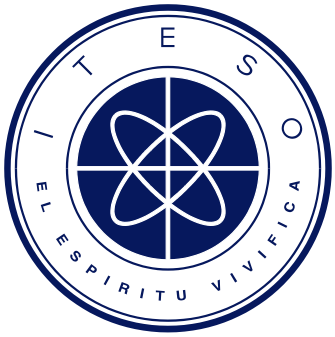
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

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



Design and development of a driver based on J2716 standard for data transmission on high electrical noise environments

Tesis para obtener el grado de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Hiram Rayo Torres Rodríguez

Asesor: José Bernardo Cotero Ochoa

San Pedro Tlaquepaque, Jalisco. 01 de diciembre de 2016

ACKNOWLEDGEMENTS

I would like to thank my parents, for always being there when I needed them the most and supporting me in all my professional studies.

I would also like to thank all my colleagues and classmates which were always there to lend a hand throughout the whole course, laboratories and projects, because without them the completion of this course would not have been possible.

I also want to acknowledge CONACYT for its support, hence, making possible for me to take this specialization course.

Lastly but not least I want to thank ITESO for offering this course, always taking in mind the scientific and technological improvement of our country.

INDEX

RESUMEN

Este trabajo consiste en el diseño e implementación de un controlador de bajo nivel basado en el estándar de la SAE J2716, mejor conocido como SENT por sus siglas en inglés (Single Edge Nibble Transmission). El objetivo de éste es el proponer una alternativa de diseño para la transmisión de información digital en ambientes con alto ruido eléctrico, de tal forma que, se aborde la solución de la pérdida de información que suele ocurrir en sistemas en los que la información debe ser compartida entre controladores.

La aproximación de diseño es descrita a detalle con la intención de establecer los requerimientos que deben ser cumplidos para asegurar la estabilidad, escalabilidad y sostenibilidad en el controlador.

Finalmente se presentan los casos de prueba y resultados para determinar la factibilidad de esta aproximación de diseño y las conclusiones a las que se ha llegado con este estudio.

ABSTRACT

This work consists in the design and implementation of a low level driver based on the SAE J2716 standard, better known as SENT (Single Edge Nibble Transmission). The objective is to propose a design alternative for the transmission of digital information in high electrical noise environments, hence, solving the information-loss issues which often occur on systems where information is required to be shared between controllers.

The design approach is described in detail in order to set the requirements that shall be met in order to ensure stability, scalability and sustainability in the controller.

Finally, results, test cases and test results are shown in order to support the feasibility of this design approach and the conclusions which has been found by this study.

ACRONYMS AND ABBREVIATIONS

ECU		Electronic Control Unit
SENT		Single Edge Nibble Transmission
SPI		Serial Peripheral Interface
OS		Operating System
RTOS		Real Time Operating System
HAL		Hardware Abstraction Layer
BIOS		Basic Input Output Software
SCI		Serial Communication Interface
AUTOSAR		AUTomotive Open System ARchitecture
BSW		Basic Software
RTE		Runtime Environment
MAL		Microcontroller Abstraction Layer
PIT		Periodic Interrupt Timer

FIGURES LIST

INTRODUCTION

SAE J2716, also known as SENT [1], is a point to point protocol, operating between a sensor and an Electronic Control Unit (ECU). The sensor emits data continuously over the link while the ECU receives and processes the data. SENT is a recently developed serial protocol used for sensors in the automotive industry. SENT exhibits several advantages over older serial links. It supports high resolution, is very reliable, immune to Electro Magnetic Perturbations and cost effective.

The SENT protocol is a one-way, asynchronous voltage interface which requires three wires: a signal line (low state < 0.5V, high state > 4.1V), a supply voltage line (5V) and a ground line.

As observed on the picture below, a SENT message consists of:

- A calibration/synchronization pulse with period of 56 clock ticks.
- One 4-bit status and a serial communication nibble pulse of 12 to 27 ticks
- A sequence from one to six 4-bit data nibble pulses (12 to 27 clock ticks each) representing the values of the signal(s) to be communicated. The number of nibbles is fixed for each application of the encoding scheme (i.e. throttle position sensors, mass air flow, etc.) but can vary between applications. For example, if two 12-bit values are transmitted, six nibbles will be communicated. One 4-bit checksum nibble pulse of 12 to 27 clock ticks and an optional pause pulse.

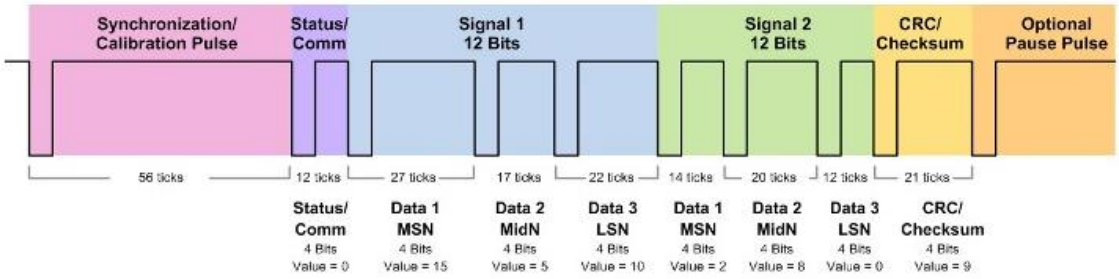


Figure 1: SENT Frame Description

In this work, the design of a driver compliant with SAE J2716 is proposed. The design covers in detail both configuration and status structures of the driver to ensure proper functionality. Also detailed information on the interfaces provided to upper layers is described.

Due to the increasing trend to move from single core to multi core arquitectures this implementation was made using an MC9S12XEP100 Microcontroller [2] with a XGATE co-processor to support parallelism of tasks and real time control algorithms. Design was made in order to ensure layered architecture to support single core implementation if needed, or if the platform microcontroller is replaced.

THEORETICAL FRAMEWORK

Layered Software Architecture

A multi-layer architecture consists in using different software layers to divide responsibilities among the whole system, which lead to the following advantages:

1. Clear separation of responsibilities – Each layer is in charge of an specific task
2. Exposed workflow – Clear and understandable code which leads to sustainability
3. Ability to replace a layer with minimum effort.
4. Portable code not a 100% dependent on the Microcontroller.

AUTOSAR Architecture

AUTOSAR [3] describes a software architecture dedicated for Automotive ECUs that have a Strong interaction between SW and HW, Communication networks and are Microcontroller based systems.

