INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS

LIN 1.3 DRIVER IMPLEMENTATION IN A DEMO9S12XEP100 BOARD

Tesina para obtener el grado de:

Especialista en sistemas embebidos

Presenta(n)
Alejandro Alcaraz Zamudio,

Director de tesina: Dr. Esteban Martinez Guerrero

San Pedro Tlaquepaque, Jalisco. Noviembre de 2016.

Abstract

In this document it is described in detail the implementation of a LIN 1.3 protocol communication between two commercial boards configured as Master and Slave nodes respectively.

This implementation is the last part of a series of labs developed in the courses of graduate program of "Speciality in Embedded Systems" of Instituto Tecnologico y de Estudios Superiores de Occidente (ITESO), which in addition of granting general knowledge of embedded systems, it is focused in automotive implementations.

Inside the text, it is explained how a driver was developed to stablish communication between boards and which consideration were made in order to create a configurable driver. First, an analysis of different communications options in LIN protocols is presented, then it is briefly described how the implemented code using resources available in LIN protocol was designed.

In particular, for the implemented code, two configurable structures were used, one is the configuration of LIN driver and the other is the configuration of LIN frames. These structures were used in order to create a status structure for each configured driver and to implement configurable schedule tables as LIN protocol specifies it.

Once designed, a series of test's code were performed. A test bench was assembled in order to perform testing the functionality of code in hardware. The test bench consists of two Freescale boards DEMO9S12XEP100, a P&E USB-Multilink-Interface debugger, a USB to Serial Adapter, an Oscilloscope and a Laptop. The main signature that communication between boards was correctly achieved was the transmission of test frames from Master to Slave by changing the operation mode of each node.

Acknowledgment

I want to acknowledge all the people involved in this phase of my personal development. As each of them, classmates, teachers, advisors and others, push me through one of the most important milestones of my personal and professional development.

At this point, it has become clearer to me, that knowledge is not only found aboard, but we can found here through sharing, experience and hard work. I had found this knowledge sharing in the right place as this postgraduate degree demands it.

I also would like to acknowledge CONACYT for its support and make possible for me to take this postgraduate degree.

Also, I want to acknowledge ITESO as it deserves, which make this possible by providing necessary tools and support though all the cycle, always finding the way to provide any necessary medium to develop a better student.

Index

Abstract 3

Acknowledgment 4

Index 5

1. Introduction 6

2. Objectives 7

3. Development of driver 9

3.1 Driver Configuration 9

3.2 Driver Status 10

3.3 Driver Functions 11

3.4 Driver Functionality 13

3.4.1 Master Node 13

3.4.2 Slave Node 14

3.4.3 State Machine implementation 15

4. Driver testing 18

5. Conclusions 21

6. Personal reflexions 21

Reference 22

Appendices 23

Signals 23

Frames 23

Schedule Tables 24

1. Introduction

Local Interconnect Network (LIN) is part of the low cost automotive communication protocols such as CAN and Flexray for embedded control [1]. “The LIN protocol as proposed is an automotive focused low speed universal asynchronous receiver transmitter (UART) based network.”[2] This is the reason of the nominal bit rate being from 1 to 20 kbit/s.

Some key characteristics that LIN protocol offers are [2]:

- Signal based communication.
- Schedule table base transfer.
- Master/Slave communication with error detection.
- Node configuration.
- Diagnostic service transportation.

LIN is based in signals communication, a single signal or a group of signals compose a frame. Signals transmission is encapsulated in a frame. Each signal is allocated within the same message's data field, with an offset to determinate start and end of signal value.

The frames are transmitted according to a schedule table, with assigned publisher and subscribers.

The frame is initiated by the Master and it contains two parts, the frame header sent by the Master and the frame response, which encompasses the actual message (signals) and a checksum field.

The frame header contains a Sync Brake (allowing the Slave to recognize the beginning of a new message), a sync field with a regular bit pattern for clock synchronization and an identifier field defining the content type and length of the frame response message. The identifier is encoded by 6 bit (allowing 64 different message types) and 2 bits for protection.

The frame response contains up to 8 data bytes and a checksum byte. Since an addressed Slave does not know a priori to the reception of the respective frame header that it has to send a message, the response time of a Slave is specified within a time window of 140% of the nominal length of the response frame.

From the Slave’s view, the LIN protocol is a plain polling protocol, since the Slaves only react on the frame header from the Master. It is the Master’s task to issue the respective frame headers for each message according to a scheduling table. The configuration of the network must ensure that each message has exactly one producer. Several nodes can subscribe to a particular message [3].

2. Objectives

In this report we present the implementation of a LIN 1.3 Protocol in DEMO9S12XEP100 board.

The implementation will test the communication between a Master and a Slave node configured to transmit 3 frames according to a schedule table.

Each node should test the frame's PID field in Master task and, then, corresponding publisher node should publish signal data (data field).

As stated in LIN standard, subscriber's node should calculate Check Sum (CRC) value and test it against transmitted CheckSum in order to validate frame reception. If a transmitted CheckSum does not match with the internally calculated CheckSum, frame should be disposed.

All received data will be assigned to a global variable (Rx_Data_Frame array), which only will be updated with the latest valid data frame received.

