



Title: LIN_Protocol_ver 2.2A
SW Component LIN_Protocol

Maturity:	Draft
Authors:	David Antonio Diaz Ramirez
	Francisco Javier Quirarte Pelayo
VERSION:	Public
Security Classification:	None



History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
1.0	Draft 26-nov-15	Francisco Quirarte	Francisco Quirarte y David Díaz	Creation of the document.
History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
2.0	Draft 27-nov-15	David Díaz	Francisco Quirarte	Added: Purpose, Definitions and abbreviations.
History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
3.0	Draft 28-Nov-15	Francisco Quirarte y David Díaz	Francisco Quirarte	Added: Realization constraints and targets.
History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
4.0	Draft 29- Nov -15	Francisco Quirarte y David Díaz	Oswaldo Garcia	Added: SW Conceptual design.

History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
6.0	Draft 30-nov-15	Francisco Quirarte y David Díaz	Luis Alvarez	Review of SW Component internal breakdown and added general corrections. General corrections at traceability matrix

History				
Issue status (Index)	Maturity/Date (draft/invalid/valid) (dd-mmm-yyyy)	Author Department	Check/Release Department	Description
7.0	Release 30-Nov-15	Francisco Quirarte y David Díaz	Oswaldo Garcia e Isamar Gálvez	Review of SW Conceptual design and added general corrections. Review of the test plan.



Table of Contents

1	INTRODUCTION	5
2	PURPOSE	6
3	COMPARISON CHART PROTOCOLS	7
4	LIN PROTOCOL CONCEPTS	7
5	CHART CONCEPTS.....	9
6	DEFINITIONS AND ABBREVIATIONS	11
7	LIN SOFTWARE DRIVER SERVICES	13
8	THE ISO/OSI REFERENCE MODEL AND LIN	14
8.1	Figure 10: reference model ISO/OSI LIN	14
9	SOFTWARE LAYERS	15
9.1	Figure 11: Layers	15
9.2	Layer descriptions.....	15
9.3	Functional Blocks Descriptions	16
10	LIN NETWORK	17
11	REALIZATION CONSTRAINTS AND TARGETS	19
11.1	Network Graphical Description	21
11.1.1	Figure 11: Deployment Diagram (A)	21
	Deployment Diagram. Represents the physical configurations of software and hardware items.....	21
11.2	Use Case Diagram	22
11.2.1	Figure 12: Use Case Diagram.....	22



11.3	Component diagram request & response (A/D)	23
11.3.1	Figure 3: Component diagram (A/D).....	23
11.3.2	Figure: Components Descriptions	23
11.3.3	Component diagram SW.....	24
11.4	Activity Diagram (R)	25
11.5	SEQUENCE.....	28
11.6	Response Sequence Diagram	29
11.7	State machine diagram	30
11.7.1	Slave 2.....	30
11.7.2	Lin manager.....	31
11.7.3	LIN Handler.....	33
11.7.4	Master node transmission handler	34
11.7.5	LED Handler.....	36
12	APPLICATION PROGRAM INTERFACE SPECIFICATION VER. 2.2A.....	37
12.1	CORE API	37



1 Introduction

This application note shows how to implement a LIN (Local Interconnect Network) slave task in a 32-bit RISC microcontroller without the need for any external components. The LIN protocol is a serial communications protocol which efficiently supports control of mechatronic nodes in a distributed network. This makes the protocol ideal for use in automotive applications. A LIN network consist of a single Master and a set of Slave nodes. This application note show how to implement the protocol for the physical layer and the data link layer according to the two lowest levels of the ISO/OSI reference model. This provides the foundation for message transmissions between nodes in the network. The physical layer of the ISO/OSI model only provides a raw bit-stream between two nodes in the network. The data link layer makes the physical layer reliable by adding error detection and control. It also adds the means to activate, maintain, and deactivate the link. The higher levels of the ISO/OSI reference model is beyond the scope of the LIN protocol and thus not explained here. For now, this is up to the user to specify and implement. The LIN protocol differs from the CAN protocol in the sense that it is below the performance and scope of the CAN. Its strongpoint is that it provides a simple, low-cost solution for applications not requiring the power of the CAN protocol.



2 Purpose

Bus Local Interconnect Network, LIN usually abbreviated Bus is a system used in current data transmission networks Automotive System. SPECIFICATION meets the LIN consortium, in its first version 1.1, released in 1999. The specification has been developed from then to version 2.0 para solve the current needs of a red. The LIN bus is a little slow system that USING As a cheap bus para- red sub UN can integrate smart devices or actuators in today's cars.

Is a concept for low cost automotive networks, which complements the existing portfolio of automotive multiplex networks. LIN will be the enabling factor for the implementation of a hierarchical vehicle network in order to again further quality enhancement and cost reduction of vehicles. The standardization will reduce the manifold of existing low-end multiplex solutions and will cut the cost of development, production, service and logistics in vehicle electronics.



3 Comparison chart protocols

	SPI	I2C	RS232	RS485[1]	LIN	CAN
TIPO DE TRANSMISIÓN	Sincrono[2]*	Sincrono	Síncrona, asíncrona	Síncrona, asíncrona[3]	Asíncrono*	Asíncrono[4]
SOPORTE MULTIMAESTRO	NO	SI	NO	SI	NO	SI[5]
PINES UTILIZADOS	SLCK, MOSI, MISO, SS	SDA, SCL	TxD, RxD, S G	TxD, RxD, SG	LIN BUS	CANH, CANL,
COMUNICACIÓN	Half dúplex, Full dúplex *	Halfdúplex, full dúplex*	simplex, half duplex o full dúplex[6]	HalfDuplex	Simplex*	Halfduplex
VELOCIDAD Y DISTANCIA DE TRANSMISION	320kbps-5Mbps[7], 2 m	100kbps - 400kbps, y alta velocidad 3.4 Mbps	20 Kbps, hasta 15m[8]	10Mbps a 10 m, y 100kbps a 1200m[9]	20 kbps [10], 40 m y aumenta con un repetidor[11]	125 Kbps a 500m -1 Mbps a 40 m[12]
CONSUMO DE ENERGIA	No utiliza Pull-ups y ahorra energía, fuente +5V	Utiliza Pull-Ups, fuente +5V	+12 y -12 Volts	fuelle +5V , 1.5v entre las salidas diferenciales.	Utiliza Pull-Ups	cable L (Low) = 0V y 2.25V, cable H= 2.75V. y 5V
DIRECCIONAMIENTO	utiliza líneas específicas para cada dispositivo	Cada dispositivo tiene una dirección única por software, 128	Es punto a punto		basado en mensajes, no en direcciones	basado en mensajes, no en direcciones
		direcciones				
CANTIDAD DE DISPOSITIVOS SOBRE EL MISMO BUS	Depende de los pines disponibles del PIC para SS	Depende de la capacitancia de las líneas 400 pf	Uno a uno	32 (Bus de 2 hilos), 64 (4 hilos bus)	16 unidades esclavas[13]	127
NOTAS			Trabaja con lógica negativa			Necesita de un transeiver x cada dispositivo
COMPAÑIA	Motorola	Philips Semiconductor	Electronic Industries Alliance	Electronic Industries Alliance		Robert Bosch GmbH

4 LIN Protocol Concepts

Single Master Multiple Slaves

The LIN protocol does not use bus arbitration. A single Master is responsible for initializing all message transfers. All slaves can respond to the Master or any other node in the network but only after being addressed and given permission by the Master.



Variable Length of Data Frames

Included in the identifier field are two bits indicating the length of the message field (see Table 1). This adds flexibility and reduces overhead bytes when only a limited amount of data is required.

Multi-cast Reception

When a message frame is transmitted from the Master or a Slave task, all connected nodes in the network can read the message. Depending on the identifier byte, the receiving nodes decides if any action is to be initiated or not. For example, a single "CLOSE ALL" command from the Master could be accepted from all nodes and could in the case of a car security system close all windows and doors.

Time Synchronization without Need for Quartz or Ceramic Resonators in the Slave Nodes

After the Synch Break, a Synch Field is transmitted from the Master, this field makes it possible for all the Slaves to synchronize to the master clock. Such synch fields are placed at the beginning of every message frame. The accuracy of the receiving Slaves need only be good enough to keep synchronization through the entire message frame. This feature allows the slave to run on an internal RC Oscillator thus reducing the overall system cost.

Data Checksum Security and Error Detection

The data in the message frame uses an inverted modulo256-checksum with the carry of the MSB added to the LSB for error detection. In addition, the identifier byte uses a XOR algorithm for error detection.

Detection of Defect Nodes in the Network

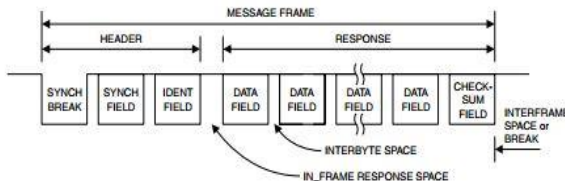

The Master task is responsible for initiating the transmission of message frames and thus has the responsibility of requesting information and checking that all nodes are alive and working correctly.

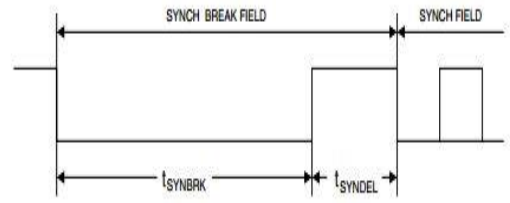
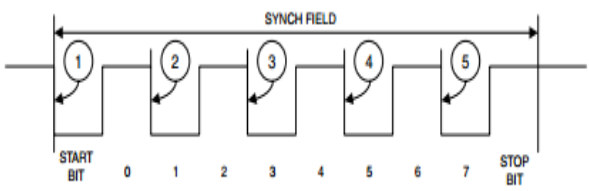
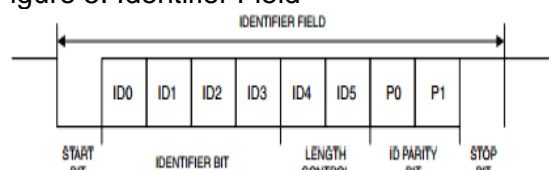
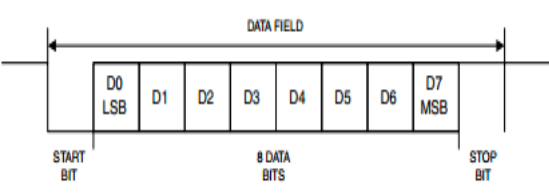
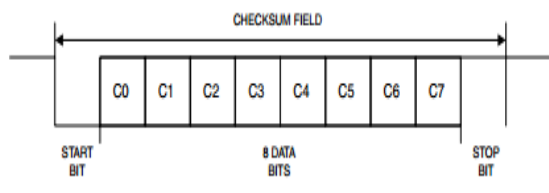
Minimum Cost Solution

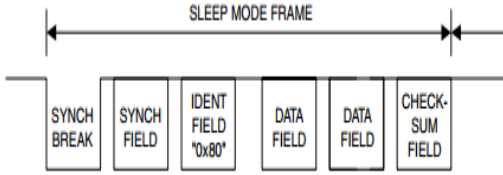
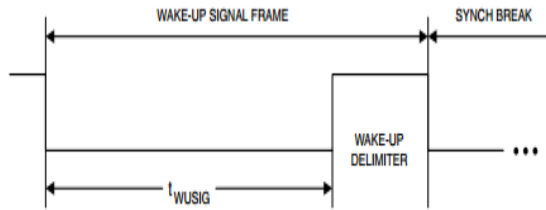
Due to the simplicity of the protocol, a slave task complying to the LIN standard can be built using a minimum of external components and does not put heavy constraints on the accuracy of the Oscillator in the Slave nodes.

