**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

# MAQUINA DE ESTADOS EN EL MICROCONTROLADOR S12X PARA UN PROTOCOLO LIN CONTROLADO POR EVENTOS

Trabajo recepcional que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Josymar Itzae Guzmán Mercado

Asesor: Raúl Campos Rodríguez

Tlaquepaque, Jalisco. 27 de octubre de 2016.

# 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

# MAQUINA DE ESTADOS EN EL MICROCONTROLADOR S12X PARA UN PROTOCOLO LIN CONTROLADO POR EVENTOS

Trabajo recepcional que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Josymar Itzae Guzmán Mercado
Becario CONACYT No. 424506

Asesor: Raúl Campos Rodríguez

Tlaquepaque, Jalisco. 27 de octubre de 2016.

# 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

# EVENT DRIVEN LIN PROTOCOL STATE MACHINE USING S12X MICROCONTROLLER.

Final work that to obtain the diploma of

EMBEDDED SYSTEM SPECIALIST

Presents: Josymar Itzae Guzmán Mercado
CONACYT Scholarship No. 424506

Advisor: Raúl Campos Rodríguez

Tlaquepaque, Jalisco, octubre de 2016.

# AKNOWLEDGMENTS

First I would like to express my sincere gratitude to my advisor Dr. Raúl Campos for the support of achieving this work done.

I also want to thank to Consejo Nacional de Ciencia y Tecnología (CONCACYT) for promoting scientific and technological research, and for his support by the Scholarship No. 424506.

Specially tanks to ITESO for having the equipment and facilities for every activity developed during the specialty, all teachers and students for their patience, motivation and immense knowledge I exchanged with and enriched my mind.

Finally, I would like thank to Continental Automotive Guadalajara, for the support in the professional growth and also for the valuable support they have offered me.

# RESUMEN

Los sistemas embebidos juegan un papel importante en un gran número de dispositivos de alta tecnología, como los requeridos en la industria automotriz. La creciente necesidad de entregar una solución de ingeniería que se pueda mantener y depurar con facilidad, lleva a hacer diseños tan configurables como sea posible, ya sea tanto en la plataforma de hardware como de software, con un gran énfasis en esta última.

Este trabajo tiene dos propósitos principales: (1) implementar el código fuente del protocolo de comunicación LIN que trata con restricciones en tiempo real y (2) demostrar que el uso de un analizador basado en software de PC proporciona una manera poderosa, rápida y fácil de generar código fuente útil mediante la construcción de archivos de texto que posteriormente pueden ser utilizados por el ingeniero de desarrollo. Las combinaciones de ambas soluciones garantizan una herramienta de ingeniería bien diseñada, configurable y que puede ser mantenida con facilidad.

La primera parte del documento especifica los objetivos, el enfoque propuesto y los antecedentes teóricos de este trabajo. Enseguida, se ofrecen detalles de implementación de la solución incorporada en la tarjeta de desarrollo y la solución de interfaz que se ejecuta en la PC, así como la fase de prueba / validación del proyecto.

Finalmente, se presenta un resumen del trabajo, conclusiones y una reflexión sobre cómo el conocimiento adquirido en este proyecto mejora mi trabajo diario. Algunas referencias bibliográficas útiles para una comprensión adecuada de este trabajo se ofrecen al final del documento, en la sección correspondiente.

# ABSTRACT

Embedded systems play an important role in a great number of high technology devices, such as those required in the automotive industry. The incremental need of delivering a maintainable engineering solutions leads to make configurable designs as much as possible.

This work has two major purposes: (1) to implement LIN source code dealing with real-time constrains and (2) to demonstrate that using a PC parser provides a powerful, faster and easy way to generate useful source code from a text file. Both solutions in combination ensure a well-designed, maintainable and configurable engineering solution.

First part of the document specifies the objectives, proposed approach and theoretical background. It's followed by implementation details of the embedded solution in the board and the interface solution running in the PC, as well the testing/validation project phase, and finally summarizing the entire work conclusion and how engineering knowledge acquired improves my daily work.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF ACRONYMS AND ABBREVIATIONS

| | |
|---|---|
| Cross-compilation | Process whereby compilation is done on a PC (host) with the purpose to be executed on other system, called target (target). When cross-compilation is done, executables for another system are produced. This happens when the target does not have the toolchain native compilation or when the host is faster and has better resources (CPU, memory, etc.). |
| SDK | Software Development Kit. |
| X11 | X Window System. |
| RTOS | Real-Time Operating System |

# 1. INTRODUCTION

**Abstract:** *This chapter briefly presents the background of the object of study, problem definition and justification.*

## 1.1. Automotive Communication Protocols Overview

Government regulations implemented in order to make automobiles environment friendly motivated the development of vehicle network technology, followed by the growth of the semiconductor industry, competition and cost pressures. Electronics played a decisive role not only to achieve reducing emission, but improving customer satisfaction, driving safety and comfort.

For many years the car radio was the unique electronic device in vehicle, until the migration of electronic control units (ECU's) which began at the end of 1970's.

The increasing number of requirements in the different domains of a car production have led to the need of a large number of different automotive networks, such as Controller Area Network (CAN), Local Interconnect Network (LIN), J1850, FlexRay, MOST, and most recently Ethernet, among others.

The most used in-car communication protocols are by far the LIN and the CAN. By the one hand, the LIN protocol works in a single master fashion, with one wire at 12V, which was adopted by the industry in the 1990's. This protocol is able to transfers data up to 20 Kbit/s. On the other hand, the CAN protocol is a multi-master communication procedure, with a typical dual wire at 5V. It was widely adopted by the industry in the second half of the 1990's. This protocol is able to communicate at rates up to 500 Kbit/s.

A common characteristic, sometimes useful o sometimes not, of the majority of the protocols and buses is that each connected ECU shares a single input port and output port. Accordingly, the information does not have a pass through the ECU's. Thus, when an ECU sends any package of information, all other ECU's receive it almost at the same time and in the same order.

Since it is quite probable that time-critical applications would be running in each ECU, then the operation of adding and/or removing nodes is one of the most common activities in an automotive communication protocols. Hence, tools for debugging activities such as measurement, diagnostics and sniffing, are aroused and created a new branch in automotive engineering in the design of developing and support tools.

## 1.2. Objectives

The main objective is to create a complete project which comprises on one hand a LIN implementation logic running in a real time environment. On the other hand, a complementary interface, running in a Windows Operating System PC, capable to receive meaningful LIN protocol related information in a text file and use it to modify the configuration in the board.

The entire project can be then broken down into two core implementations:

- *LIN implementation*. Using MC9S12XEP100 Freescale (now NXP and then Qualcomm) demo board and a predefined scheduling environment, take advantage of microcontroller and peripherals capabilities to create a real-time source code to support LIN protocol requirements.

- *LIN configuration parser*. Create a syntactic analyzer application running on Windows Operative Sytem PC, capable to receive as input a text file containing information describing the LIN configuration and as a result, generate a header and a source file ready to be used in the LIN implementation. The input file (text file) must have the information arranged with predefined rules so the parser can analyze it. The output files must manage all the information provided in the input file without any change.

# 2. BACKGROUNG

***Abstract:*** *This chapter briefly reviews some background concepts and protocols that are relevant in the development of this project*.

All project implementation requires research and knowledge to know requirements and constrains; this is not the exception. In this section it is provided an introduction to LIN protocol, state machines, human machine interface, and syntactic analyzer and interrupt driven development; which are the core topics to achieve the project implementation.

## 2.1. LIN Protocol

The LIN is a SCI/UART-based serial, byte-oriented, low cost and low speed and time triggered communication protocol used for short distance networks.

LIN protocol was developed as cost-effective alternate to CAN protocol, to become the link between the intelligent control module and the remote sensors/actuators such as:

- Vehicle roof (rain sensor, light sensor, light control)
- Doors/windows/seats (mirrors, windows lift, door locking, mirror switch)
- Steering wheel (cruise control, wipers, turning light)
- Seat (occupancy sensor, control panel)

#### 2.1.1.1. Data exchange

As already mentioned, a LIN network comprises of one master node and one or more, up to 12, slave nodes. Only the master node initiates the communication. The master node defines the transmission speed, sends synchronization pulses, monitors data and switches slave nodes to sleep/wake up mode. It also receives the Wakeup Break from slave nodes when the bus is inactive and they request some action.

### 2.1.1.2. LIN Message Frame

The LIN protocol uses frames for the communication. A frame consists of a header, a response and some response space. The master sends out a message header containing a synchronization break, synchronization byte and the message identifier, each part begins with a start bit and ends with a stop bit.

A slave node waits for the synchronization pulse, processes the message identifier and according the ID, it transmits, receive data or do nothing. The response contains one to eight data bytes and one checksum byte.

Once the bus becomes inactive, either the master starts over the communication or the slave sends a weak up break to request the master to do something.

### 2.1.1.3. Header

The LIN protocol is byte oriented; this means the data is sent byte by byte, the LSB first.

Synch Break is the field transmitted by the master to notify the start of the LIN message frame. It must be at least 13 bit periods in duration to allow master-slave synchronization.

Synch Byte is a specific pattern for determination of time base. It precedes any message frame.

The identifier incorporates information about the sender, the receiver, the purpose and the data field length.

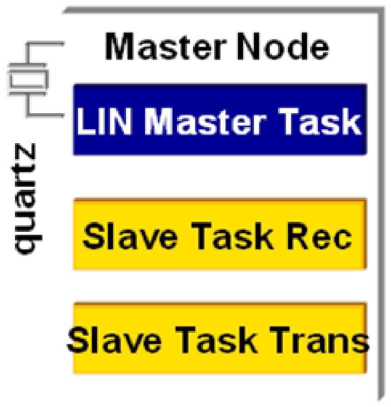


**Figure 1. Master node representation**

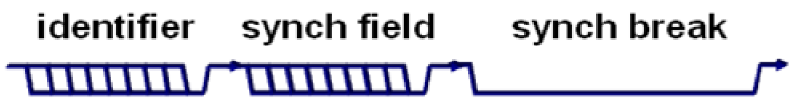


**Figure 2. Master message stream flow.**

2.1.1.4. Response

The response is composed mainly by the data and checksum, the data sent could be 1, 2, 4 or 8 data bytes, the checksum to identify data integrity.

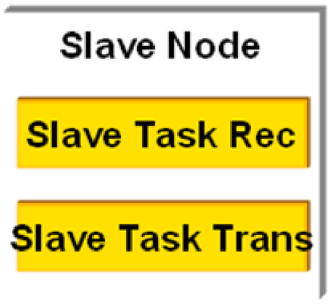


**Figure 3. Slave node representation.**

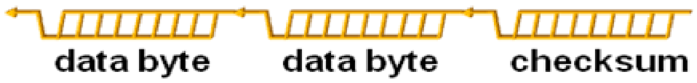


**Figure 4. Slave response message flow.**

### 2.1.1.5. State Machines

A finite state machine is one of the most common design patterns used in the embedded systems world. The main purpose of a state machine is to define the proper transition amount pre-defined states; a state is a condition that captures relevant aspects of the system’s history. Each state transition is event triggered. The event could be due to an interrupt, timer, signal or input from other module in the system.

It is common to find state machine diagrams to have a big picture of the state machine life cycle. A state is regularly represented as a rectangle with rounded corners and the state name inside the rectangle, but using a circle is also accepted. A transition is represented by an arrow leading from one state to another and an event is simply the occurrence of something in the system indicating a change.

One of the simplest and intuitive state machines is that of a puss button. It has two states (Power On and Power Off), two transitions (Power On – Power Off, Power Off – Power On) and one event (Button click). The Figure 5 shows an example of a State Machine capturing the button behavior.

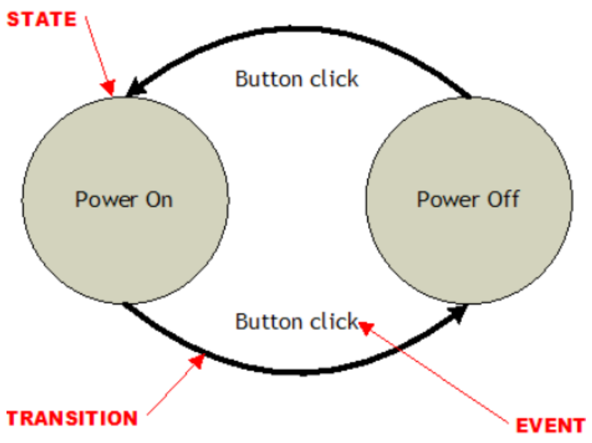


**Figure 5. State Machine example.**

### 2.1.1.6. Human Machine Interface

A Human Machine Interface (HMI) did not start as it is known nowadays. The first approach of a Human Machine Interface was using buttons, lights and switches to monitor machine operations. Since the possibility to replace the buttons with an electronic terminal, in 1990, along with integrated circuit evolution, HMI were considered as a system add-on.

Then, the PC was born, and the HMI grew exponentially; a lot of improvements have been done to make what we know today as displays (late in 1990's), not only for visualization purposes but for creating an interaction mean.

HMI history is huge, and we will focus only on the HMI purpose and definition.