The implementation must support Master and Slave LIN nodes and configured to transmit 3 frames according to following specification:

```
SCH_TBL1
{
        Frame1 delay 15 ms;
        Frame2 delay 25 ms;
        Frame3 delay 40 ms;
}

Frame1:19,MST
{
        RearFogLampInd,0;
        PositionLampInd,1;
        FrontFogLampInd,2;
        IgnitionKeyPos,3;
        SLVFuncIllum,8;
        SLVSymbolIllum,12;
 }
Frame2:34,SLV
{
        SLVSWPartNo,0;
        SLVHWPartNoB0,8;
        SLVHWPartNoB1,24;
 }
 Frame3:59,SLV,6
{
        FanIdealSpeed,0;
        FanMeasSpeed,16;
        WaterTemp,32;
 }
```

In order to achieve proposed LIN protocol implementation, it was required the following resources. It is used either for debugging purposes or for testing analysis.

Hardware

- 2 Freescale boards DEMO9S12XEP100
- P&E USB-Multilink-Interface debugger
- Oscilloscope
- USB to Serial Adapter
- Laptop

Software

- CodeWarrior for S12 version 5.1 Special Edition
- Docklight terminal

3. Development of driver

According to Fig.1, to implement the communication between Master and Slave nodes is to define its role; so as first step in the design of the driver, we define one as Master and the other as Slave. As in LIN protocol the Master node will always start the communication sending the message header, in which a Sync Break field, a Sync Byte field and a Protected Identifier (PID) field data are encapsulated.

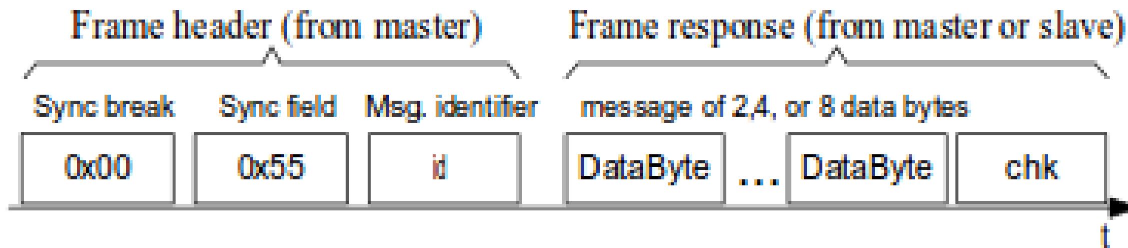

Figure 1: LIN protocol frame format [3]

According to the PID content, the response message can be transmitted from the Master node or from the Slave node to its subscribers. In the first case, the Master node will transmit the whole frame, while in the second case it will only transmit the header and waits for a Slave to transmit the message response.

This implementation was developed to cover both roles, Master and Slave, where compilation switches were used in order to only compile the needed code according to the node to be implemented. This was done using the preprocessor macro (MST – Master node, SLV – Slave node) in *cnf_lin_protocol.h* file:

```
#define LIN_CONFIGURED_DEVICE  SLV
```

This file is found under LIN filter folder, along with the following configuration and program files:

- *cnf_lin_protocol.h & cnf_lin_protocol.c:* Frame and Signals configuration files.
- *lin_protocol.h:* Frame and Signals structures declaration file.
- *cnf_lin_driver.h & cnf_lin_driver.c:* Driver Configuration Structures declaration and assign.
- *lin_driver.h & lin_driver.c:* Program code of LIN driver.

3.1 Driver Configuration

The second step is the driver configuration; it is stated that driver works with the Serial Communication Interface (SCI) [3] module and a LIN transceiver (MC33661) that provides needed voltage levels. This configuration needs to be routed in hardware to desired output. Also, SCI module provides the capability to detect and send break characters to the bus, using assigned control registers (SCICR) [4].

Using SCI driver software the used module was configured to work along with developed software. This SCI configuration holds the information related to bit rate and interruptions services, as callbacks (see code lines below).

```
const tSCIchannel_config SCI_channel_cfg[] =
{
   {
      (UINT32)9600,                    /* SCI_baudrate */
      (tCallbackFunction)NULL,         /* SCI_TX_callback */
      /* SCI_RX_callback */
      #if (LIN_CONFIGURED_DEVICE == MST)
      (tCallbackFunction)LIN_Send_Frame,
      #elif (LIN_CONFIGURED_DEVICE == SLV)
      (tCallbackFunction)LIN_Send_Frame,
      #endif
      SCI_CH0,                 /* SCI_Channel */
      (UINT8)ENABLE,           /* SCI_TX_enable */
      (UINT8)ENABLE,           /* SCI_RX_enable */
      (UINT8)ENABLE,           /* SCI_TIE_enable  */
      (UINT8)ENABLE,           /* SCI_RIE_enable */
      (UINT8)SCI_TX_MAX_SIZE,    /* SCI_TX_MAX_BUFFER_SIZE */
      (UINT8)SCI_RX_MAX_SIZE     /* SCI_RX_MAX_BUFFER_SIZE */
   }
};
```

The configuration structures for LIN driver will help to basic information according to number of channels configured and LIN mode that this will work with, this is specified in code lines below:

```
/* Channel Configuration Structure */
typedef struct{
  enum tLIN_Channel u8Channel_ID;       /*Channel ID*/
  UINT8 u8Channel_Node;                 /*Channel Node: Master or Slave*/
  UINT8 u8Channel_Enable;               /*Channel Enabled*/
  UINT8 u8Rx_Buffer_Size;               /*Max buffer size for RX*/
  UINT8 u8Tx_Buffer_Size;               /*Max buffer size for TX*/
}tLIN_Channel_Config;
```

3.2 Driver Status

The third step is to make the proper configuration that will help to determinate the driver status. Therefore, a status structure was implemented, which helps the driver and other modules, to know the current state of such driver itself. The information could be used only internally or could be readout externally using interfaces.