5 Chart Concepts

In the following chart will describe the concepts LIN protocol:

<p>Signal Transmission</p>	<p>The LIN bus consists of a single channel that carries both data and synchronization information. The physical medium is a single wire connected to VCC via a pull-up resistor (see Figure 1). The idle state on the bus is high or “recessive” and the active state is low or “dominant”. In automotive applications VCC will typically be the positive battery node. The LIN protocol does not define an acknowledgment procedure for the Slave tasks. The Master task uses its own slave task to verify that the sent message frame is identical with the one received by this slave task. If any discrepancy is detected the message frame can be retransmitted. The data rate for transmitted data is limited to 20 Kbits/s due to EMI (Electro Magnetic Interference) requirements for a single wire transmission medium.</p>
<p>Message Frames Figure 1. LIN Message Frame</p> 	<p>All information transmitted on the LIN bus is formatted as Message Frames. As shown in Figure 2, a Message Frame consists of the following fields:</p> <ul style="list-style-type: none"> • Synch Break • Synch Field • Identifier Field • Data Field • Checksum Field
<p>Byte Fields Figure 2. LIN Byte Field</p> 	<p>The Byte Field format, shown in Figure 3, is identical to the commonly used UART serial data format with 8N1 coding. This means that each Byte Field contains eight data bits, no parity bits, and one stop-bit. Every Byte Field has the length of 10 bit-times (Tbit). As shown in the figure, the start bit marks the beginning of the Byte Field and is “dominant” while the stop-bit is “recessive”. The eight data bits can be either “dominant” or “recessive”.</p>
<p>Synch Break Figure 3. Synch Break Field</p>	<p>The Synchronization Break marks the beginning of a Message Frame. This field is always sent by the Master task, and provide a means for the Slave task to prepare for the synchronization field. The Synch Break consists</p>

	<p>of two different parts: a dominant (low) level that should be minimum 13 bit-times (Tbit) which is followed by a recessive (high) period that should be in the range one to four Tbit. The second field is necessary to detect the dominant (low) level of the start bit of the following Synch Field.</p>
<p>Synch Field Figure 4. Synch Field</p> 	<p>The Synch Field contains the signaling required for the slave to synchronize with the master clock. The Synch Field is a Byte Field containing the data "0x55" giving the waveform shown in Figure 4.</p>
<p>Identifier Field Figure 5. Identifier Field</p> 	<p>The Identifier Field contains information about contents and length of a message. As shown in Figure 6, this field is divided into three sections: identifier bits (four bits), length control bits (two bits) and parity bits (two bits). This divides the set of 64 identifiers into four subsets of 16 identifiers each.</p>
<p>Data Field Figure 6. Data Field</p> 	<p>The data frame contains from two to eight Data Fields containing eight bits of data each. Transmission is done with LSB first. The data fields are written by the responding Slave task. Since there is no bus arbitration, only one slave task should be allowed to respond to each identifier. All other Slave tasks are limited to reading the response and act accordingly.</p>
<p>Checksum Field Figure 7. Checksum Field</p> 	<p>The last field in the Message Frame is the Checksum Field. This byte contains the inverted modulo-256 sum of all data bytes (data frame not including the identifier). This sum is calculated by doing an "ADD with carry" on all data bytes and then inverting the answer. The properties of the inverted modulo-256 sum are such that if this number is added to the sum of all data bytes the result will be "0xFF".</p>
<p>Sleep Mode Frame Figure 8. Sleep Mode Frame</p>	<p>The frame structure for the Sleep Mode Frame is identical to an ordinary Message Frame and is distinguished by the identifier byte "0x80". The contents of the data fields are not specified and could be used to distribute system parameters for the sleep mode.</p>

	
<p>Wake-up Signal Figure 9. Wake-up Signal Frame</p> 	<p>The sleep mode can be terminated by any connected slave task by sending a wake-up signal. The wake-up signal is only allowed when the bus is in sleep mode and a node internal request for wake-up is pending. The wake-up signal is the character "0x80".</p>

6 Definitions and abbreviations

Definitions

LIN 2.2A	Serial network protocol used for communication between components in vehicles
Master (node)	Node chosen to control all communications. Bus arbiter.
Slave (node)	Node that responds to the master in three possible ways: Receive data, transmit data or ignore data.

Abbreviations

LIN	Local Interconnect Network
RX	Reception
TX	Transmission
MSTR	Master
SLV	Slave
MSG	Message
HAL	Hardware Abstraction Layer
LIN	Local Interconnect Network
CAN	Controller Area Network



ECU	Electronic Control Unit
API	Application User Interface
IDE	Integrated Development Enviroment
UART	Universal Asynchronous Receiver Transmitter
ASC	Async/Synchronous Serial Interface
ID	Identifier
EMI	Electro Magnetic Isolation
t_slv_stat	Signal to display slave node status.
t_cmdType	Signal to control LED states and slave node state.
t_LEDstat	Signal to display LED status.
t_boolean	Signal response that indicates slave node status.
array	Signal that contains the team member's initials for the Msg ID.
scalar	Signal that contains the team assigned number for the Msg ID

References

N°	Document name
1	<i>Traceability Matrix Template.xls</i>
2	MPC5604B/C Microcontroller Reference Manual.pdf
3	Test_template.xls
4	http://www.freescale.com/products/power-architecture-processors/mpc5xxx-5xxx-32-bit-mcus/mpc56xx-mcus/mpc5606b-startertrak-development-kit:TRK-MPC5606B
5	http://www.freescale.com/products/power-architecture-processors/mpc5xxx-5xxx-32-bit-mcus/mpc56xx-mcus/ultra-reliable-mpc56xb-mcu-for-automotive-industrial-general-purpose:MPC560xB#pspFeatures
6	SW_C_Code_Review_Template.docx
7	Window-lifter-requirements.docx
8	LIN_Protocol_Conti.ppt
9	LIN_Network_Database
10	LIN-Spec_2-2A.doc
11	LIN-Spec_2-2A.pdf
12	http://www.atmel.com/images/doc1637.pdf



7 LIN Software Driver Services

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

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

LIN Software Driver Requirements

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

Hardware Requirements

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

Memory Requirements

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

Slave task:

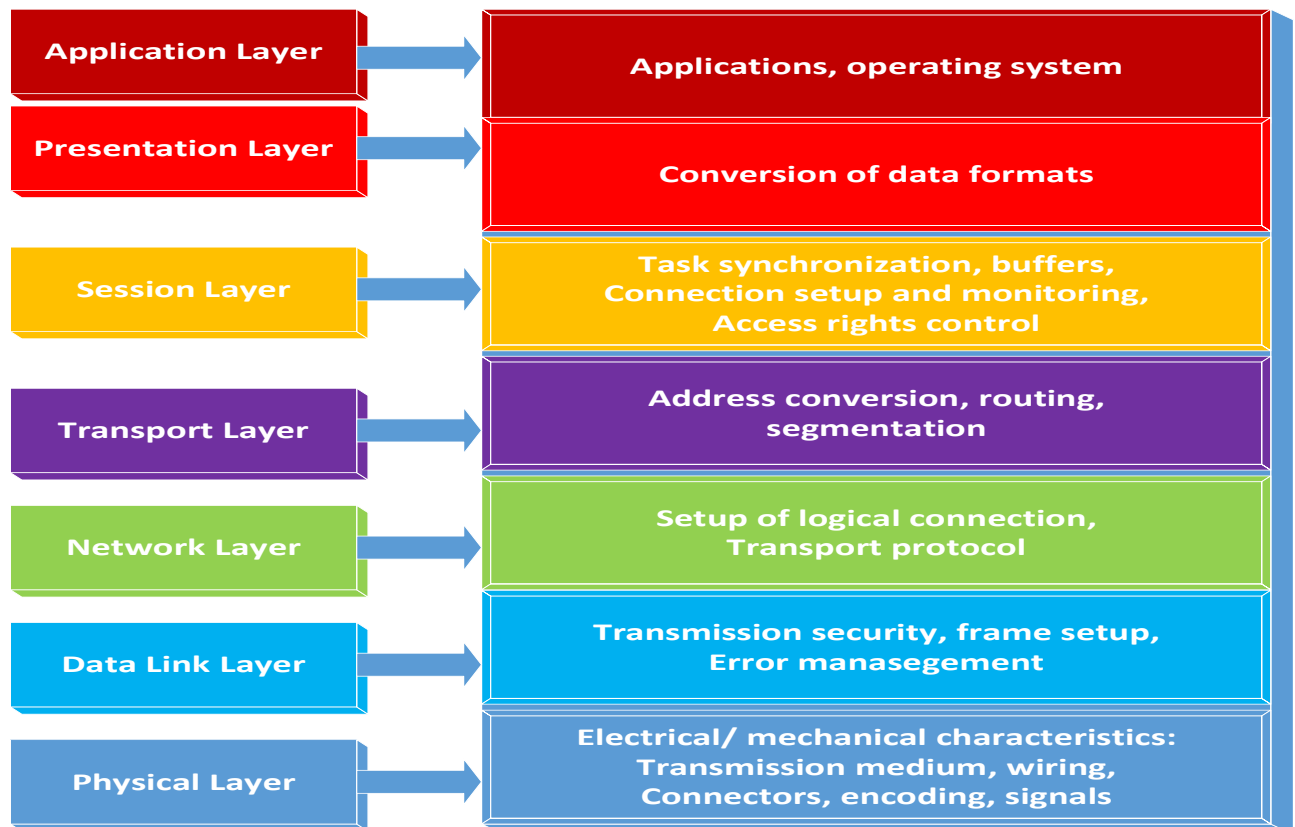
1. Receive-ID-Field (message filtering), copy data to LIN transfer buffer and send first databyte
2. Receive first databyte and send second data byte
3. Receive second databyte and send third data byte
4. Receive third databyte and send fourth data byte
5. Receive fourth databyte and calculate / send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