AUTOSAR architecture currently proposes 3 principal software layers as shown in the picture below:

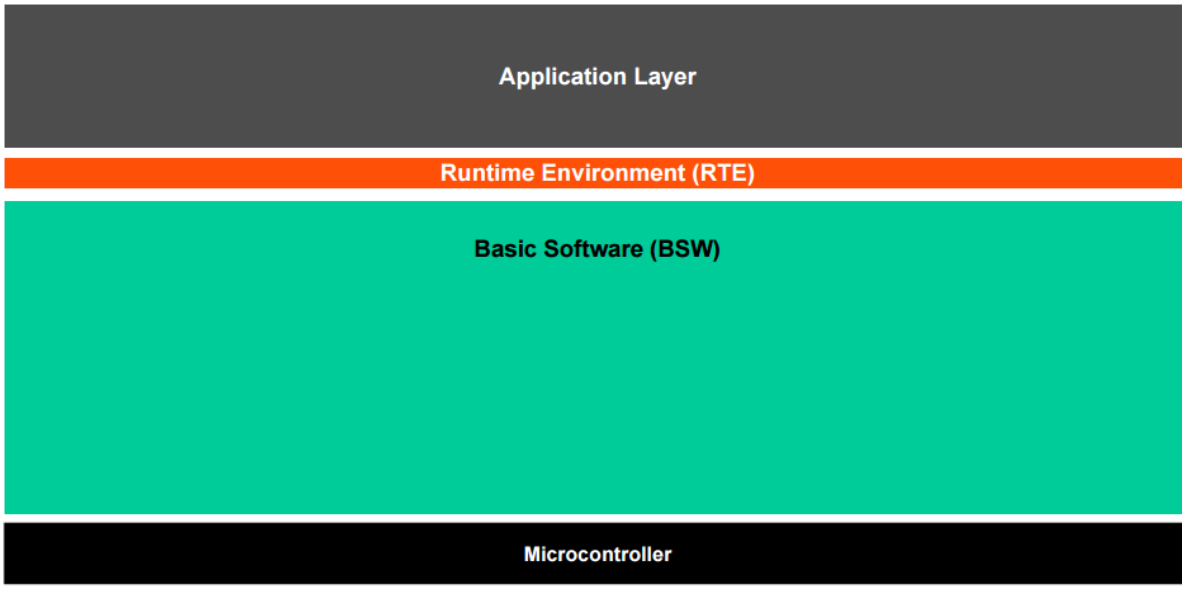


Figure 2: AUTOSAR Layered Architecture

Basic Software is then divided in the following sub-layers:

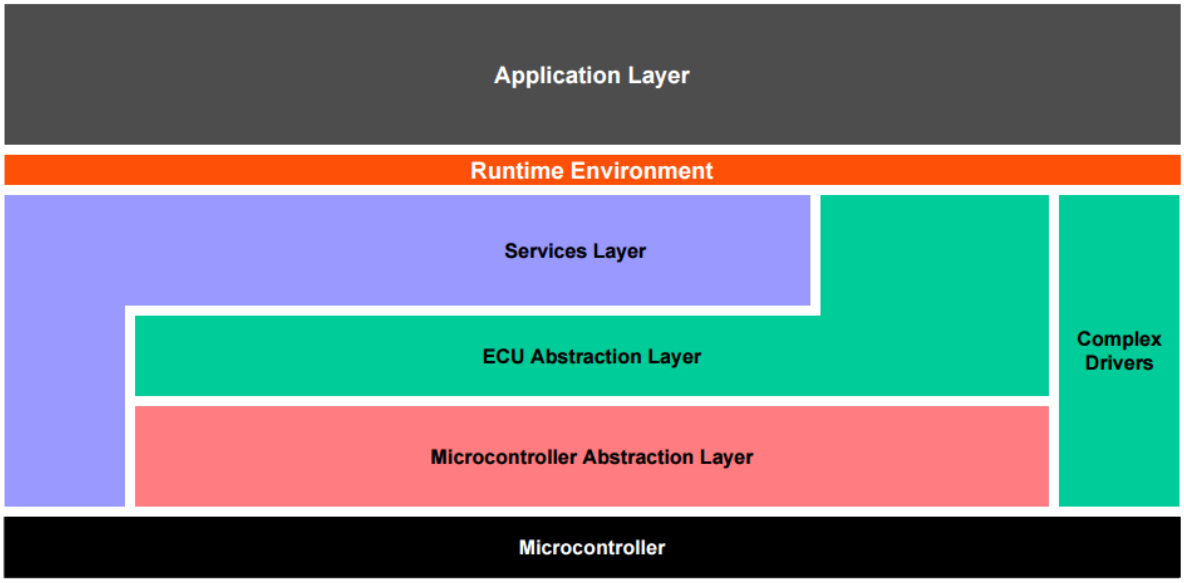


Figure 3: AUTOSAR BSW layered Architecture

Microcontroller Abstraction Layer (MAL)

The MAL layer contains the internal drivers, which are software modules that possess a direct access to the microcontroller. The task of this layer is to abstract the microcontroller from the rest of the software in such a way that only this layer has direct access to microcontroller registers and configurations. An advantage of this layer is that if the microcontroller wants to be replaced, only this layer needs to be replaced and minor changes need to be made to ensure full functionality.

ECU Abstraction Layer

This layer provides API's to access information or services provided by each individual driver.

Example:

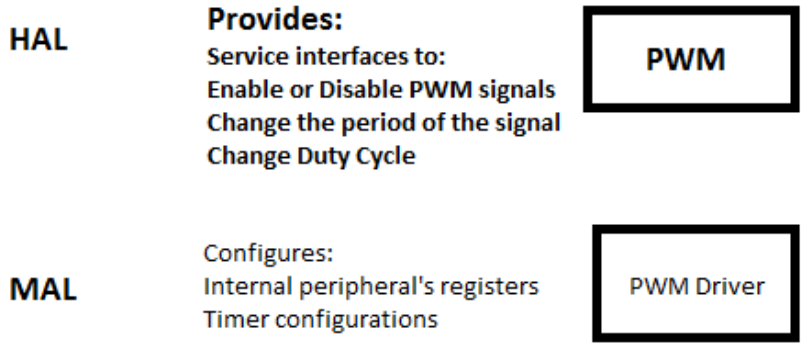


Figure 4: PWM Layered Driver Example

Complex Driver Layer

Provides a way to integrate drivers for special functionalities which have very high timing constraints or cannot be divided in layers. Implementation is Hardware dependent. Examples of this drivers are: Ignition, Injection and Engine position drivers.

Services Layer

This layer offers services such as:

- Real time operating system tasks (RTOS)
- Safety services
- Memory services

Driver configurability

When designing a device driver a set of requirements shall be met in order to ensure proper functionality. The first of these requirements is configurability. Configurability means that the driver will be designed in such a way that the user will be able to configure it to work accordingly to the project needs.

This provides advantages such as:

- The user does not need to fully understand how the driver works, only how to configure it.
- The driver can be re-used on other projects

This configurability is supported by defining configuration structures in which the user can decide different parameters on which the driver will work. For example, on a PWM driver the user could choose the following configurations:

- Type of PWM (A, B, C or D)
- Period
- Duty Cycle

The Configuration concept relies on the understanding that the design of the driver has to be generic and consider configurations given in the structure to write this data into the microcontroller registers.

Driver Scalability

Another requirement on the design of a device driver is scalability. The scalability concept means that the driver shall support the need to add more configuration levels on configuration structure. According to AUTOSAR, 3 basic levels of configuration are defined for a driver:

Driver Level:

This level refers to the whole system, in this configuration, pointers to the second configuration level are included as well as a variable to indicate how many configuration structures are on the second level.

Device Level:

This level refers to amount of physical controllers available on the microcontroller. In this configuration level, pointers to the third level of each device are included as well as a variable to indicate how many members are on the third level.

Channel Level:

This level refers to the amount of channels that are being configured on every particular device. In this level all configurations are included depending of the driver and the microcontroller.