**Figure 6. Old Human Machine Interface with buttons, switches, levels and light indicators.**

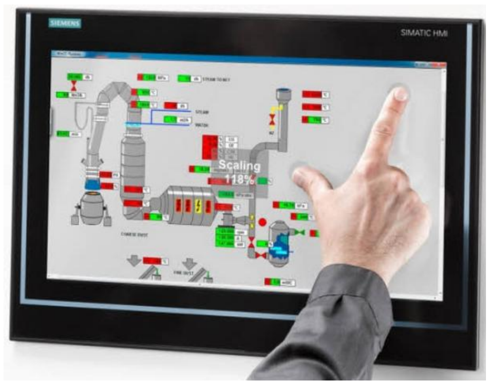


**Figure 7. Nowadays Human Machine Interface with visualization and touch screen.**

2.1.1.7. Syntactic Analyzer

Some applications provide tools to create, modify and verify the syntax of their configuration files, and this tools have a graphical interface. In some applications a more understandable file is consider as input to convert into a complex file used in the system.

Such tools are known as parsers or syntactic analyzer which processes a string of symbols to produce an advanced file usable by the system.

The parsing process consists of two parts: Scanner and Parser. The Scanner analyses the input file to separate the meaningful information into tokens with a predefined pattern. Once the Scanner finishes a stream of tokens are sent to the Parser. The Parser then builds nodes according predefined rules to finally take an action, for example, commit information to a database, send information to a server or, add text to a file.

There are a lot of high level frameworks and programming languages that can help achieving this task (.net, java, LabView/TestStand, python, among others). The Figure 8 shows a conceptual diagram of an analyzer.

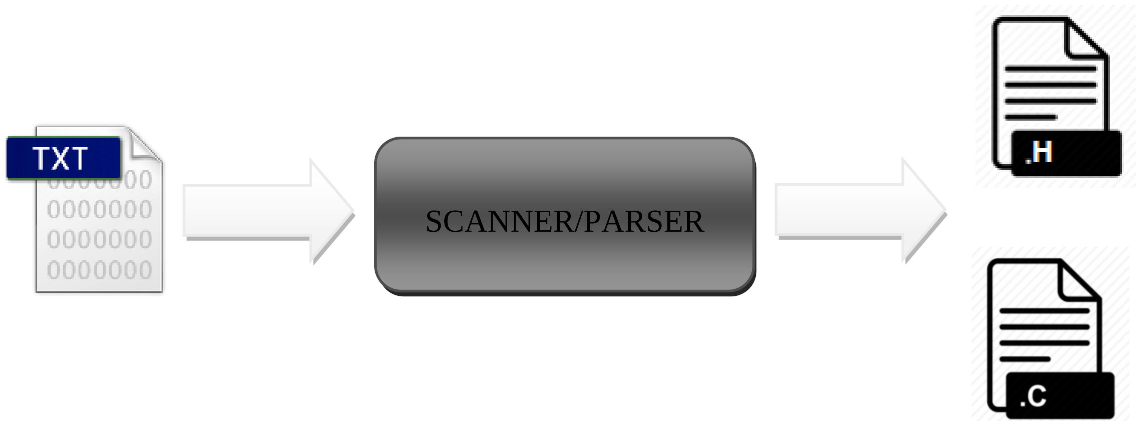


**Figure 8. Syntactic Analyzer Diagram example.**

### 2.1.1.8. Interrupt Driven Development

A rough definition of interrupt is that: interrupt is a mechanism by a microcontroller interrupts the normal processing of the processor and requests the device to execute a specific action. They can be categorized in program, input/output, timer and hardware failure.

- Program: generated as a result of an instruction execution, such as arithmetic overflow or an invalid machine instruction.

- Input/Output: generated by an I/O event, generally hardware related to remark an event.

- Timer: generated by timer expiration in the processor/microcontroller.

- Hardware failure: generated due to a hardware failure.

We will focus mainly in Input/Output and Timer interrupts for this document, since are the base for the software developed.

### 2.1.1.9. Interrupt Service Routine

Interrupts are one of the most powerful and useful features in embedded systems development to make the system more efficient and responsive to critical events or time constrain actions.

When an interrupt is generated, the microcontroller jumps to the Interrupt Handler or Interrupt Service Routine to execute a dedicated code the designer/developer has prepared for the interrupt source. For every interrupt, there is a fixed location in memory that holds the address of its interrupt service routine. Once the interrupt is attended, the microcontroller returns from the interrupt and it resumes the normal code execution as interruption has never occurred.

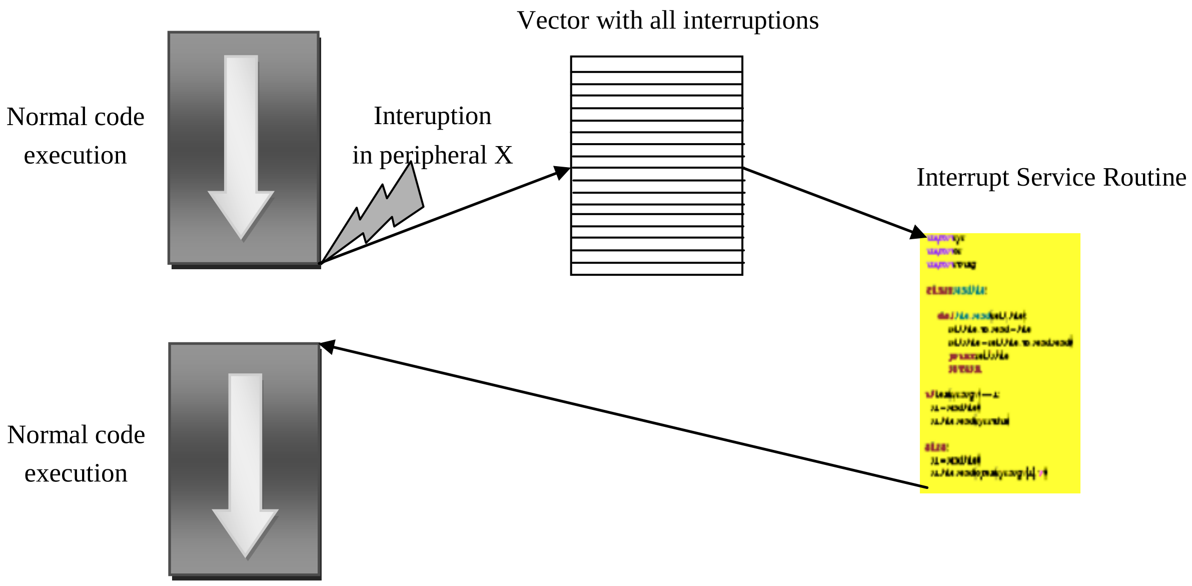

Vector with all interruptions
Normal code execution
Interuption in peripheral X
Interrupt Service Routine
Normal code execution

**Figure 9. Interrupt Service Routine sequence.**

# 3. DESIGN AND IMPLEMENTATION

***Abstract:*** *This chapter details some of the most relevant aspects of the design and implementation of this work.*

## 3.1. Proposed approach

#### 3.1.1.1. LIN Implementation

All automotive communication protocols require a quick response to acquire incoming data and response; hence, interrupt scheme in microcontrollers is suitable for this task, besides a real-time scheduling is available to achieve this.

Once information is available in the microcontroller, the next step is to arrange it properly according the LIN protocol. In order to know what information is already processed, completed, remaining and what action is coming next, a state machine is proposed to know current process state and transition to take action if needed.

#### 3.1.1.2. LIN Configuration Parser

Most of enterprise computers run on Windows Operating System. Most of the engineering tools as well run one version of the Windows Operating System. Considering this, either a Windows framework, .NET framework for example, or a multiplatform language, such as Java, can be selected. Due previous experience working with Window framework, .NET is proposed to build the parser application.

## 3.2. Implementations description

This work focuses mainly in solving two activities, its design and related problems and challenges. These activities are briefly described as follows:

1. Implementing a time-constraint solution to handle LIN protocol messages;

2. To provide the user the capability to create a LIN configuration easily.

An ISR, the involved state machines and configuration files compound the solution to accomplish the point number 1. These elements are fully described later in this chapter of the document.

Syntactic Analyzer tool is the solution to accomplish the second point, which is as well described this chapter of this document.

The combination of these two solutions provides a real-world industry implementation solution.

#### 3.2.1.1. LIN Implementation

#### 3.2.1.2. Interrupt Driven State Machine

For implementing the state machine, this work used a conditional statements approach. This approach is quite simple, it consists of switch-case statement containing each predefined state, an evaluation is made to determine the proper time to transition or not to another state.

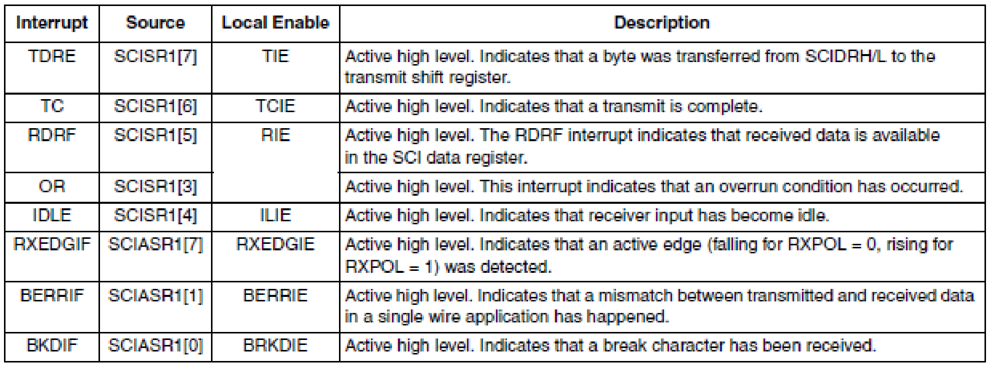
Master and Slave behavior were handled by using state machines. This work defined 4 states machines, depending on the selection of the node behavior. One is dedicated for the transmission as master, one is dedicated for reception as Master, one dedicated for transmission as Slave and finally one dedicated for reception as Slave.

To make package transmission and reception faster, easier and better, this work decided to use an interrupt event driven mechanism.

This event driven mechanism is ISR (interrupt Service Routines). The microcontroller provides ISR which are external events the microcontroller. Each interrupt has a unique ID which can be used in the software to refer to certain interrupt. In this way, interrupts allow a rapid response to time-critical events often occur in measurement and control applications (also in automotive industry).

Since the LIN protocol is SCI based protocol, this work has made use of the SCI block. The S12X microcontroller data sheet provides a detailed description of the SCI related registers and considerations for its operation.

The Figure 10 depicts the registers related to the SCI interrupt module in the S12X microcontroller.



| Interrupt | Source | Local Enable | Description |
|---|---|---|---|
| TDRE | SCISR1[7] | TIE | Active high level. Indicates that a byte was transferred from SCIDRH/L to the transmit shift register. |
| TC | SCISR1[6] | TCIE | Active high level. Indicates that a transmit is complete. |
| RDRF | SCISR1[5] | RIE | Active high level. The RDRF interrupt indicates that received data is available in the SCI data register. |
| OR | SCISR1[3] | | Active high level. This interrupt indicates that an overrun condition has occurred. |
| IDLE | SCISR1[4] | ILIE | Active high level. Indicates that receiver input has become idle. |
| RXEDGIF | SCIASR1[7] | RXEDGIE | Active high level. Indicates that an active edge (falling for RXPOL = 0, rising for RXPOL = 1) was detected. |
| BERRIF | SCIASR1[1] | BERRIE | Active high level. Indicates that a mismatch between transmitted and received data in a single wire application has happened. |
| BKDIF | SCIASR1[0] | BRKDIE | Active high level. Indicates that a break character has been received. |

**Figure 10. S12X microcontroller SCI interruption registers.**

### 3.2.1.3. Master Task

The Master task is the one in charge of handling the communication flow, once the task starts running the first state is IDLE, which means the task is ready to start the information transmission, it transitions to SendSynchBreak state, here the task fulfill a byte array to be loaded in an SCI buffer and be transmitted, until the last byte is transmitted it then transitions to SendSynchField state, in similar way as the previous state, it arranges a byte array with 0X55 value, once the last byte is transmitted, the task starts preparing the PID; if the publisher is the master it proceeds to send data, if it’s a slave, the master waits for receiving the response from the corresponding slave.

The Figure 11 details the state machine for the master node task. The machine is composed of seven states and ten transitions.