As the status structure held everything related to current state of the driver, it should include all variables used by the driver allocated dynamically.

Using the information provided by configuration structures, the following status structure has been defined:

```
typedef struct
{
```

```
    enum tLIN_Channel u8Channel_ID;   /*Channel ID*/
    UINT8 u8Channel_State;           /*Further Implementation */
    UINT8 PID;                       /*Protected ID*/
    UINT8 Mode;                      /*Slave Task Mode: TX / RX*/
    UINT8 * Tx_Buffer;               /*Pointer to TX Buffer (Start Reference)*/
    UINT8 * ptrTx_Buffer;            /*Pointer to TX Buffer*/
    UINT8 Tx_Buffer_Length;          /*TX buffer Length*/
    UINT8 * Rx_Buffer;               /*Pointer to RX Buffer (Start Reference)*/
    UINT8 * ptrRx_Buffer;            /*Pointer to RX Buffer*/
    UINT8 Rx_Buffer_Length;          /*RX buffer Length*/
    UINT16 CheckSum;                 /*CheckSum Calculation*/
    UINT8 u8TaskState;               /*State Machine Task State*/
}tLIN_Channel_Status;
```

In this implementation it is proposed an internal buffer allocated dynamically to hold any transmitted and received information, this is used only by the driver. Once that all frame has been received and validated with the checksum, the received data frame is moved to Rx_Data_Frame array.

For implementation testing purposes valid received frames will be hold in a global variable. Such variable is Rx_Data_Frame array, in which last valid received frame will be placed. In current implementation this variable is not extern with any interface, it is just updated by frame.

The PID element will be used to be tested against Frames PID and proceed to configure the message response as transmission or reception mode (Mode element), if the node is configured in transmission mode, it will be responsible of publishing the data in Data field. On the other hand, if it is configured in reception mode, it will read data provided by the publisher.

The CheckSum element is used to calculate the data checksum and, either sends it when Transmit mode is selected, or validate it if Reception mode is selected.

The u8TaskState is used in state machine control to perform any needed operation accordingly. The different states used in this implementation are based in the following definitions of LIN protocol, these are displayed in descending order:

```
#define SYNC_BRK        0x00
#define SYNC_FIELD      0x01
#define PID_BYTE        0x02
#define DATA_BYTE       0x03
#define CHECKSUM        0x04
#define EOF             0x05
```

3.3 Driver Functions

This driver implementation resides in the following public and private functions. These functions are part of the developed code in order to achieve a stable implementation with a proper abstraction.

void vfnLIN_Initialization(const tLIN_Driver_Config * LINDriver_Config):

High Level Initialization of LIN configured channels; reception parameter is a pointer to configuration structure. In this Initialization, buffers and status structures are allocated into the dynamic memory, then buffers (vfnLIN_Rx_Clear and vfnLIN_Tx_Clear) are cleared and status structures are updated to initial values. After, a low level Initialization is performed with vfnLIN_Init() to send the state machine to initial state.
Also, SCI driver is initialized according to its configuration structure, this SCI driver initialization should be part of LIN driver as in this hardware LIN and SCI modules are based in same physical architecture.

void vfnLIN_Rx_Clear (enum tLIN_Channel channel):

Reception buffer clean; in this stage, data reception pointer (ptrRx_Buffer) is set to start buffer pointer (Rx_Buffer), buffer length and CheckSum are cleared (logical 0). Also, reception buffer's allocated memory is cleared.

void vfnLIN_Tx_Clear (enum tLIN_Channel channel):

Same as vfnLIN_Rx_Clear, but no physical memory clear is performed. In this case data is being overridden instead of cleared.

void vfnLIN_Write_Buffer (enum tLIN_Channel channel, UINT8* pu8Data, UINT8 u8DataLength):

With this instruction, buffer is queued in dynamic memory, then data transmit pointer (ptrTx_Buffer) is set to start buffer pointer (Tx_Buffer), and data Length is set accordingly. This data information is the already arranged data to be transmitted byte per byte.

void vfnLIN_Mode_Select (enum tLIN_Channel channel, UINT8 PID, UINT8 Mode, UINT8 frame_id):

Once the transmitted/received knows frame's PID, the Mode is set according to the PID publisher and, then, CheckSum is initialized to 0. If PID publisher is the same node, the frame is created by calling vfnLIN_Tx_Frame_Handler function. Otherwise, PID's configured frame size (u8frame_size) is set to reception buffer length (Rx_Buffer_Length).

void vfnLIN_Data_Tx_Task (enum tLIN_Channel channel):

