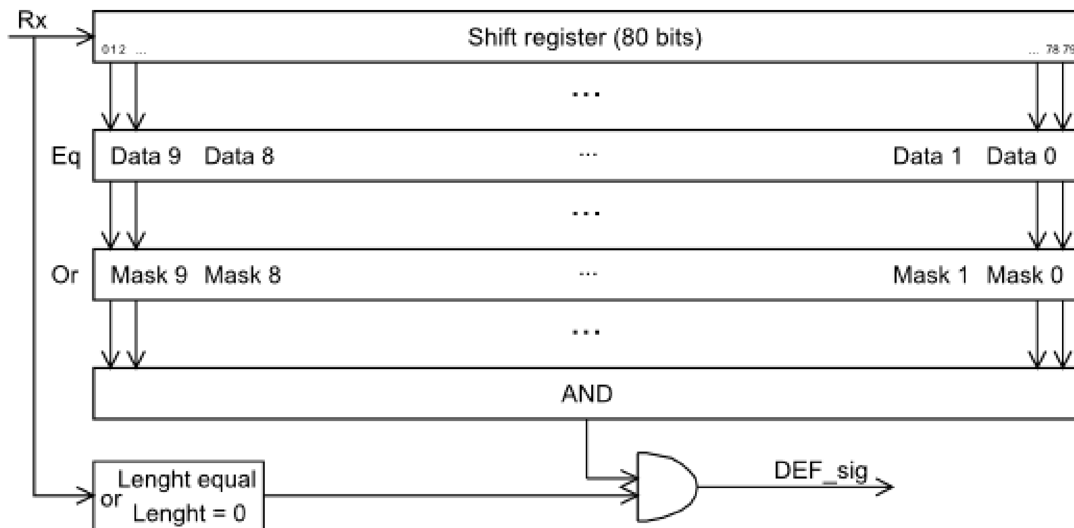

Figure 2.5: Data comparison schematics

```
first byte = 1100 xxxx
```

where "x" means "do not care".

In this example the following registers must be set:

```
DATA8 = 1000 0000
DATA9 = 1100 0000

MASK8 = 0000 0000
MASK9 = 0000 1111

LENGTH = 0001 0000

MASK0-7 = 1111 1111 (reset state)
```

If the defined sequence appears on the bus, the interrupt is set.

2.6 Output

The interrupt output is shown in the figure 2.6. Any number of interrupt sources can be selected from the CONTROL1 register. Only one can be selected from the INTSRC register.

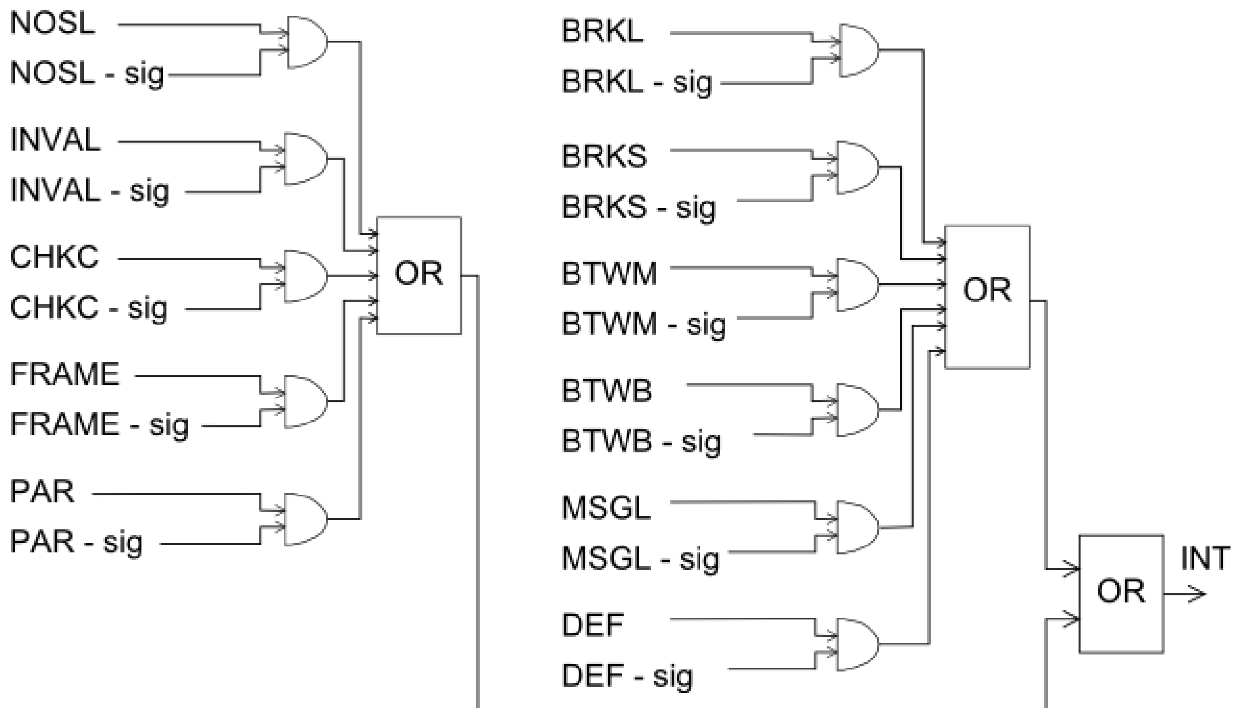

Figure 2.6: Interrupt output

The interrupt signal is set only if an event occurred. That means the signal can sometimes be very short (one clock cycle), sometimes it can be longer.

The interrupt signal is one clock cycle long if any event specified in the CONTROL1 register or BRKL in INTSRC is set.

The interrupt signal defined in the INTSRC register, except for BRKL, is cleared when the event is not on the bus anymore.

2.7 Controller architecture

The architecture is based on the Xilinx LIN controller [1] again. However, this model was significantly reduced. The schematics, which consists of three main components, is shown in the figure 2.7. These components are: configuration registers, receiver and core state machine (parity generator, majority sampler and clock divider are described in the LIN controller).

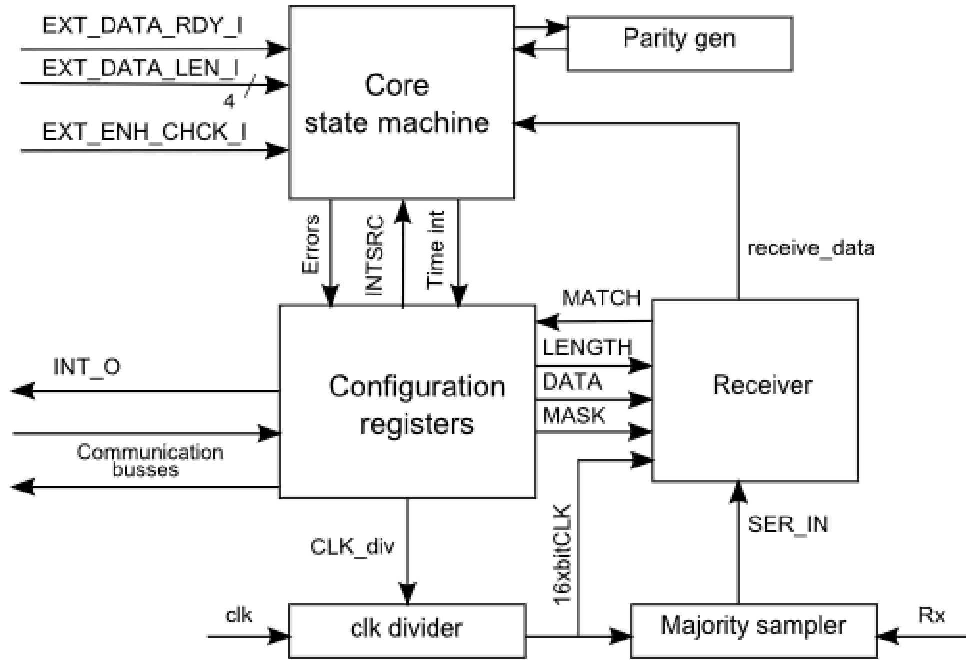

Figure 2.7: Schematics of the LIN trigger

2.7.1 Configuration registers

The configuration registers store all settings which are described in the table 2.2. All values are stored in registers and no RAM is needed.