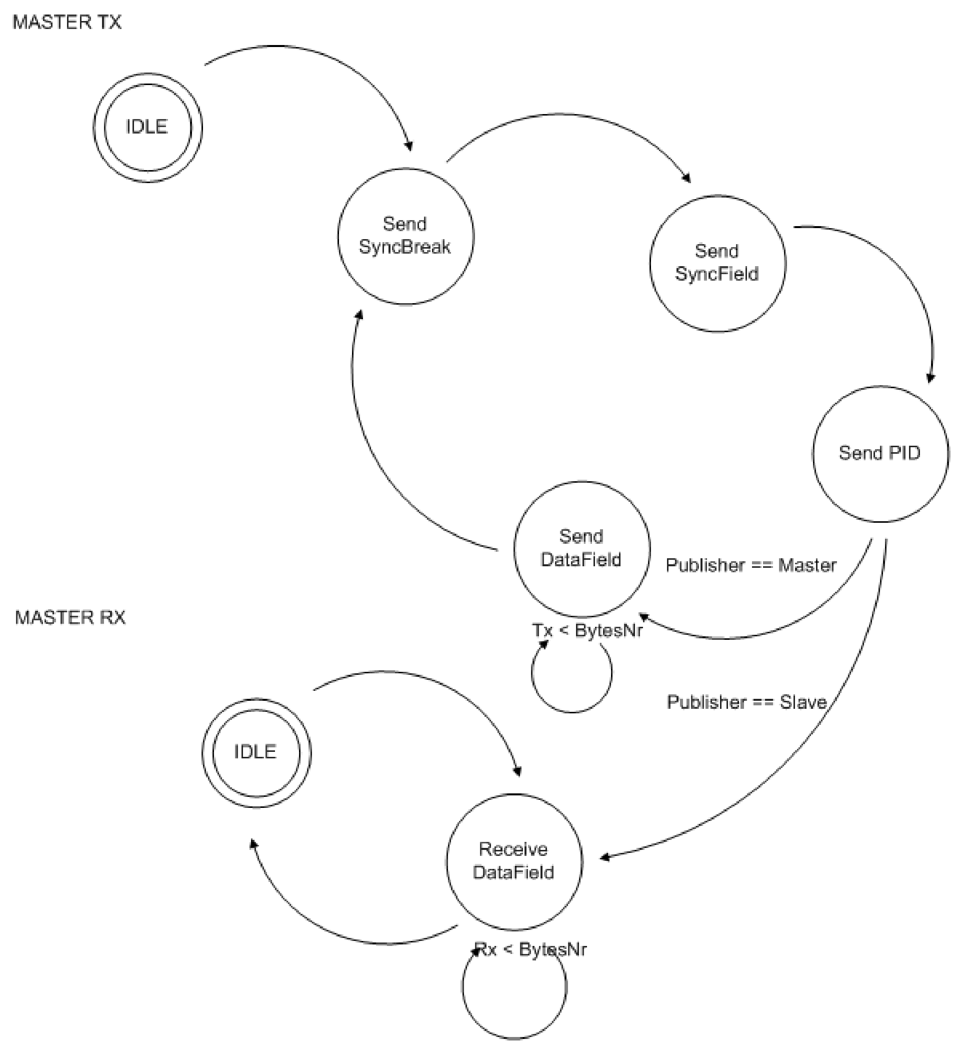


**Figure 11. State machine chart for Master node task.**

### 3.2.1.4. Slave Task

Following the same concept as the Master task, once it is detected the SCI buffer has information the Slave task verifies it is the SynchBreak and then wait for the SynchField reception, if there are no errors the state machine transitions to Receive PID state, depending on the PID, the slave task evaluates to either receiving data or transmit data.

The Figure 12 depicts the state machine of the slave task. It is composed of seven states and nine transitions. This machine includes two initial states, one for the reception of a message and one for the transmission of a message.

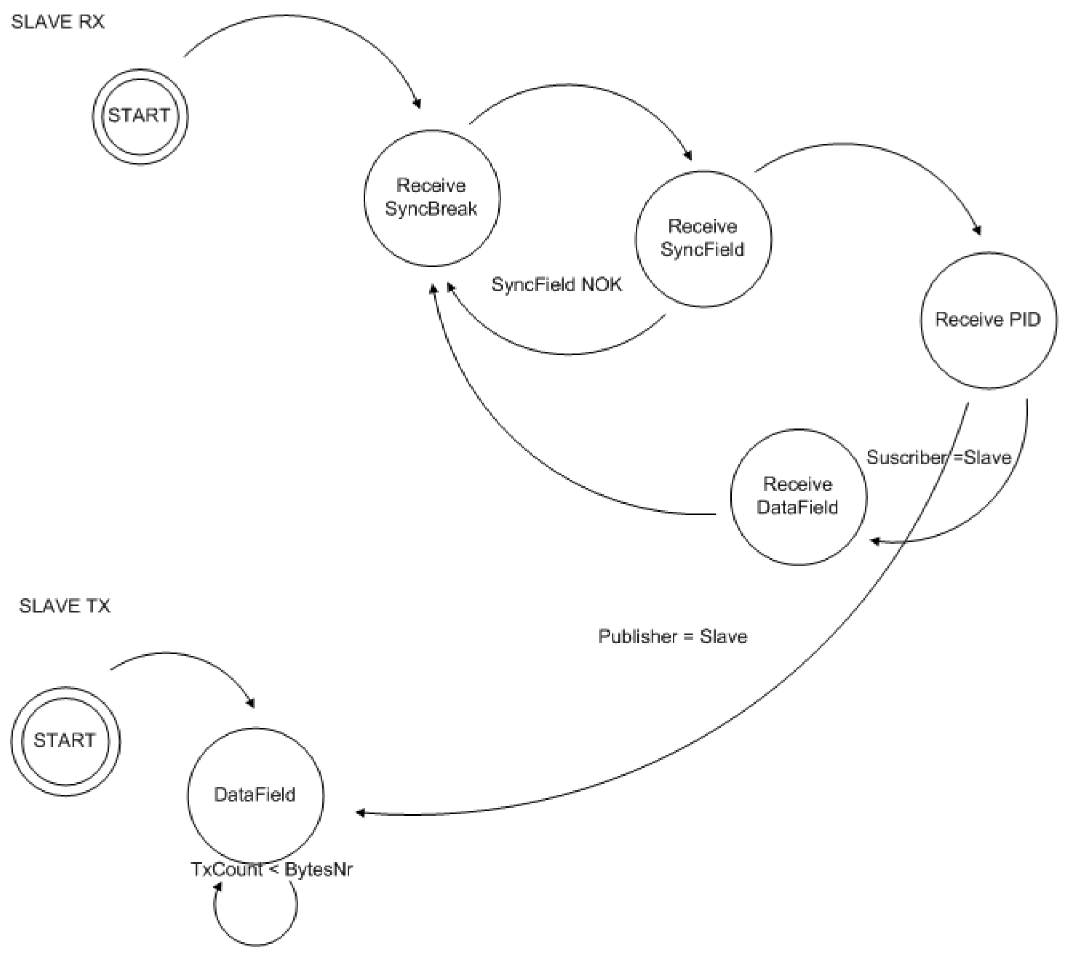


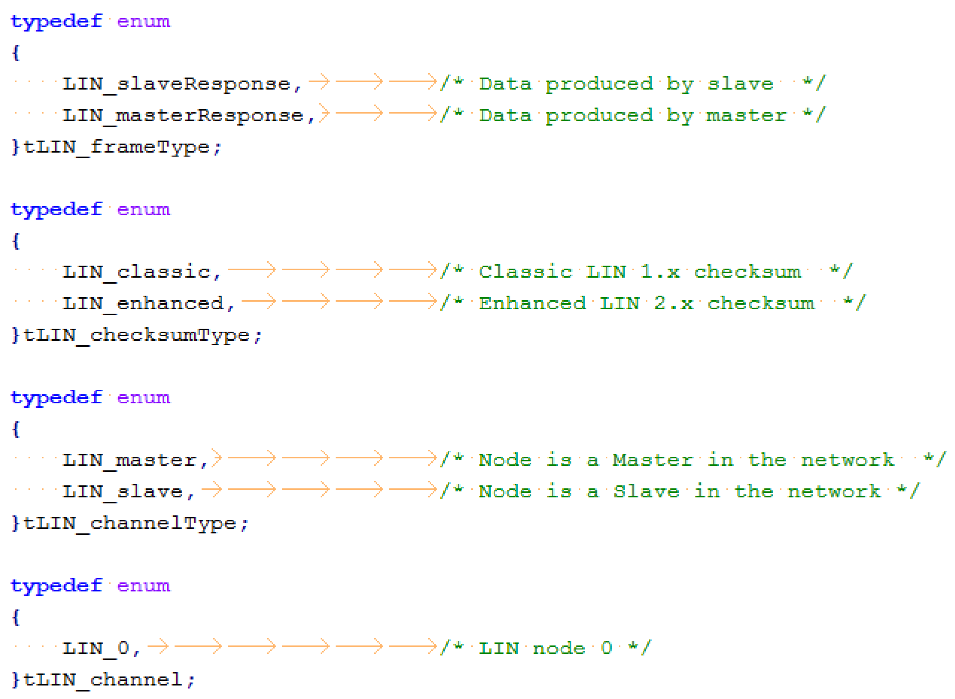
**Figure 12. State machine chart for Slave node taks.**

### 3.2.1.5. LIN Configuration File

In computing area and embedded systems as well the configuration files are intended to set parameter or initial setting an application will use.

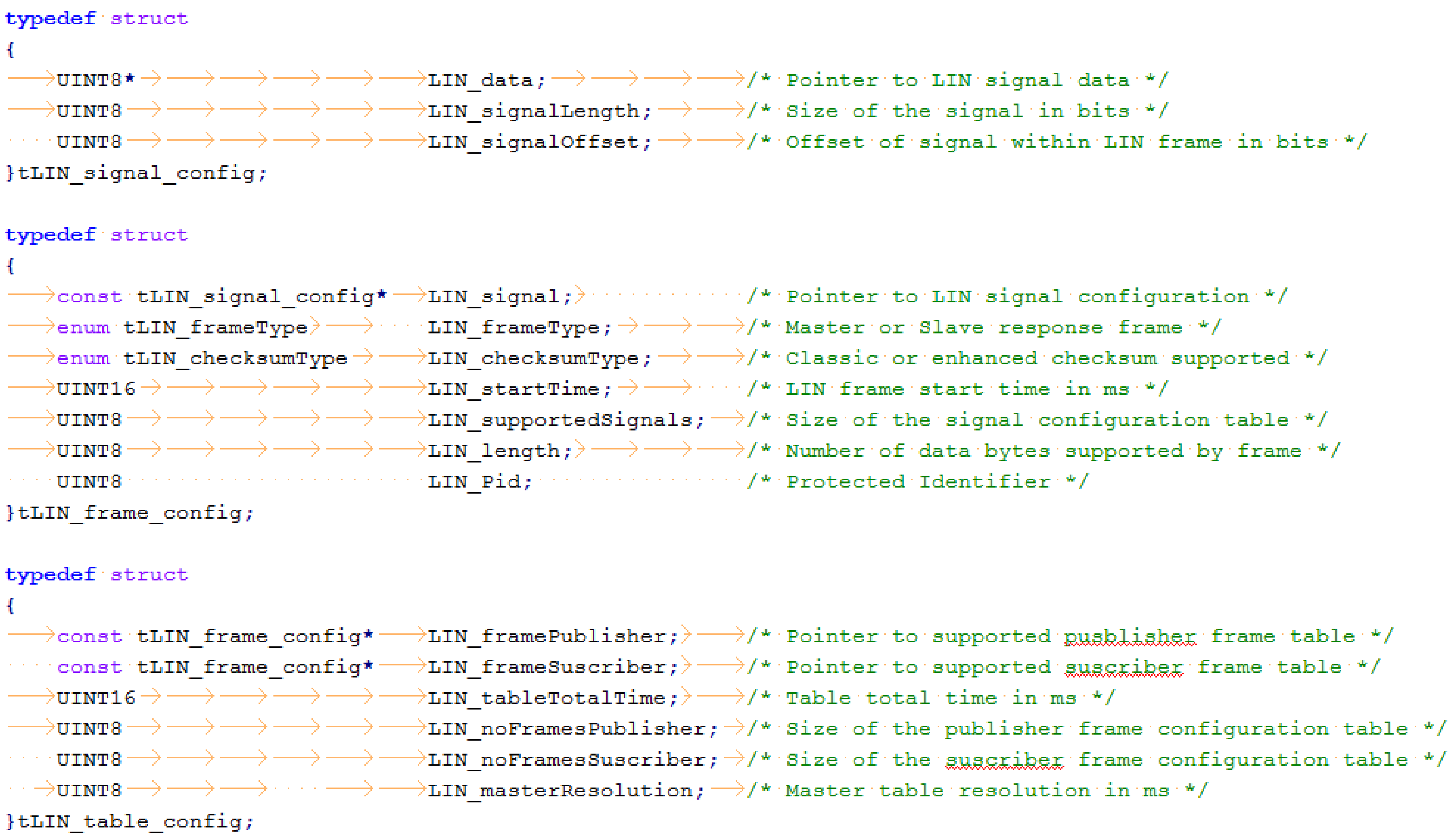
In this case the configuration file resides in the pair header-source file (.h and .c data type files). The header file (.h) includes only data types created to make easier handling the frames transmission-reception used by the application, the data type definitions is based in enumerations and structures.

The Figure 13 shows the type definition for the enumeration required by the implementation.



