

|  |
| --- |
| Master  Degree in Electronic design / Embedded systems specialty program |
| Advanced Embedded Systems  Homework # 3  SCI Driver Configuration |
|  |

**Provide SCI driver specific functionality**

Professor: Abraham Tézmol Otero

Alumni: César Tomás Gómez Cruz

Andrés Eduardo Sánchez Rodríguez

Juan Carlos Morales Quintana

Pablo Esteban Camacho

**October 19th, 2015**

I T E S O A.  C., Universidad Jesuita de Guadalajara

Periférico Sur Manuel Gómez Morín #8585, Tlaquepaque, Jalisco, México

Technical Report Number: ESE-O31015-001

® ITESO A.C.

1. **Introduction :**

Serial Communications Interface (SCI) is an old communication protocol originally used to connect input/output terminals to mainframe computers. The common name for this protocol is Universal Asychronous Receiver Transmitter or UART. When microcomputers came onto the scene in the 1980s, this serial protocol was used to perform input/output. The protocol is neither fast, nor reliable. However, it is simple and continues to exist in state-of the- art microcontrollers because there is still a wide range of external devices using the protocol, such as GPS and LCD graphics displays.

Serial transmission involves sending one bit a time, where the data is spread out over time.

Engineers have found that one can send data farther, faster, less expensively, and more reliably using serial versus parallel channels. This is because it is easier to control capacitive loading and added noise within a serial cable. The total number of bits transmitted per second is called the baud rate. Most of the Freescale embedded microcomputers supports at least one Serial Communications Interface or SCI.

This project goal is to be able to use and implement basic TX/RX operations of serial communication devices, with general features common to all devices. Common features that affect the entire SCI module include:

* A baud rate control register used to select the transmission rate
* A mode bit M used to select 8-bit (M0) or 9-bit (M1) data frames

Each device most be capable of creating its own serial port clock with a period that is an integer

multiple of the E clock period. The programmer will select the baud rate by specifying the

Integer divide-by used to convert the E clock into the serial port clock. This will be done trough static configuration structure.

A frame is the smallest complete unit of serial transmission. The difficulty with serial transmission is synchronizing the receiver with the transmitter. With the SCI protocol, a start bit that always has a 1 to 0 transition is sent by the transmitter signifying the start of the frame. Figure 8.1 plots the signal versus time on a serial port, showing a single frame, which includes a start bit (0), 8 bits of data (least significant bit first) and a stop bit (1). The stop bit is also required so that when a start bit occurs there will be a 1 to 0 transition. This protocol is used for both transmitting and receiving. The information rate, or bandwidth, is defined as the amount of data or usable information transmitted per second.

Our main goal while doing this project will be to implement SCI driver using XEP100 development board, with a well-defined configuration to make it as generic as possible using industry standard techniques.

1. **Exercise Statement**

Perform re-architecture of the SCI driver embedded in the Dynamic Memory Allocation example project as per the attached document "Software component - Configuration and Status".

Submit a unique \*.zip file with your project and your report.

## Project 3 Static configuration and Internal structures

## Configuration Type Definitions

Types used by the SCI driver shall be declared as follows:

typedef **enum tSCI\_Channel**

{

SCI\_CH0,

SCI\_CH1,

SCI\_NULL

};

typedef struct

{

UINT32 SCI\_baudrate; /\* Baudrate \*/

tCallbackFunction SCI\_TX\_callback;

tCallbackFunction SCI\_RX\_callback;

enum tSCI\_Channel SCI\_channel;

UINT8 SCI\_TX\_enable;

UINT8 SCI\_RX\_enable;

UINT8 SCI\_TIE\_enable; /\* Transmit Interrupt Enable \*/

UINT8 SCI\_RIE\_enable; /\* Receive Interrupt Enable \*/

UINT8 SCI\_TX\_MAX\_BUFFER\_SIZE;

UINT8 SCI\_RX\_MAX\_BUFFER\_SIZE;

}**tSCIchannel\_config**;

typedef struct

{

UINT8 u8Number\_of\_SCI\_channels;

const tSCIchannel\_config \* ptr\_SCIchannels\_config;

}**tSCIdriver\_config**;

Note: SCI\_baudrate can only have the following values:

Type UINT32

Range: {115200, 57600, 38400, 19200, 9600, 4800}

Those type definitions shall be used internally by the driver.

## Status Type Definitions

typedef struct

{

UINT8 \* u8SCI\_RxData;

UINT8 \* u8SCI\_TxData;

UINT8 \* pu8SCI\_Receive\_Data\_ptr;

UINT8 \* pu8SCI\_Read\_Receive\_Data\_ptr;

UINT8 \* pu8SCI\_Read\_Transmit\_Data\_ptr;

UINT8 u8SCI\_RxLength;

UINT8 u8SCI\_TxLength;

UINT8 u8SCI\_Comm\_Status;

enum tSCI\_Channel SCI\_channel;

}**tSCIchannel\_status**;

typedef struct

{

tSCIchannel\_status \* SCIchannel\_status;

UINT8 u8Number\_of\_SCI\_channels;

}**tSCIdriver\_status**;

## Driver Functionality

SCI driver shall provide the following functionality:

1. Driver shall take a unique configuration and configure all channels as per their corresponding static configuration
2. Require, upon initialization, dynamically allocated RAM for their TX and RX buffers
3. Require, upon initialization, dynamically allocated RAM for their internal status structures
4. Provide a unique series of functions as per the provided examples. i.e. no duplicate functions for each channel.
5. Provide the same level of functionality as the provided example.
6. Exceptions to point e) are those who would improve modularity, reusability, and security of the driver

Dynamic memory allocation driver can be taken as example.

A well-documented report that shows the key improvements and rationale behind those improvements is required.

\* \* \*

**III. Description of data transmission & reception process, using SCI and using interrupts**

Just to start our explanation in some point, we will reviewing some of the key files which compound our project: the files for the SCI Driver Configuration and Status Control:

These four files correspond to the Serial Configuration Driver:

2 files for the Driver configuration itself.