Example:

For the design of a Serial communication Interface (SCI) Driver, we can consider the configuration levels as follows:

```
const tSCIdriver_config  SCI_config[] =
{
    {
        sizeof(SCI_device_cfg)/sizeof(SCI_device_cfg[0]),          /* number of configured devices */
        &SCI_device_cfg[0]                                         /* SCI devices configuration structure */
    }

};
```

Figure 5: SCI Driver Configuration Structure

It is observed that the first member of the configuration structure is a variable which will contain how many devices (physical controllers) are present on the driver itself. The second member of the structure is the pointer itself to the device structure.

```
const tSCIdevice_config  SCI_config_device[] =
{
    {
        sizeof(SCI_channel_cfg)/sizeof(SCI_channel_cfg[0]),          /* number of configured channels */
        &SCI_channel_cfg[0]                                          /* SCI channels configuration structure */
    }

};
```

Figure 6: SCI Device configuration structure

It is observed that, as in the Driver level, the first member will contain how many channels are configured on each device, and the second member will be the pointer to the device channels. In this configuration level there can be as many devices equals to the amount of physical controllers present on the microcontroller. If the microcontroller has 2 physical SCI controllers a configuration like the following could be observed:

```
const tSCIdevice_config  SCI_config_device[] =
{
    {
        sizeof(SCI_channel_cfg)/sizeof(SCI_channel_cfg[0]),          /* number of configured channels */
        &SCI_channel_cfg[0]                                          /* SCI channels configuration structure */
    },

    {
        sizeof(SCI_channel_cfg_2)/sizeof(SCI_channel_cfg_2[0]),      /*2nd device: number of configured channels */
        &SCI_channel_cfg_2[0]                                        /* SCI 2nd device channels configuration structure */
    }

};
```

Figure 7: SCI configuration with 2 devices

```
const tSCIchannel_config SCI_channel_cfg[] =
{
    {
        (UINT32)57600,              /* SCI_baudrate */
        (tCallbackFunction)NULL,    /* SCI_TX_callback */
        (tCallbackFunction)NULL,    /* SCI_RX_callback */
        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 */
    },
    {
        (UINT32)19200,              /* SCI_baudrate */
        (tCallbackFunction)NULL,    /* SCI_TX_callback */
        (tCallbackFunction)NULL,    /* SCI_RX_callback */
        SCI_CH1,                    /* 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 */
    },
};
```

Figure 8: SCI channel configuration structure

In this configuration level a structure for each channel is included with its particular configuration parameters. For this example, the SCI driver provides the possibility to configure for each channel: baudrate, Transmission callback function, Reception callback function, Enable or disable the Transmission and reception interrupts and the definition of the buffer size for both transmission and reception.

DESIGN AND IMPLEMENTATION

SENT was implemented as if the ECU was the smart sensor, hence, encoding an analog value obtained by an analog sensor and transmitting it through the SENT link.

The Driver was designed as HAL driver which relies on 2 different MCAL level drivers, one for Inputs and Outputs (GPIO module) another for the timer management: Periodic Interrupt Timer (PIT module). Data sent in the implementation is taken by the ADC but any other data can be used. The SENT driver should remain independent of these 2 modules and contain the logic to perform the ticks calculation to create the entire frame accessible from both cores.

Therefore, creation of 2 different Drivers to provide coherency to the needed actions was required. These different drivers will be described in the following sections.

PIT Driver:

The Periodic interrupt timer is a controller which consists in an array of 24 bit timers which is used to trigger time based interruptions.

In this implementation the main focus of the PIT is to raise interruptions based on the 3us configured tick, which will be attended by the XGATE core to remove load on the main CPU.

In order to manage configurability and scalability the following configuration structures were defined. These parameters take into consideration most of the available PIT configurations:

Data Type	Element	Description
enum tPIT_Channel	PIT_Channel	PIT Channel enumeration that correspond to the specified channel
UINT8	u8Timeout_Value	Numeric Value to be loaded in the Channel Counter
UINT8	u8Cntr_Frz	Enable/Disable Timer Freeze when debugging
UINT8	u8Microtimer_Mux	Microtimer multiplexor selection (at device level)
UINT8	u8PIT_Enable	Channel Enable/Disable
UINT8	u8Interrupt_Enable	Periodic Interrupt Enable/Disable
UINT8	u8CPU_Select	S12/XGATE interrupt mapping
tCallbackFunction	ptrCallbackFunction	Callback Function (if applicable)

Table 1: – tPIT_Channel_Config Structure, Channel Configuration

Data Type	Element	Description
enum tPIT_Device	PIT_ Device	PIT Device enumeration that correspond to the specified Device
UINT8	u8Microtimer_Base0	Numeric Value to be loaded in the Microtimer 0
UINT8	u8Microtimer_Base1	Numeric Value to be loaded in the Microtimer 1
const tPIT_Channel_Config*	ptrPIT_Channel_Config	Pointer to Configuration Channel structures.
UINT8	u8Number_of_Channels	Number of Channels configured per Device

Table 2: tPIT_Device_Config Structure, Device Configuration

Data Type	Element	Description
const tPIT_Device_Config*	ptrPIT_Device_Config;	Pointer to Configuration Device structures.
UINT8	u8Number_of_Devices	Number of Devices configured.

Table 3: tPIT_Driver_Config Structure, Driver Configuration

This configurations will provide the necessary parameters in order to configure the driver accordingly. In this case it was necessary to include the device structure, instead of going directly to the channels configurations, since there are 2 elements that are configurated in a device level, which are the Microtimer 0 and 1. These microtimer affects directly to all channels and therefore should be considered as device elements.

Due to protocol standard, it is required that SENT works in a 3-90 µS ticks period, which was achived with the configuration structures of the driver, modifing 2 differents macros called *PIT_MICROTIMER1_DIV* (MicroTimer 1 value) and *PIT_TARGET_PERIOD_US* which are mapped to the SENT macro *TICK_PERIOD* that manipulates the tick period.

It is needed to configurate the Microtimer load (PITMTLD), Multiplexor selection and Timeout Value (PITLD) accordingly, having in mind two contraints as per time-out period formula:

$$time - out\ period\ = \ \frac{(PITMTLD + 1) * (PITLD + 1)}{f_{BUS}} \quad \#1$$

$$where\ \ f_{BUS} = \ 24\ MHz \quad \#2$$

Converting the formula in function of Timeout Value (PITLD):

$$PITLD\ = \frac{time-out\ period * f_{BUS}}{(PITMTLD + 1)} - 1 \quad \#3$$

Calculating the constraint with the minimun value time-out period in this case (3 µS = $\frac{3}{1{,}000{,}000\ Hz}$) we had as result:

$$PITLD\ = \frac{\left(\frac{3}{1{,}000{,}000\ Hz}\right) * 24{,}000{,}000\ Hz}{(PITMTLD + 1)} - 1 \quad \#4$$

$$PITLD\ = \frac{3 * \left(\frac{24{,}000{,}000\ Hz}{1{,}000{,}000\ Hz}\right)}{(PITMTLD + 1)} - 1 \quad \#5$$

$$PITLD\ = \frac{3 * 24}{(PITMTLD + 1)} - 1 \quad \#6$$

This new formula gives an important constraint to be considered in the implementation. Since the 2 variables are integer the values need to be selected carefully, trying to select a (*PITMTLD + 1)* multiple of 3 will prevent cutting the value in the division part. the selected value was:

$$PITMTLD = 8 \quad \#7$$

$$PITLD = \frac{3*24}{9} - 1 : PITLD = 7 \quad \#8$$

Once the *PITMTLD* is selected with the smaller period we can implement a formula to calculate the *PITLD* according to the target tick period (SENT macro *TICK_PERIOD*), calculated by the PIT macro *PIT_VALUE2*:

$$PITLD = \frac{time-out\ period*24}{9} - 1 \quad \#9$$

It is important to mention that given the behaviour of this implemented formula, the maximum accurency between the target period and the real period will be achieved when a multiple of 3 is entered.