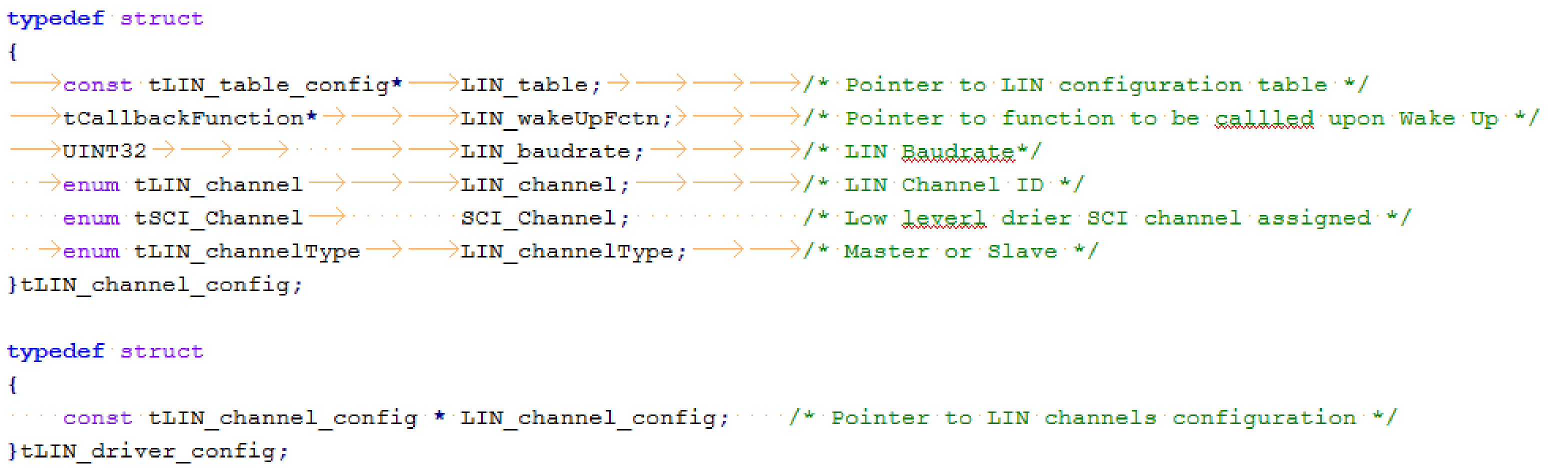
```c
typedef enum
{
    LIN_slaveResponse,              /* Data produced by slave  */
    LIN_masterResponse,             /* Data produced by master */
}tLIN_frameType;

typedef enum
{
    LIN_classic,                    /* Classic LIN 1.x checksum  */
    LIN_enhanced,                   /* Enhanced LIN 2.x checksum  */
}tLIN_checksumType;

typedef enum
{
    LIN_master,                     /* Node is a Master in the network  */
    LIN_slave,                      /* Node is a Slave in the network */
}tLIN_channelType;

typedef enum
{
    LIN_0,                          /* LIN node 0 */
}tLIN_channel;
```

**Figure 13. Type definition for enumerations.**



```c
typedef struct
{
    UINT8*                  LIN_data;               /* Pointer to LIN signal data */
    UINT8                   LIN_signalLength;       /* Size of the signal in bits */
    UINT8                   LIN_signalOffset;       /* Offset of signal within LIN frame in bits */
}tLIN_signal_config;

typedef struct
{
    const tLIN_signal_config*   LIN_signal;             /* Pointer to LIN signal configuration */
    enum tLIN_frameType         LIN_frameType;          /* Master or Slave response frame */
    enum tLIN_checksumType      LIN_checksumType;       /* Classic or enhanced checksum supported */
    UINT16                      LIN_startTime;          /* LIN frame start time in ms */
    UINT8                       LIN_supportedSignals;   /* Size of the signal configuration table */
    UINT8                       LIN_length;             /* Number of data bytes supported by frame */
    UINT8                       LIN_Pid;                /* Protected Identifier */
}tLIN_frame_config;

typedef struct
{
    const tLIN_frame_config*    LIN_framePublisher;     /* Pointer to supported pusblisher frame table */
    const tLIN_frame_config*    LIN_frameSuscriber;     /* Pointer to supported suscriber frame table */
    UINT16                      LIN_tableTotalTime;     /* Table total time in ms */
    UINT8                       LIN_noFramesPublisher;  /* Size of the publisher frame configuration table */
    UINT8                       LIN_noFramesSuscriber;  /* Size of the suscriber frame configuration table */
    UINT8                       LIN_masterResolution;   /* Master table resolution in ms */
}tLIN_table_config;
```

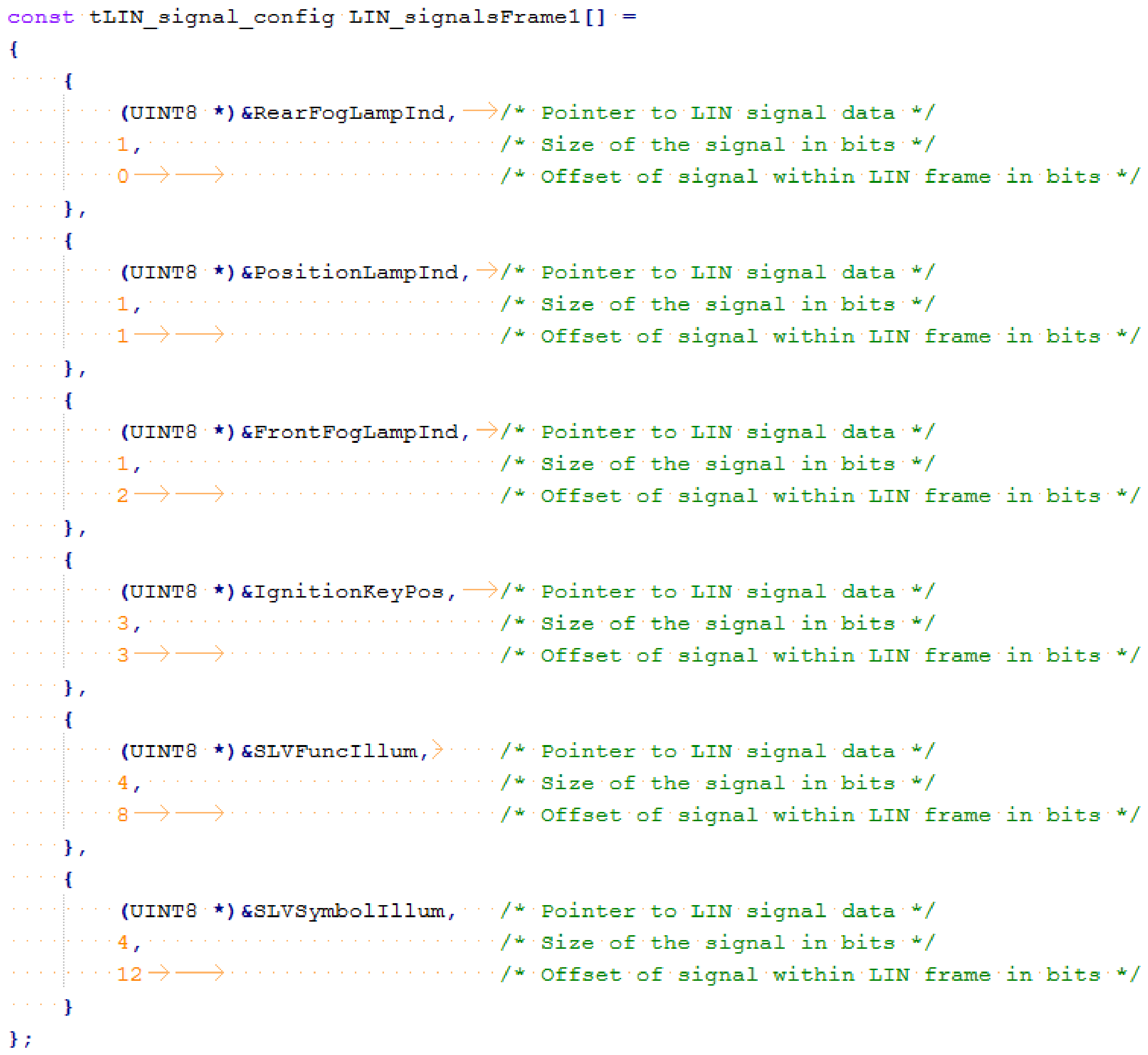
**Figure 14. Type definition for signal, frame and table structures.**



```c
typedef struct
{
    const tLIN_table_config*    LIN_table;              /* Pointer to LIN configuration table */
    tCallbackFunction*          LIN_wakeUpFctn;         /* Pointer to function to be callled upon Wake Up */
    UINT32                      LIN_baudrate;           /* LIN Baudrate*/
    enum tLIN_channel           LIN_channel;            /* LIN Channel ID */
    enum tSCI_Channel           SCI_Channel;            /* Low leverl drier SCI channel assigned */
    enum tLIN_channelType       LIN_channelType;        /* Master or Slave */
}tLIN_channel_config;

typedef struct
{
    const tLIN_channel_config * LIN_channel_config;    /* Pointer to LIN channels configuration */
}tLIN_driver_config;
```

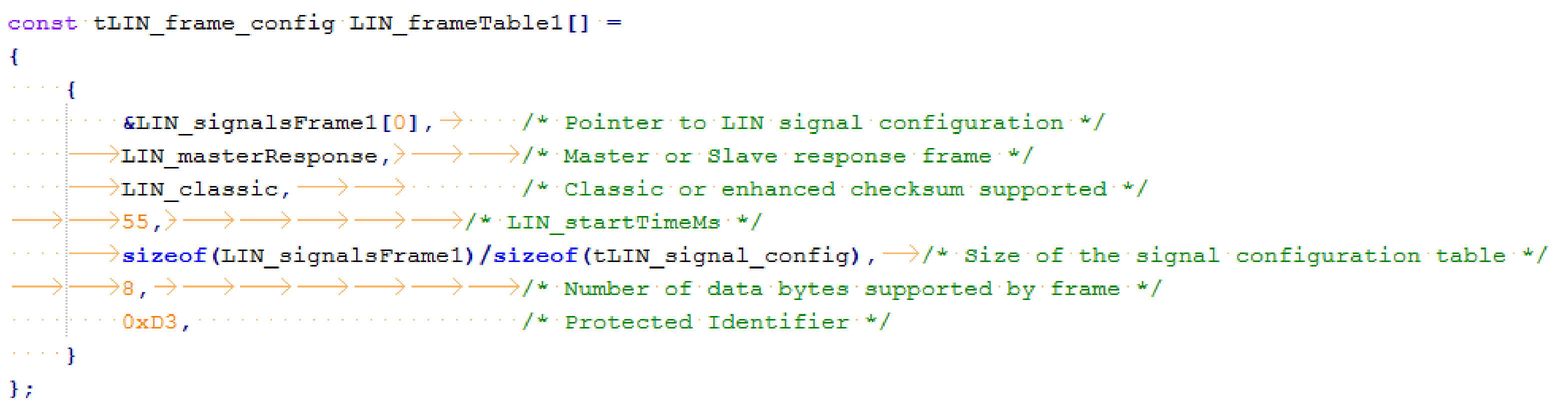
**Figure 15. Type definition for channel and driver structures.**

The source file (.c) uses the predefined data types to construct constant tables, in this way the application is configurable and maintainable, it's easier to modify the "shape" of the frames to adapt them according the application needs.



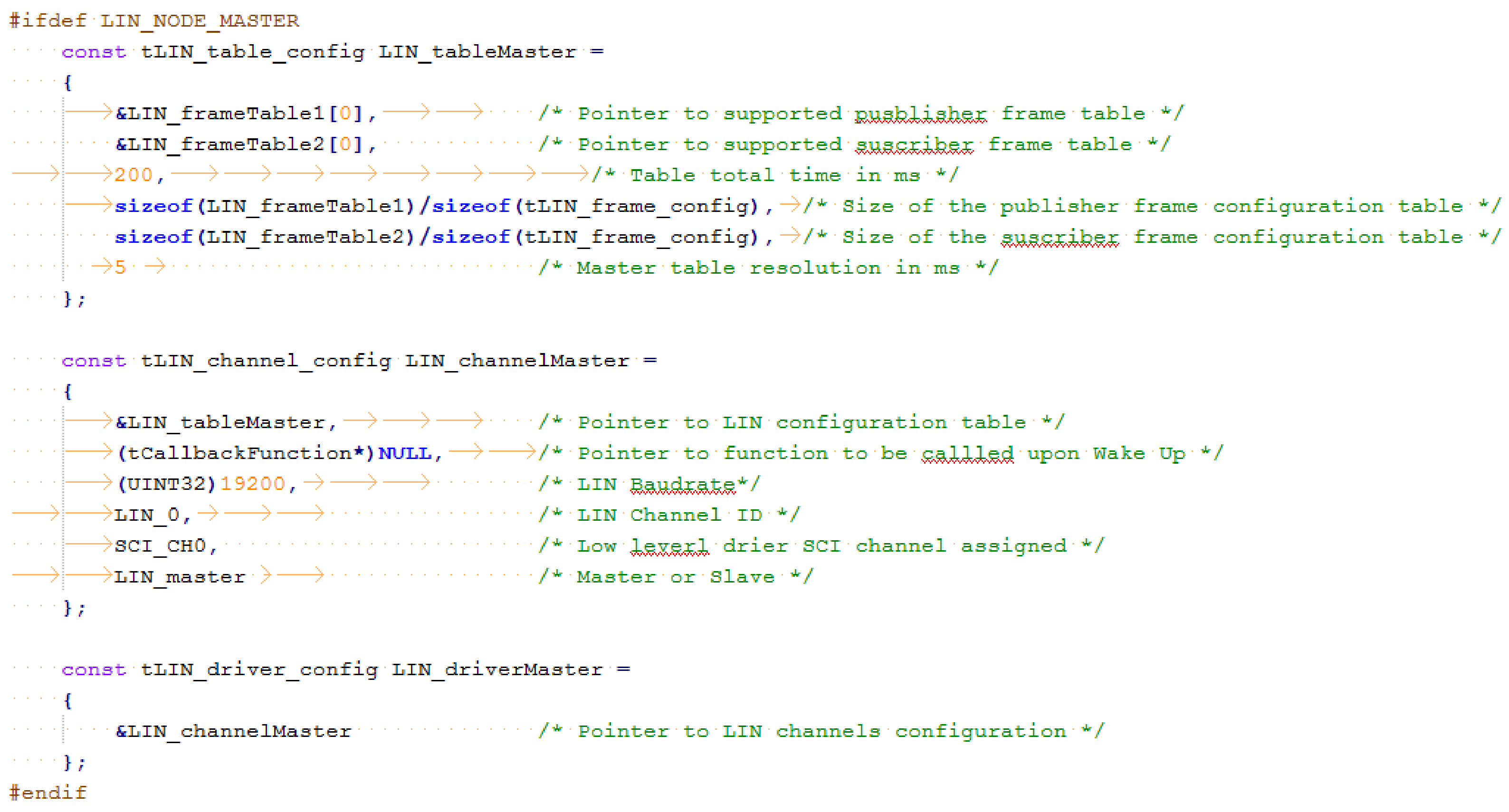
```
const tLIN_signal_config LIN_signalsFrame1[] =
{
    {
        (UINT8 *)&RearFogLampInd,   /* Pointer to LIN signal data */
        1,                          /* Size of the signal in bits */
        0                           /* Offset of signal within LIN frame in bits */
    },
    {
        (UINT8 *)&PositionLampInd,  /* Pointer to LIN signal data */
        1,                          /* Size of the signal in bits */
        1                           /* Offset of signal within LIN frame in bits */
    },
    {
        (UINT8 *)&FrontFogLampInd,  /* Pointer to LIN signal data */
        1,                          /* Size of the signal in bits */
        2                           /* Offset of signal within LIN frame in bits */
    },
    {
        (UINT8 *)&IgnitionKeyPos,   /* Pointer to LIN signal data */
        3,                          /* Size of the signal in bits */
        3                           /* Offset of signal within LIN frame in bits */
    },
    {
        (UINT8 *)&SLVFuncIllum,     /* Pointer to LIN signal data */
        4,                          /* Size of the signal in bits */
        8                           /* Offset of signal within LIN frame in bits */
    },
    {
        (UINT8 *)&SLVSymbolIllum,   /* Pointer to LIN signal data */
        4,                          /* Size of the signal in bits */
        12                          /* Offset of signal within LIN frame in bits */
    }
};
```