2 files for the control of the Driver status.

serial\_cfg.h

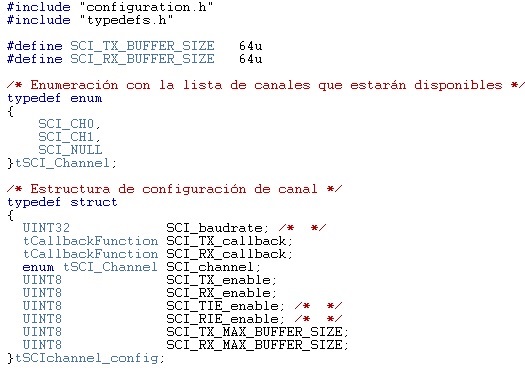
serial\_cfg.c

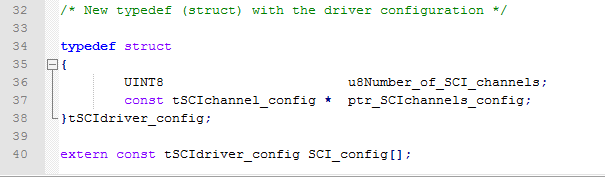
serial.h

serial.c

**serial\_cfg.h**

This header configuration driver file shows that our project matches the provided requirements, as shown in “1. Configuration Types Definition”.





**serial\_cfg.c**

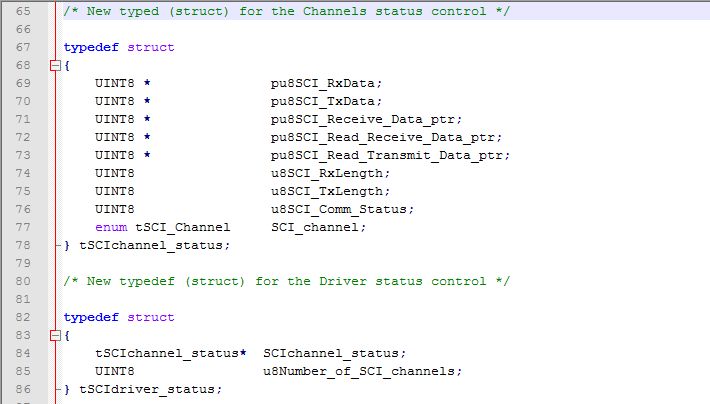
This \*.c configuration driver file shows the initialization values for the data types created in the previous header file; in our example, it is setup for two channels.



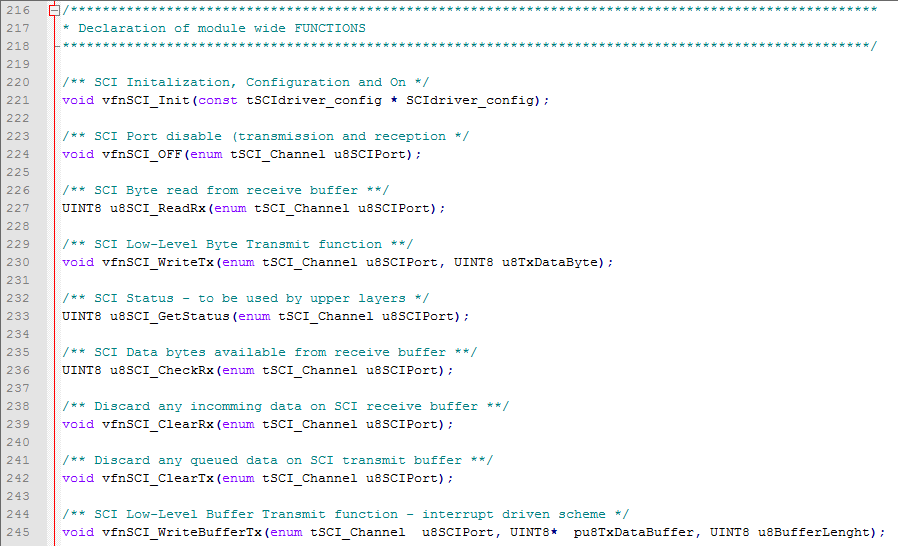
**serial.h**

The main compoenet structures which we want to highlight first in header file are following ones:

a) The data structures created for the control of both Channels and Driver status:

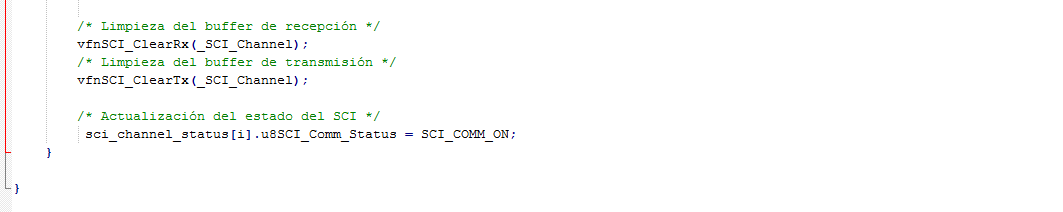
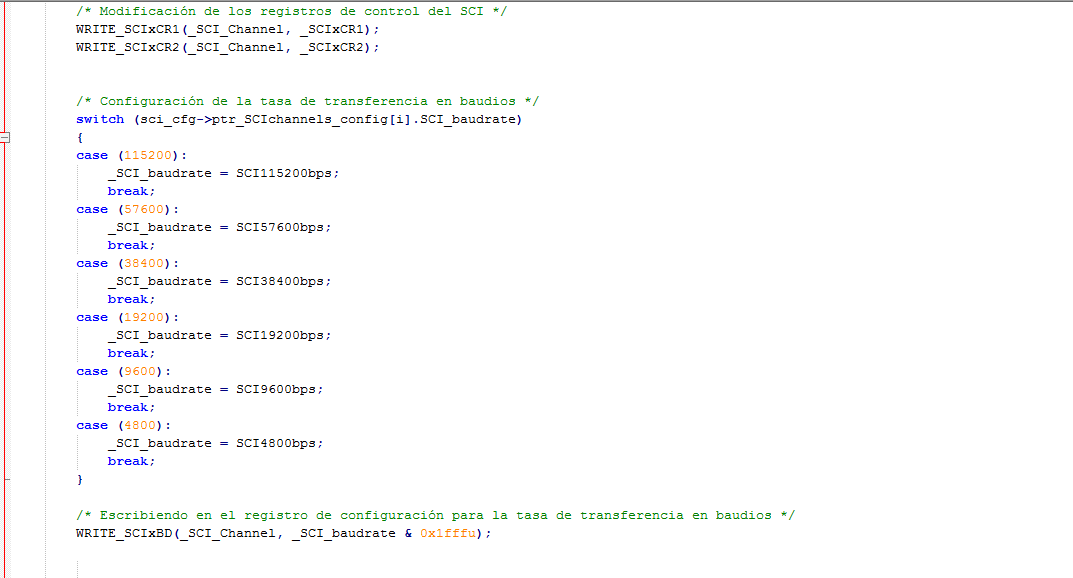
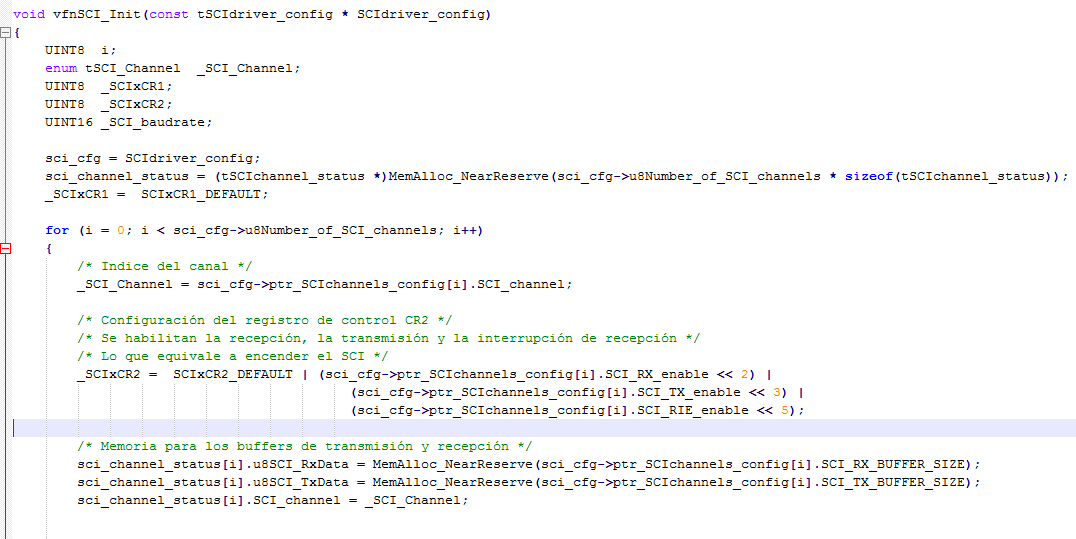


b) The declaration (prototypes) of key functions for the Driver, such as initialization, reading, writing, so forth. (Note: in the following image, only some of them are shown just as example):



**serial.c**

This \*.c Driver file provides, principally, de declaration (body) of the functions declared in the previous header file. We are showing only one function body (the Init one), just as an example:

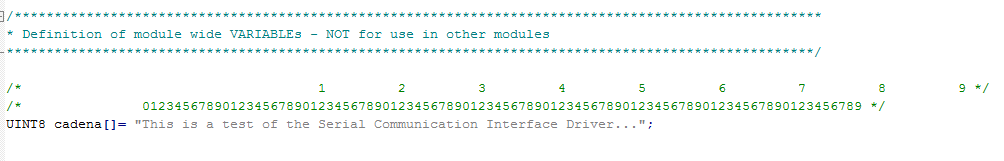


Switch – case to call the appropriate macro corresponding to the select baudrate.

Configuration of the control register for channels.

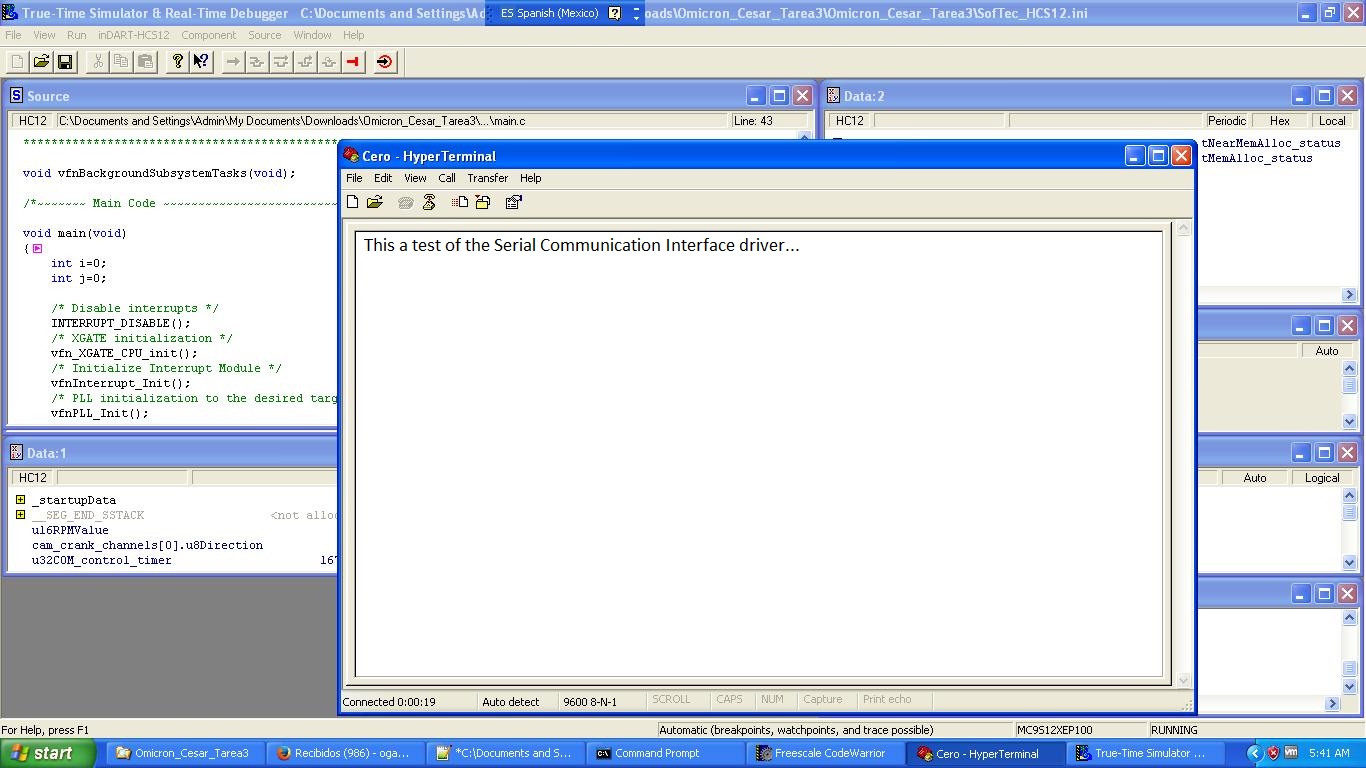
Even if we will turn back to some specific points of the Driver files, we want now to point to the main.h and main.c files.