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



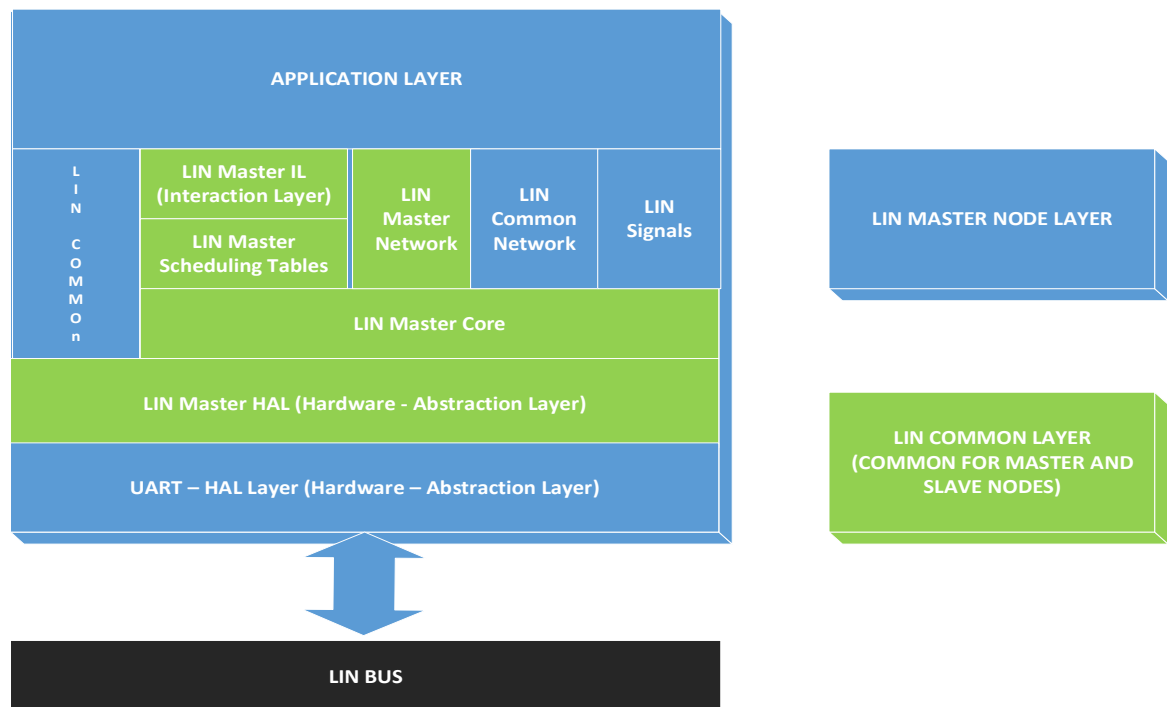
8 The ISO/OSI reference model and LIN

8.1 Figure 10: reference model ISO/OSI LIN



9 Software Layers

9.1 Figure 11: Layers



9.2 Layer descriptions

Layer	Description
Application Layer	This layer correspond to the application functionally which uses the LIN data being received by LIN and writes the data being transmitted by LIN.
LIN Master Node Layer	This layer implements the functionally of the LIN Master node
LIN common layer	This layer provides implementation of functionally that is common either for a LIN master node or a LIN slave node.
UART Layer	This layer implements all UART related interfaces to be used by the LIN functionally



9.3 Functional Blocks Descriptions

Functional Block Name	Description	Related source code files
LIN Master IL	Provides the “skeleton” of the callbacks being called by the Lin master node. Some of those callbacks are associated to messages defined in the LIN network so this file changes defined in the LIN network	LIN_mstr_hal.c LIN_mstr_hal.h
LIN Master Network	Defines and implements all functionality related to the LIN messages of the network where the master node is connected. This file changes each time any of the the LIN messages changes in the LIN network definition.	LIN_mstr_network.c LIN_mstr_network.h
LIN Master Scheduling Tables	Defines and implements all functionality related to the LIN master scheduling tables. This file changes each time a scheduling table or messafe changes in the LIN network	LIN_mstr_sched.c LIN_mstr_sched.h
LIN Master Core	Implements the core functionality of the LIN master node, i.e., the functionality for sending and receiving LIN frames. This functionality does not depend on the LIN network definition so it does not change upon changes of it	LIN_mstr_core.c LIN_mstr_core.h
LIN Master HAL	Implements the interfaces of the LIN master core functionality with the UART hardware.	LIN_mstr_hal.c LIN_mstr_hal.h
LIN Common	Provides functionality that is common among any node of the LIN network, i.e., either the master node or any slave node. For example, the checksum calculation is implemented in this functional block.	LIN_common.c LIN_common.h
LIN Common Network	Provides definitions of the LIN network which is common to	LIN_common_network.h

	all LIN nodes (master and slaves). For example, the ID and data length of the messages, etc.	
LIN Signals	Provides data structures to ease the access of the individual signals of the LIN frames.	LIN_signals.h
UART HAL	Implements the API's for sending and receiving bytes with the UART hardware.	UART_hal.c UART_hal.h

10 LIN NETWORK

Messages defined for this network.

Msg ID	Msg Name	Msg Data Length (byte)	Msg Publisher	Msg Subscribers	Msg Callback
CF	MASTER_CMD_ALL	1	MASTER	SLAVE1, SLAVE2, SLAVE3, SLAVE4	Auto
50	MASTER_CMD_SLV1	1	MASTER	SLAVE1	Auto
11	MASTER_CMD_SLV2	1	MASTER	SLAVE2	Auto
92	MASTER_CMD_SLV3	1	MASTER	SLAVE3	Auto
D3	MASTER_CMD_SLV4	1	MASTER	SLAVE4	Auto
20	SLAVE1_RSP	2	SLAVE1	MASTER	Auto
61	SLAVE2_RSP	2	SLAVE2	MASTER	Auto
E2	SLAVE3_RSP	2	SLAVE3	MASTER	Auto
A3	SLAVE4_RSP	2	SLAVE4	MASTER	Auto
F0	SLAVE1_ID	7	SLAVE1	MASTER	Auto
B1	SLAVE2_ID	7	SLAVE2	MASTER	Auto
32	SLAVE3_ID	7	SLAVE3	MASTER	Auto
73	SLAVE4_ID	7	SLAVE4	MASTER	Auto

Messages used for and by slave 2

Signal Types defined for this network.

Type Name	Elements...					
	0	1	2	3	4	5
t_slv_stat	Disabled	ENABLED				
t_cmdType	cmd_NONE	cmd_LED_on	cmd_LED_off	cmd_LED_toggling	cmd_disable_slv	cmd_enable_slv
t_LEDstat	OFF	ON	TOGGLING			
t_target_active	INACTIVE	ACTIVE				
t_boolean	FALSE	TRUE				

Signals defined for this network.

Signal Name	Signal Msg	Signal Start Byte	Signal Start Bit	Signal Len (bits)	Signal Type	Signal Endianness
master_cmdForAll	MASTER_CMD_ALL	0	0	4	t_cmdType	LE
master_cmdForSlave1	MASTER_CMD_SLV1	0	0	4	t_cmdType	LE
master_cmdForSlave2	MASTER_CMD_SLV2	0	0	4	t_cmdType	LE
master_cmdForSlave3	MASTER_CMD_SLV3	0	0	4	t_cmdType	LE
master_cmdForSlave4	MASTER_CMD_SLV4	0	0	4	t_cmdType	LE
slave1_LEDstat	SLAVE1_RSP	0	0	2	t_LEDstat	LE
slave1_enabled	SLAVE1_RSP	1	0	1	t_boolean	LE
slave1_supplier	SLAVE1_ID	0	0	8	scalar	LE
slave1_serial	SLAVE1_ID	1	0	48	array	LE
slave2_LEDstat	SLAVE2_RSP	0	0	2	t_LEDstat	LE
slave2_enabled	SLAVE2_RSP	1	0	1	t_boolean	LE
slave2_supplier	SLAVE2_ID	0	0	8	scalar	LE
slave2_serial	SLAVE2_ID	1	0	48	array	LE
slave3_LEDstat	SLAVE3_RSP	0	0	2	t_LEDstat	LE
slave3_enabled	SLAVE3_RSP	1	0	1	t_boolean	LE
slave3_supplier	SLAVE3_ID	0	0	8	scalar	LE
slave3_serial	SLAVE3_ID	1	0	48	array	LE
slave4_LEDstat	SLAVE4_RSP	0	0	2	t_LEDstat	LE
slave4_enabled	SLAVE4_RSP	1	0	1	t_boolean	LE
slave4_supplier	SLAVE4_ID	0	0	8	scalar	LE
slave4_serial	SLAVE4_ID	1	0	48	array	LE



11 Realization constraints and targets

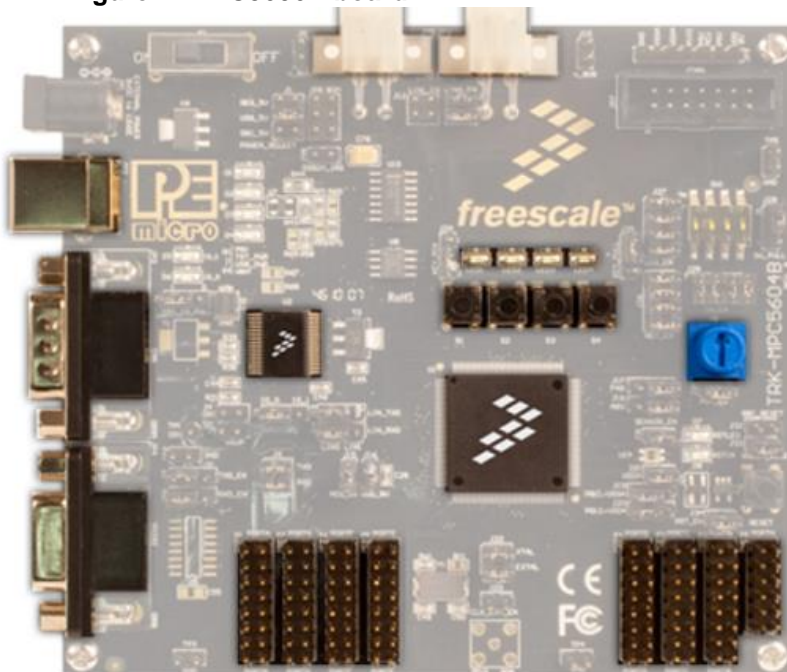
System must ensure reliability on window behavior and security with the anti-pinch function, push buttons functions shall be much delimited in any unusual situation from ambiguous scenarios. The constraints to accomplish this targets are the use of microcontroller freescale TRK-MPC5606B, emulation just for 10 led bar and anti-pinch function managed only by push button with scheduler control.

The system will be running in the MPC 5606B. With the following specifications:

- MPC5606B MCU in a 144LQFP package.
- On-board JTAG connection via open source OSBDM circuit using the MPC9S08JM MCU
- MCZ3390S5EK system basis chip with advanced power management and integrated CAN transceiver.
- CAN and LIN interface.
- Analog interface with potentiometer.
- High-efficiency LEDs.
- SCI serial communication interface.

TRK-MPC5606B: MPC5606B StarterTRAK (Development Kit)

Figure 1: MPC5606B board.





Operating Frequency (Max): 64 MHz.

Total DMA Channels 16.

Internal Flash (KB): 512

GPIOs: 149.