**Figure 16. Configuration for signals in frame1.**



```c
const tLIN_frame_config LIN_frameTable1[] =
{
    {
        &LIN_signalsFrame1[0],      /* Pointer to LIN signal configuration */
        LIN_masterResponse,         /* Master or Slave response frame */
        LIN_classic,                /* Classic or enhanced checksum supported */
        55,                     /* LIN_startTimeMs */
        sizeof(LIN_signalsFrame1)/sizeof(tLIN_signal_config),   /* Size of the signal configuration table */
        8,                          /* Number of data bytes supported by frame */
        0xD3,                       /* Protected Identifier */
    }
};
```

Figure 17. Configuration table for frame1.



```c
#ifdef LIN_NODE_MASTER
    const tLIN_table_config LIN_tableMaster =
    {
        &LIN_frameTable1[0],        /* Pointer to supported pusblisher frame table */
        &LIN_frameTable2[0],        /* Pointer to supported suscriber frame table */
        200,                            /* Table total time in ms */
        sizeof(LIN_frameTable1)/sizeof(tLIN_frame_config),  /* Size of the publisher frame configuration table */
        sizeof(LIN_frameTable2)/sizeof(tLIN_frame_config),  /* Size of the suscriber frame configuration table */
        5                           /* Master table resolution in ms */
    };

    const tLIN_channel_config LIN_channelMaster =
    {
        &LIN_tableMaster,           /* Pointer to LIN configuration table */
        (tCallbackFunction*)NULL,   /* Pointer to function to be callled upon Wake Up */
        (UINT32)19200,              /* LIN Baudrate*/
        LIN_0,                      /* LIN Channel ID */
        SCI_CH0,                    /* Low leverl drier SCI channel assigned */
        LIN_master                  /* Master or Slave */
    };

    const tLIN_driver_config LIN_driverMaster =
    {
        &LIN_channelMaster          /* Pointer to LIN channels configuration */
    };
#endif
```

Figure 18. Configuration table for Master node.

When the application is running, it looks for the constant data specified by these two files to work.

#### 3.2.1.6. LIN Configuration Parser

#### 3.2.1.7. Human Machine Interface

HMI stands for Human Machine Interface, as stated in 1.4 Human Machine Interface section, the HMI provides an interface the user can interact with the system, in this case an HMI was developed to be an interface to a parser to provide an easy way the final user can generate configuration files and use them directly in the system without making any change.

Due previous experience using .NET framework and National Instruments tools, in the design of the solution described in this work, it has been decided to use the combination of both environments, LabVIEW and Visual C#. Object Oriented Programming was used, integrating the full functionality of both implementations using .NET Assembly to be called from vi(LabVIEW).

- The development was divided in the following sections:
- Open the configuration file.
- Scan sections.
- Generate configuration objects.
- Use configuration objects to generate .c sections.
- Concatenate sections and generate .c content.
- Save to .c file.

#### 3.2.1.8. Object Oriented Programming Approach

Objects are the key to understand object-oriented programming. Real-world objects are involved in our life: a dog, a desk, a television, a bicycle, etc. Software objects are conceptually similar to real-world objects; they both consist of state and related behavior, an object stores its state in fields (variables) and exposes its behavior through methods (functions). Methods operate on an object’s internal state. For example, a bicycle, its state could be: current speed or current pedal cadence or current gear; and providing methods to change its state, accelerate, break, changeGear, etc.

Object oriented programming provides benefits, such as:

- Modularity. Source code for an object can be written, maintained and replace easily.
- Information hiding. Object’s methods hide the implementation details.
- Code re-use. Existing code can be used in you program/application.

Since the number of elements can change and its information, OOP fits to solve the problem of getting from a text file a configuration with a specific format and information arrangement.

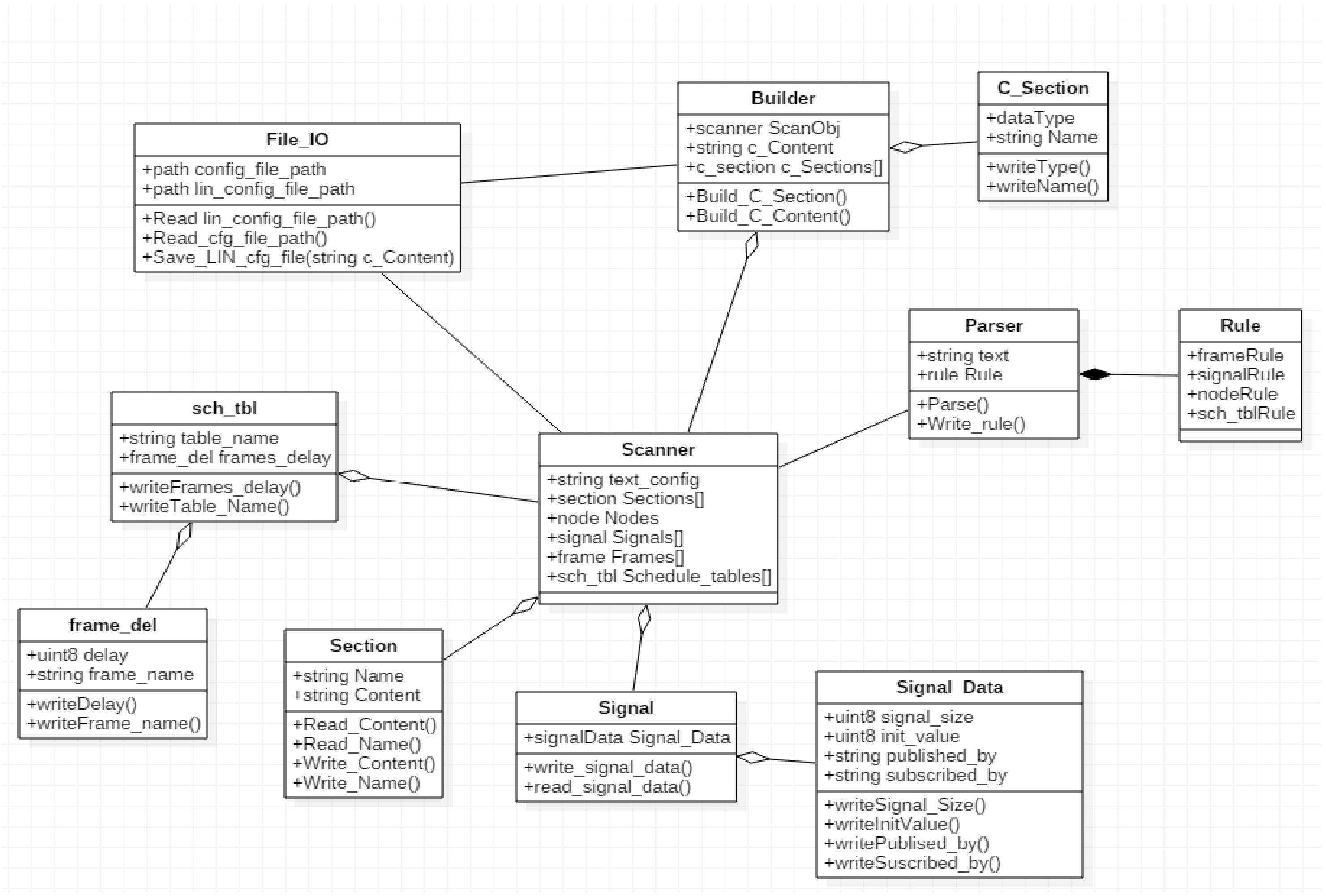


**Figure 19. UML diagram break down implementation.**

The intention of this implementation is to split all into smaller procedures and objects. As it can be seen, the **Scanner** object takes an important part, because it is the one that contains all the objects. Based on the sections it has, it will translate them to objects with the help of the **Parser**.

#### 3.2.1.9. DLL Library

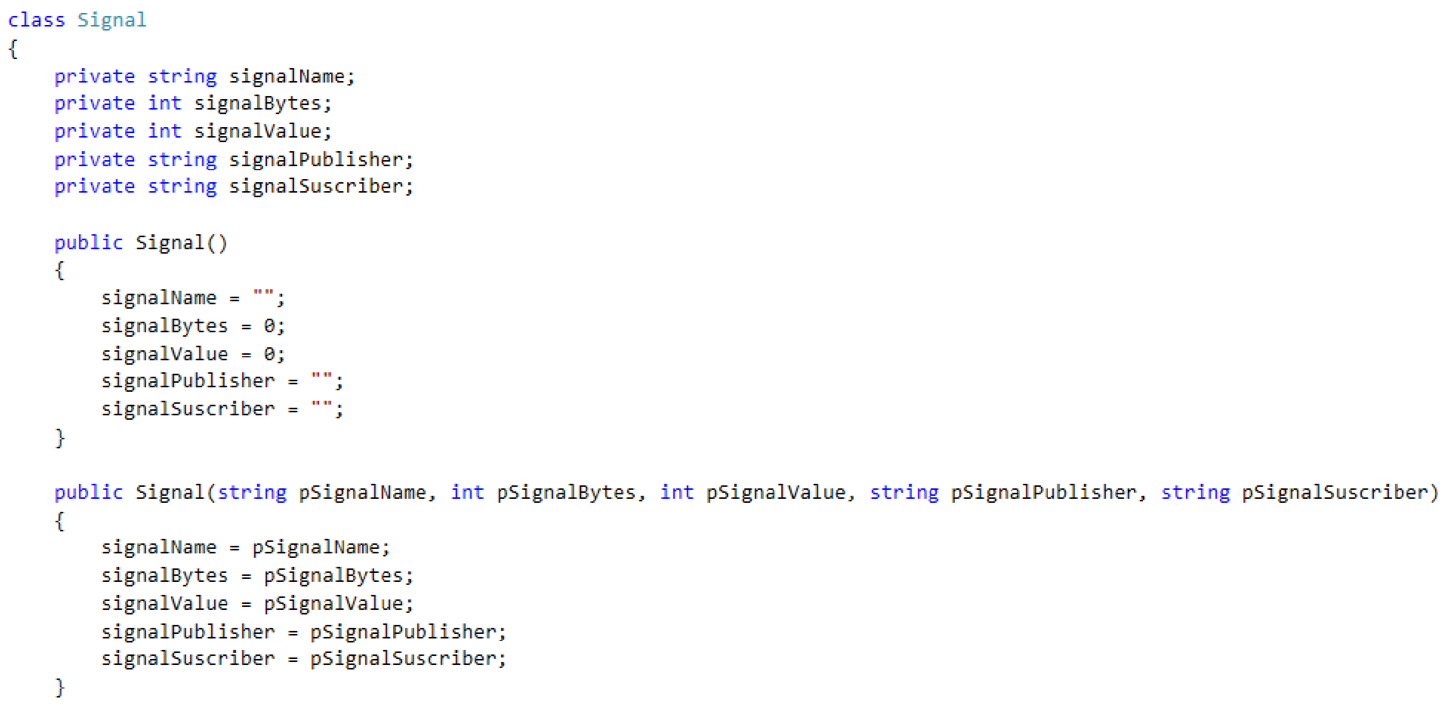
For Microsoft Windows operating system much of the functionality is provided by Dynamic Link Libraries (DLL). For example, some programs may contain many different modules, and each module of the program is contained and distributed in DLLs.