Once the configuration is performed, there will be another set of structures that will give us continuous status parameters which will allow us to trace the behaviour of the software. These structures are defined as follows:

Data Type	Element	Description
enum tPIT_Channel	PIT_Channel	PIT Channel enumeration that correspond to the specified channel
UINT8	u8Interruption_Enable	Indicates if the Channel Interrupt is Enable
UINT8	u8PIT_Enable	Indicates if the Channel is Enable
UINT8	u8Interruption_Flag	Indicates if there is an Interruption for the Channel
tCallbackFunction	ptrCallbackFunction	Callback Function (if applicable)

Table 4 : tPIT_Channel_Status Structure, Channel Status

Data Type	Element	Description
enum tPIT_Device	PIT_ Device	PIT Device enumeration that correspond to the specified Device
tPIT_Channel_Status*	ptrPIT_Channel_Status	Pointer to Channel Status structures.
UINT8	u8Number_of_Channels	Number of Channels configured per Device

Table 5: tPIT_Device_Status Structure, Device Status

Data Type	Element	Description
const tPIT_Device_Config*	ptrPIT_Device_Config;	Pointer to Configuration Device structures.
UINT8	u8Number_of_Devices	Number of Devices configured.

Table 6: tPIT_Driver_Status Structure, Driver Status

The API given by the PIT driver consist of 3 functions described bellow:

Name of the Function	**Void PIT_Init(const tPIT_Driver_Config* PIT_Driver);**
Input parameters	const tPIT_Driver_Config* - pointer to a driver configuration structure
Return value	void
Description	Configurate all necessary registers as speficed by the 3 configuration structures in the cnf files, this function also updates accordendly the ncessary status structures as the registers are being modify. It also calls another private function where the channels are configured separetly:

Table 7: PIT_Init Function Description

Name of the Function	**void vfnPIT_Start (const tPIT_Driver_Config* PIT_Driver);**
Input parameters	const tPIT_Driver_Config* - pointer to a driver configuration structure
Return value	void
Description	Enables the PIT Driver as a whole by modifying the PITE register and maps the interruption vector to the selected core in the configuraion structure.

Table 8: PIT_Start Function Description

Name of the Function	**void interrupt vfnSENT_PIT_Isr (void);**
Type	Interrupt
Input parameters	void
Return value	void
Description	Function mapped to XGATE_Interrupt_Vector, this serves as a interrupt handler for the PIT ISR used by the SENT driver, this interrupt contains all the SENT logic.

Table 9: SENT_PIT_Isr Function Description

Once the tick interruptions have been configured accordingly the SENT driver will take care of the logic to generate the frame, however, it is required to implement a GPIO driver to manipulate the logic levels on the communication link.

GPIO Driver:

The GPIO Driver will take care to configure each pin according to its configuration structure. For this implementation it provides a set of API's which will take care of toggling the communication line to generate the SENT frame.

For the GPIO driver the following structures where the ones that were considered necessary in order to provide full configurability to the driver

Data Type	Element	Description
UINT8	U8Channel	GPIO Channel enumeration that correspond to the specified channel
Enum GPIO_IO_State	IO_State	Determines the input/output configuration for the channel
UINT8	u8PIT_Enable	Indicates if the Channel is Enable

Table 10: Channel Configuration structure members

Data Type	Element	Description
Enum GPIO_IO_State	Config_Port	GPIO Device enumeration that correspond to the specified device (PORT)
tGPIO_Channel_Config*	GPIO_Channel_Config	Pointer to the Channel configuration structure
UINT8	u8GPIO_No_Of_Channels	Indicates how many channels are currently configured

Table 11: Device configuration structure members

Data Type	Element	Description
tGPIO_Device_Config*	GPIO_Device_Config	Pointer to the device configuration structure
UINT8	u8GPIO_No_Of_Devices	Pointer to the Channel configuration structure

Table 12: Driver configuration structure member

The API given by the GPIO driver consist of the functions described bellow:

Name of the Function	**void GPIO_Init(tGPIO_Driver_Config* GPIO_Driver_Config)**
Input parameters	tGPIO_Driver_Config* - pointer to driver configuration structure
Return value	void
Description	Configurate all necessary registers as speficed by the 3 configuration structures in the cnf files, this function also updates accordendly the ncessary status structures as the registers are being modify. It also calls another private function where the channels are configured separetly:

Name of the Function	**void GPIO_Toggle_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)**
Input parameters	Enum GPIO_Port- GPIO Portx which will be used Channel – PORTx pin which will be toggled
Return value	void
Description	Toggles the logic state on the GPIO Portx channel

Name of the Function	**void GPIO_Set_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)**
Input parameters	Enum GPIO_Port- GPIO Portx which will be used UINT8 channel – pin which will be set HIGH
Return value	void
Description	Sets a logic state HIGH on the PORTx pin

Name of the Function	**void GPIO_Clear_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)**
Input parameters	Enum GPIO_Port- GPIO Portx which will be used UINT8 channel – pin which will be set LOW
Return value	void
Description	Sets a logic state LOW on the PORTx pin

SENT Driver:

As it was mentioned above this driver was designed as a HAL driver that works based in the PIT module in order to generate a time base and an ADC in order to take the information and encrypt it in the SENT format. Below are the configuration structures that conform the SENT driver.

Data Type	Element	Description
tSENT_Channel_Config*	ptrSENT_Channel_Config	Pointer to the channel configuration structure
UINT8	u8Number_of_Devices	Variable that stores the number of SENT channels that are being configured.

Table 13: SENT driver configuration structure

Data Type	Element	Description
tSENT_Channel	SENT_Channel	Variable that stores the number of SENT channel that is being configured
UINT8	u8Tick_Period	Variable that stores the number of SENT channels that are being configured.
UINT8	u8Pause_Enable	Variable that configures whether a pause pulse is gonna be transmited or not.
UINT8	u8No_Nibbles	Number of nibbles that are gonna be transmitted (for this project this is forced to 6 since we have the constraint of sending 3 nibbles coming from the potentiometer and 3 nibbles coming from the photoresistor)
GPIO_Port	SENT_Port	Port in which the SENT frame is gonna be transmitted
UINT8	u8SENT_PIN	Pin in which the frame is gonna be transmitted

Table 14: SENT driver channel configuration structure

SENT Frame Generation

To ensure the precision of the algorithm we have a table that is being updated in each SENT frame. A pointer variable is defined pointing to the base of the table in order to have an index that implies which nibble is being managed at the moment. The table contains the following fields with the following information:

Field	Ticks
Calibration	Constant 56 tick value
Status	Constant 12 tick value
Nibble 1	Read value
Nibble 2	Read value
Nibble 3	Read value
Nibble 4	Read value
Nibble 5	Read value
Nibble 6	Read value
CRC	Calculated Cyclic Redundancy Check (CRC)
Pause pulse	Constant 77 tick value (optional by configuration)
End of frame	Auxiliary variable to detect the end of the table

Table 15: SENT Frame Table

The SENT algorithm works in the following way: Each time the PIT counter times out an interruption is generated. This occurs every 3us (depending on the configured value) this service is attended by the XGATE core. At the beginning it request the S12 the conversion of the information read by the ADC and then it update the required fields of the table in ticks. First it evaluates if the interruption has occurred 5 times in order to ensure the constant low drive. After the 5 ticks have passed the signal is pulled HIGH and keeps counting until it reaches the value in ticks of the current field of the table. When this value has been reached, the signal is pulled LOW, the counter is restarted and the pointer is incremented in such a way that we have visibility to the next field of the table. Once the end of the table is reached if the pause pulse is enabled it is generated and the pointer is reloaded to the start of the table. If the pause pulse it's not enabled, the pointer is reloaded automatically.