2.7.2 Receiver

The receiver is responsible for receiving messages[5]. The receiver also contains 80-bit shift register which stores the whole message. This register stores data bits only (start and stop bits are ignored). The comparison works as follows:

```
dataMatch <= '1';
for i in 0 to 79 loop
        if ((DATA(i) /= SHIFT_REG(i)) and MASK(i) = '0') then
                dataMatch <= '0';
        end if;
end loop;
```

and the output is defined as:

[5]The majority sampler delays the input for 0.5 bit and receiver samples in the middle of the bit. Therefore, the bit reception is delayed for one bit.

data width[bit]	total logic elements	total registers	total memory bits	max freq. [MHz]
8	914	442	0	51.21

Table 2.4: Compilation results

```
data_matches <= '1' when (dataMatch = '1' and
                (LENGTH = 0 or LENGTH = biteCounter)) else '0';
```

Therefore, the mask must be set to 1 for all unused bits.

2.7.3 Core state machine

The core state machine contains a slave machine only. This process determines errors on the bus (parity, frame, checksum, no slave response), and it also measures events defined in the INTSRC register. Time measurement is based on a very simple process - when a defined event occurs on the bus, the time stamp is stored to the register, and an interrupt is set when

$$savedTimestamp + DifferenceTimestamp \leq ActualTimestamp \tag{2.3}$$

The savedTimestamp register is set to the actual time stamp when the defined event is not on the bus. Evaluating of this equation takes two clock cycles. In the first cycle the addition and in the second one the comparison is executed.

This principle is used for all time events defined in the INTSRC, except for BRKL, which generates an interrupt when a short break is detected (it watches for a situation when the break has ended and the equation above does not hold).

2.8 Conclusion

The LIN trigger has been tested on the Cyclone II EP2C35F672C6 with a 32-bit time stamp and it meets functionality mentioned in this document. The compilation was made with the respect to the smallest area on the chip in the Quartus II version 9.1. The result can be found in the table 2.4.

Many logic elements and registers are needed for the trigger. However, we must consider that the trigger contains three very long registers, which take the most of these elements.

Chapter 3

LIN injector

3.1 Overview

An injector (a programmable generator) is a very important component, which allows to generate a desired waveform. These generators are able to generate any waveform, which can be transmitted to the bus and recognized as a valid message. In this case, a normal controller could be used, and it would be certainly better. On the other hand, a generator offers much more. Testing requires generating frames that are corrupted, or that are not even message frames, which is not possible to achieve with a normal controller.

The generator takes a sequence of bits (the length is adjustable) and transmits it to the bus without any feedback. This sequence is programmable, and thus it can generate any desired waveform. If the desired sequence is a valid LIN message, it must contain a synchronization break, synchronization byte 0x55, protected identifier, and another data bytes, including a checksum. The injector sends the message as a stream of bits, and therefore the sequence must contain the start and stop bit, synchronization break with a valid length, and so on.

The advantage of this injector is that it can generate invalid characters, as well as a message that is corrupted. It can generate invalid frames such as parity, framing, checksum, or a short frame error, as well as an invalid bit rate, a frame that is longer than the maximum frame length, or a break that is too short or too long, and many others.

The injector offers two possibilities how to start the transmitting. It can either be started by an outer signal, or it can be triggered by a μController.

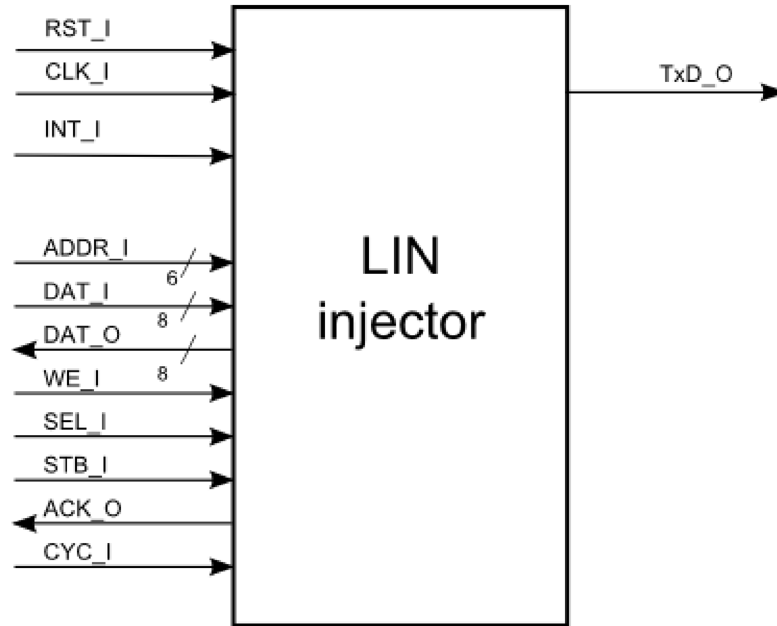

Figure 3.1: Interface

3.2 Interface

The LIN injector IP is a core with a WISHBONE slave interface [2]. The interface is compliant with the WISHBONE Slave Rev. B.3 interface with separated 8-bit input/output data buses[1]. The interface signals are shown in the figure 3.1 and their description can be found in the table 3.1.

The communication with the LIN injector is shown in the figure 1.2 (writing to registers, ignoring the dash line) and 1.3 (reading from registers, ignoring the dash line).

Pin	Activity	Description
RST_I	HIGH	Reset signal
CLK_I	-	Clock input
ADDR_I[5 : 0]	-	Address of the register
DAT_I[7:0]	-	Data input (8 bits)
DAT_O[7:0]	-	Data output (8 bits), DAT_O is equal to DAT_I when the device is not selected
WE_I	HIGH	Bus access signal: HIGH for the write transfer, LOW for the read transfer
SEL_I	HIGH	Device select bit
STB_I	HIGH	Strobe input indicated a valid data transfer cycle
ACK_O	HIGH	"Acknowledge output" indicates the termination of a normal bus cycle
CYC_I	HIGH	"Valid bus cycle in progress" bit
INT_I	HIGH	"Interrupt input" bit which starts the transmitting of the sequence (The injector must be enabled)
TxD_O	-	Data output

Table 3.1: Interface description

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	CLK_DIV1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x01	CLK_DIV2	-	-	-	-	BIT 11	BIT 10	BIT 9	BIT 8
0x02	LENGTH	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x03	CONFIG	STAT	-	-	-	-	-	NOW	ACTV
0x04	DATA0	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x05	DATA1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x06	...	...	...	...	...	...	...	...	...
0xNN	DATAN	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0

Table 3.2: Registers

3.3 Register description

3.3.1 CLK_DIV

This read/write registers set the bit rate

$$CLK_DIV = \frac{clk}{16 \cdot bit_rate} \tag{3.1}$$

Divisors 1 or 0 are invalid.

3.3.2 LENGTH

This read/write register determines how many bits are to be transmitted.

3.3.3 CONFIG

ACTV 0 - device disabled
1 - device enabled

NOW Writing 1 to the register sends the message immediately. If the trasmitting is already in progress, this bit has no effect.

STAT 1 - transmitting is in progress
0 - waiting for a trigger event

3.3.4 DATAx

This read/write registers set the data bits which are to be transmitted. The number of these registers is adjustable in the source code (the minimal value is 1, the maximal value is 31; the number specified in "generic" in the source code represents the length in bites, that is 8 and 248 respectively).

DATA0 (bit 0) is transmitted first; DATAN (bit 7) is transmitted last [2].

[1] The design offers two interfaces. The first one, described in this document, is wishbone compatible. The other one has the same signals as the first one, except for the DAT_I and DAT_O. These two signals are replaced with DAT_IO. If WE_I = 1 or chip is not selected, then DAT_IO is in the state of high impedance.

[2] only if the whole buffer is transmitted (LENGTH is equal to 8·N)

CLK_DIV1	0x2
CLK_DIV2	0x0
LENGTH	0x0
CONFIG	0x0

Table 3.3: Reset values

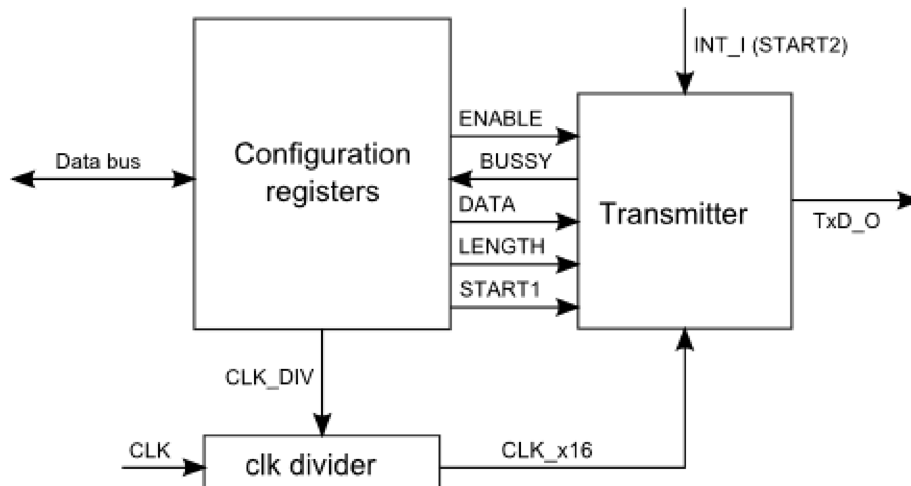

Figure 3.2: Architecture

3.4 Reset values

Default values after reset are specified in the table 3.3.