Using DLLs helps modularize the code, reuse it, make memory usage more efficient and also, reduce disk space. Therefore, the operating system and programs load faster.

DLL is a library that contains code and data that can be used by one or more programs at the same time, in other words, it’s sharable code amount programs. The DLL module can be loaded into programs at run time.

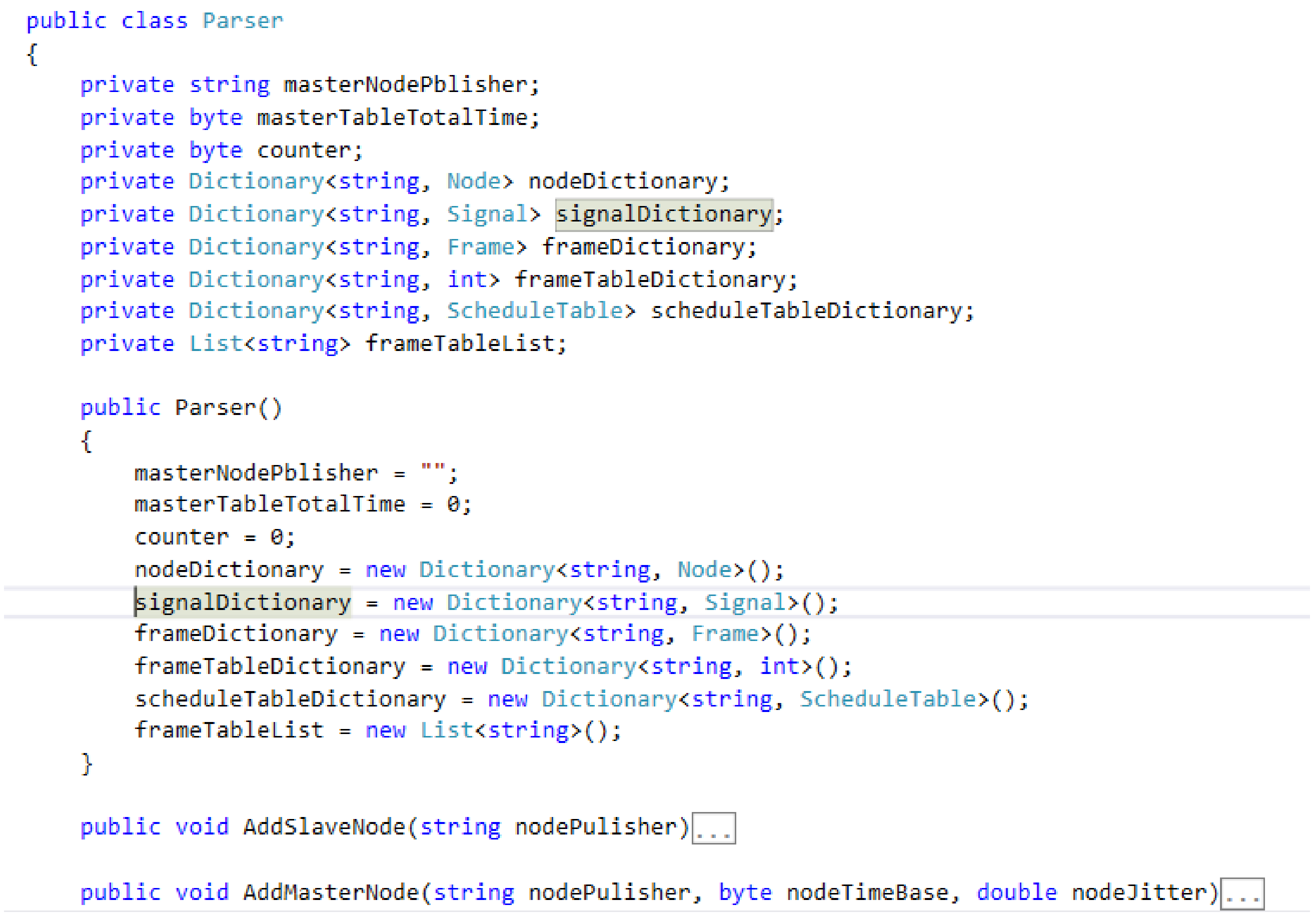
For the developing the library Microsoft Visual C# was used; it gives you the option to develop software as a DLL project type.

For developing the DLL this work has considered while the scanning process the Interface uses DLL methods to create proper objects with its attributes, so when it finishes scanning, all the needed information is available and ready to start parsing and constructing the output source files.



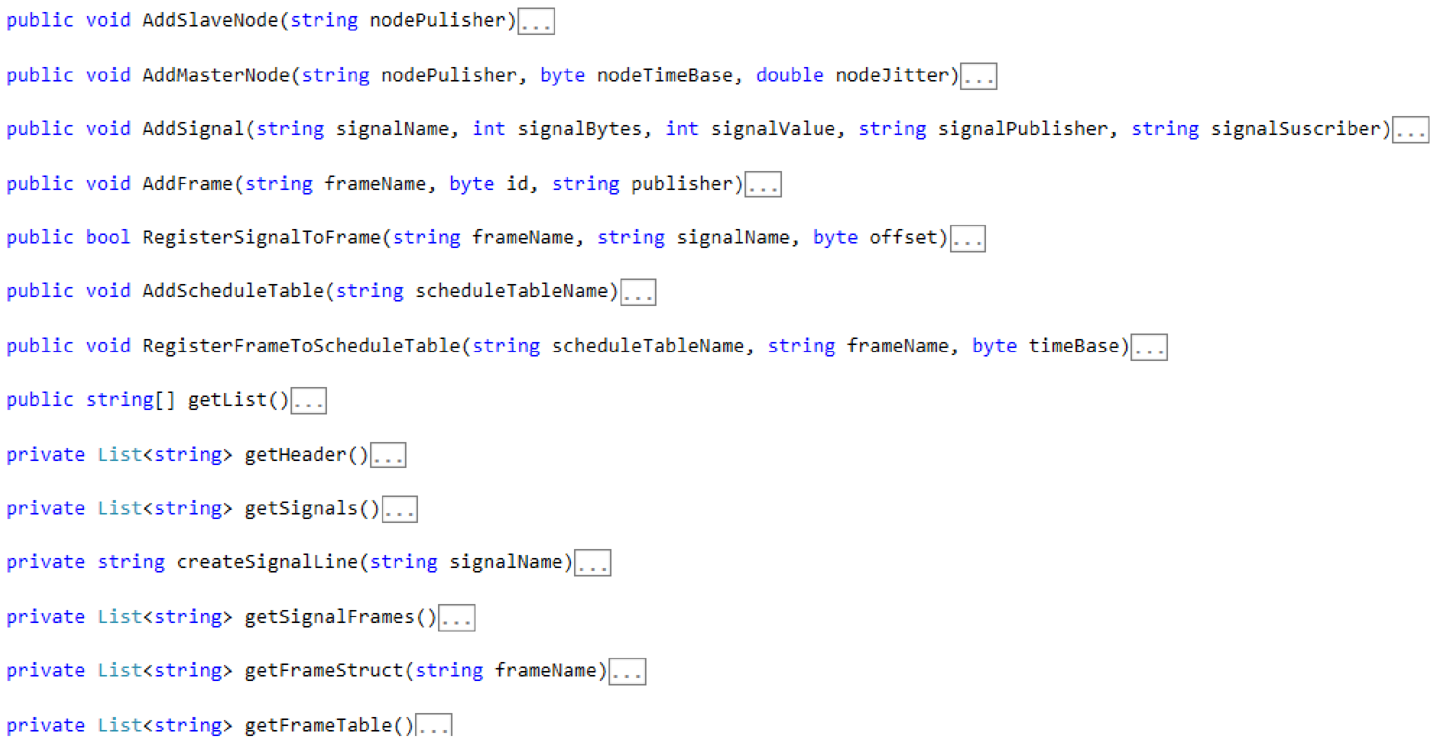
```
class Signal
{
    private string signalName;
    private int signalBytes;
    private int signalValue;
    private string signalPublisher;
    private string signalSuscriber;

    public Signal()
    {
        signalName = "";
        signalBytes = 0;
        signalValue = 0;
        signalPublisher = "";
        signalSuscriber = "";
    }

    public Signal(string pSignalName, int pSignalBytes, int pSignalValue, string pSignalPublisher, string pSignalSuscriber)
    {
        signalName = pSignalName;
        signalBytes = pSignalBytes;
        signalValue = pSignalValue;
        signalPublisher = pSignalPublisher;
        signalSuscriber = pSignalSuscriber;
    }
```

**Figure 20. Constructor for signal class and attributes.**



```
public class Parser
{
    private string masterNodePblisher;
    private byte masterTableTotalTime;
    private byte counter;
    private Dictionary<string, Node> nodeDictionary;
    private Dictionary<string, Signal> signalDictionary;
    private Dictionary<string, Frame> frameDictionary;
    private Dictionary<string, int> frameTableDictionary;
    private Dictionary<string, ScheduleTable> scheduleTableDictionary;
    private List<string> frameTableList;

    public Parser()
    {
        masterNodePblisher = "";
        masterTableTotalTime = 0;
        counter = 0;
        nodeDictionary = new Dictionary<string, Node>();
        signalDictionary = new Dictionary<string, Signal>();
        frameDictionary = new Dictionary<string, Frame>();
        frameTableDictionary = new Dictionary<string, int>();
        scheduleTableDictionary = new Dictionary<string, ScheduleTable>();
        frameTableList = new List<string>();
    }

    public void AddSlaveNode(string nodePulisher)...

    public void AddMasterNode(string nodePulisher, byte nodeTimeBase, double nodeJitter)...
```

**Figure 21. Parser withpublic interface exposed to LabVIEW GUI.**

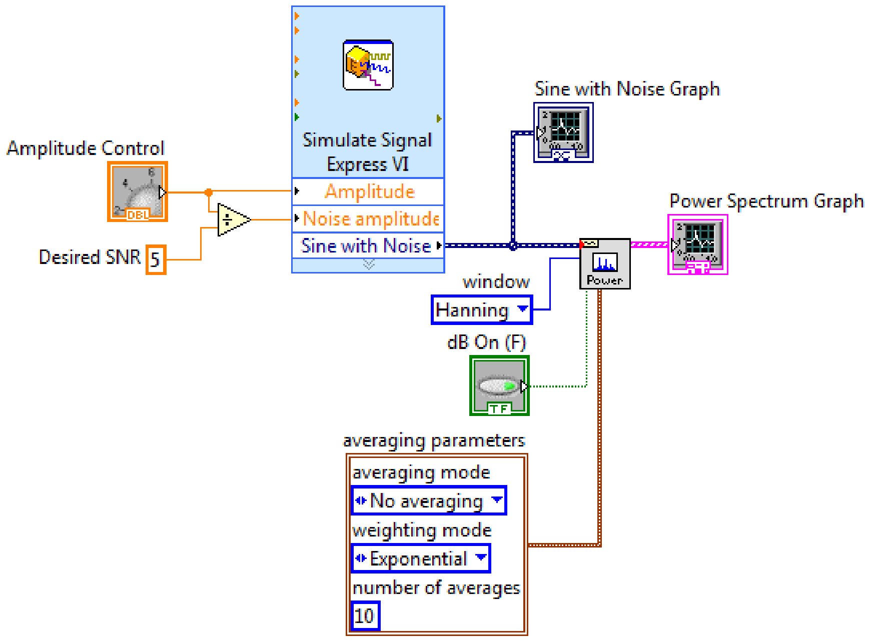


```
public void AddSlaveNode(string nodePulisher)...

public void AddMasterNode(string nodePulisher, byte nodeTimeBase, double nodeJitter)...

public void AddSignal(string signalName, int signalBytes, int signalValue, string signalPublisher, string signalSuscriber)...

public void AddFrame(string frameName, byte id, string publisher)...

public bool RegisterSignalToFrame(string frameName, string signalName, byte offset)...

public void AddScheduleTable(string scheduleTableName)...

public void RegisterFrameToScheduleTable(string scheduleTableName, string frameName, byte timeBase)...

public string[] getList()...

private List<string> getHeader()...

private List<string> getSignals()...

private string createSignalLine(string signalName)...

private List<string> getSignalFrames()...

private List<string> getFrameStruct(string frameName)...

private List<string> getFrameTable()...
```

**Figure 22. Parser Class showing public methods available to be used and private methods.**

### 3.2.1.10. LabVIEW Complementary Application

LabVIEW is a software development environment with numerous components. It's a graphical-based programming tool own by National Instruments. It main feature is selecting amount the plenty building blocks to wire, connect and control many different tasks.



Desired SNR 5

window Hanning

averaging mode No averaging

weighting mode Exponential

number of averages 10

**Figure 23. LabVIEW with graphical code.**

LabVIEW, as graphical programming, already provides an easy way to construct an HMI, it is also compatible with .NET framework, so it can consume DLL's to access the system and other applications.

A Producer-Consumer like architecture was used. The producer (First loop on top) handles the events generated by the user. The consumer handles instructions that require more processing, like the scanning, parsing and building processes.