**main.h – Testing the transmission functionality**

The following snapshot is a extract from the main.h file where we can see the initialization value of the array to test the transmission functionality of the Driver:

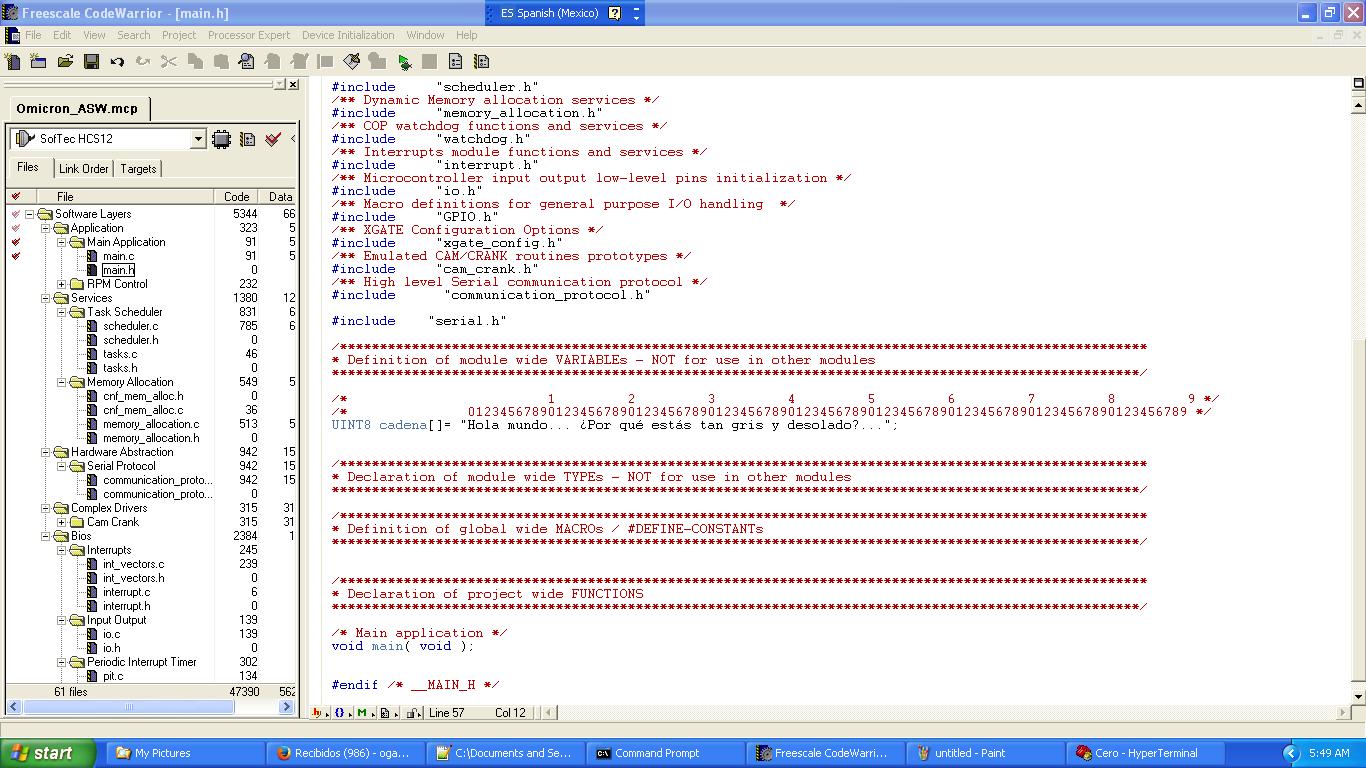
**main.c**

In the following snapshot, we can observe the output of the Driver transmission shown by HyperTerminal (chosen terminal) according to our purpose:

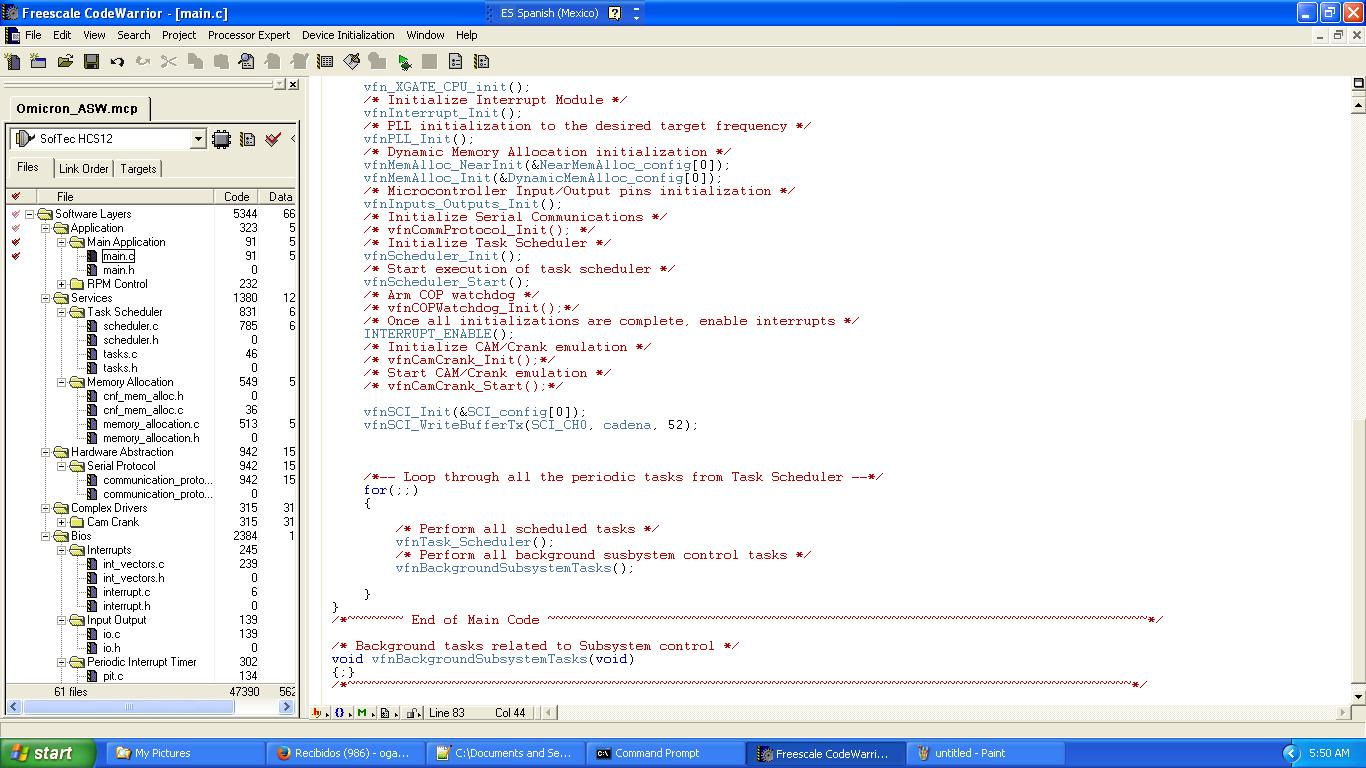


The next snapshot turns back to the **main.h** file, in this case to show a change on the text to be transmitted:

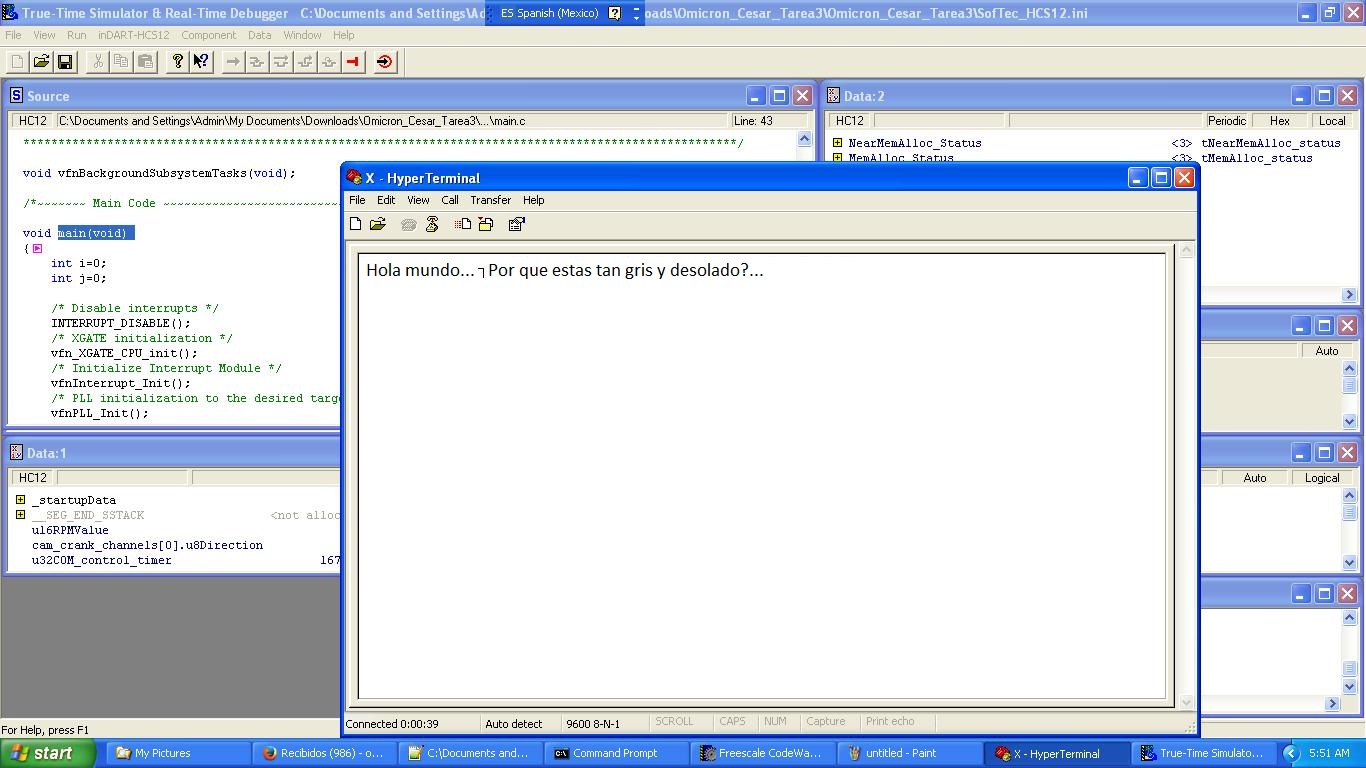
1. Changing the text:



1. Changing the chain extension to be transmitted:

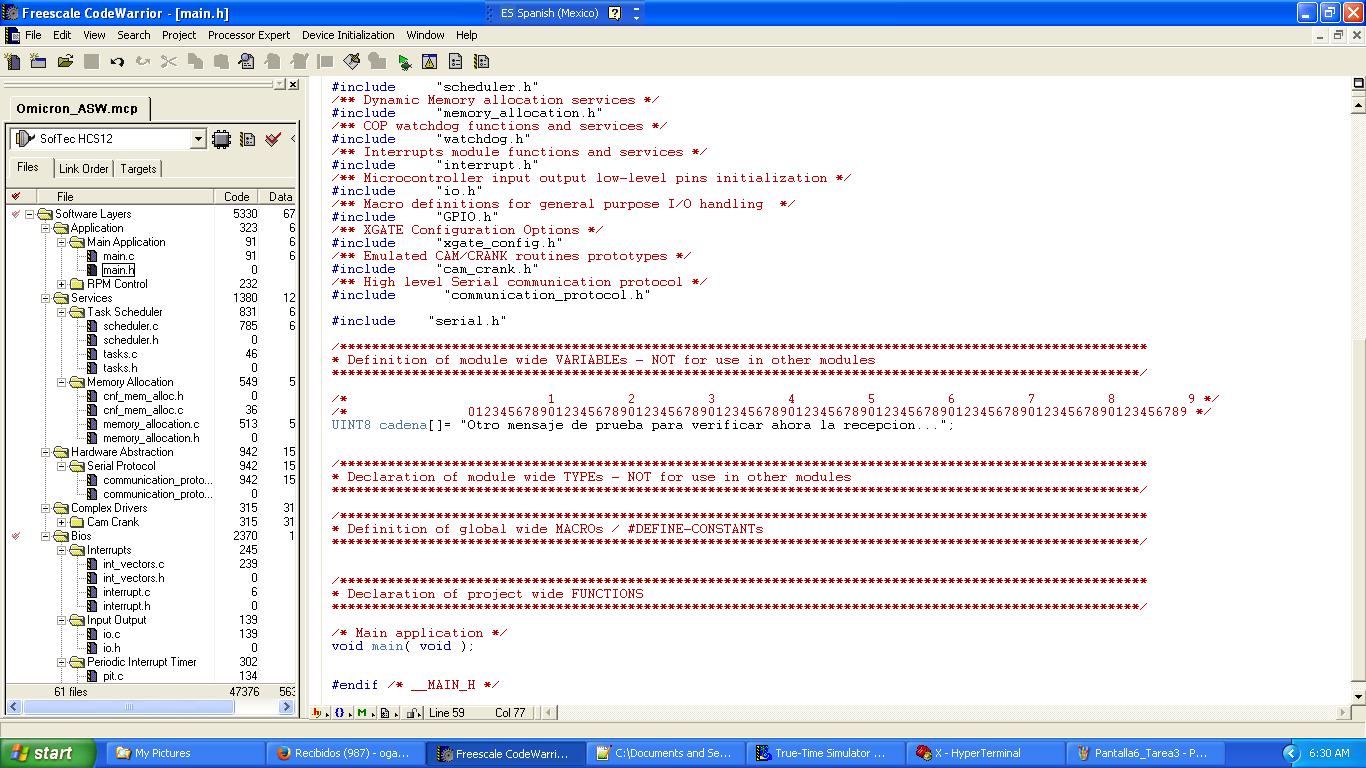


Again, we can see the output in the terminal tool:

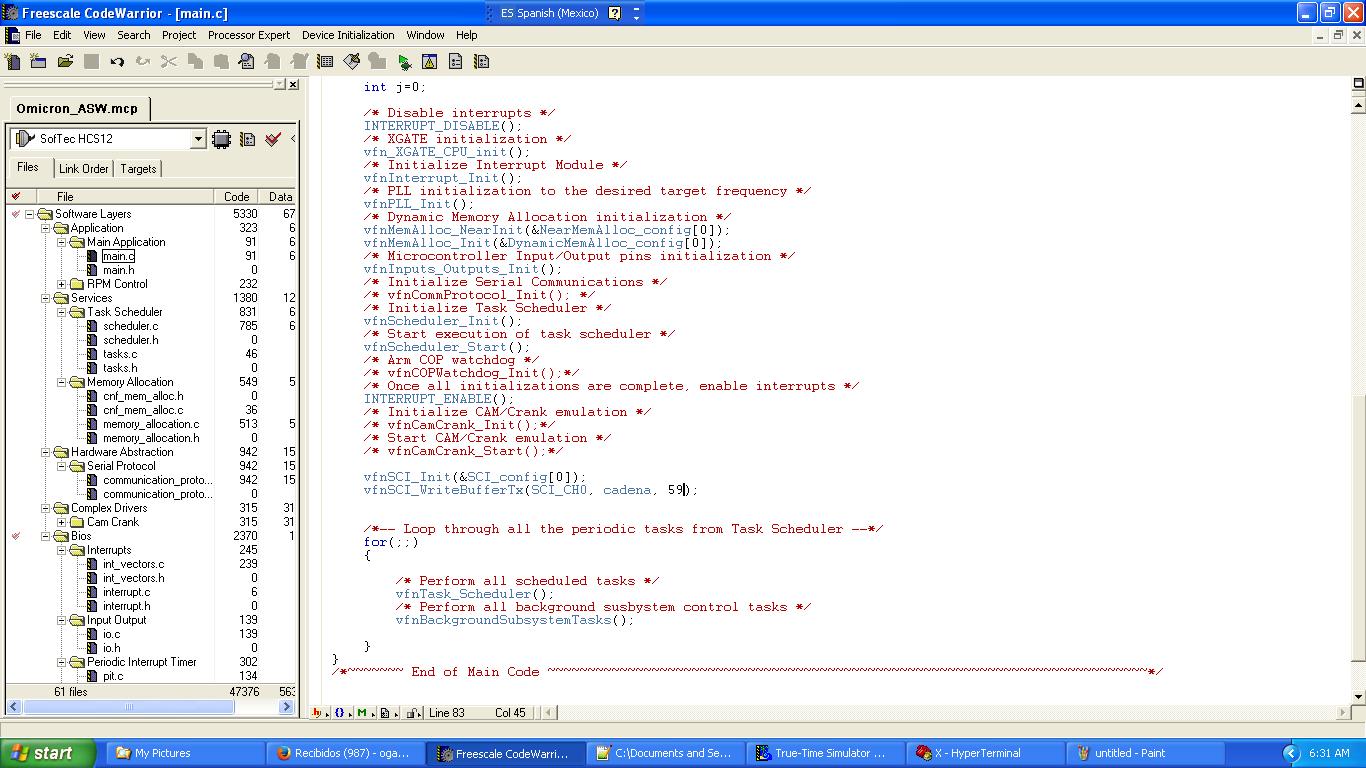
­­­­

Let’s now turn once more to the **main.h** file to apply changes in order to test the reception funcionality of our Driver:

1. Frame to be transmitted:



1. Updating the message lenght:



And, following the previous steps, we can perform any change we would like to implement for transmission process.

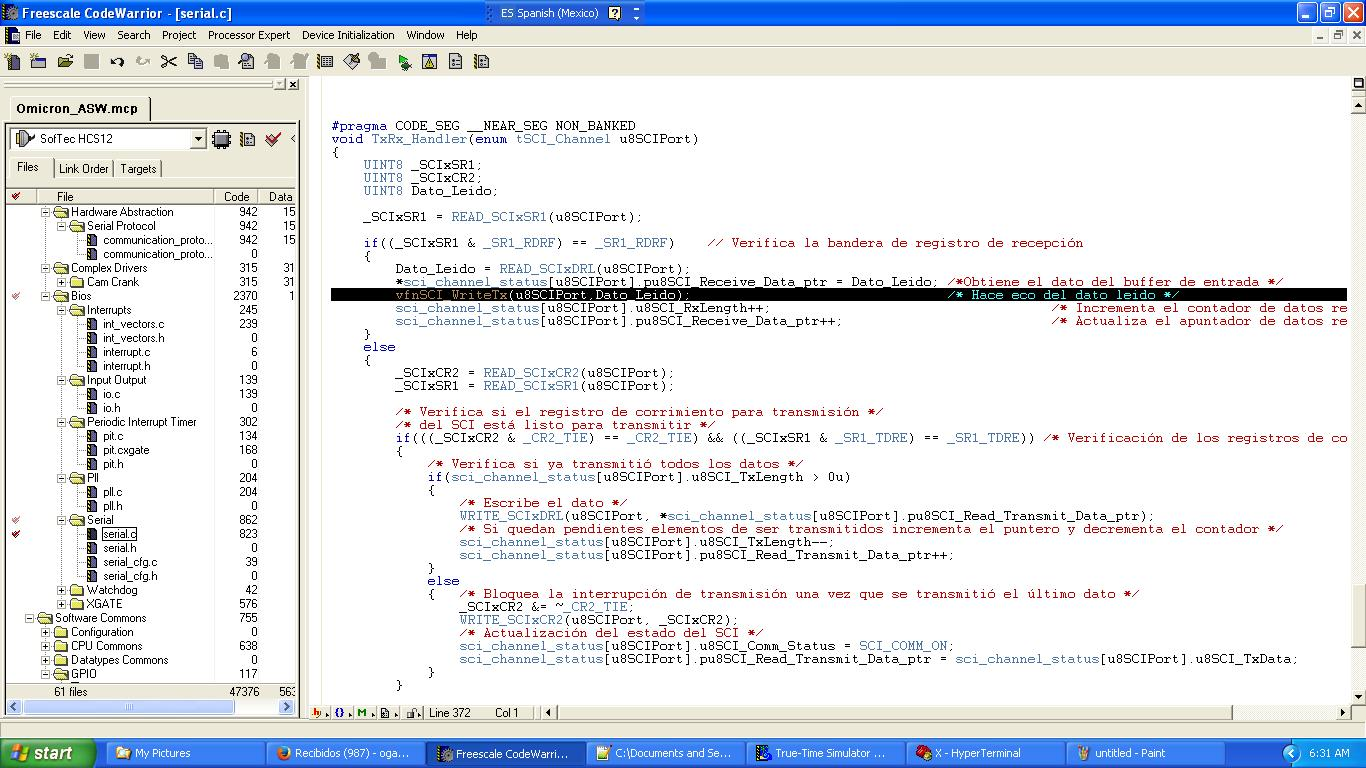
**serial.c – Testing the reception functionality**

As we promissed in previous paragraphs, we are turning back now to the Driver functionality, in this case to show the Handler function and to test the recepcion functionality of the Driver.

In the next snapshot, we have highlighted one code line belonging to the TxRx\_Handler function, contained in the **serial.c** file.

This line of code lets the program replicate characters on the terminal, characters which keys in the keyboard the user have been pressed. Remember that the terminal is just emulation terminal software and, using it, the character is sent to our PCB through the serial port.

This is a way to see that some data has been sent from the terminal to the PCB via the SCI driver, and that the microcontroller also responds using the same media, the SCI.