EEPROM: 64 KB DataFlash®

RAM: Up to 96 KB

Timer: 16 bits up to 64 channels

ADC:

10 bits up to 36 channels

12 bits up to 16 channels

Up to six CAN

Up to six SPI

Up to 10 LINFlex

MPC560xB/C Block Diagram

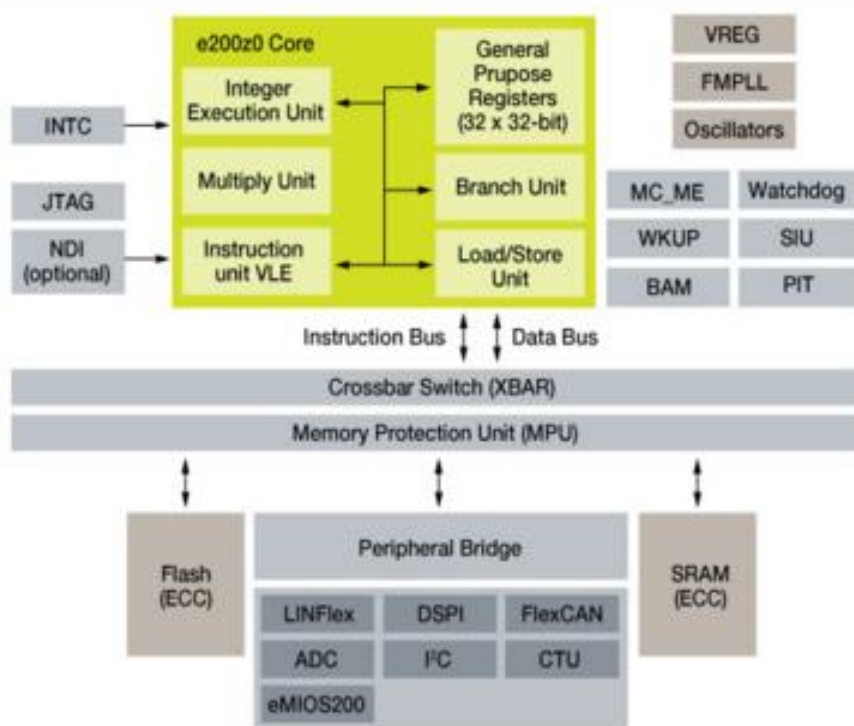
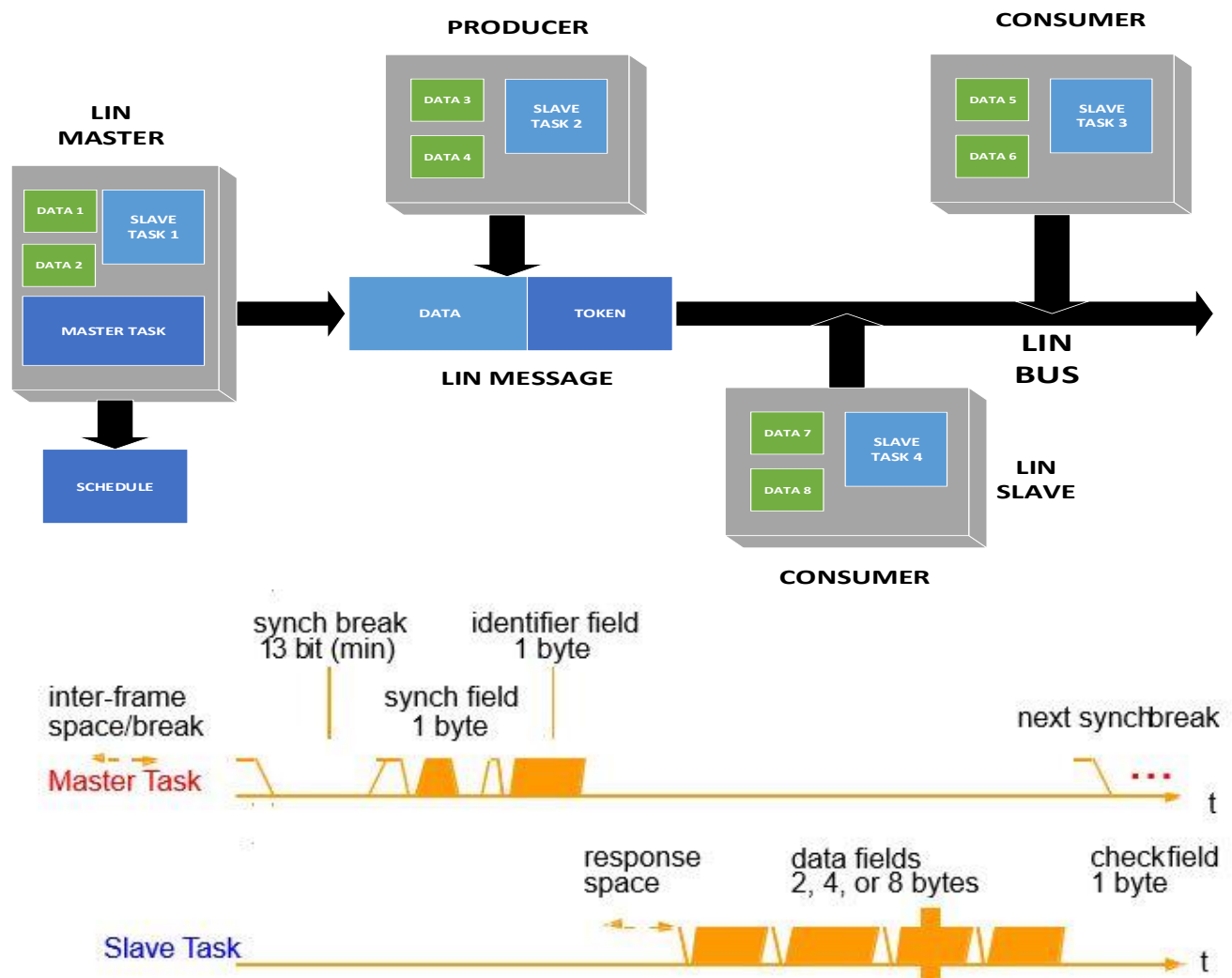


Figure 2: MPC56X B/C Boolero Architecture Family

11.1 Network Graphical Description

11.1.1 Figure 11: Deployment Diagram (A)

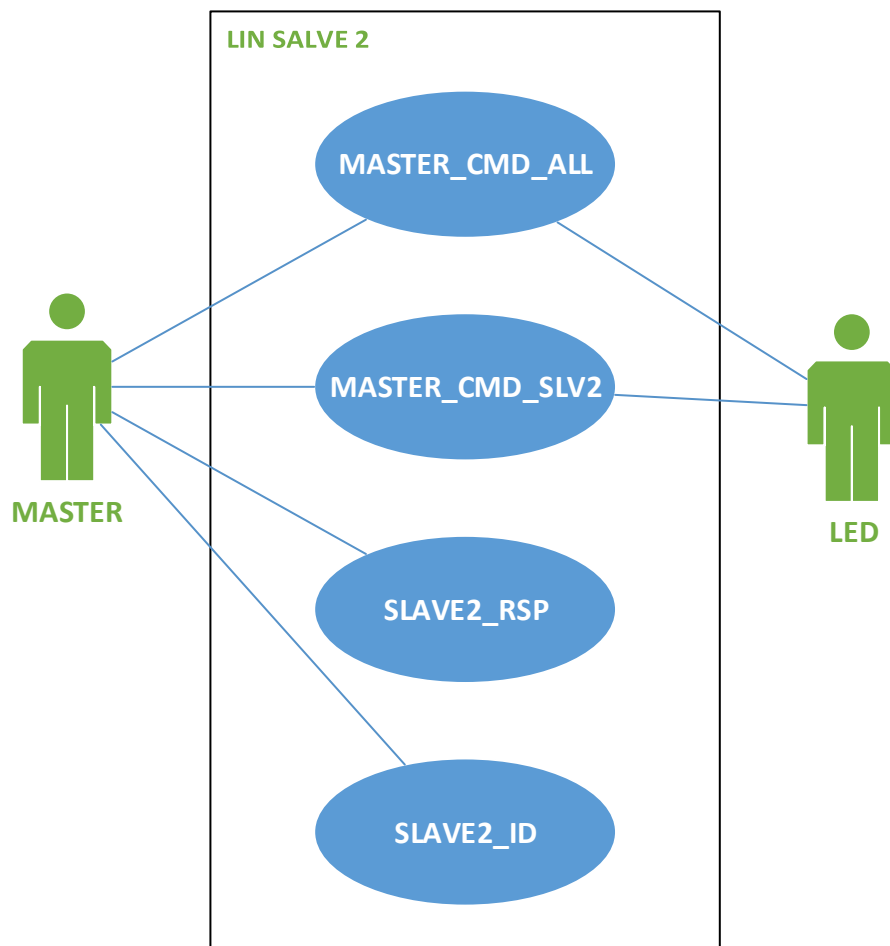


Deployment Diagram. Represents the physical configurations of software and hardware items.

The network showed in the figure above represents the full integrated system, which consists of the master node and the four slaves shown.

Our assignment is to develop the slave node labeled as Slave3, with all its internal components and the required actions to perform.

11.2 Use Case Diagram



11.2.1 Figure 12: Use Case Diagram

Master: Node that controls the network. This node can send up to of 13 pre-defined messages, but in this diagram only the messages relevant to Slave3 node are shown.

LED: Physical On-board LEDs which will respond to the commands given by the system.

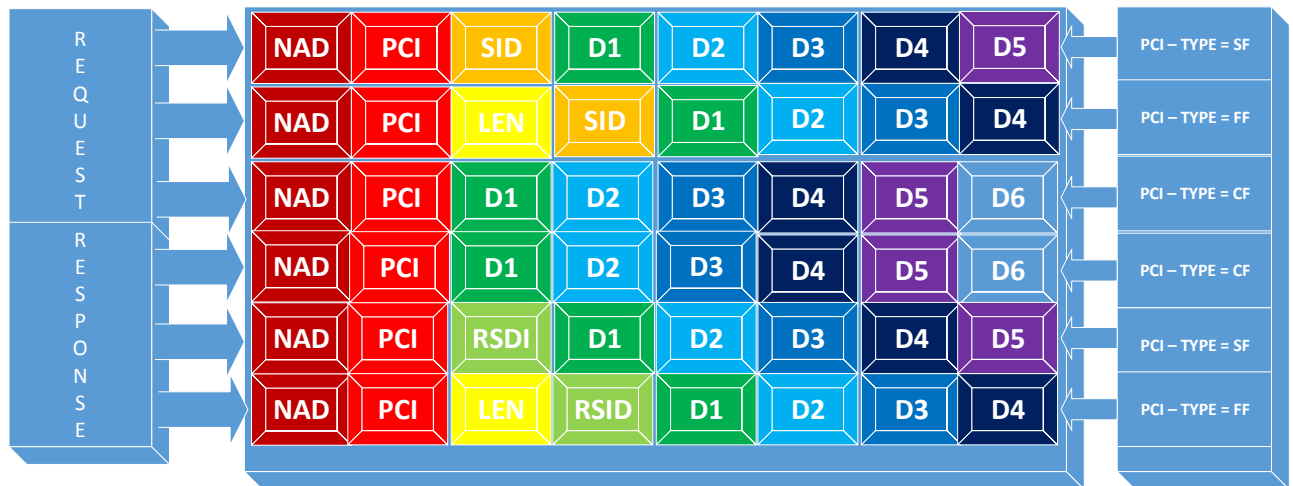
MASTER_CMD_ALL: Message sent to every slave node to execute a command.

MASTER_CMD_SLV2: Message sent specifically to slave node 3 to execute a command.

SLAVE2_RSP: Message sent from Slave3 to the master node with a specific response.

SLAVE2_ID: Message sent from Slave2 node to the master with the corresponding ID.