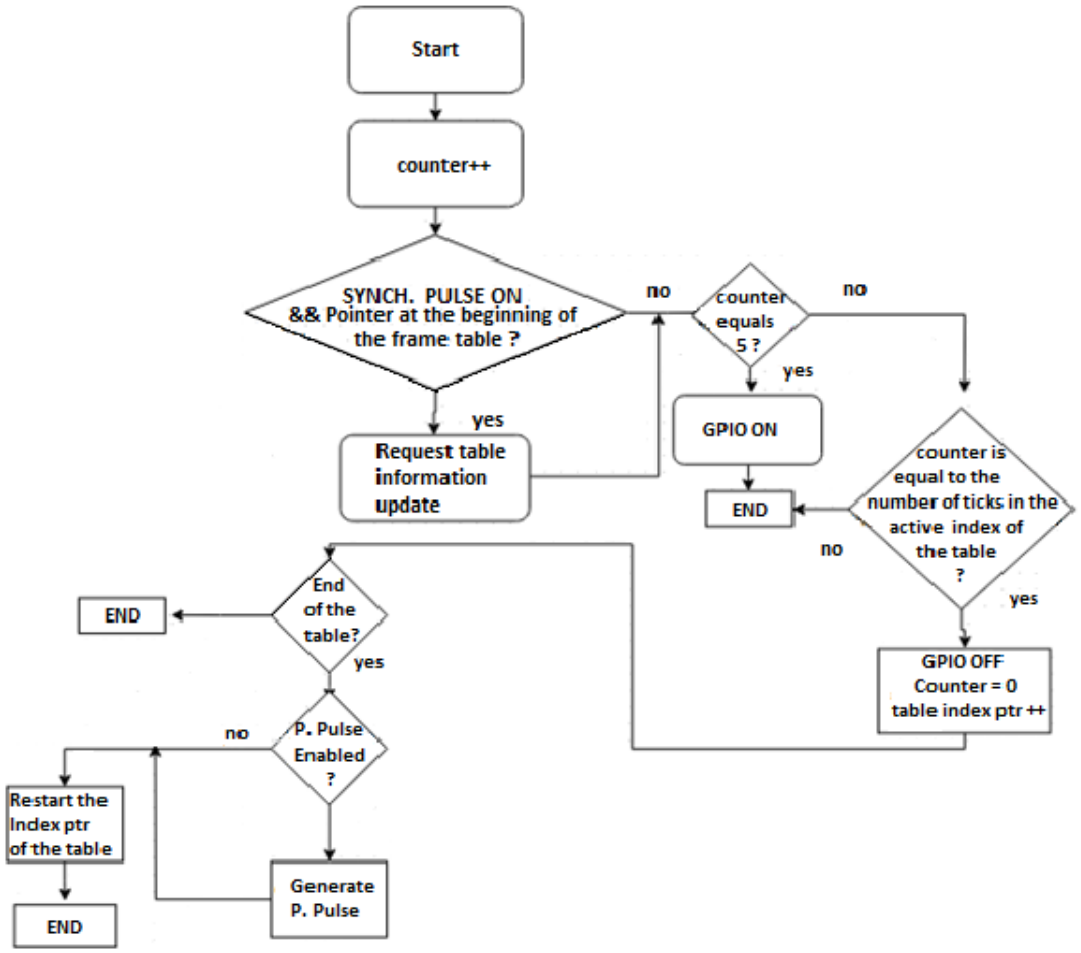


Figure 9: SENT Driver Frame generation Flowchart

DRIVER TESTING

In the scope captures we can clearly appreciate the generation of the SENT frame with all the desired characteristics

Tick : 3 us

Synchronization pulse

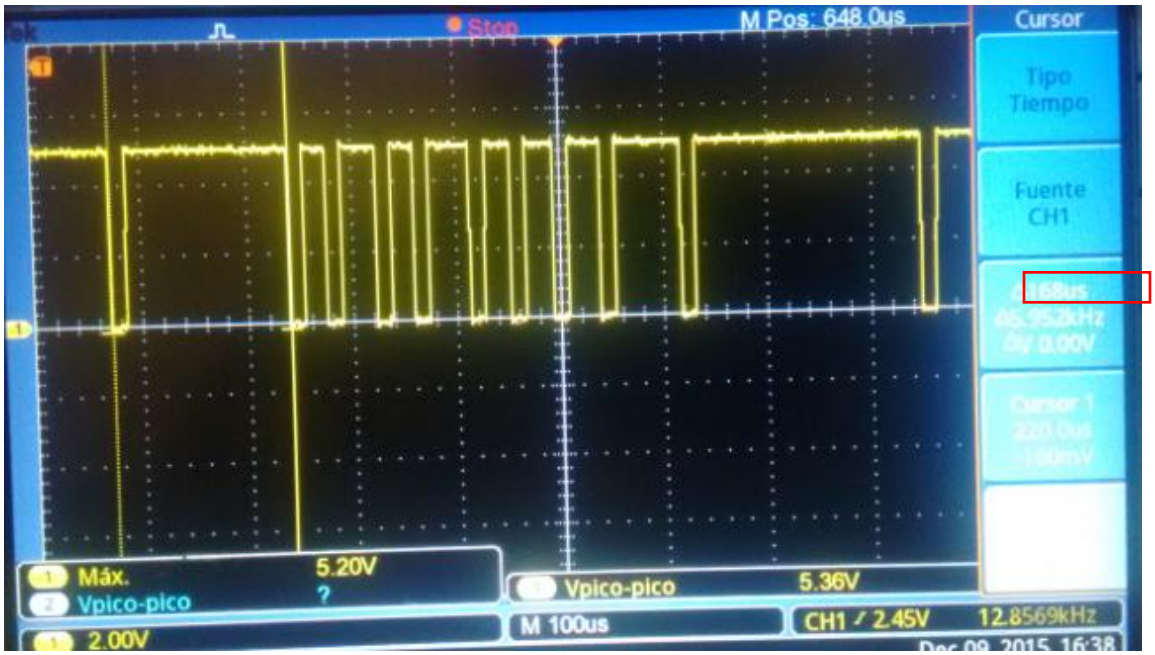


Figure 10: Synchronization Pulse Test Result

Pulse width is equal to 168 us as expected to the 56 ticks that the synchronization pulse must have.

Status pulse is equal to 12 ticks , as it is observed the measured value is correct.