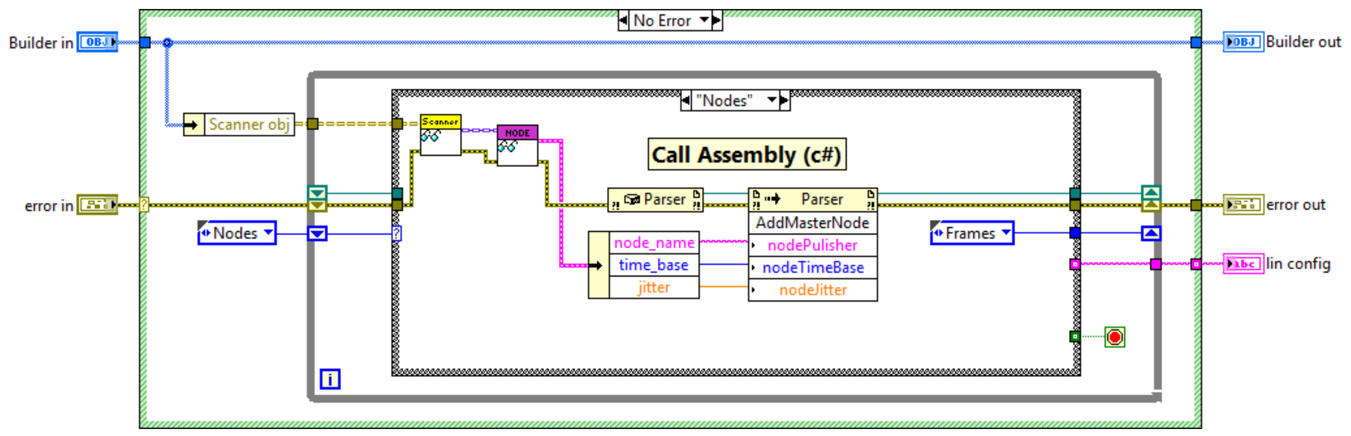


Figure 24. Producer-Consumer Architecture.

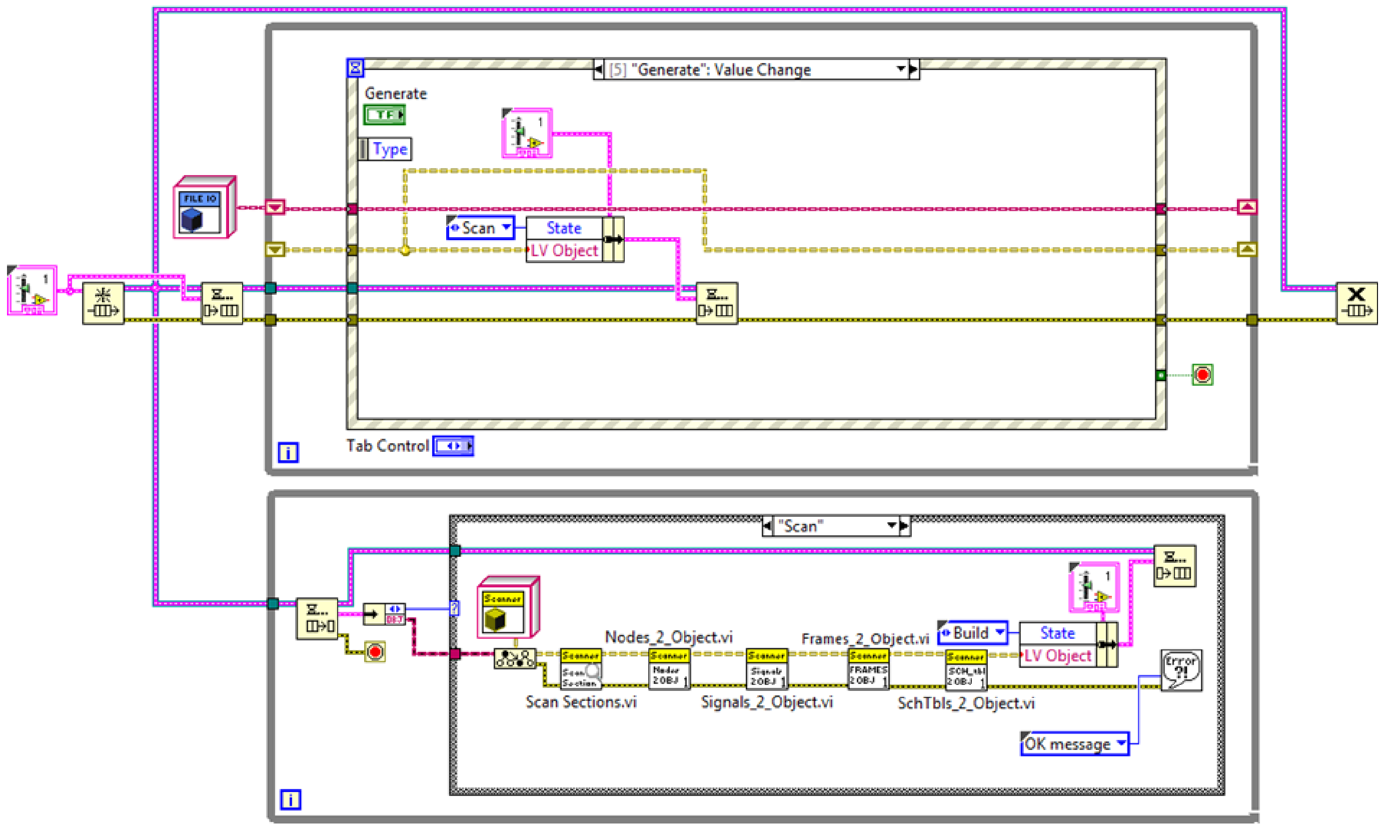


Figure 25. Producer-Consumer Architecture.

# 4. TEST AND RESULTS

***Abstract:*** *In this chapter, the results obtained after executing a set of tests to the implementations are shown. The experimental scenario is briefly described.*

## 4.1. Experiment and Testing Scenario

In order to verify the functionality, it is needed the master and at least one node. This work decided to set the code configuration as MASTER and use as serial communication tool to simulate the SLAVE response. Additionally, a Docklight tool is used, and an oscilloscope to see the frames in the physical bus.

LIN bus is running at 19200 baudrate, the serial communication tool must be configured at the same bus speed. S12EX board Tx/Rx channels connected to PC's Rx/Tx channels accordingly. Put the scope probe to the Tx channel in the board and start both, the application in the target and the pc simulator tool.

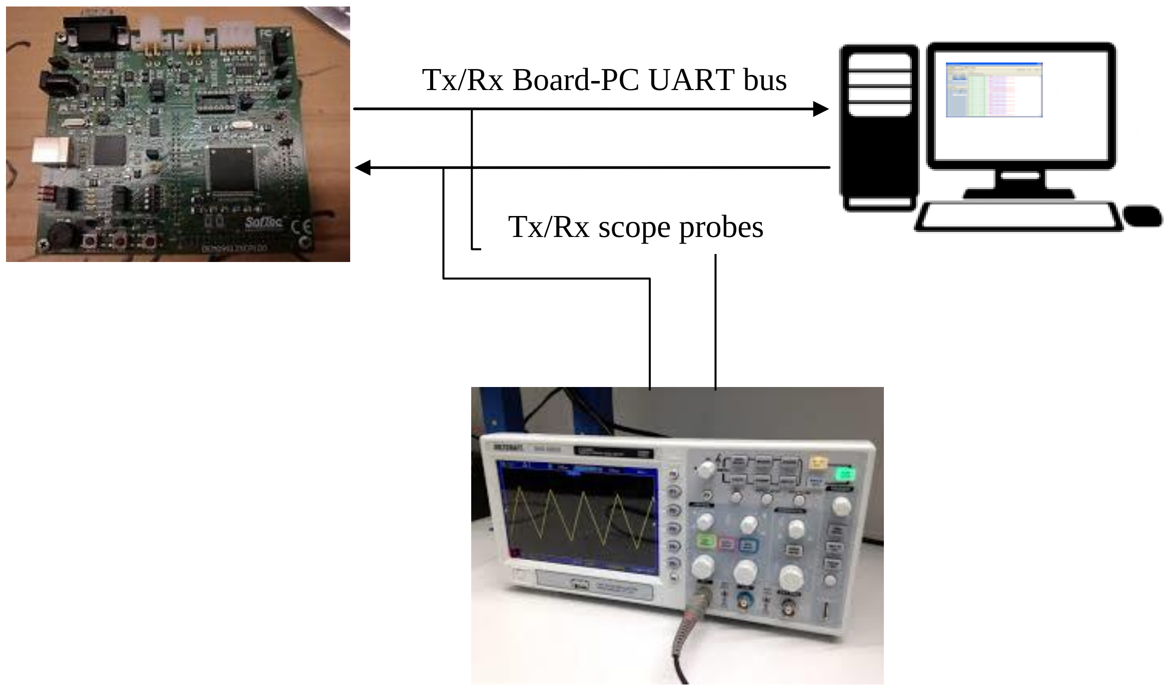


**Figure 26. Target board – PC/Oscilloscope testing environment.**

## 4.2. Results

The MASTER configuration implements both, the start of bus communication and reception of response. The first step is to verify the fix data in the protocol, which is: The SynchBreak, SynchField and PID. Therefore, the PID must be the SLAVE. In the image it can be seen the initial transition indicating the start of the bus activity, followed by the SynchField, which is 0x55 and then the PID. Here the MASTER is ready to receive information coming from the SLAVE.

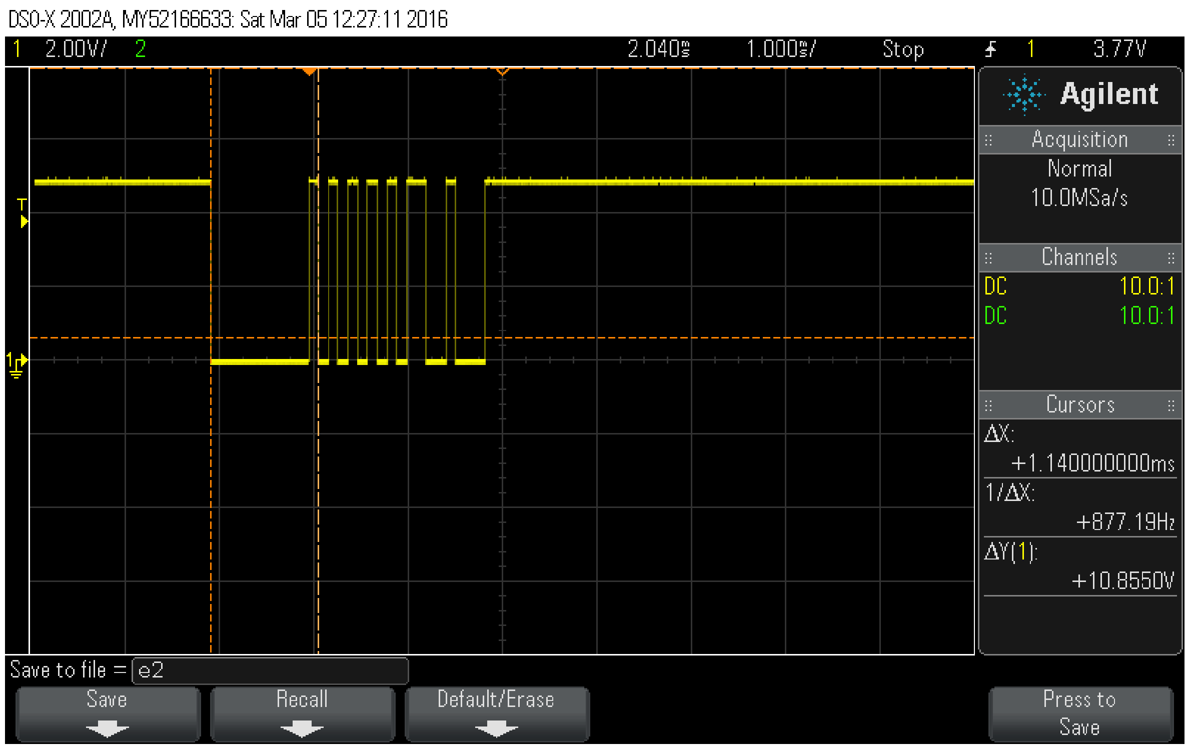


**Figure 27. Master node initiating the bus activity: SynchBreak, SynchField and PID.**

The application after receiving the information coming from the PC simulation tool, continuing running sending the frames scheduled at the right timing.

Now the next step is to verify of the MASTER sending not only the initial frame, but also the complete frame corresponding to be published by the MASTER.