11.3 Component diagram request & response (A/D)



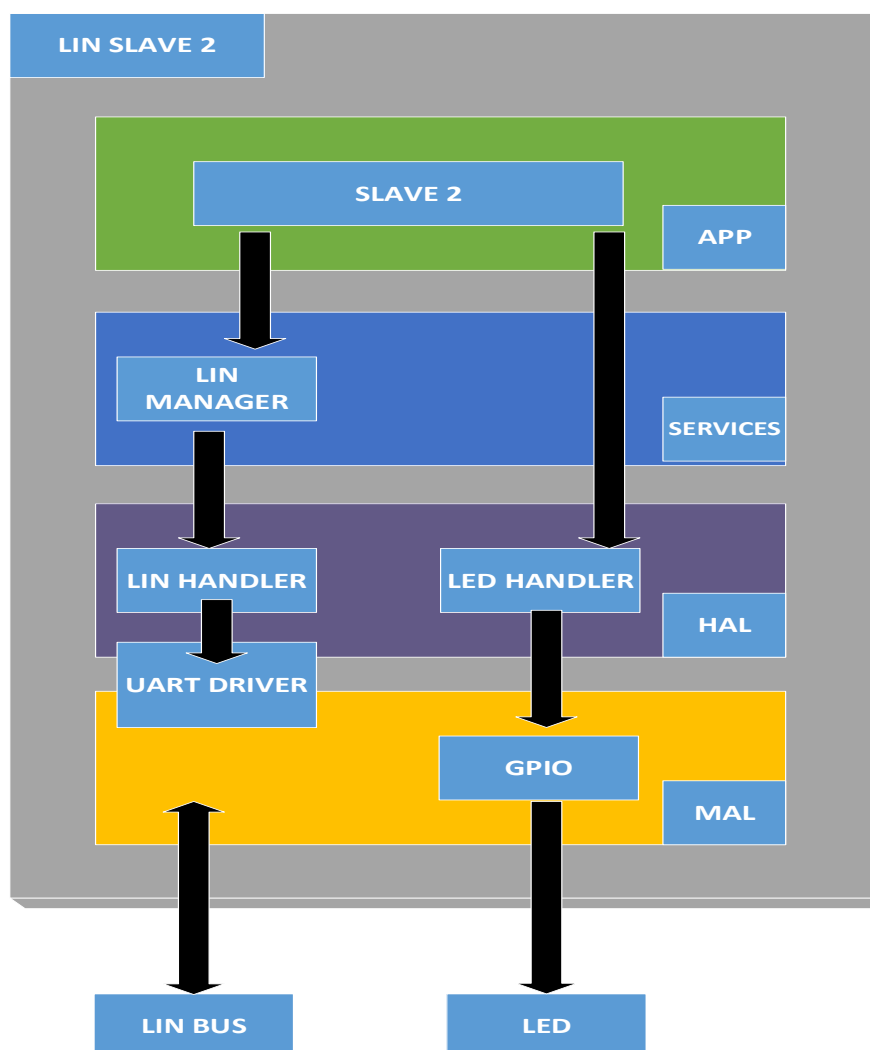
11.3.1 Figure 3: Component diagram (A/D).

11.3.2 Figure: Components Descriptions

NAD	Is the address of the slave node being addressed in a request, only slave nodes have an addressed NAD is also used to indicate the source of a response. NAD values are in the range 1 to 127 (0x7F) while 0 and 128 (0x80) to 255 (0xFF) are reserved for other purpose																																												
PCI	<table><tr><th rowspan="2">Type</th><th colspan="4">PCI type</th><th colspan="4">Additional information</th></tr><tr><th>B7</th><th>B6</th><th>B5</th><th>B4</th><th>B3</th><th>B2</th><th>B1</th><th>B0</th></tr><tr><td>SF</td><td>0</td><td>0</td><td>0</td><td>0</td><td colspan="4">Length</td></tr><tr><td>FF</td><td>0</td><td>0</td><td>0</td><td>1</td><td colspan="4">Length/256</td></tr><tr><td>CF</td><td>0</td><td>0</td><td>1</td><td>0</td><td colspan="4">Frame counter</td></tr></table> <p>Table 3.1: Structure of the PCI byte.</p> The PCI (Protocol Control Information) contains the transport layer flow control information. Three interpretations of the PCI byte exist,	Type	PCI type				Additional information				B7	B6	B5	B4	B3	B2	B1	B0	SF	0	0	0	0	Length				FF	0	0	0	1	Length/256				CF	0	0	1	0	Frame counter			
Type	PCI type				Additional information																																								
	B7	B6	B5	B4	B3	B2	B1	B0																																					
SF	0	0	0	0	Length																																								
FF	0	0	0	1	Length/256																																								
CF	0	0	1	0	Frame counter																																								
LEN	A LEN byte is only used in FF; it contains the eight least significant bits of the message length. Thus, the maximum length of a message is 4095 (0xFFFF) bytes.																																												
SID	The service identifier (SID) specifies the request that shall be performed by the slave node addressed. 0 to 0xAF																																												

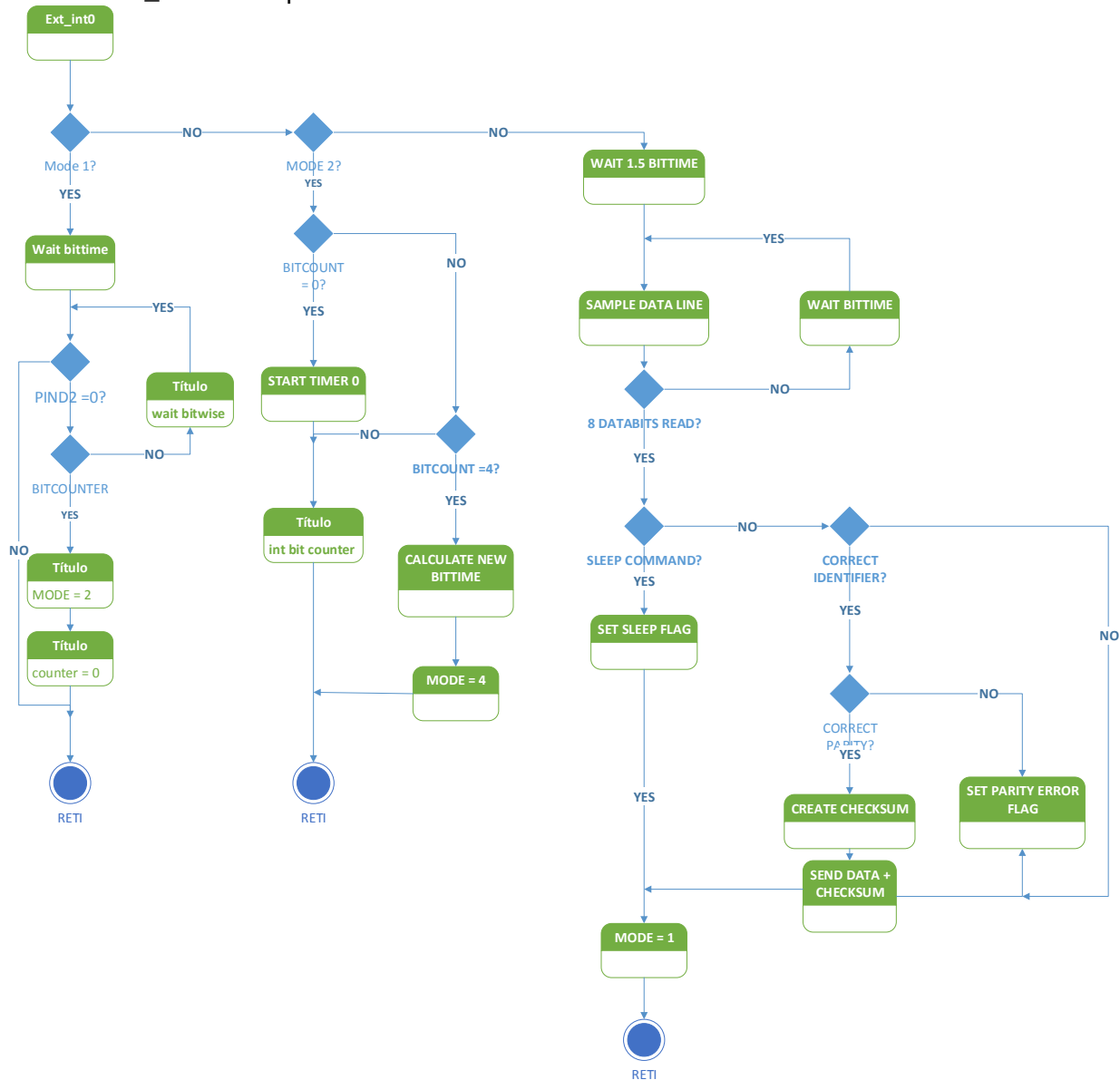
RSID	The response service identifier (RSID) specifies the contents of the respons. The RSID for a positive response is always SID + 0x40. If the service is supported in the slave node the response is mandatory (seven if the request is broadcast). The support of a specific service will be listed in the NCF.
D1 TO D5	The interpretation of the data bytes (up to five in a node configuration PDU) depends on the SID or RSID. If a PDU is not completely filled the unused bytes shall be recessive, their value shall be 255 this is necessary since a diagnostic frame is always eight bytes long.

11.3.3 Component diagram SW



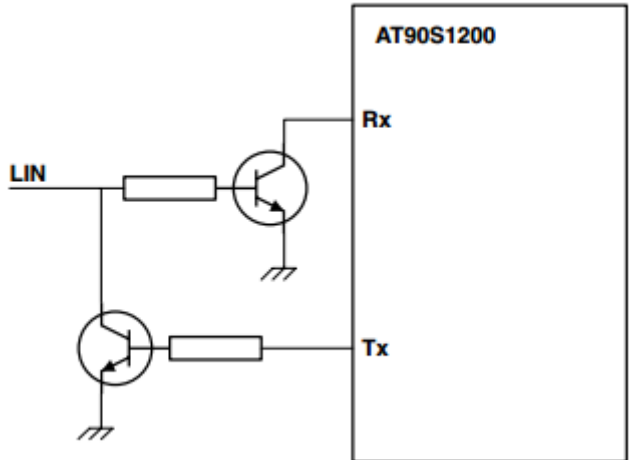
11.4 Activity Diagram (R)