Using this function one can transmit data from data transmit buffer using serial interface, also it performs CheckSum calculation (add with carry) and decreases data transmit length (Tx_Buffer_Length). Once reception data length is 0, CheckSum is calculated for the last time and state machine (u8TaskState) is moved to the next state.

void vfnLIN_Master_Rx_Handler(UINT8 channel):

Using this function, serial data received will be enqueued in reception data buffer. Also it performs CheckSum calculation (add with carry) and decreases expected data length (Rx_Buffer_Length). Once reception data length is 0, CheckSum is calculated for the last time, data buffer pointer (ptrRx_Buffer) is restarted and state machine (u8TaskState) is move to the next state.

void vfnLIN_Tx_Frame_Handler (enum tLIN_Channel channel, UINT8 frame_id):

This function is used to join frame signals into transmit bytes; it uses two 32 bytes arrays to arrange the different signals according to its values (u8init_value), size (u8signal_size) and offsets (u8signal_offset). Then data is enqueued in the transmit buffer.

3.4 Driver Functionality

In this implementation, the main driver functionality is performed with vfnLIN_Frame_Transmit function, which contains state machine for both cases (Master and Slave nodes). Each state machine is handled in different ways according to the node configuration.

3.4.1 Master Node

When the driver is configured as Master (MST) node, the state machine is first accessed when a frame (u8Active_LIN_Frame) is activated according to its delay values (u8delay). This action will set a state machine flag (u8State_Machine_set). Flag is used to transmit data in a background task, and to perform a low level initialization (vfnLIN_Init).

This low level initialization (vfnLIN_Init) clears LIN task state (u8TaskState) and configures SCI to perform a Break Sync transmission (vfnLIN_Break_config) event (see Fig. 2).

The frame activation test is performed in a cyclic 1 ms task; if a frame has been activated for continuous transmission, it will be performed in the background task. Continuous transmission is performed by accessing to state machine using vfnLIN_Frame_Transmit function.

When the state machine validates the PID_BYTE data, it can select its mode with function vfnLIN_Mode_Select. For instance, if reception mode (RX_MODE) is selected the state machine is accessed via SCI callback function. Otherwise, transmission control will proceed to be performed in the background task (see Fig. 2). If reception mode (RX_MODE) is selected the last received byte, CheckSum, will be tested against the one calculated internally. If both values match, data reception buffer will be copied into Rx_Data_Frame array. However, if values do not match, data reception buffer will be cleared in the next initialization step, and no data is updated on Rx_Data_Frame array.

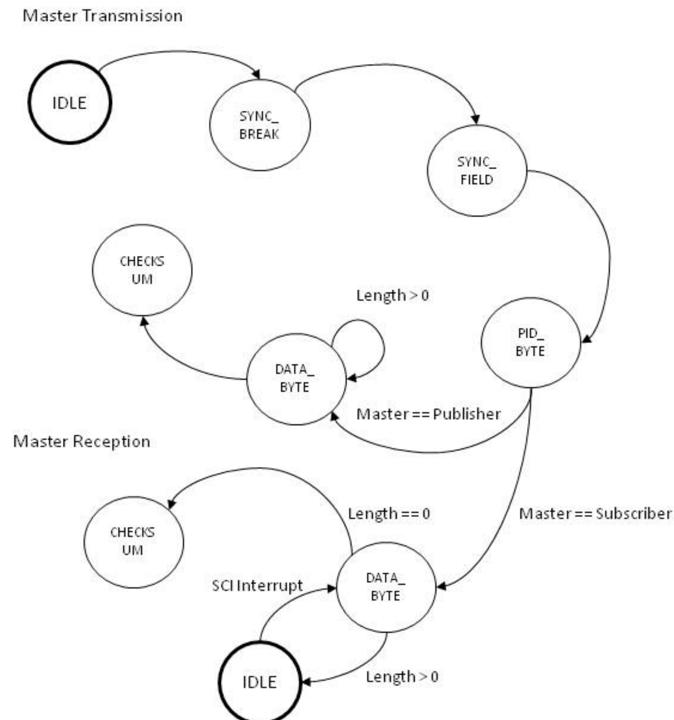

Figure 2: Master node machine state representation

3.4.2 Slave Node

When the node is configured as Slave, the state machine function (vfnLIN_Frame_Transmit) is accessed only by the callback function until PID_BYTE data is validated. In the low level initialization (vfnLIN_Init) the reception interrupt is disabled (RIE) and the Sync Break detection feature is enabled (BKDFE and BKDIE [3]). This action will ensure that the only interruption associated to the SCI is Sync Break detection; therefore initial state is preserved until this event.

Once the break is detected, the SCI is configured to be interrupted by byte reception (RIE) until PID_BYTE data is validated; the PID_BYTE is compared against configured PID frames (u8frame_id), if received value matches a configured frame a mode, Publisher or Subscriber, is selected (vfnLIN_Mode_Select). If the received PID_BYTE does not match with any configured PID frame (u8frame_id), state machine is sent to Sync Break state (see Fig. 3).

Once a mode has been selected (vfnLIN_Mode_Select), the state machine behavior is the same as the Master.