3.5 Injector architecture

The schematics, which consists of three main components, is shown in the figure 3.2. The components are: configuration registers, clock divider, and transmitter.

3.5.1 Configuration registers

The configuration registers store all settings which are described in the table 2.2. All values are stored in registers and no RAM is needed.

3.5.2 Clock divider

The clock divider receives the clock, and divides it according to CLK_DIV settings.

3.5.3 Transmitter

The transmitter transmits the data stored in the DATA registers to the bus. It can be in two different states: idle or transmitting. If the transmitter is in the idle state (and transmitting is enabled) , it waits for the start signal (NOW in the CONFIG or the outer signal INT_I). Then it goes to the transmitting state, and transmits data until the value in LENGTH is equal to the number of the transmitted bits, or the device is disabled. This procedure is decribed by the following pseudo code:

```
while(1){
        if(state = transmit){
                TxD = DATA(bitCounter);
        } else {
                TxD = 1;
        }
        switch state {
                idle :
                        if (enable = 1 && start_x = '1') {
                                bitCounter = 0;
                                state = transmit;
                        }
                transmit :
                        if (bit_clk) {
                                bitCounter++;
                        }
                        if (bitCounter = LENGTH || enable = 0) {
                                state = idle;
                        }
        }
}
```

total logic elements	total registers	total memory bits	max freq. [MHz]
334	203	0	107.31

Table 3.4: Compilation results

3.6 Conclusion

The LIN injector has been tested on the Cyclone II EP2C35F672C6 with 18 DATA registers (144 bits). The compilation was made with the respect to the smallest area on the chip in the Quartus II version 9.1. The result can be found in the table 3.4.

The injector meets all requirements described in the the documentation, and therefore it is suitable for an automotive testing system.

Chapter 4

LIN database

4.1 Overview

LIN specifications 1.x and 2.x are slightly different. The main difference is in the checksum and the length of the message. The length does not depend on the ID (as it did in version 1.1), and the type of checksum (enhanced/normal) can be different for different IDs. Some devices, available on the market, must be adjusted by a μController to the correct type and length of the message after the identifier is transmitted to the bus. However, this solution is not very suitable for a system consisting of many buses.

The LIN database is a device which solves this problem. This database holds records of all identifiers (the type of checksum and the length) which are adjustable by the μController. These records are adjusted when the system starts or are left default, which is compatible with the LIN 1.1 standard. The database is connected to the bus, and receives the frame. When an identifier is transmitted to the bus, the database generates output signals (shown in the figure 4.1). These signals contain information about the message length and the type of checksum.

The LIN controller and the trigger, which are compatible with these inputs, read these signals, and therefore they do not need to be adjusted by the μController. The recommended block connection between the database, LIN controller and trigger is shown in the figure 4.2.

This component is not very useful for a single block, which receives one identifier only. However, this database is a very important component for a system which transmits and receives the full range of IDs.