Dataflow of ext_int0 Interrupt Handler Routine



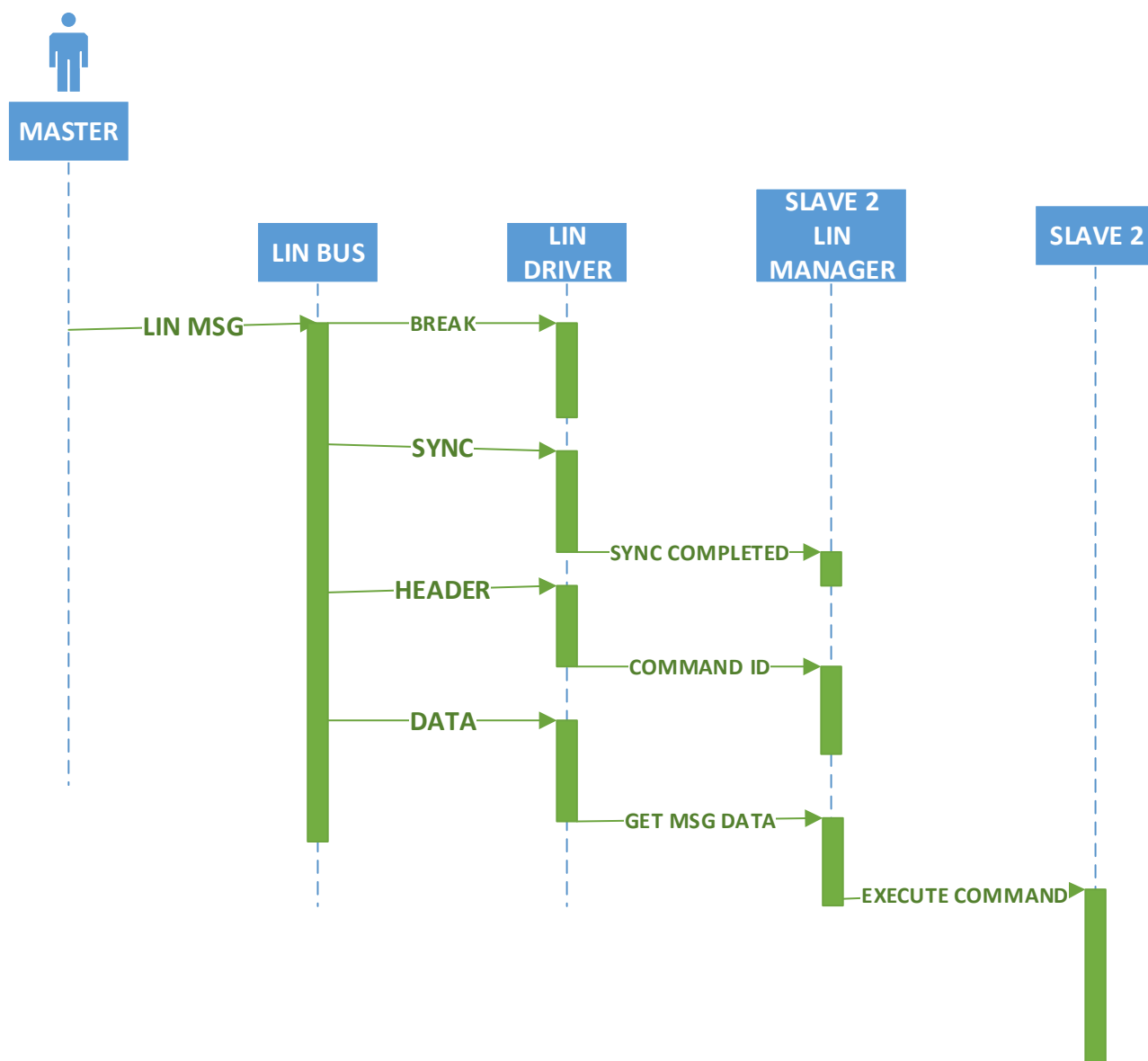


check_parity	This routine counts the number of recessive bits in the counter variable. If the number is even, the LSB of the bitcount variable is 0. If the number is odd, the LSB of the bitcount variable is 1. This is used to check the counter variable for odd or even parity.
Timer0 Overflow: tim0_ovf	Timer/Counter0 used in this application is an 8-bit counter. To be able to count eight bittime, this routine is called when Timer0 overflows, incrementing the "counter" variable. By combining the "counter" value, and the Timer/Counter1 value, the result gives a 16- bit number indicating the time for eight bits. Dividing this by eight gives the bittime. The microcontroller runs on a chosen frequency of 64 MHz and the counter increments every clock cycle. Thus, the bittime is given directly in microseconds.
putchar	The putchar routine is used for sending data via the LIN bus. When sending a "0", PIND2 is set as output with a "0" in the Port Register. When sending a "1", PIND2 is set as input and the LIN bus' pull-up resistor pulls the level to logical "1". This acts like an open collector output. The data to be sent should be placed in adjacent registers in ascending order (e.g., data1 to datan is placed in register rx to rx+n). The Slave sends either two, four, or eight data bytes decided by bit four and five in the Identifier Field. The constant data_addr in the assembly file holds the register number of the first data to send.
delay	The delay routine gives a delay of [temp2] cycles \pm 1 cycle. This routine is used to create the required delay between the

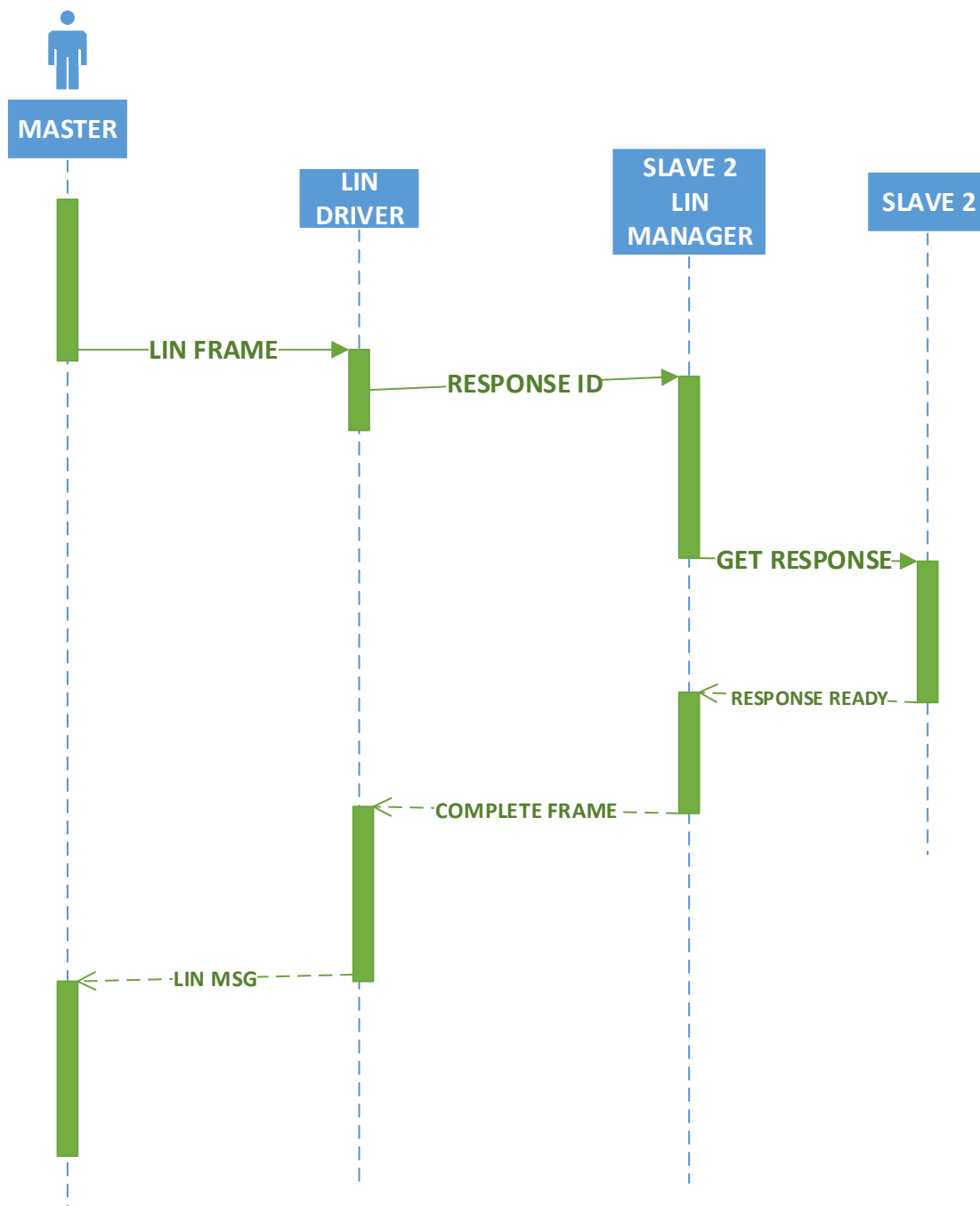
	sampling of bits.
wake-up	To generate a wake-up call on the bus this routine may be called from the main program. The routine is not used by the ext_int0 routine.
Hardware Considerations	The LIN standard specifies an operating voltage range for the "dominant" voltage between 9V and 18V (40V while not operating). This application is constructed for a LIN voltage of 5V. Since the AVR microcontroller used in this application requires a stable 5V supply, the most cost-effective and easiest solution is to let the LIN bus dominant voltage be 5V. If this is not acceptable there are some hardware solutions to interface the 5V LIN to a 9 - 18V LIN.
Method	 <p>Using two bipolar transistors (in the same housing) may be a more cost-effective solution (see Figure 14). These transistors are obtainable complete with base resistors and are less expensive than FET transistors.</p>