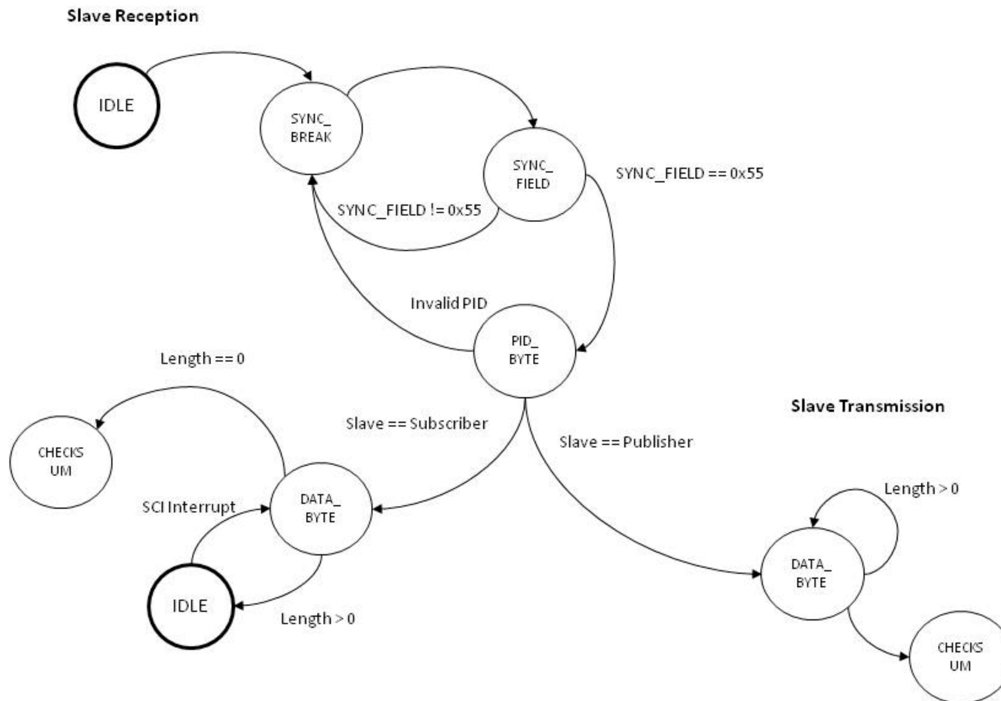

Figure 3: Slave node machine state representation

In both modes a timeout mechanism has been implemented to avoid losing data or frames mis-reception. Although, this timeout mechanism is implemented in different way for each node; in both cases the timeout will call the low initialization function from LIN. The nominal timeout value selected in this development was of 5 ms according to the resolution value of schedule table.

When driver is configured as Master timeout mechanism is easier, given that, if the frame was not transmitted in its totality in the space between scheduled frames, so it will invalidate received information and restart state machine.

On the other hand, if driver is configured as Slave, a timeout timer will be started after receiving Sync Break byte and when a 5 ms has occurred and the frame has not been transmitted, a low level initialization is performed as well disposing all information from this frame.

Data became available to internal transmit buffer (ptrTx_Buffer) once the PID has been decided and operation mode has been selected. Data from signals are arranged according to its lengths and offsets, and copied into transmission buffer. This buffer is used to send data byte in data field, and received by subscriber.

This signals arrangement is performed by vfnLIN_Tx_Frame_Handler function, which uses configuration structure to access to signals configuration according to frame ID to be transmitted.

3.4.3 State Machine implementation

As described above, the core logic of implemented driver resides in the state machine. The state machine implementation depends directly in a compiler switch (LIN_CONFIGURED_DEVICE) used to determinate implementation type.

The main difference is in Master Task part of LIN frame transmission, defined by LIN standard. Each state of the implemented state machine is described below.

SYNC_BRK (0x00)

This is the state where Sync Break is expected, either to be transmitted or to be received.
Master node: If LIN node is configured as Master, this node is responsible of transmitting Sync Break. So to start frame transmission, SCI registers is configured to transmit Sync Break byte. This is performed by writing in SBK (Sync Break) register control register 2 (CR2) [4].
Slave node: If node is configured as Slave, it will wait until a Sync Break byte is received. Therefore registers AMAP (registers map), BKDFE (Break Field Enable) and BKDIE (Break Interrupt Enable) should be configured accordingly, and RIE (Reception Interrupt Enable) will be disabled, so only Sync Break reception is detected. After the Sync Break reception RIE is enabled again.

SYNC_FIELD (0x01)

SYNC_FIELD is the state where sync byte (0x55) is expected.
In this state Master node is responsible of transmitting Sync byte value (0x55). Once transmission has been completed, state machine moves to next state (PID_BYTE) (see Fig. 2).
As Slave node concerns, it expect Sync byte reception interrupt from SCI module, if interrupted, it will validates Sync byte value (0x55). If received value is correct, state machine will moved to next state (PID_BYTE), otherwise driver is restarted to initial state (SYNC_BRK) using low level initialization routine (vfnLIN_Init) (see Fig. 3).

PID_BYTE (0x02)

In this state, Protected Identifier is transmitted according to schedule table.
According to schedule table value, Master node transmits PID and moves to the next state. In PID_BYTE state, Master node will select also operation mode (publisher or subscriber according to frame configuration) (see Fig. 2).
Regarding Slave node at time when PID value is received, it searches in frames table the received value. If PID was found, it will select the operation mode and move to next state. If PID is not found, driver is restarted to initial state (SYNC_BRK) and waits for a Sync Break interruption (see Fig. 3).