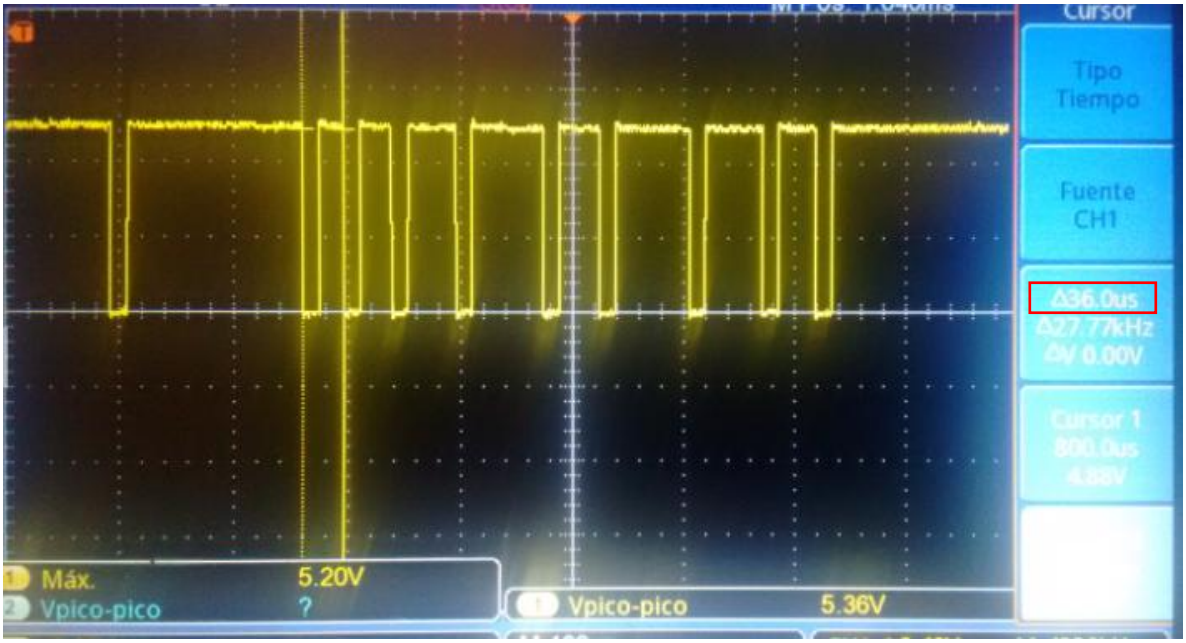


Figure 11: Sent Nibble Measurement

Due to the potentiometer and the photoresistor not giving stable measurements it is very difficult to make a measurement of a read value.

Here a Data nibble is shown

Figure 12: SENT nibble measurement 2

Low Drive time = 15 us 1)

High Drive time = 36us -15us = 21us 2)

High Drive in ticks = 21/3 = 7 ticks 3)

Low Drive in ticks = 15/3 = 5 ticks 4)

As we established that a “0” value is 12 ticks, we can calculate the value of the pulse in the following way:

Value = 7+5 -12 = 0 5)

CONCLUSIONS AND REFLEXIONS

As it was shown previously, it can be clearly appreciated that the driver worked as expected.

During the testing phase, it was observed that the driver behaved in a predictable manner, sending the encoded data in the SENT protocol over the course of time without showing any interruption in the transmission, thus, proving to be stable.

In this implementation, SENT driver was established as a HAL layer controller, in such a way that it relied on 2 low level drivers. It is important to emphasize that this comes as a problem in the implementation due to the usage of resources to implement this solution in comparison to other more commonly on chip serial protocols like SPI. However, in new generation controllers, SENT is now being added as an on chip protocol with dedicated resources, hence, enhancing the functionality and providing better performance. This solution proves to be reliable, however, a detailed analysis on resource allocation needs to be made first in order to ensure overall functionality.

REFERENCES

[1] SAE International - SENT - Single Edge Nibble Transmission for Automotive Applications, (Jan 27, 2010).

[2] MC9S12XEP100 Reference Manual Covers MC9S12XE Family, MC9S12XEP100RMV1 Rev. 1.25 (2013, February), Freescale.

[3] AUTOSAR Release 4.2 Overview and Revision History, (May 31, 2015), [Online] Available: https://www.autosar.org/fileadmin/files/releases/4-2/AUTOSAR_TR_ReleaseOverviewAndRevHistory.pdf

APPENDIX

SENT DRIVER – cnf_sent.c

```
/** Dynamic Memory allocation configuration definitions */
#include    "cnf_sent.h"

/**********Definition of  VARIABLEs - ***************/

/* SENT Channel Configuration */
#pragma DATA_SEG SHARED_DATA
const tSENT_Channel_Config SENT_Channel_Config [] =
{
 {
 SENT_Channel0,              /*tSENT_Channel */
 (UINT8) TICK_PERIOD,        /*u8Tick_Period*/
 (UINT8) ENABLE,             /*u8Pause_Enable*/
 (UINT8) NO_OF_NIBBLES,      /*u8No_Nibbles*/
 PORT_B,                     /*SENT_Port*/
 (UINT8)PIN0,                /*u8SENT_PIN*/
 },
```

```
/*Second Channel*/

//  {
//   SENT_Channel1,              /*tSENT_Channel */
//  (UINT8) TICK_PERIOD,         /*u8Tick_Period*/
//  (UINT8) ENABLE,              /*u8Pause_Enable*/
//  (UINT8) NO_OF_NIBBLES,       /*u8No_Nibbles*/
//  PORT_B,                      /*SENT_Port*/
//  (UINT8)PIN1,                 /*u8SENT_PIN*/
//  },


};
#pragma DATA_SEG DEFAULT

/* SENT Driver Configuration */
#pragma DATA_SEG SHARED_DATA
const tSENT_Driver_Config SENT_Driver_Config [] =
{
  {
   &SENT_Channel_Config[0],     /*SENT Channel Configuration Structure*/
   sizeof(SENT_Channel_Config)/sizeof(SENT_Channel_Config[0]),        /* number of configured channels */
  },
};
#pragma DATA_SEG DEFAULT
```

SENT DRIVER – cnf_sent.h file

```
/******************/
#ifndef __CNF_SENT_H        /*prevent duplicated includes*/
#define __CNF_SENT_H

/*********Include files*************/

/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"

/** Used Modules */
/** PLL definitions and prototypes*/
  #include    "cnf_gpio.h"

/************Declaration of module wide TYPES*************/
typedef enum{
  SENT_Channel0,
  SENT_Channel1,
  SENT_ChannelNull,
}tSENT_Channel;
```

```
/*------------------------------ Configuration Structures ----------------------------------------*/

typedef struct {
  enum tSENT_Channel SENT_Channel;
  UINT8 u8Tick_Period;
  UINT8 u8Pause_Enable;
  UINT8 u8No_Nibbles;
  enum GPIO_Port SENT_Port;
  UINT8 u8SENT_PIN;
}tSENT_Channel_Config;

typedef struct {
  const tSENT_Channel_Config * ptrSENT_Channel_Config;
  UINT8 u8Number_of_Devices;
}tSENT_Driver_Config;

/*************************Definition of  VARIABLEs - ****************/

#pragma DATA_SEG SHARED_DATA
extern const tSENT_Driver_Config SENT_Driver_Config[];

#pragma DATA_SEG DEFAULT
```

```
/*************Definition of module wide MACROS / #DEFINE-CONSTANTS***************/

/** Periodic Interrupt Timer definitions */
#define    TICK_PERIOD          3    /* Period bewteen 3 - 90 uS*/
#define    NO_OF_NIBBLES        6
#define    NO_CHANNELS          1

#endif /* __CNF_SENT_H */
```