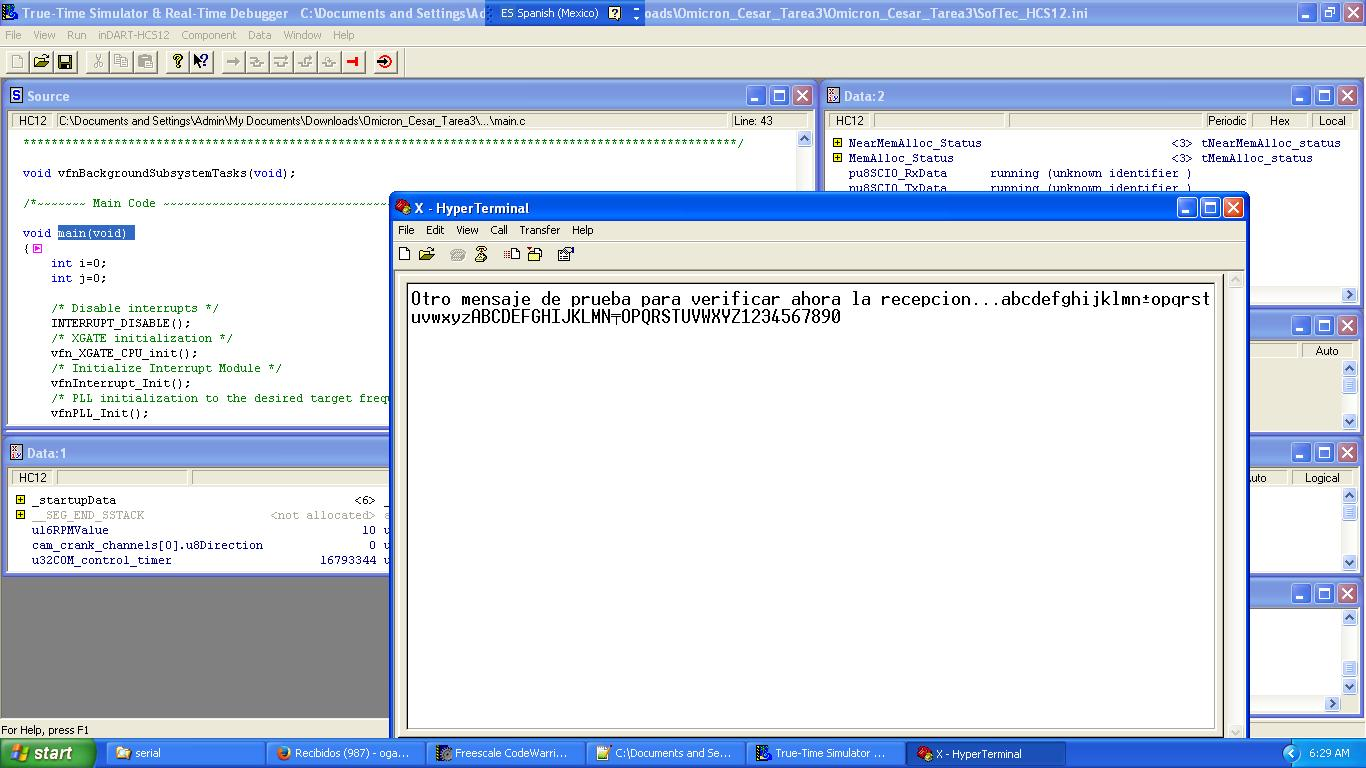
****

**It is important to highlight that this process is performed using interruptions, which proves in an easy way the requested functionality for this project**.

**It is also important to highlight that, in case an interruption occurs in order to perform the data RX / TX, the corresponding function is included in the interrupt vector list (int\_vectors.c).**

**Therefore, even though in the code we show a difference between the called function upon the used channel (channel0, channel1…), really the function which handle all the channels is only one (as we can see in the below copied snapshot, taken from int\_vector.c file); thus, it is the TxRx\_Handler the responsible to choose (in an internal manner) which channel it should handle.**

Let’s check, finally, an example of the previous description, shown in the following snapshot:

****

**IV. Conclusions : A More Technical Process Description**

Note, please, the following:

As mentioned before, we start with the registers initialized to 0 (0x00).Almost the immediately next step is the bitwise OR, which will setup to 1 the fundamental bits for Transmission and Reception.The register SCIDRH/L is the responsible to send the 8-bits-data (that is: each character) through the serial port.To control the transmission through interrupts, we use the previously mentioned 2 registers on PORT0: SCI0CR1 and SCI0CR2, while we turn the corresponding bits to 1 or to 0 respectively (as we will show soon).Our array (ai8AnalogSignalName0) contains the data used to fill up the queue. We have a pointer which points to the first array element (position 0). Then, each character of the array passes through the SCIDRH/L and, at the same time, we activate / deactivate the corresponding bits of SCI0CR1 and SCI0CR2 registers.Is important to notice that, before transmitting, we clean the TC flag. To achieve this goal, we just need to access that register (We perform this action in order to have the interruption active to call the callback function. In fact, if we do not clean the flags, we experience gaps in the communication and, therefore, the calling of callback function is not possible).A common problem could be present when changing clock frequency without modifying the baud rate register, in this case SCI will operate at an incorrect baud rate.

**V. References**

We used the following documentation to correctly setup our Project:

1. **MC9S12XEP100 Reference Manual Covers MC9S12XE Family**: this is the most important document used for this project.
2. ***SCI – Serial Communication Interface: Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O)*** (by [Louis Frenzel](http://www.amazon.com/s/ref=dp_byline_sr_ebooks_1?ie=UTF8&text=Louis+Frenzel&search-alias=digital-text&field-author=Louis+Frenzel&sort=relevancerank)).
3. ***Serial Port Complete: Programming and Circuits for RS-232 and RS-485*** (by Janet Louise Axelson & Jan Axelson).
4. **Serial communication in Matlab With Proteus and Code Vision AVR using Interrupts**: <https://www.youtube.com/watch?v=s7sQuucqd7w>
5. <http://www.learncpp.com/cpp-tutorial/78-function-pointers/>
6. <http://www.cprogramming.com/tutorial/function-pointers.html>
7. <http://www.cprogramming.com/tips/tip/passing-a-pointer-to-a-function>
8. <http://cboard.cprogramming.com/c-programming/94522-callback-functions-concept.html>