DATA_BYTE (0x03)

This is the state where data transmission or reception is handled. This state considers both case and it is handled according to PID's subscriber/publisher mode.
If node's mode is publisher, it will retrieve values of current pointer and allocate them in data buffer (status structure). In each byte transmission, it will perform CheckSum operation, describes by LIN protocol as add with carry of received bytes.

Publisher will write data to SCI transmission register, then moves pointer to next data byte in buffer, and decrease length value. When length value reaches 0, it inverts CheckSum value and moves to next state.
If node's mode is subscriber, data reception is handled by callback interruption. At interruption, it will read received byte from SCI register and holds it in the internal buffer (status structure). The subscriber will calculate an internal CheckSum value; as described by LIN protocol it performs an addition with carry of received bytes.
At each data reception, subscriber will increase buffer pointer to the next byte location and decrease length, according to expected length from frame configuration. When length value reaches 0, the pointer's buffer is moved to start of buffer, internal calculated CheckSum value is inverted and state machine moved to next state (see Fig. 2 and 3).

CHECKSUM (0x04)

In this state the CheckSum validation is performed. To explain this task, it is divided by publisher and subscriber mode:
In Publisher mode, it will transmit CheckSum value using SCI, and then state machine is restarted to initial state (SYNC_BRK) using low level initialization routine (vfnLIN_Init).
In Subscriber mode at SCI reception interruption, CheckSum value is readout from SCI register and validated against internally calculated value. If CheckSum values matches, internal buffer, from status structure, is copied to global Rx_Data_Frame variable. Otherwise, no copy operation is performed. Afterward, the state machine is restarted to initial state (SYNC_BRK) using low level initialization routine (vfnLIN_Init) (see Fig. 2 and 3).

EOF (0x05)

This state (EOF (0x05)) was not implemented in the present work.

4. Driver testing

The way the system was tested was by loading the same code in 2 different DEMO9S12XEP100 boards (see Fig. 4). As indicated above, one board was configured as Master (macro configuration as MST) node and other one as Slave node (macro configuration as SLV).

Figure 4: Used set up of LIN protocol communication tests: a Laptop, two DEMO9S12XEP100 boards, a P&E USB-Multilink-Interface debugger and a USB to Serial adapter.

Physical connections of modules were made and the oscilloscope as snipper to transmission signals was enabled to validate frame composition and Slave/Master node interaction.
First part of testing consist in verifying frame header transmission according to expected length, in order to ensure that either Master node send it and Slave node received. In this stage, Master's and Slave's state machine reacts to a frame header reception, but no data transmission is performed (see Fig. 5).

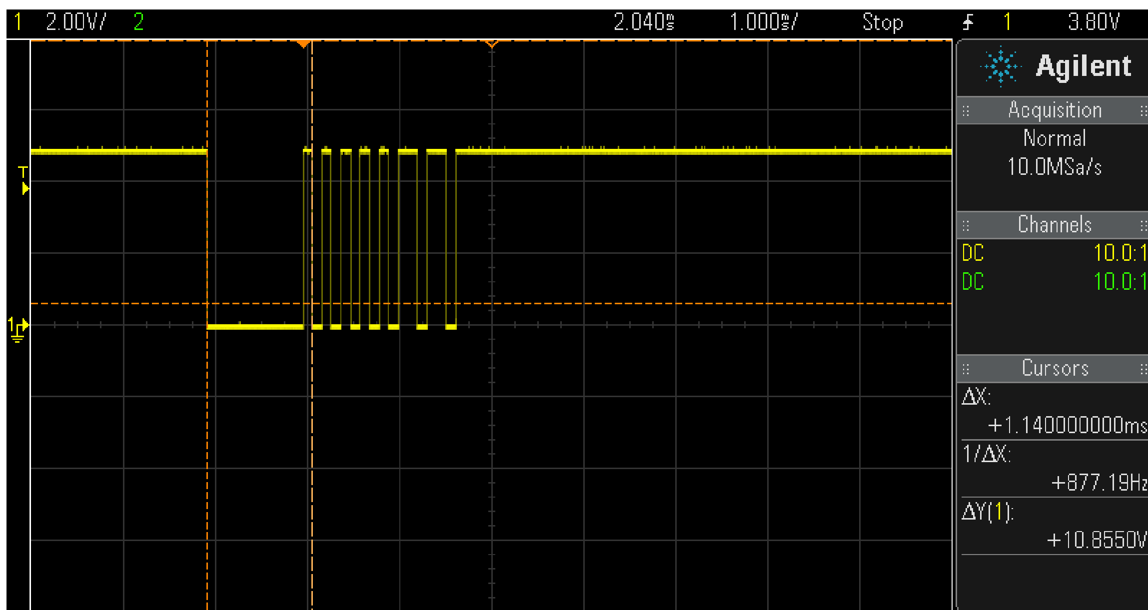

Figure 5: Master node transmits frame header, but no data transmission is performed.