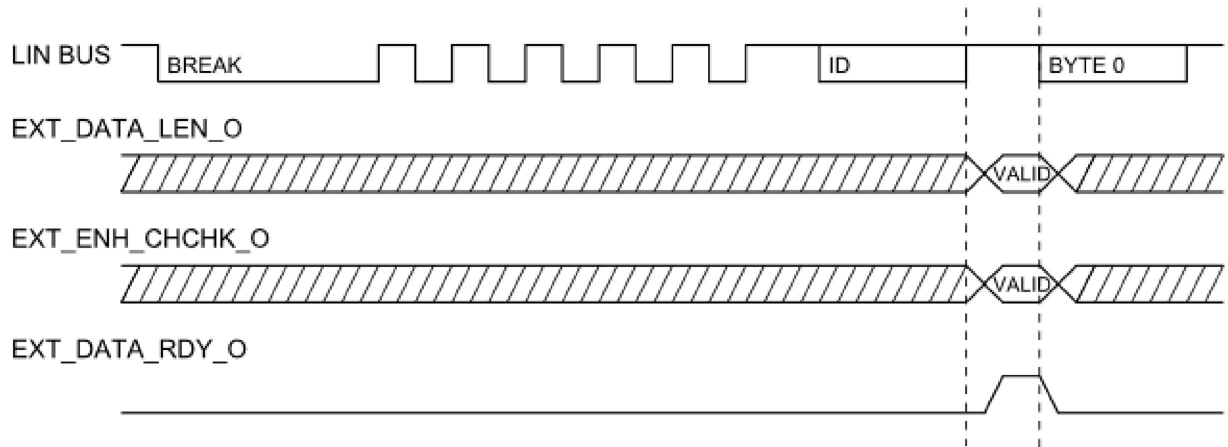

Figure 4.1: Signals from LIN database

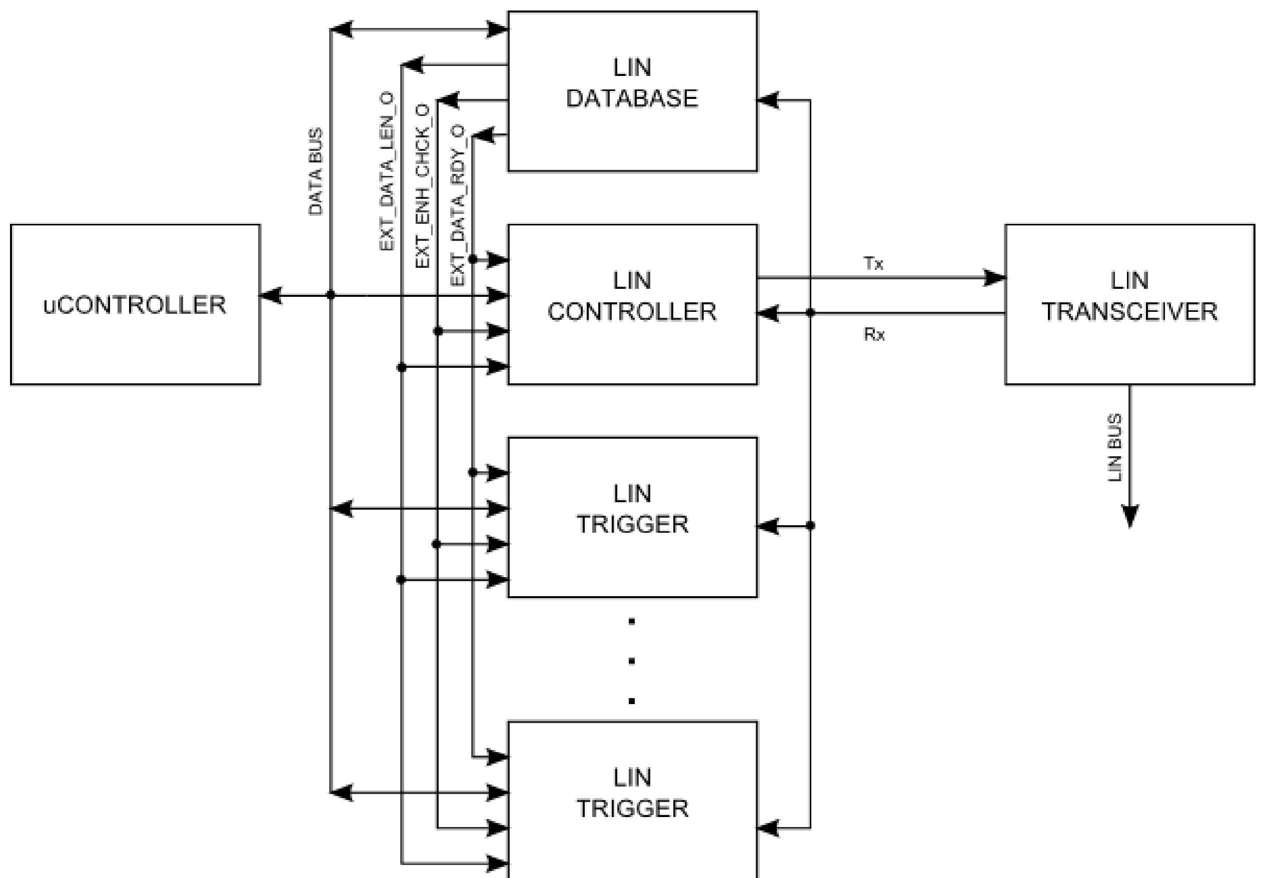

Figure 4.2: Block connection with controller and triggers

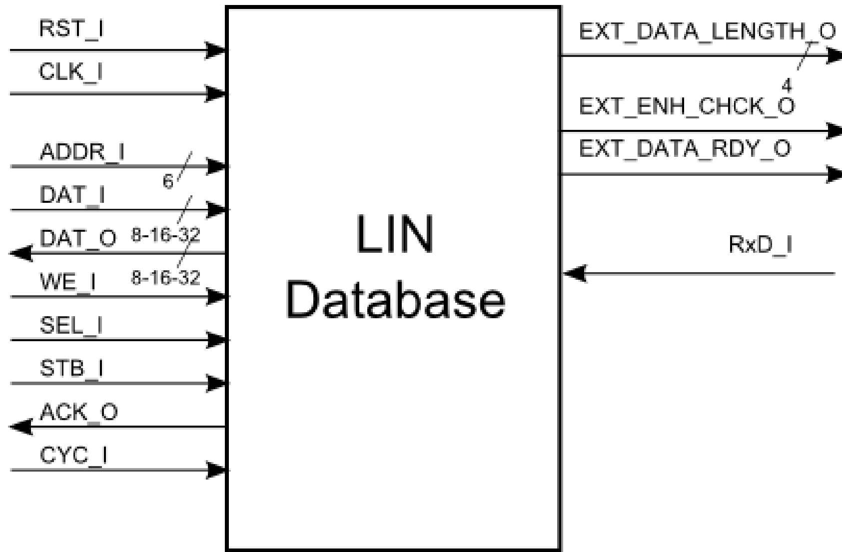

Figure 4.3: Interface

4.2 Interface

The LIN database is a core with the WISHBONE slave interface [2]. The Interface is compliant with the WISHBONE Slave Rev. B.3 interface with separated 8, 16 or 32-bit input/output data buses[1]. The interface signals are shown in the figure 4.3 and their description can be found in the table 4.1.

The LIN database interface offers three specific configurations for 8, 16, and 32-bit bus width - configuration LINDatabase8bit_WISHBONE, LINDatabase16bit_WISHBONE, and LINDatabase32bit_WISHBONE, respectively [2]. The DATA_LENGTH constant in generic must be set according to the data width.

The addressing of the registers differs for different data widths. For example, it is not possible for a 16-bit interface to address the registers 0x1 and 0x2. It is possible to address the registers 0x0 and 0x1, or 0x2 and 0x3. The address bits used for the addressing are shown in the table 4.2

[1]The design offers two interfaces. The first one, described in this document, is wishbone compatible. The other one has the same signals as the first one, except for the DAT_I and DAT_O. These two signals are replaced with DAT_IO. If WE_I = 1 or chip is not selected then DAT_IO is in the state of high impedance.

[2]A non WISHBONE compatible interface offers 8, 16, and 32-bit configurations LINDatabase8bit, LINDatabase16bit, and LINDatabase32bit, respectively

Pin	Activity	Description
RST_I	HIGH	Reset signal
CLK_I	-	Clock input
ADDR_I[5 : 2,1,0]	-	Address of the register (for the 32, 16 and 8-bit interface, respectively. Described in the table 4.2)
DAT_I[31,15,7:0]	-	Data input (32,16 or 8 bit)
DAT_O[31,15,7:0]	-	Data output (32,16 or 8 bit), DAT_O is equal to DAT_I when the device is not selected
WE_I	HIGH	Bus access signal : HIGH for write transfer, LOW for read transfer
SEL_I	HIGH	Device select bit
STB_I	HIGH	Strobe input indicated a valid data transfer cycle.
ACK_O	HIGH	Acknowledge output indicates a termination of a normal bus cycle
CYC_I	HIGH	”Valid bus cycle in progress“ bit
EXT_DATA_LENGTH_O[3:0]		Data length from an external LIN database
EXT_ENH_CHCK_O	HIGH	Data checksum type signal from an external database
EXT_DATA_RDY_O	HIGH	A signal from an external database that data are ready
RxD_I	-	Receive data from LIN transceiver

Table 4.1: Interface description

Data width[bit]	Address bits used
8	ADDR_I[5 : 0]
16	ADDR_I[5 : 1]
32	ADDR_I[5 : 2]

Table 4.2: Address bits

Offset	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x00	ID0	ENHA	SET	-	-	-	LEN 2	LEN 1	LEN 0
0x01	ID1	ENHA	SET	-	-	-	LEN 2	LEN 1	LEN 0
...	...	...	-	-	-	...	...	...	...
0x3D	ID3D	ENHA	SET	-	-	-	LEN 2	LEN 1	LEN 0
0x3E	CLK_DIV1	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
0x3F	CLK_DIV2	ABAU	-	-	-	BIT 11	BIT 10	BIT 9	BIT 8

Table 4.3: Registers

4.3 Registers

The registers are shown in the table 4.3. The registers contain IDs and a record about the length and the type of checksum. The address corresponds to the ID. The addresses 0x3E and 0x3F are used for the bit rate settings[3]

4.3.1 IDx

Read/write registers.

LEN - sets the length of the message; 0 means 8-byte message.

SET - 0 - data were not set in the register[4]
1 - data were set in the register

ENHA - 0 - classic checksum
1 - enhanced checksum

4.3.2 CLK_DIV

Read/writes registers

[3]Reserved frames shall not be used in a LIN 2.x cluster. Their frame identifiers are 62 (0x3E) and 63 (0x3F), LIN specification 2.1.

[4]A flag is set only for the groups of four. For example, if the ID0 register is set, this bit is set to ID1, ID2, and ID3.

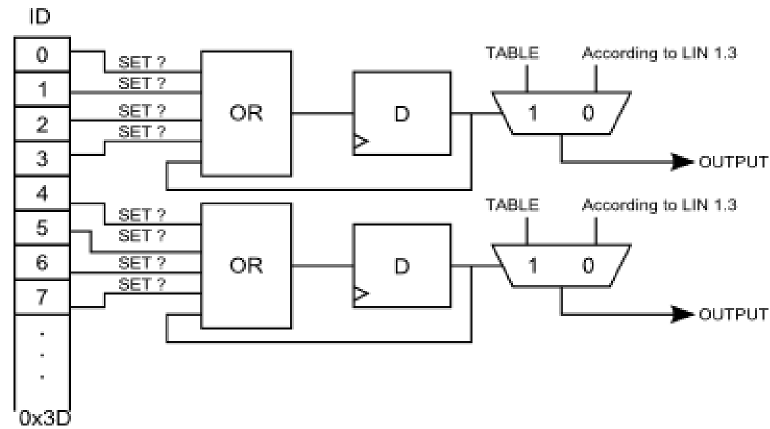

Figure 4.4: Choosing default or table values

CLK_DIV - sets bit rate

$$CLK_DIV = \frac{clk}{16 \cdot bit_rate} \quad (4.1)$$

Divisors 1 or 0 are invalid.

ABAU - 0 - auto baud rate disabled
1 - auto baud rate enabled [5]

4.4 Reset values

By default, the reset values are set to the classic checksum and length according to LIN 1.3. If any ID is set, then the bit indicating the "use table value" is set for all IDs with the same address (5 downto 2). This is shown in the figure 4.4.

4.5 Database architecture

The database architecture is based on the LIN controller core. Most of unnecessary functionalities were removed, and thus the core is significantly smaller than the LIN controller. The schematics of the LIN database is shown in the figure 4.5. It consists

[5]The CLK_DIV register must be set to the default value, or the bit rate must be smaller than the smallest bit rate on the bus.