11.5 SEQUENCE

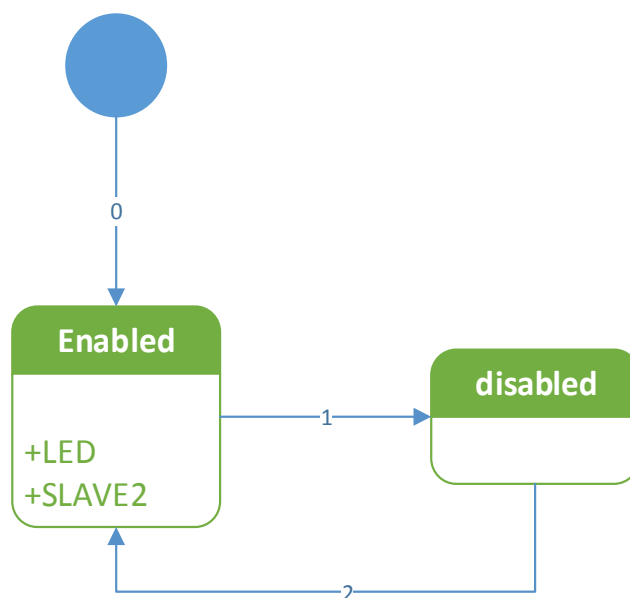


11.6 Response Sequence Diagram



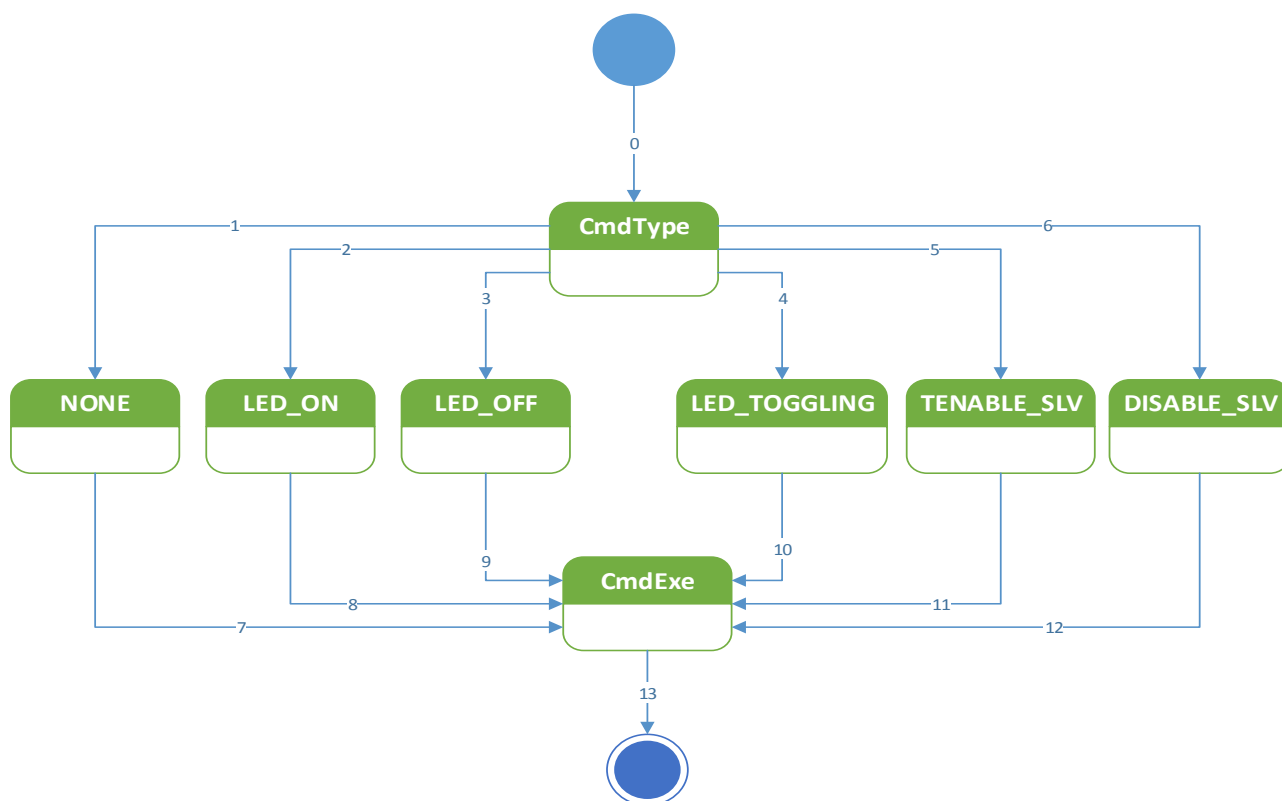
11.7 State machine diagram

11.7.1 Slave 2



TRANSITION	CONDITION	ACTION
0	Initialize LIN Slave2	Initialize node as enabled.
1	t_cmdType==cmd_disable_slave	Node==Disabled
2	t_cmdType==cmd_enable_slave	Node==Enabled / Access to either super state.

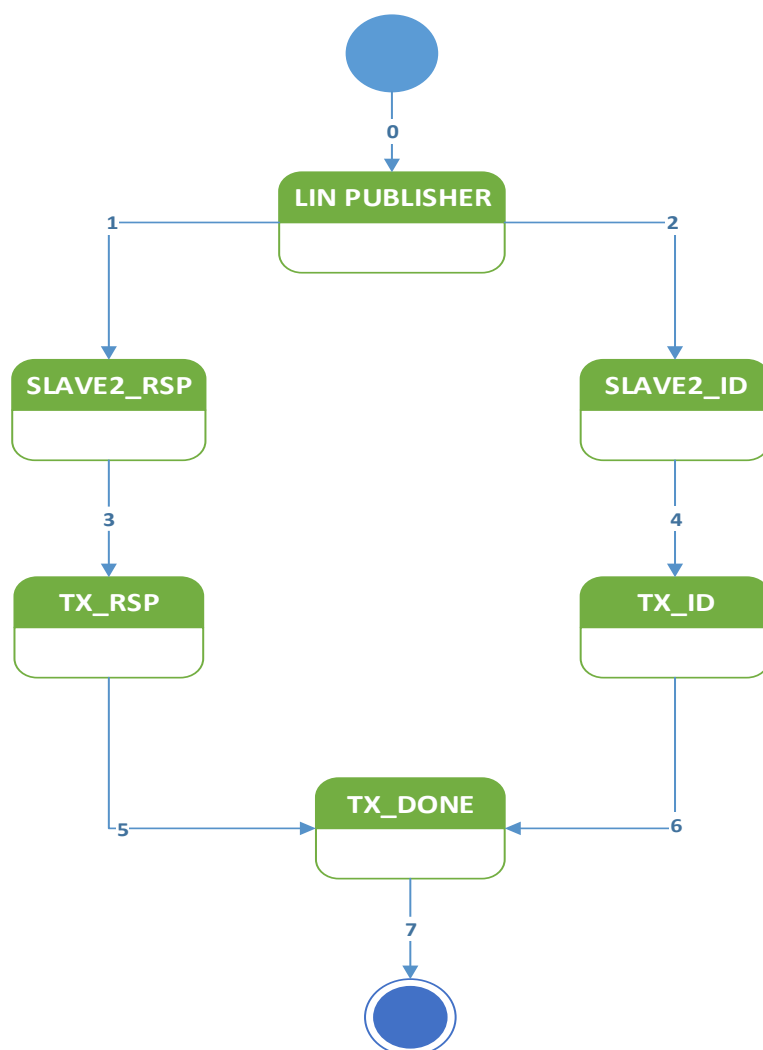
11.7.2 Lin manager



TRANSITION	CONDITION	ACTION
0	Signal Msg == MASTER_CMD	Initialize CmdType
1	t_cmdType==cmd_NONE	Initialize none.
2	t_cmdType==cmd_LED_on	Initialize LED_ON
3	t_cmdType==cmd_LED_off	Initialize LED_OFF
4	t_cmdType==cmd_LED_toggling	Initialize LED_TOGGLING
5	t_cmdType==cmd_enable_slv	Initialize ENABLE_SLV
6	t_cmdType==cmd_disable_slv	Initialize DISABLE_SLV
7	Command variable updated to None.	Go to CmdExe
8	Command variable updated to LED on.	Go to CmdExe
9	Command variable updated to LED off.	Go to CmdExe
10	Command variable updated to LED toggling.	Go to CmdExe
11	Command variable updated to Enable.	Go to CmdExe
12	Command variable updated to	Go to CmdExe



	Disable.	
13	Cmd==executed.	Exit state machine.

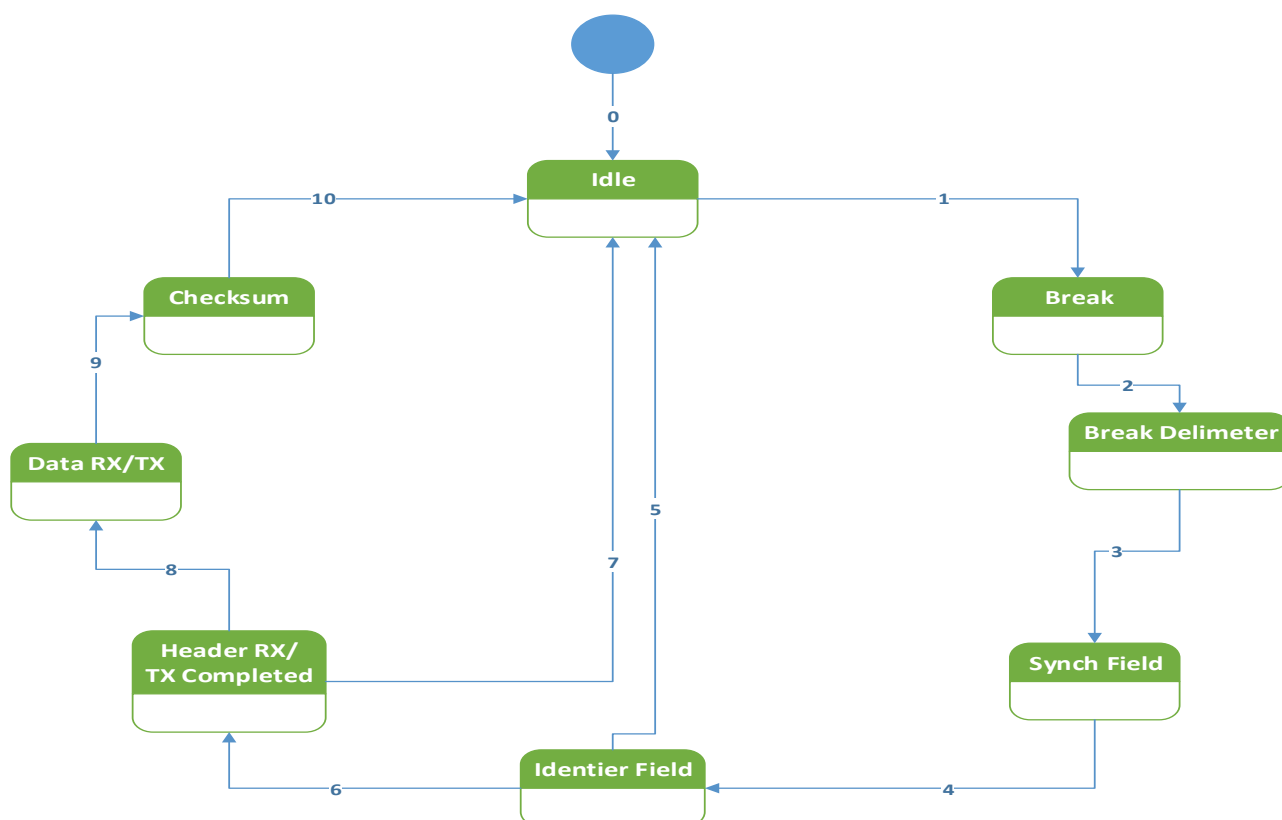


TRANSITION	CONDITION	ACTION
0	(Signal Msg == SLAVE2_RSP) (Signal Msg == SLAVE2_ID)	Initialize LIN Publisher.
1	If Signal Msg == SLAVE2_RSP	Initialize SLAVE2_RSP.
2	If Signal Msg == SLAVE2_ID	Initialize SLAVE2_ID.