Once frame header transmission is validated, one can says both nodes were tested with entire functionality. Then, Master node starts frame transmission with message header, Master task, and switching between different PIDs values (see Fig. 6). Each node should test PID value and publisher start with the Slave task of the message.

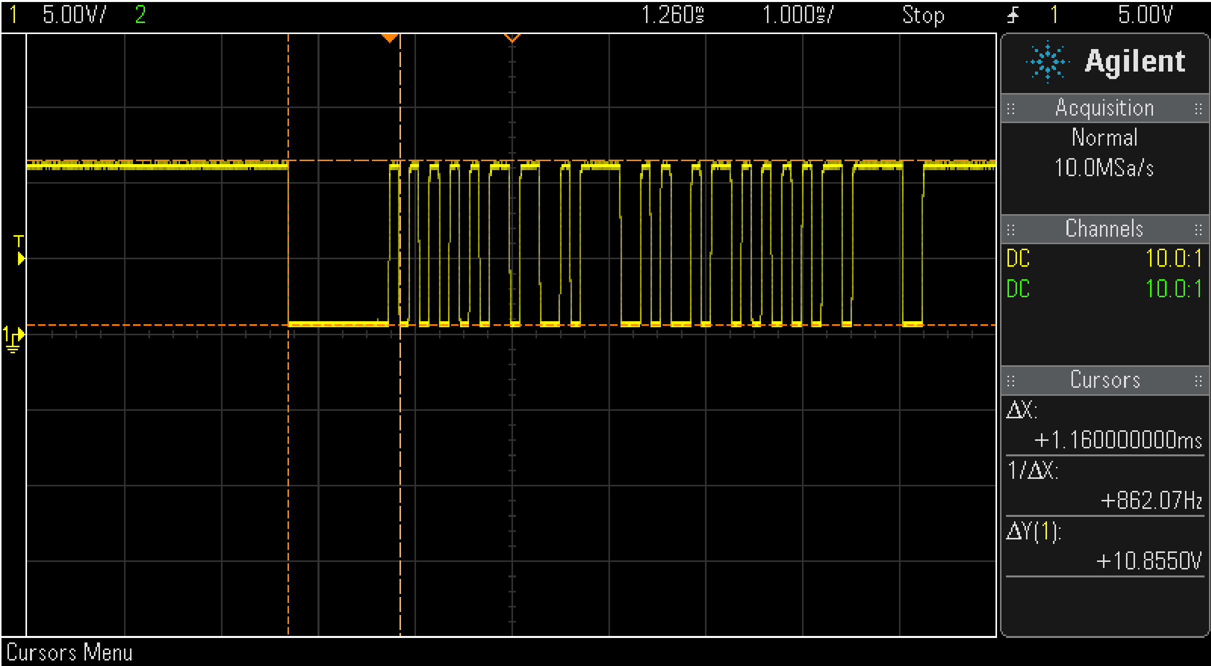

Figure 6: LIN frame transmission, Master node starts frame with header and publisher performs data transmission.

At CheckSum validation all frames were validated and only positive frames updated to global variable. The CheckSum validation is performed by Slave node, which reaches this validation state after frame transmission.

In error validation test mode a “hard-coded” CheckSum was transmitted instead of calculated one. Therefore no data was updated to global variable as CheckSum validation fails and data was disposed.

Schedule table was tested with the help of the oscilloscope; to do this we have frozen 2 frames, then the time between start of Sync Break transmission was measured and compared with the configured one (see Fig. 7).

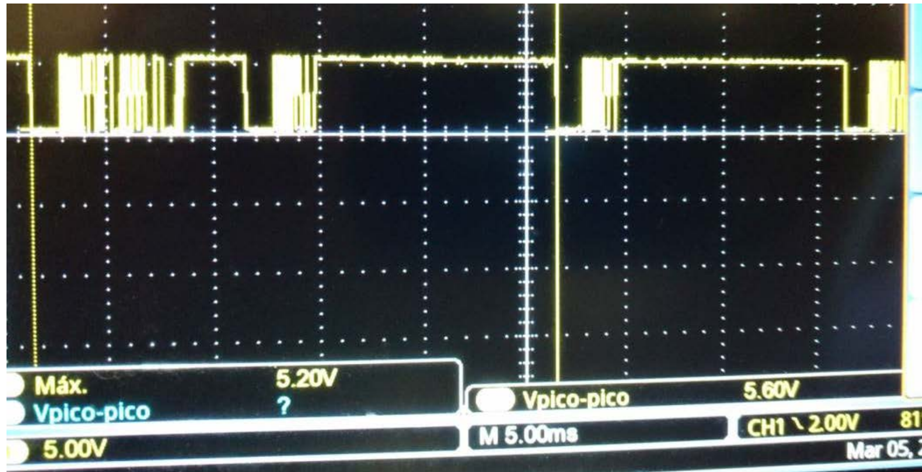

Figure 7: Frame delay measures, this case the delay between frames are 25 ms.

Based on the above presented results, one can said that all performed tests were according to LIN specifications.

5. Conclusions

Given the time constrains, frames validation and composition, LIN protocol is a complex driver to develop with fulfillment of key characteristics of the protocol itself. However, it is a very deterministic protocol that became attractive for low cost implementations in automotive embedded control.