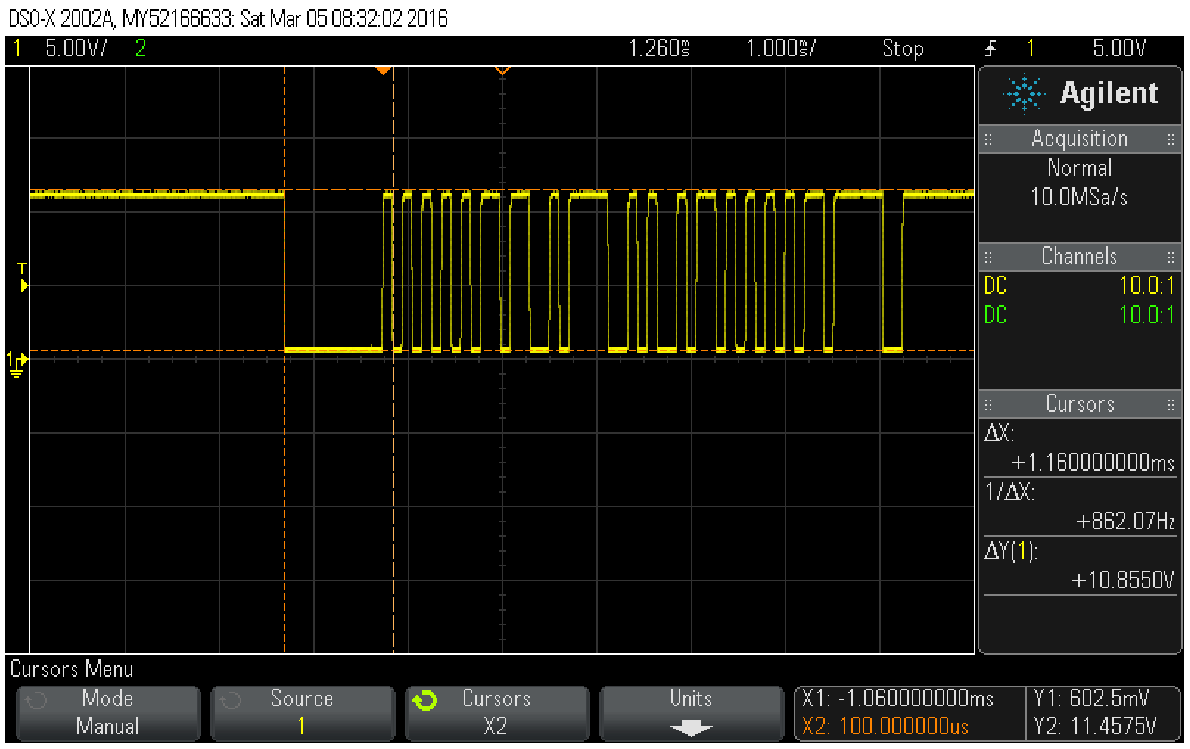


**Figure 28. Frame with publisher as MASTER, therefore SynchBreak, SynchField, PID and data are sent.**

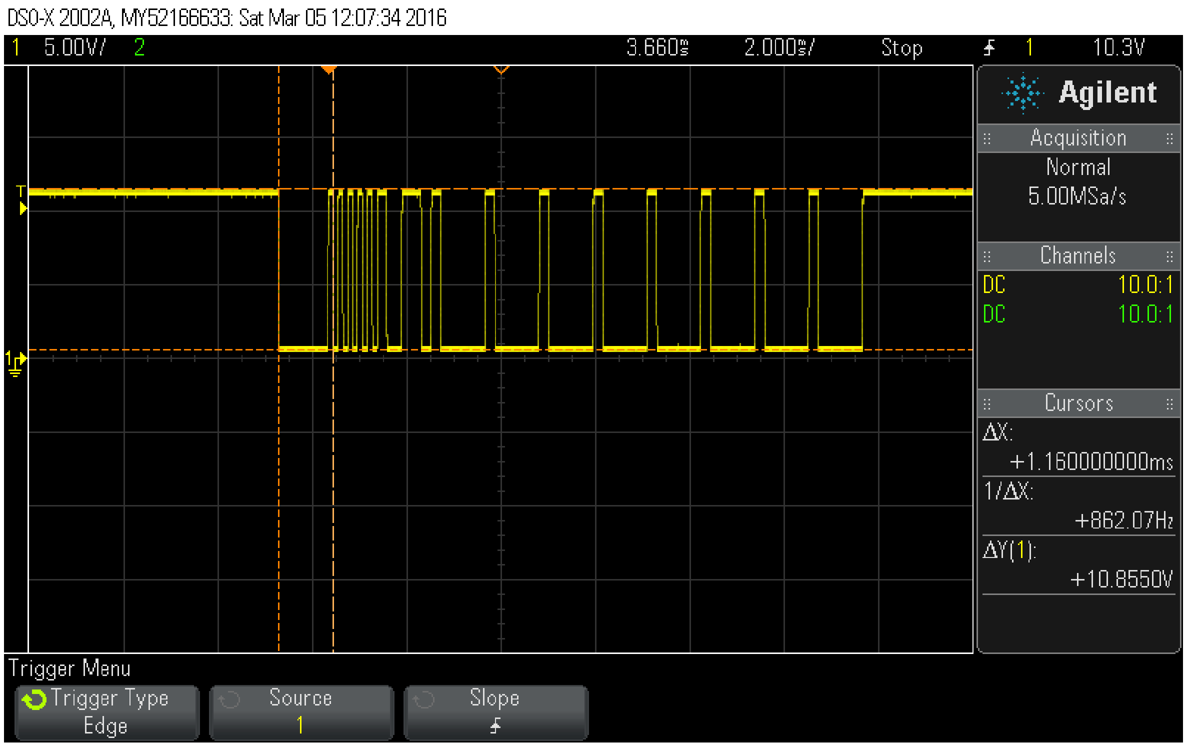
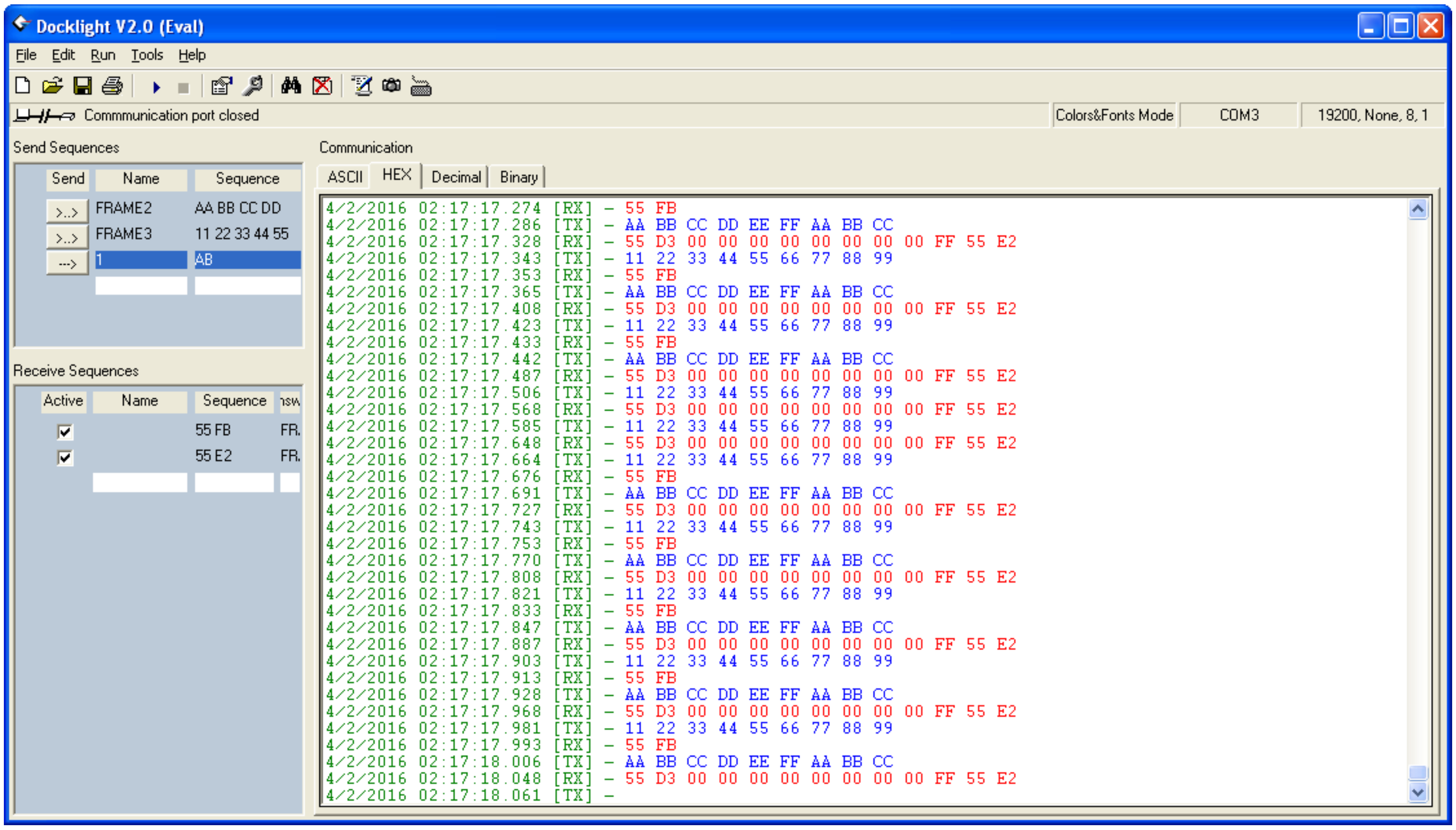


**Figure 29. The Master sends the sleep message when a button is pressed.**



| Send | Name | Sequence |
|---|---|---|
| >..> | FRAME2 | AA BB CC DD |
| >..> | FRAME3 | 11 22 33 44 55 |
| ---> | 1 | AB |

| Active | Name | Sequence | nsw |
|---|---|---|---|
| ✓ | | 55 FB | FR. |
| ✓ | | 55 E2 | FR. |

```
4/2/2016 02:17:17.274 [RX] - 55 FB
4/2/2016 02:17:17.286 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.328 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.343 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.353 [RX] - 55 FB
4/2/2016 02:17:17.365 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.408 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.423 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.433 [RX] - 55 FB
4/2/2016 02:17:17.442 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.487 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.506 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.568 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.585 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.648 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.664 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.676 [RX] - 55 FB
4/2/2016 02:17:17.691 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.727 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.743 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.753 [RX] - 55 FB
4/2/2016 02:17:17.770 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.808 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.821 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.833 [RX] - 55 FB
4/2/2016 02:17:17.847 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.887 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.903 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.913 [RX] - 55 FB
4/2/2016 02:17:17.928 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:17.968 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:17.981 [TX] - 11 22 33 44 55 66 77 88 99
4/2/2016 02:17:17.993 [RX] - 55 FB
4/2/2016 02:17:18.006 [TX] - AA BB CC DD EE FF AA BB CC
4/2/2016 02:17:18.048 [RX] - 55 D3 00 00 00 00 00 00 00 00 FF 55 E2
4/2/2016 02:17:18.061 [TX] -
```

**Figure 30. PC Serial Communication Tool with Rx data frames from the MASTER and Tx data frames from the SLAVE.**

Knowing the configuration file information frames, in the PC serial simulation tool it is possible to verify that the MASTER data transmitted is correct.

# 5. CONCLUSIONS

**Abstract:** *In this chapter, some conclusions and how the abilities I have developed through this work was improved my daily activities at my job are described.*

## 5.1. Conclusions

Regarding the results got testing the implementation in the target board (S12EX), there are three major things to get this working. A well-defined state machines, make use of the interruption scheme to send and receive information in the time constrain established, build the information in a backup task to avoid time consumption critical to achieve the frame transmission and the compilation switches to decide whether using as Master or Slave.

I experienced a weird behavior at time of sending the sync break we started using a serial monitor interface, I noticed the sync break byte was sometimes inverted, I decided to use an oscilloscope to get a second look to the same bytes, the signal was sent every time in the right way. The problem was the sync break is not a native serial data and the tool could have misunderstood the information.

Regarding the HMI parser and library implementation, this assignment taught me that sometimes, in order to have a complete implementation, it is necessary to use high-level tools. It is good to know how to use them because a good usage can help to have a really good solution in all aspects.

I had some difficulties when scanning the text configuration file because of the curly braces. Sometimes more than one level exists, so this leveling should be taken into account.
In case of the c-file parsing, I had the difficulty of building the file because of the names of the variables. At the end, the OOP approach helped a lot to split up the processes and achieve the objectives.

Summarizing, going from low level development with interrupt event handling, state machine and Operative System task running in backup to the high level HMI parser development and DLL library, it creates a whole engineering solution.

This project has been divided in two main solutions (one as the solution running in the target device and the other one as HMI parser). This has remarked the importance of requirements definition, development scope, engineering constrains, development environment and development skills; it’s all different in each solution.

All knowledge acquired enriches the work flow in my dailies activities, since are base in the software development for automotive industry; microcontrollers uses interruption to forward information to the Operative System, regardless which is, to execute specific actions. Besides, high level tools are always needed to interact with the target devices.

# BIBLIOGRAPHY

[1]. DEMO9S12XEP100: Demo Board for the 16-bit MC9S12XE and XS-Families, http://www.nxp.com/products/automotive-products/microcontrollers-and-processors/16-bit-s12-s12x-mcus/demo-board-for-the-16-bit-mc9s12xe-and-xs-families.:DEMO9S12XEP100

[2]. LIN Consortium, http://www.lin-subbus.org/

[3]. ISO 17987-6:2016 Road vehicles -- Local Interconnect Network (LIN) -- Part 6: Protocol conformance test specification, http://www.iso.org/iso/catalogue_detail.htm?csnumber=61227

[4]. LabVIEW System Design Software, http://www.ni.com/labview/#

[5]. Docklight - Test & Simulate Serial Protocols, http://docklight.de/

[6]. Agilent Oscilloscope, http://www.keysight.com/en/pcx-x2015004/oscilloscopes?cc=MX&lc=eng