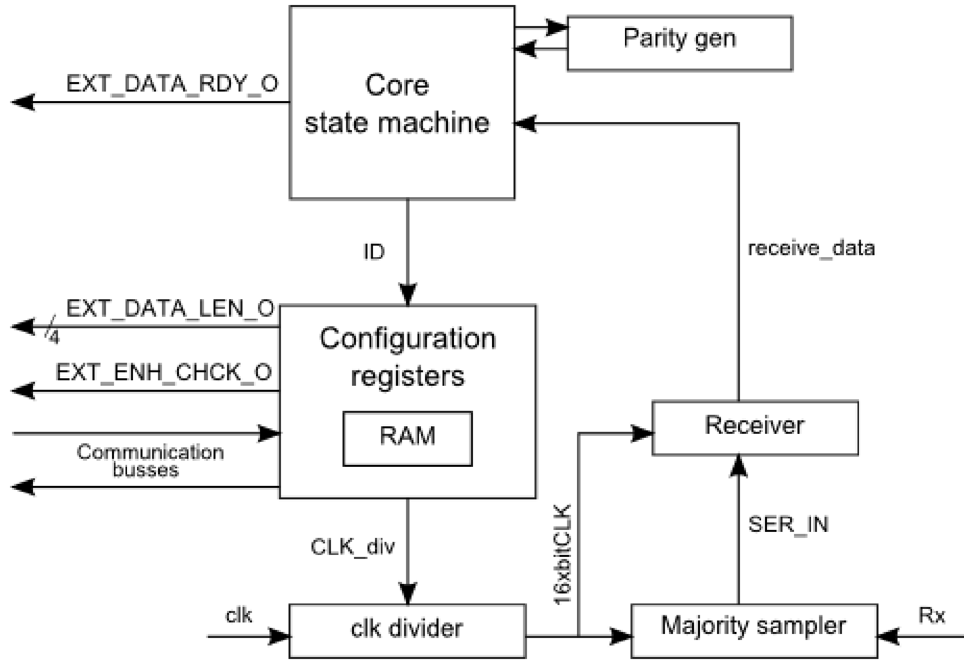

Figure 4.5: Schematics

of six simple blocks: configuration registers, core state machine, parity generator, clock divider, majority sampler, and receiver. The description of all blocks, except for the configuration registers and the core state machine, can be found in the LIN controller documentation or in [1].

4.5.1 Core state machine

The core state machine is a very simple process which can be described by the following pseudo code:

```
while(1){
        switch state {
                idle :
                        if(break detected){
                                state = synch0x55;
                        }
                synch0x55 :
                        if(char 0x55 received){
                                state = rec_id;
                        }
```

```
            rec_id :
                    if (id received &&
                    parity_ok){
                            state = signal_out;
                    }
        signal_out :
            EXT_DATA_RDY_O = 1;
            if (data_in = 0){
                    state = idle;
            }
    }}
```

4.5.2 Configuration registers

The configuration registers hold the settings (the clock divider), records about the IDs, the message length and the type of checksum. Configuration registers also contain registers (described in the figure 4.4) which determine whether to use the LIN 1.3 value or the tabled value.

The tabled values are stored in the dual port RAM whose structure depends on the chosen database interface width (shown in the figure 4.6).

4.6 Conclusion

The LIN database has been tested on the Cyclone II EP2C35F672C6 and it meets specifications mentioned in this document. The compilation was made with the respect to the smallest area on the chip in the Quartus II version 9.1. The compilation result can be found in the table 4.4 for the 8, 16 and 32-bit interface..

The maximal frequency is over 60 MHz which is more than sufficient to support the maximum LIN bit rate of 19200 bps.

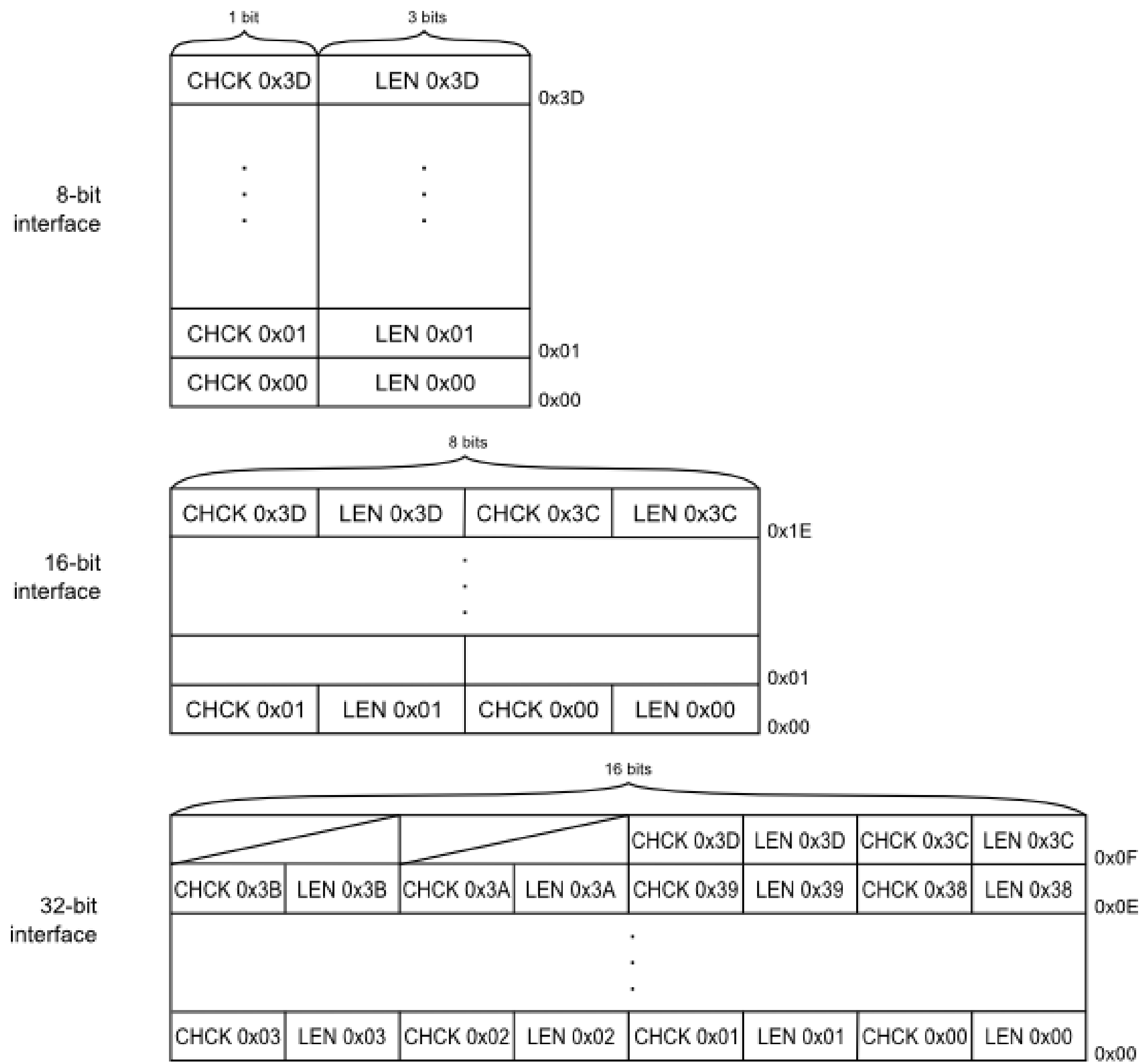

Figure 4.6: Memory map

data width[bit]	total logic elements	total registers	total memory bits	max freq. [MHz]
8	261	159	248	61.84
16	271	172	248	60.7
32	286	175	256	63.79

Table 4.4: Compilation results

Chapter 5

System conception

5.1 Overview

The system must be able to cover all possible configurations that will be required for testing. Therefore, it may contain several trigger units, controllers, and injectors for every bus. The number of these components depends on test requirements and, of course, on the FPGA chip size.

The system may use either a μController that is a part of the system inside the FPGA chip, or an external μController.

An external μController is usually faster than the synthesized one, and it does not take any space inside the chip. However, if the FPGA chip contains enough logic gates, registers, and RAM (and if not, it may be suitable to choose a larger one), the internal μController is a better option. This controller does not require any external pins (except the pins for communication). It is possible to increase or decrease the RAM size, or to add any needed peripherals, and this controller also simplifies the design of the circuit board.

The internal based processor system is therefore a better option for solutions with a few external components. This conception is also suitable for a system that tests several buses as well as for a single-bus system.

The system, developed in this project, is based on the Altera FPGA chip Cyclone II EP2C35F672C6, mounted on the DE2 Development and Education Board. This chip contains over 30 thousand programmable logic elements and registers, almost 500 thousand memory bits, and it can be connected via 475 I/O pins. This chip is therefore more than sufficient for the system that tests devices on a single LIN bus.