SENT Driver – sent_protocol.c

```
/** Used headers*/
/** PIT driver header*/
#include    "pit_2.h"
/** IO driver header*/
#include    "io.h"
#include "ADC.h"
#include "xgate_config.h"
#include "interrupt.h"
#include "sent_protocol.h"

/** Used modules */
/***********Declaration of module wide FUNCTIONs **************/

/** Privet Initialization of SENT Driver*/
void vfnSENT_Driver_Init (const tSENT_Driver_Config* ptrSENT_Driver);

/** CheckSum Calculation */
/**********Definition of module wide MACROs / #DEFINE-CONSTANTs ************/
#define SENT_SEED        5
 const UINT8 CRCLoookUp [16] = {0, 13, 7, 10, 14, 3, 9, 4, 1, 12, 6, 11, 15, 2, 8, 5};
```

```
/*******Declaration of module wide TYPEs ***************/

/*------------------------------- SENT FRAME ------------------------------------------------*/
#pragma DATA_SEG SHARED_DATA
/* SENT Array Frame */
UINT8 u8SENT_Frame [11] =
{ SENT_CALIBRATION, 12, 16, 12, 18, 13, 14, 15, 26, 77, SENT_END_OF_FRAME };

/* SENT Status structure*/
tSENT_Driver_Status SENT_Driver_Status;

tSENT_Channel_Status SENT_Channel_Status [NO_CHANNELS] =
{
  {SENT_ChannelNull, NULL, DISABLED, LOW},
};

#pragma DATA_SEG DEFAULT

UINT16 SENT_ADC0 = 0;
UINT16 SENT_ADC1 = 0;

#pragma DATA_SEG SHARED_DATA
UINT8 SENT_Pause_Enable[2];
UINT8 SENT_cPORT[2];
UINT8 SENT_PIN[2];
UINT8 SENT_Devices;
#pragma DATA_SEG DEFAULT
```

```
void vfnSENT_Init (const tSENT_Driver_Config* ptrSENT_Driver)
{

  /* Initialize ADC channels used by SENT*/
  HAL_ADC_Init();

  /* Initialize PIT channel used by SENT*/
  vfnPIT_Init(PIT_Driver_Config);
  vfnPIT_Start (PIT_Driver_Config);

  /* Initialize SENT driver*/
  /* Initializing Driver Status structure */
  vfnSENT_Driver_Init (ptrSENT_Driver);

}
```

```
void vfnSENT_Driver_Init (const tSENT_Driver_Config* ptrSENT_Driver)
{
  UINT8 u8Index;

  /* Lock Hardware Semaphore for SENT structure */
  XGATE_lock_hw_semaphore_3( );

  /* Initializing Driver Status structure */
  SENT_Driver_Status.ptrSENT_Channel_Status = &SENT_Channel_Status[0];

  for (u8Index = 0; u8Index < ptrSENT_Driver->u8Number_of_Devices; u8Index++)
  {
    /* Channel enumeration assign*/
    SENT_Driver_Status.ptrSENT_Channel_Status[u8Index].SENT_Channel = ptrSENT_Driver-
>ptrSENT_Channel_Config[u8Index].SENT_Channel;
    /* Frame Total Lenght Calculation*/
    SENT_Driver_Status.ptrSENT_Channel_Status[u8Index].Frame_Total_Lenght = 3 +
(ptrSENT_Driver->ptrSENT_Channel_Config[u8Index].u8No_Nibbles) + \
                                                (ptrSENT_Driver-
>ptrSENT_Channel_Config[u8Index].u8Pause_Enable);
    /* Output Initial State LOW*/
    SENT_Driver_Status.ptrSENT_Channel_Status[u8Index].Output_State = LOW;
    /* Pointer to SENT Frame Array*/
    SENT_Driver_Status.ptrSENT_Channel_Status[u8Index].ptrFrame_Table = &u8SENT_Frame[0];
    /* Increment SENT channels*/
    SENT_Driver_Status.u8Number_of_Devices++;
```

```
    /* SENT Pause Pulse enabled*/
   SENT_Pause_Enable[u8Index]  = ptrSENT_Driver-
>ptrSENT_Channel_Config[u8Index].u8Pause_Enable;

   SENT_cPORT[u8Index]  = ptrSENT_Driver->ptrSENT_Channel_Config[u8Index].SENT_Port;
   SENT_PIN[u8Index]  = ptrSENT_Driver->ptrSENT_Channel_Config[u8Index].u8SENT_PIN;
   SENT_Devices = ptrSENT_Driver->u8Number_of_Devices;
  }
  /* Realese Hardware Semaphore for SENT structure */
  XGATE_release_hw_semaphore_3( );
}

UINT8 u8SENT_Calc_CRC (UINT8* ptr_Table)
{
  UINT8 i = 0;
  UINT8 u8CheckSum = SENT_SEED;

  for (i=SENT_Status; i<=7; i++){
    u8CheckSum ^= ptr_Table[i];
    u8CheckSum = CRCLoookUp[u8CheckSum];
  }

  return u8CheckSum;

}

/**
```

```
void  vfnSENT_Frame_Update (void)
{
  UINT8 i;
  /* Lock Hardware Semaphore for SENT Frame */
  XGATE_lock_hw_semaphore_2( );
  /* SENT Frame writting*/
  u8SENT_Frame [SENT_Status] = 0;
  /*Mask for Nibbles*/
  for (i = 0; i <3; i++)
  {
    u8SENT_Frame [SENT_Nibble0 + i] = (UINT8)(ADC_PotValue & SENT_NIBBLE_MASK);
    ADC_PotValue = ADC_PotValue>>4;
  }
    /*Mask for Nibbles*/
  for (i = 0; i <3; i++)
  {
    u8SENT_Frame [SENT_Nibble3 + i] = (UINT8)(ADC_FotoValue & SENT_NIBBLE_MASK);
    ADC_FotoValue = ADC_FotoValue>>4;
  }
  u8SENT_Frame [SENT_CRC] = (u8SENT_Calc_CRC (&u8SENT_Frame[0]));
  u8SENT_Frame [SENT_Pause] = SENT_PAUSE_PULSE;
  for (i = 1; i < 9; i++)
  u8SENT_Frame[i] = u8SENT_Frame[i] + SENT_NIBBLE_ZERO;

  /* Release Hardware Semaphore for SENT Frame */
  XGATE_release_hw_semaphore_2( );
}
```

```
#pragma CODE_SEG __NEAR_SEG NON_BANKED
void interrupt vfnSENT_SW_Trigger (void)
{
  vfnSENT_Frame_Update ();
  XGATE_SW_TRIGGER(SOFTWARE_TRIGGER_2,SOFTWARE_TRIGGER_DISABLE);
}
#pragma CODE_SEG DEFAULT
```