The development of the LIN driver was focused in the usage of the state machines that are manipulated according to the reception data or by de absent of it by the timeout mechanism. Each state from the state machine provides valuable information of driver's functionality and inner logic.

With this LIN driver implementation, a stable communication between the two boards was achieved. In each node we were able to change its operation mode as Publisher/Subscriber, and start transmission data.

Although the implemented code shows reliability and configurability, a better abstraction could be implemented in order to provide a simpler and easier communication. Timeout mechanism is another parameter that could be improved as well; so it can be either a configurable timeout, or attached to LIN protocol timeout values.

As a future work of this project, LIN 2.0 compatibility and configurable feature could be included, which will make a more scalable and re-usable code.

6. Personal reflexions

Despite the previous knowledge and experience obtained in previous projects and driver developments, the LIN driver proved to be a challenge in different manners. The first challenge was to implement configurable and scalable driver that should include frame configurations, despite the result being far from perfect.

Another challenge that was faced is the development of a driver that could serve both purposes, master and slave, in which it was decided to implement something new from developers knowledge, compiler switches. Granting the developed driver the ability to serve different purposes in other projects. This technique is well known in current job field, so this way same developed code can be reused in multiple cases by a proper configuration.

Without a doubt while developing this driver it was learn the real usage and functionality of several techniques taught to us during the course, in this case configuration and status structures, that are part of the core development to create reconfigurable drivers. Also this kind of techniques helps to code re-usability and code abstraction itself.

Also, as stated before, the usage of state machines in communication protocols, which compiles and are a core part of communication protocols implementation. This helps to distinguish communication mechanism and the perform data segregation where multiple nodes relays in the same bus.

Reference

[1] LIN Specification Package Revision 1.3, (Dec. 12, 2002), [Online] Available: https://www.cs-group.de/fileadmin/media/.../LIN_Specification_Package_2.2A.pdf

[2] ISO 17987-6:2016(en) Road vehicles — Local Interconnect Network (LIN) — Part 6: Protocol conformance test specification. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso:17987:-6:ed-1:v1:en

[3] W. Elmenreich and S. V. Krywult (2004, March) A Comparison of Fieldbus Protocols: LIN 1.3, LIN 2.0, and TTP/A. [Online] Available: https://pdfs.semanticscholar.org/7fb5/59fd071bdda4aede5d514fe2e250c6099ab9.pdf

[4] Freescale, MC9S12XEP100 Reference Manual Covers MC9S12XE Family, MC9S12XEP100RMV1 Rev. 1.25 (February 2013) pp. 723-760.

Appendices

Signals

Test signal used in implementation follows below structure, this structure describes signals configuration, size, offset and publisher of such signal. This information is used to create configuration files for each node.

```
Signals
{
        [<signal_name>:<size>,<init_value>,<publisher>[,<subscriber>];]
}
```

Following such structure the used test signals in this implementation were:

```
Signals
{
        RearFogLampInd:1,0,MST,SLV;
        PositionLampInd:1,0,MST,SLV;
        FrontFogLampInd:1,0,MST,SLV;
        IgnitionKeyPos:3,0,MST,SLV;
        SLVFuncIllum:4,0,MST,SLV;
        SLVSymbolIllum:4,0,MST,SLV;
        SLVSWPartNo:8,0,SLV,MST;
        SLVHWPartNoB0:12,0,SLV,MST;
        SLVHWPartNoB1:12,0,SLV,MST;
        FanIdealSpeed:16,0, SLV,MST;
        FanMeasSpeed:16,0, SLV,MST;;
        WaterTemp:16,0,SLV,MST;;
}
```

Frames

Frames composition relays above signals. These test frames structure describes the frame itself and which signals should be included. Also, they are assigned with a PID and a publisher node. These frames structure will be read at run time to determinate which node should act as a publisher when a given PID is transmitted.

```
Frames
{
        [<frame_name>:<frame_id>,<published_by>(,<frame_size>)
        {
                [<signal_name>,<signal_offset>;]
        }]
}
```

Used test frames configuration were the following:

```
Frames {
        Frame1:19,MST
        {
                RearFogLampInd,0;
```

```
                PositionLampInd,1;
                FrontFogLampInd,2;
                IgnitionKeyPos,3;
                SLVFuncIllum,8;
                SLVSymbolIllum,12;
        }
        Frame2:34,SLV
        {
                SLVSWPartNo,0;
                SLVHWPartNoB0,8;
                SLVHWPartNoB1,24;
        }
        Frame3:59,SLV,6
        {
                FanIdealSpeed,0;
                FanMeasSpeed,16;
                WaterTemp,32;
        }
}
```

Schedule Tables

Schedule table determinates the periodicity of each frame to be transmitted. Also, it is describe the delay between frames and the order of transmission. Schedule table follows below structure:

```
Schedule_tables
{
        [<schedule_table_name>
        {
                [<frame_name> delay <frame_time> ms ;]
        }]
}
```

Used test frames were schedule within a range of 40 ms, only 3 frames are transmitted and publisher and subscriber are described by frame itself. The schedule table just determinate the time between frame transmissions.

```
Schedule_tables
{
        SCH_TBL1
        {
                Frame1 delay 15 ms;
                Frame2 delay 25 ms;
                Frame3 delay 40 ms;
        }
}
```