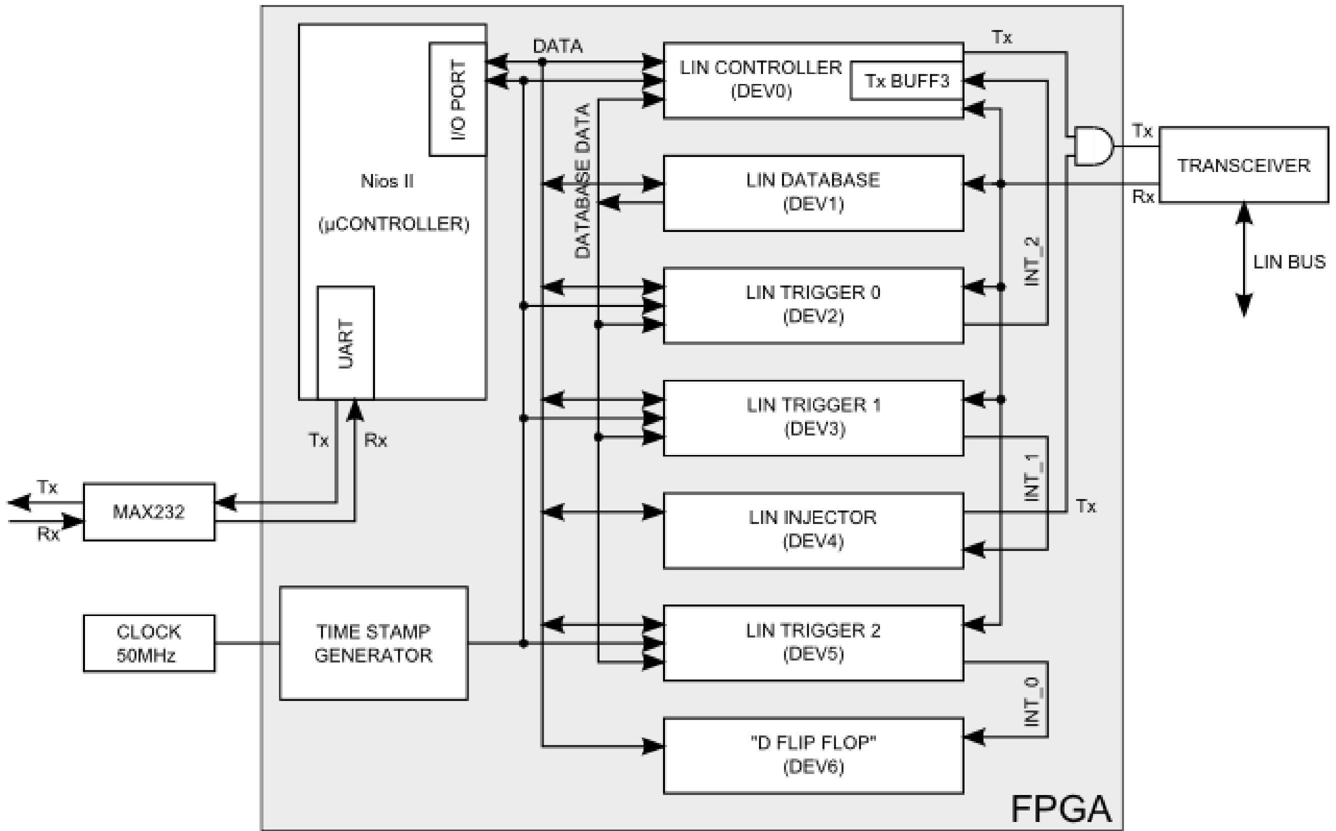

Figure 5.1: Structure of the implemented system

The system (shown in the figure 5.1) consists of a μController, which is synthesized inside the chip, one LIN controller, a database, an injector, a time generator, a D flip-flop component, and three triggers. External peripheries, such as 50 MHz crystal oscillator, MAX232, and physical LIN transceiver TJA 1020, provide clock timing, communication via RS232, and connection to the LIN bus, respectively.

The task of the μController is to communicate with a computer. A computer may request execution of some test. The controller sets all components to execute this test, and when a test is finished, the response (containing the result) is sent to the computer.

5.2 Implemented system

The system (Figure 5.1), implemented for the purpose of this thesis, is capable of undertaking real-time testing of the LIN bus network. It offers RS232 interface as well as

a LIN interface for communication. For the communication with a computer, a simple message format (described in the table 5.3) was created.

This testing system consists of the Nios II, 32-bit μController, one LIN controller (described in the chapter 1), the LIN database (chapter 4), the LIN injector (chapter 3), a time stamp generator, and three LIN triggers (chapter 2). The system uses 32-bit time stamp, LIN controller contains four Tx and eight Rx buffers and the injector is configured to contain a 152 bit buffer.

As shown in the figure 5.1, the μController and all another components are connected to the 8-bit bidirectional data bus[1]. The Nios II μController is a master on the bus, and controls components with 6-bit address bus and another signals specified for the WISHBONE interface.

5.2.1 Block connection

Blocks must be interconnected in order to provide communication channels. I/O ports of the Nios II are used for communication with other components. They are connected to data and address buses and signals SEL(), STB, ACK, and CYC.

As the figure 5.1 shows, these blocks are also connected with another signals in order to provide a real-time response, that does not depend on the communication with the processor. The following list describes these connections:

- Database data bus interconnects the LIN database, the controller and triggers. The LIN database, described in chapter 4, receives an identifier from the bus and sends info about the checksum and frame length to the LIN controller and the triggers.
- Time stamp generator generates 32-bit time stamp, which is broadcasted to the μController, the LIN controller, and the triggers via the 32-bit bus.
- Trigger 0 interrupt output is connected to the LIN controller, Tx buffer 3 (buffers are indexed from 0).
- Trigger 1 interrupt output is connected to the LIN injector interrupt input.
- Trigger 2 interrupt output is connected to the D flip-flop.

[1]Non WISHBONE compatible interface is used for these components; the only difference is the bidirectional bus.

- LIN trigger, database, and controller blocks are connected to the Rx signal, which is connected to the transceiver.
- Tx signals from the LIN controller and injector blocks are connected to one "AND" gate, and then this signal is connected to the transceiver's Tx pin.

5.2.2 μController Nios II

The Nios II is a processor synthesized inside the FPGA. It is fully adjustable, and therefore it is up to the developer which configuration will be used. Altera also provides a compiler that compiles a source code written in C programming language and loads into it to the μController. A tutorial "how to begin with the Nios II system" can be found in [3].

5.2.2.1 Configuration description

Altera provides a SOPC builder tool. This tool allows a designer to specify the processor features for a particular Nios II hardware system. The designer must therefore choose a configuration that is suitable for the implemented system. This tool offers three basic configurations of the controller: Nios II/e, Nios II/s and Nios II/f. These configurations are described in [4] as follows:

- Nios II/f: The Nios II/f "fast" core is designed for fast performance. As a result, this core presents the most configuration options allowing a developer to fine-tune the processor for performance.
- Nios II/s: The Nios II/s "standard" core is designed for small size while maintaining performance.
- Nios II/e: The Nios II/e "economy" core is designed to achieve the smallest possible core size. As a result, this core has a limited feature set, and many settings are not available when the Nios II/e core is selected.

The system implemented in this thesis does not require a high-performance μController, and therefore a Nios II/e variant was chosen. However, the basic setting contains 4 kB of RAM[2], which was not enough (the code for this controller, written in C, is larger than 4 kB), and so it was increased to 16 kB. This RAM is inside the FPGA chip, and no

[2]this processor uses Von Neumann architecture

module name	description	Base addr	End addr
CPU_0	Nios II processor, IRQ	0x2800	0x2FFF
Switches	Parallel input (Switches SW0-SW7)	0x3000	0x300F
LEDs	Parallel output (LEDs LED0-LED7)	0x3010	0x301F
JTAG	JTAG UART	0x3020	0x3017
UART	UART (RS232 serial port)	0x0000	0x001F
Address	Parallel output pin (address)	0x0020	0x002F
Data	Parallel input/output pin (bidirectional data bus)	0x0030	0x003F
RW	Parallel output pin (signal read/write; bit addressable)	0x0040	0x005F
Device select	Parallel output pin (selects device to/from which will be written/read; bit addressable)	0x0060	0x007F
on chip memory	On-chip memory (RAM, 16kB)	0x4000	0x7FFF
time stamp	Parallel input pin (receives a 32-bit time stamp)	0x0080	0x008F

Table 5.1: Nios II system, configuration

external RAM is required. The table 5.1 shows the RAM address allocation as well as another peripheries that were configured inside the processor.

The processor contains JTAG interface in order to be possible to load the program into the memory, run it, and debug it[3].

5.2.2.2 Program inside the μController

The program controls the whole process of testing and communication with a computer, and therefore it is very important to describe it. The program is written in the C programming language.

In simple terms, the program receives commands from a computer, executes tests, and then sends the results to the computer. The controller receives a message in the format described in the section 5.3, and then executes a routine (usually a test) which depends on the meaning of the message.