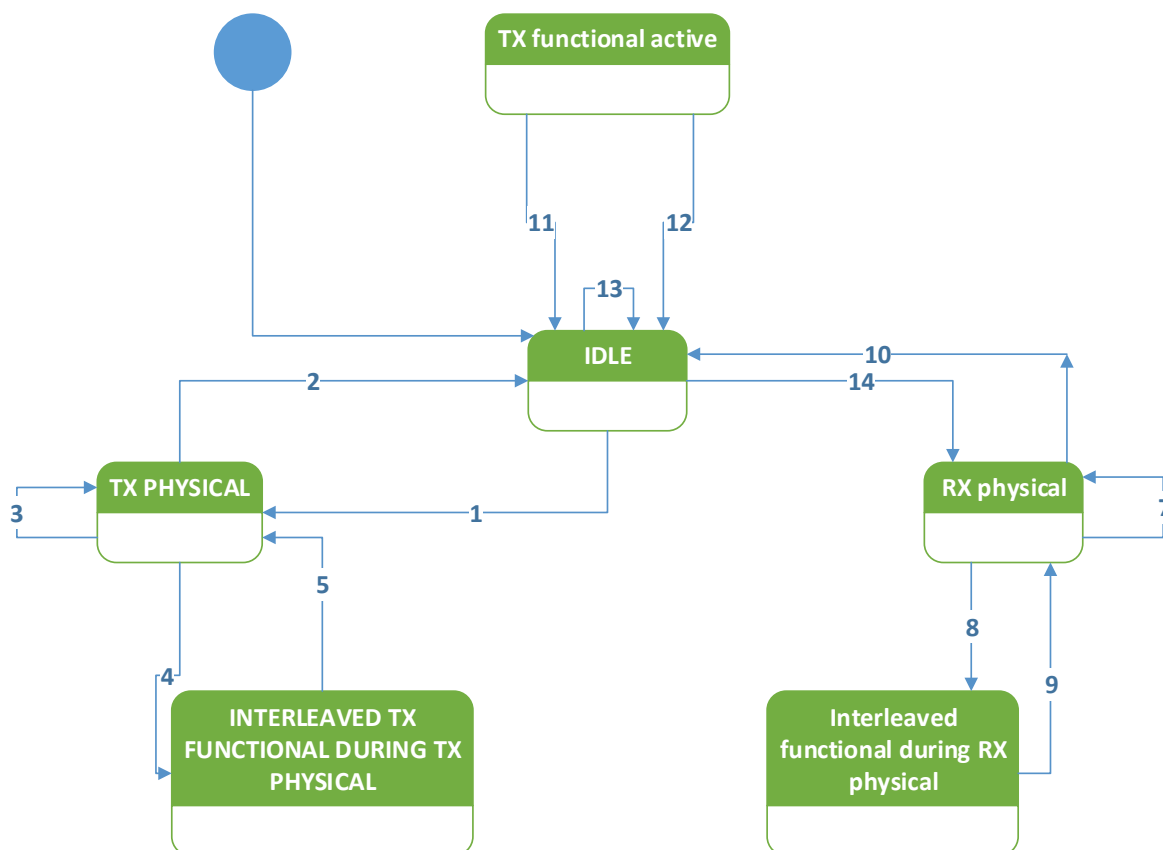
3	(ID field==valid)&&(Header==sent)	Initialize TX_RSP.
4	(ID field==valid)&&(Header==sent)	Initialize TX_ID.
5	Data TX_RSP==transmitted	Go to TX_Done.
6	Data Tx_ID==transmitted	Go to TX_Done.
7	Transmission completed.	Exit state machine.

11.7.3 LIN Handler



TRANSITION	ACTION
0	Initialize Idle
1	Go to break.
2	Enter Break Delimeter.
3	Enter Synch Field.
4	Initialize Identifier Field.
5	Return to Idle.
6	Enter Header Rx/ Tx compl.
7	Return to Idle.
8	Go to Data Rx/Tx.
9	Enter Checksum.
10	Return to Idle.

11.7.4 Master node transmission handler

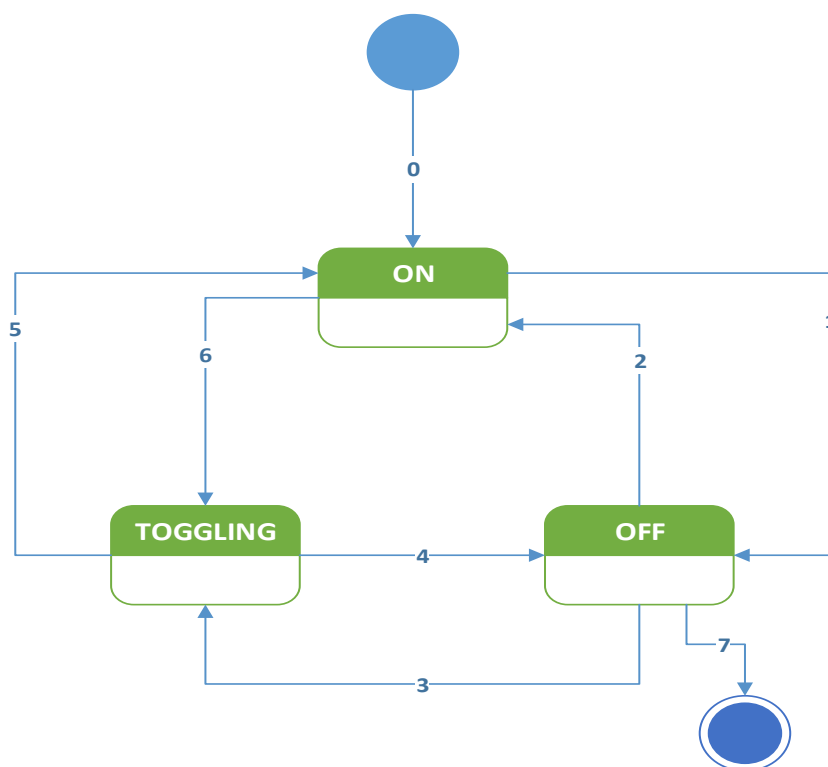


1.- IDLE STATE	<p>Tx physical active state</p> <p>Trigger: Start of a physical transmission from the back-bone bus to a slave node.</p> <p>Effect: Start executing diagnostic master request frame schedule tables and handle the transport protocol.</p>
2.- Tx physical active state	<p>Idle state</p> <p>Condition: routing of the physical transmission from the back-bone bus to the cluster has finished or a transport protocol transmission error</p> <p>Action: stop executing master request frame schedule tables.</p>
3.- Tx physical active state	<p>Tx physical active state</p> <p>Condition: routing of the physical transmission</p>



	<p>from the back-bone bus to the slave node has not finished</p> <p>Action: continue executing master request schedule table and handle transport protocol on LIN</p>
4.- Tx physical active state	<p>Interleaved functional during Tx physical state</p> <p>Condition: functional addressed request from the back-bonebus has been received.</p> <p>Action: interrupt the physical transmission to the slave node and execute a single master request frame schedule to transmit the functional addred request onto the cluster.</p>
5.- interleaved functional during Tx physical state	<p>Tx physical active state</p> <p>Condition: functional addressed request has been routed onto the cluster.</p> <p>Action: continue the interrupted physical transmission to the slave node.</p>
6.- Tx physical active state	<p>Rx physical active state</p> <p>Condition: functional transmission to the slave node has been successfully completed (according to the transport layer specification).</p> <p>Action: stop executing master request frame schedule tables, start executing slave response frame schedule tables and handle the incoming response from the previously addressed slave node.</p>
7.- interleaved functional during Rx physical state	<p>Rx physical active state</p> <p>Condition: functional addressed request has been routed onto the cluster.</p> <p>Action: restart executing slave response frame schedules and continue the interrupted reception from the slave node.</p>

11.7.5 LED Handler



TRANSITION	CONDITION	ACTION
0	Initialize MCU	Default state is ON
1		Go to OFF
2		Return to ON.
3		Set to TOGGLING
4		Go to OFF.
5		Go to ON.
6		Go to TOGGLING.
7		Exit the state machine.



12 Application Program Interface Specification ver. 2.2a

12.1 CORE API

The LIN core API has a set of functions all based on the idea to give the API a separate name space, in order to minimize the RISK of conflicts with existing software. All functions and types will have the prefix "I_" (lowercase L followed by an underscore).

The LIN core shall define the following types:

- L_bool - 0 is false, and non- zero (>0) is true
- L_ioctl_op - implementation dependent
- L_irqmask - implementation dependent
- L_u8 - unsigned 8 bit integer
- L_u16 - unsigned 16 bit integer

Name	Prototype	Description
L_sys_init	L_bool I_sys_init (void);	Performs the initialization of the LIN core. The scope of the initialization is the physical node. The call to the I_sys_init is the first call a user must use in the LIN core before using any other API function. The function returns: Zero if the initialization succeeded Non-zero if the initialization failed
Static Prototype		Dynamic Prototype
None		None

Name	Prototype	Description
Scalar signal read	none	Reads and returns the current value of the signal
Static Prototype		Dynamic Prototype
L_bool I_bool_rd_read (void); L_u8 I_u8_rd_read (void); L_u16 I_u16_rd_read (void);		L_bool I_bool_rd (I_signal_handle read); L_u8 I_u8_rd (I_signal_handle read); L_u16 I_u16_rd (I_signal_handle read);

Name	Prototype	Description
Scalar signal write	none	Sets the current value of the signal to v.
Static Prototype		Dynamic Prototype
Void I_nool_wr_write (I_bool v); Void I_u8_wr_write (I_u8 v); Void I_u16_wr_write (I_u16 v);		Void I_bool_wr (I_signal_handle write, I_bool v); Void I_u8_wr (I_signal_handle write, I_u8 v); Void I_u16_wr (I_signal_handle write, I_u16 v);



Name	Prototype	Description
Byte array read	none	Reads and returns the current values of the selected bytes in the signal. The sum of start and count shall never be greater than the length of the byte array.
Static Prototype		Dynamic Prototype
Void I_bytes_rd_data_read (I_u8 start, L_u8 count, L_u8* const data);		Void I_bytes_rd (I_signal_handle data_read, L_u8 start, /*first byte to read from*/ L_u8 count, /*number of bytes to read*/ L_u16 const data); /*where data will be written*/

Name	Prototype	Description
Byte array write	none	Sets the current value of the selected bytes in the signal specified by the name write data to the value specified.
Static Prototype		Dynamic Prototype
Void I_bytes_wr_write_data (I_u8 start, L_u8 count, Const I_u8* const data);		Void I_bytes_wr (I_signal_handle write_data, L_u8 start, L_u8 count, Const I_u8* const data);

Name	Prototype	Description
L_flg_tst	none	Returns a C boolean indicaing the current state of the flag specified by the name flag_state, returns zero if the flag is cleared, non-zero otherwise
Static Prototype		Dynamic Prototype
L_bool I_flg_tst_flag_state (void);		L_bool I_flg_tst (I_flag_handle flag_state);

Name	Prototype	Description
L_flg_clr	none	Sets the current value of the flag specified by the name clear_flag to zero
Static Prototype		Dynamic Prototype
Void I_flg_clr_clear_flag (void);		Void I_flg_clr (I_flag_handle clear_flag);

Name	Prototype	Description
L_flg_tick	none	The I_sch_tick function follows a schedule. When a frame becomes due, its transmission is initiated. When the end of the current schedule is reached, I_sch_tick starts again at the beginning of the schedule.
Static Prototype		Dynamic Prototype
L_u8 I_sch_tick_tickflg		L_u8 I_sch_tick (I_ifc_handle tickflg);



Name	Prototype	Description
L_sch_set	none	Sets up the next schedule to be followed by the I_sch_tick function for a certain interface tickflg. The new schedule will be activated as soon as the current schedule reaches its next schedule entry point.
Static Prototype		Dynamic Prototype
Void I_sch_set_tickflg (I_schedule_handle schedule, I_u8 entry);		Void I_sch_set (I_ifc_handle tickflg, L_schedule_handle schedule, L_u8 entry);

Name	Prototype	Description
L_ifc_goto_sleep	none	This call request slave nodes on the cluster connected to the interface to enter bus sleep mode by issuing one go to sleep command,. The go to sleep command will be scheduled latest when the next schedule entry is due.
Static Prototype		Dynamic Prototype
Void I_ifc_goto_sleep (I_ifc_handle gsleep);		Void I_ifc_goto_sleep_gsleep (void);

Name	Prototype	Description
L_ifc_wake_up	none	The function will transmit one wake up signal. The wake up signal will be transmitted directly when this function is called. It is the responsibility of the application to retransmit the wake up signal according to the wake up sequence defined
Static Prototype		Dynamic Prototype
Void I_ifc_wake_up_wup (void);		Void I_ifc_wake_up (I_ifc_handle wup);