The code for the main loop is as follows:

```
while(1){
 messageLength = receiveMessage();
```

[3]This is not necessary. It is possible to let the RAM be initialized by the code when the processor is being initialized in the FPGA chip. Nevertheless, JTAG offers easier way how to do it.

```
 if(messageLength <= 0)
  continue;
 switch (Rx_data[0]) { // message type, execute test
  case 0 : {
   for (i = 0; i < messageLength - 1; i++){
    Tx_data[0] = 0x80; //response, message type
    Tx_data[1] = Rx_data[i + 1]; //test number
    Tx_data[2] = executeTest(Rx_data[i + 1]);//rest result
    transmitData(3, Tx_data); // transmits data

                //save data if someone would like to read them later
    testRecorder[testRecorderPointer] = Tx_data[1];
    testRecorder[testRecorderPointer + 1] = Tx_data[2];
    testRecorderPointer += 2;
    if(testRecorderPointer >= TEST_RECORDER)
     testRecorderPointer = 0;
    }
    break;
   }

  case 1 : { // mesage type : read all test results
    Tx_data[0] = 0x81; // response, message type
    for(i = 0; i < TEST_RECORDER && i < TxBufLength; i++)
     Tx_data[i + 1] = testRecorder[i];
    transmitData(TEST_RECORDER + 1, Tx_data);
    break;
    }
  case 2 : { // message type
       // reads data from the LIN controller and transmits them
    receiveAllMessagesAndTransmit();
    break;
  }
 }
}
```

The controller waits until it receives a valid message. The function 'receiveMessage' checks the format of the incoming message, and if it does not fulfill the format described in 5.3, the message is ignored. The program executes a fixed routine according to the message type. There are three different kinds of routines:

- If the message type is '0', then the processor executes tests and returns results for these tests. The processor sends the result of the test immediately after the test is finished. If more tests are requested in one single message, the processor sends the result in the format specified in the section 5.3 for each test separately.

  These tests can either run independently on the processor, or the processor can control this process during the test. Each test consists of three parts. In the first one, the processor configures the LIN controller, injector and triggers. In the second part, the controller, the triggers, and the injector (and even the processor in some tests) control the execution of the test. In the last part, the processor reads the result of the test. The test is usually repeated couple of times, so the processor adjusts the components again, and the test is executed.

- If the message type is '1', then the processor returns last 24 test results in the specified format.

- If the message type is '2', then the processor reads all messages stored in the Rx buffer of the LIN controller, and sends them to the computer in the specified format.

5.2.2.3 Compiling and downloading the software

Altera provides two programs, Nios II EDS and Altera Monitor Program. Nios II EDS is a compiler of the C code. This program takes a respect to the configured Nios II system. That means that this program takes the configuration of the whole Nios II system, and compiles the program for this configuration. As mentioned above, the controller may contain many features, such as hardware multipliers or special DSP blocks, so the compiler takes the advantage of knowing this, and compiles the code exactly for the specific configuration.

There are two ways how to load the software into the RAM (as mentioned above, this processor uses the internal FPGA RAM).

The first one is to let the RAM be initialized during configuration the FPGA chip. Altera SOPC builder provides options for the internal RAM 'Memory initialization'. If

these options are enabled, it is possible to use the *.hex file which is loaded to the the FPGA core when the FPGA is being initializied (this process is described in section 5.2.7).

Another possibility (more convenient for this project) is to configure a JTAG interface inside the processor. This JTAG provides loading and debugging the code. Altera Monitor Program was developed for this purpose. However, before this is possible, the whole system must be configured in the FPGA chip. Obviously, if no system exists inside the FPGA chip, the Altera Monitor Program cannot connect to the JTAG interface, because there is neither a μController, nor a JTAG interface.

So, at first, a system must be configured in the FPGA (which is described in section 5.2.7), and then Altera Monitor Program can connect to the JTAG interface[4]. Altera Monitor Program uses USB-Blaster interface - it is the same interface as the one for the FPGA configuration.

5.2.3 Time stamp generator

The time stamp generator produces 32-bit time stamp with the resolution of 5.2 μs. This resolution was chosen because the bit rate that is used on the LIN bus is 19.2 kbps, and therefore one bit is 5.2 μs long. The time stamp is broadcasted to the μController, the LIN controller and the triggers.

5.2.4 D Flip-Flop

This D flip-flop block is a very simple component. Its interrupt input is sensitive to the logic one. If this level appears on its input, it sets the internal register to a logic one. This register can be read by the μController as a normal device on the bus, and when it is read, the register is set back to a logic zero. The μController is therefore able to catch even very short (one clock cycle) impulses generated by the trigger 2.

5.2.5 System input/output pins

The system must provide some input/output signals in order to be able to communicate with another devices, such as a computer or devices on the LIN bus. As already said,

[4]The switch on the development board must be switched to 'RUN' position, so that the program could access the FPGA board.

PIN name	I/O	Description	Location
RxD_uart_0	Input	Rx signal from the MAX232 circuit	PIN_C25/RS232 Rx
TxD_uart_0	Output	Tx signal to the MAX232 circuit	PIN_B25/RS232 Tx
LIN_RX	Input	Rx signal from the LIN transceiver	PIN_E26/GPIO_0(IO-3)
LIN_TX	Output	Tx signal to the LIN transceiver	PIN_D25/GPIO_0(IO-1)
CLOCK_50	Input	Clock input 50 MHz	PIN_N2
KEY	Input	Reset button	PIN_G26/KEY0 button

Table 5.2: Input / output pins

the system is based on the DE2 Development and Education Board. This board contains many elements, but only a few of them are important for the implemented system. The important pins are described in the table 5.2.

The development board contains MAX232 (an interface for the RS232 communication), a crystal oscillator 50 MHz, a button used for reset, and general I/O pins that are connected to the LIN transceiver.

5.2.6 LIN transceiver TJA 1020

The TJA1020 is the interface between the LIN master/slave protocol controller and the physical bus in a Local Interconnect Network (LIN). It is primarily intended for in-vehicle sub-networks using baud rates from 2.4 up to 20 Kbaud [5].

This chip has the following properties, that are important for the testing circuit:

- Input levels are compatible with 3.3 V devices (the general I/O pins at the FPGA board use 3.3 V input/output levels).
- Very low electromagnetic emission (an important property for a testing device, because another devices on the bus are not affected).
- High electromagnetic immunity (also an important property, because it is possible to communicate even during electromagnetic tests).
- Low slope mode for an even further reduction of electromagnetic emission

The chip also contains the following protections:

- Thermal protection

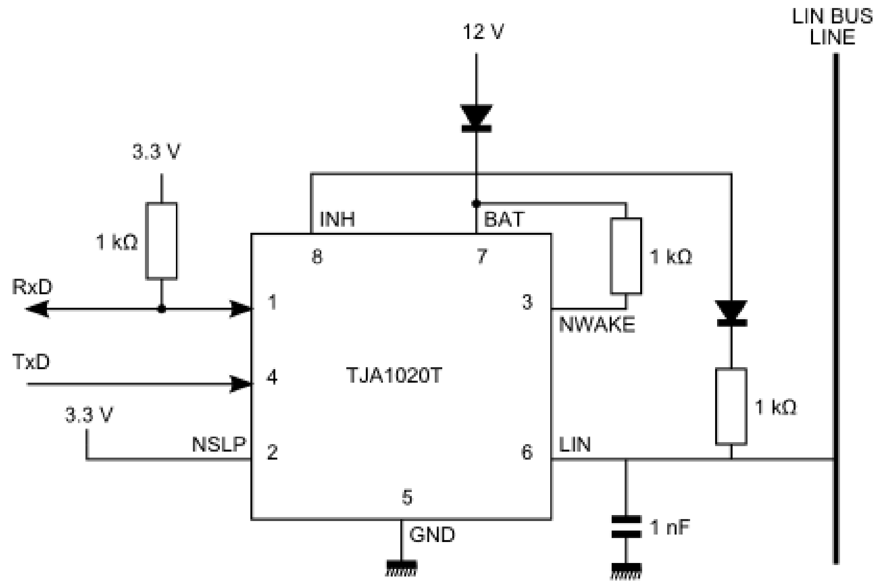

Figure 5.2: Transceiver connection

- Bus terminal and battery pin protected against transients in the automotive environment
- Transmit data (TXD) dominant time-out function

The last protection is little inconvenient. This protection prevents the bus line from being driven to a permanent dominant state if the pin TXD is permanently low state. This chip includes a timer, that is triggered by a negative edge of the pin TXD, and reset by a positive edge of TXD. If the timer exceeds the internal timer value (min 6 ms, max 20 ms), the transmitter is disabled, and the output is returned to the recessive state.

This property might be inconvenient for testing in case a break longer than 6 ms would be required. However, this time interval is more than sufficient fot the bit rate 19.2 kbps.

The connection of the TJA 1020 chip is shown in the figure 5.2. It is a recommended connection for a device which is a master on the LIN bus. Two general FPGA I/O pins (LIN_RX and LIN_TX) are connected to the RXD and TXD signal, and therefore the system inside the FPGA board has an access to the LIN bus.

5.2.7 Configuring the FPGA

FPGAs do not include any memory which would store the configuration when the chip is not powered. Therefore, it is necessary to configure the chip every time after the start,

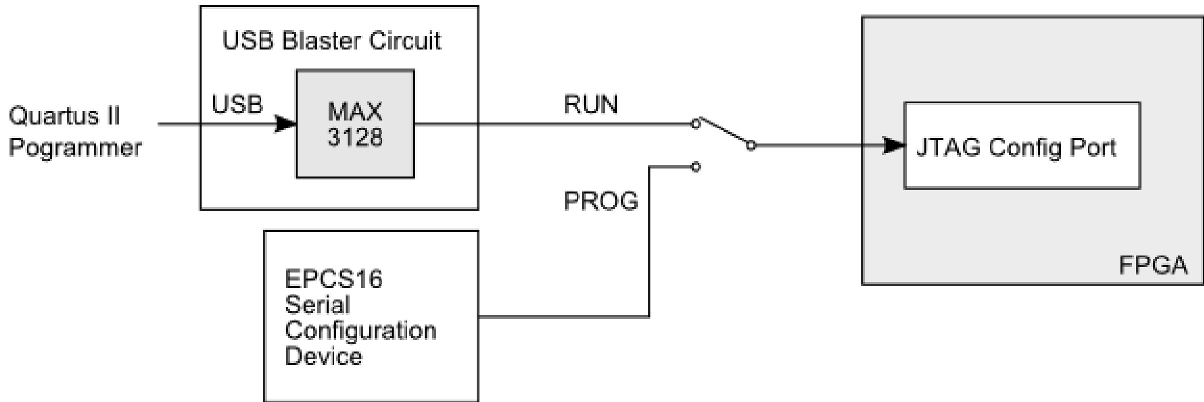

Figure 5.3: The JTAG configuration scheme

which is a task for the configuration device. This device reads the data from some flash memory, and configures the FPGA chip.

The DE2 Development and Education Board offers two ways of how to configure the chip. The first one (shown in the figure 5.3) uses a USB-blaster interface. A computer connects to the board via USB. The switch on the development board is in 'RUN' position. The Quartus II programmer can configure the FPGA chip via this interface.

When using the second way, the switch is in 'PROG position'. In this case, the configuration is not loaded to the FPGA, but to the serial configuration device. After the board is reset, the configuration device configures the FPGA chip.

More information about the configuration of the FPGA for this development board (configuration step by step) can be found in [6].

5.3 Communication protocol

The Nios II μController contains an UART interface, and it is connected to the Rx and Tx pins on the MAX232 chip that provides physical interface to the RS232 bus. This controller can therefore communicate with a computer (fixed baud rate 115.2 kbps, no parity). For this purpose, a special protocol was created. It would be possible to use any other protocol, however, this protocol is uncomplicated and fully sufficient for the simple communication with the computer.

The table 5.3 explains the meaning of each byte. The protocol starts with chars 0x55 and 0xAA. The next byte, 'data length', is the length of the data field in bytes. This

	Data	$Length_{min}$	$Length_{max}$
Header	0x55	1	1
	0xAA	1	1
	Data length	1	1
Data	Protocol type	1	1
	Payload	0	49
Checksum	x	1	1

Table 5.3: Communication protocol

is followed by the 'protocol type' byte. The payload can be 0 up to 49 bytes long. The length of this field is limited, because the μController contains 50-byte buffer for the incoming message.

The checksum is calculated using the following formula:

```
checksum = Protocol type
for (i = 0; i < Payload length; i++)     checksum ^= Payload(i);
```

For the communication a programm Oscillo (attached on the CD) was used. This program, described in [7], offers sending and receiving bytes in the HEX format (the program has many another features, but only these two are required for the communication).

5.3.1 Protocol types

The protocol type byte determines the meaning of the payload. There are two different groups of messages: requests and responds.

- Request is always sent by the computer. This byte has values between 0x0 and 0x7F.
- Respond is always sent by the system, which is inside the FPGA chip, and it has values between 0x80 and 0xFF.

The computer sends the request to the syste, which replies with the 'protocol type' byte:

$$respond = request|0x80 \tag{5.1}$$

The table 5.4 shows all kinds of messages.

Protocol type	description
0	execute test/tests
1	read last executed tests
2	read Rx buffer from the LIN controller

Table 5.4: Messages type

Protocol 0 On this request, the controller executes a set of tests. The 'protocol type' byte is followed by bytes that determine which tests are to be executed. The number of tests is not limited, however, the size of the Rx buffer is limited. So it is possible to execute only 49 tests in one go. The processor sets the test, and when the test is finished, it returns the result. The format of the respond is as follows:

```
0x80(response)|test number|test result
```

Protocol 1 When the processor receives this request, it returns results of last 24 tests in the following format:

```
0x81(response)|test number0|test result0|test number1|
test result1.. test number23|test result23
```

Protocol 2 When the processor receives this request, it reads all messages in the Rx buffers in the LIN controller and sends the following message:

```
0x82|0xFF|number of messages|x|x|x|x|x|x|x|x
```

If the number of messages in the Rx buffer is greater than zero, these messages are sent to the computer in the following format (each message separately):

```
0x82|ID|byte0|byte1 |...|byte7
```

5.4 Conclusion

This chapter describes the implemented system, which is used for testing of automotive buses. The system consists of one Nios II μController, one LIN controller, a database, an injector, one time stamp generator, one 'D flip-flop' unit, and three LIN triggers. It also describes how these components are connected together.

total logic elements	total registers	total memory bits	max freq. [MHz]
6155(19%)	3390(10%)	144.984(30%)	53.56

Table 5.5: Compilation results

This structure is sufficient for all tests used for one LIN bus. A larger system could contain a programmable multiplexer, which would connect triggers outputs to Tx buffers of the LIN controller, the LIN injector, or to the 'D flip-flop' unit. This solution would have an advantage - it would be possible to program this multiplexer (and therefore the connection between blocks). Thus, it would not be necessary to have three triggers. Only one would suffice (except for the tests where two or more triggers are required). Nevertheless, the solution described in this chapter is appropriate, because the Cyclone II EP2C35 chip contains enough logic elements even for a system with three triggers (as the table 5.5 shows, the system needs less than 20% of logic elements of the chip).

The code for the Nios II μController is written in the C programming language, and this chapter describes the basic structure of the code as well as of the protocol, which is used for the communication with a computer.

This system was synthesized for the Cyclone II EP2C35F672C6, and the table 5.5 shows how many logic elements and memory is used (the percentage tells how many elements of the whole FPGA is used). The development board contains a 50 MHz crystal oscillator, so it is important that the system can run at higher frequency than 50 MHz.

The system, which consists of more than 7 thousand lines of code[5] written in VHDL, and over one thousand lines of code written in C, meets all requirements for real-time testing of devices connected to the LIN bus. The LIN controller, trigger, and injector are the basic components that are required for testing, and therefore it is possible to use them in a system which tests M LIN, N CAN and K FlexRay buses in general.

[5]Including white space and comments. In fact, with the Nios II system included, it would be more than 12 thousand lines. However, the Nios II code was automatically generated.

Bibliography

[1] XILINX, *Implementing a LIN Controller on a CoolRunner-II CPLD, XAPP432 (v1.1)*, April 2007. http://www.xilinx.com/support/documentation/application_notes/xapp432.pdf.

[2] G. Auerbach and D. Fisman, *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B.3*. OpenCores, September 2002. http://cdn.opencores.org/downloads/wbspec_b3.pdf.

[3] Altera Corporation, *Introduction to the Altera SOPC Builder Using VHDL Design*, 2008.

[4] Altera Corporation, *Nios II Processor Reference Handbook*, July 2007.

[5] Philips, *TJA1020 LIN Transceiver datasheet*, January 2004.

[6] Altera Corporation, *DE2 Development and Education Board User Manual, v1.2*, 2005.

[7] M. Patak, “Data acquisition system based on the controller cy7c68xxx,” 2009. Bachelor thesis, Czech Technical University in Prague.