SENT DRIVER – sent_protocol.cxgate

```
/** Own headers */

/** SENT communication protocol definitions */

#include    "sent_protocol.h"


/** Used modules */

#include "xgate_config.h"

#include "pit_2.h"

#include "gpio2.h"

#include "io.h"

#include "cnf_sent.h"


/********* Declaration of module wide FUNCTIONs *************/


#pragma DATA_SEG XGATE_DATA

UINT8* ptrSENT_Frame = &u8SENT_Frame[0];

UINT8 u8Index = 0;

UINT8  u8Tick_Counter = 0;

UINT8 Sent_State =  SENT_SYNC_STATE_ON;

#pragma DATA_SEG DEFAULT
```

```
#pragma CODE_SEG XGATE_CODE
void interrupt vfnSENT_PIT_Isr (void)
{
  /* Clear PIT interrupt flag */
  PIT_WRITE_TF(PIT_Channel2, ENABLE);
  vfnSENT_Frame_Generation();
}
#pragma CODE_SEG DEFAULT
```

```
#pragma CODE_SEG XGATE_CODE
void vfnSENT_Frame_Generation(void){
/* ~~~~~~~~~~~~~~ Table Update Handler ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
 /* update how many ticks have passed */
 u8Tick_Counter++;
 /* if we are in the synchronization request the update of the table*/
 if((Sent_State == SENT_SYNC_STATE_ON) && (ptrSENT_Frame == &u8SENT_Frame[0]))
  {
   XGATE_SW_TRIGGER_X (SOFTWARE_TRIGGER_2, SOFTWARE_TRIGGER_ENABLE);
   Sent_State = SENT_SYNC_STATE_OFF;
  }
 /* determine if the pin must be driven LOW or HIGH */
  if(u8Tick_Counter == 5)
  {
   if(SENT_cPORT[0])
   SET_PORTB_PTB0(ON);

   if(SENT_cPORT[1])
   SET_PORTB_PTB1(ON);
  }
  else  if (u8Tick_Counter == *ptrSENT_Frame)
  {
   if(SENT_cPORT[0])
   SET_PORTB_PTB0(OFF);
   if(SENT_cPORT[1])
   SET_PORTB_PTB1(OFF);
   u8Tick_Counter = 0;
   ptrSENT_Frame++;
```

```
    /* verify if the pause pulse is enabled */
    if((ptrSENT_Frame == &u8SENT_Frame[9]) && SENT_Pause_Enable[0] == 0)
    {
    Sent_State = SENT_SYNC_STATE_ON;
    ptrSENT_Frame = &u8SENT_Frame[0];
    }
    /* once the end of frame is reached reload the pointer */
    else if(ptrSENT_Frame == &u8SENT_Frame[10])
    {
     Sent_State = SENT_SYNC_STATE_ON;
     ptrSENT_Frame = &u8SENT_Frame[0];
    }
   }
  }

#pragma CODE_SEG DEFAULT
```

PIT DRIVER – PIT2.c file

```
/** Configuration Options */
        #include        "configuration.h"
/** S12X derivative information */
        #include        __MCU_DERIVATIVE
/** Variable types */
        #include        "typedefs.h"
/** PIT configuration*/
  #include "pit_config.h"
/** PIT routines prototypes*/
  #include "pit_2.h"
  #include "xgate_config.h"
/********Definition of module wide VARIABLEs ****/
/*-- Configuration Variables---*/
#pragma DATA_SEG SHARED_DATA
tPIT_Driver_Status PIT_Driver_Status;
tPIT_Device_Status PIT_Device_Status [CONFIG_DEVICES] =
{
  {PIT_DeviceNull, NULL, DISABLED}
};
tPIT_Channel_Status PIT_Channel_Status [CONFIG_CHANNELS] =
{
  {PIT_ChannelNull, DISABLED, DISABLED, DISABLED, NULL}
};
#pragma DATA_SEG DEFAULT

/******* Definitions of module wide FUNCTIONs *****************/
```

```
void vfnPIT_Channel_Init ( const tPIT_Channel_Config* PIT_Channel, tPIT_Channel_Status*
Channel_Status);


/*******Code of module wide FUNCTIONS**************/


void vfnPIT_Init(const tPIT_Driver_Config* PIT_Driver)
{
  UINT8 u8Index;
  /* Initializing Driver Status structure */
  PIT_Driver_Status.ptrPIT_Device_Status = &PIT_Device_Status[0];
  for(u8Index = 0; u8Index < PIT_Driver->u8Number_of_Devices; u8Index++)
  {
     UINT8 u8IndexChannel;
     /* Disable Periodic Interrup Timer */
     PITCFLMT_PITE       =  DISABLED;
     /* Initializing Device Status structure*/
     PIT_Driver_Status.ptrPIT_Device_Status[u8Index].ptrPIT_Channel_Status =
&PIT_Channel_Status[0];
     for (u8IndexChannel = 0; u8IndexChannel < PIT_Driver-
>ptrPIT_Device_Config[u8Index].u8Number_of_Channels; u8IndexChannel++)
     {
        /* Call of Channel Initialization routine*/
        vfnPIT_Channel_Init (PIT_Driver->ptrPIT_Device_Config[u8Index].ptrPIT_Channel_Config,
&PIT_Driver_Status.ptrPIT_Device_Status[u8Index].ptrPIT_Channel_Status[u8IndexChannel]);
        /* Callback assignation per channel */

PIT_Driver_Status.ptrPIT_Device_Status[u8Index].ptrPIT_Channel_Status[u8IndexChannel].ptrCallbac
kFunction = \
        PIT_Driver-
>ptrPIT_Device_Config[u8Index].ptrPIT_Channel_Config[u8IndexChannel].ptrCallbackFunction;
        /* Increase number of configured channels*/
```

```
        PIT_Driver_Status.ptrPIT_Device_Status[u8Index].u8Number_of_Channels++;
    }
    /* Device enumeration assign*/
    PIT_Driver_Status.ptrPIT_Device_Status[u8Index].PIT_Device = PIT_Driver-
>ptrPIT_Device_Config[u8Index].PIT_Device;
    /* Microtimer count load*/
    PIT_WRITE_MTLD(MUX_TIMER0,PIT_Driver-
>ptrPIT_Device_Config[u8Index].u8Microtimer_Base0);
    PIT_WRITE_MTLD(MUX_TIMER1,PIT_Driver-
>ptrPIT_Device_Config[u8Index].u8Microtimer_Base1);

  /* Increase number of configured device*/
    PIT_Driver_Status.u8Number_of_Devices++;
 }
}
```

```
void vfnPIT_Channel_Init ( const tPIT_Channel_Config* PIT_Channel, tPIT_Channel_Status*
Channel_Status)
{
  Channel_Status->PIT_Channel = PIT_Channel->PIT_Channel;
  /* Enables PIT channel */
  PIT_WRITE_CE(PIT_Channel->PIT_Channel, PIT_Channel->u8PIT_Enable);
  Channel_Status->u8PIT_Enable = ENABLE;
  /* 16-bit timer route to micro time base */
  PIT_WRITE_MUX(PIT_Channel->PIT_Channel, PIT_Channel->u8Microtimer_Mux);
  /* Clear PTI interrupt flag */
  PIT_WRITE_TF(PIT_Channel->PIT_Channel, ENABLE);
  Channel_Status->u8Interruption_Flag = CLEAR;
  /* Precalculated PIT count down value */
  PIT_WRITE_LD(PIT_Channel->PIT_Channel, PIT_Channel->u8Timeout_Value);
  /* Access to configuration data registers for interrupts */
  INT_CFADDR            = 0x70;  /* with vectors from 0xFF70 to 0xFF7E */
  if (PIT_Channel->u8CPU_Select ==__XGATE_CORE)
  {
        PIT_INT_CFDATA(PIT_Channel->PIT_Channel) = 0x86;          /* Periodic Interrupt Timer
Channel, CPU handled, priority 6 */
  }
  else
  {
        PIT_INT_CFDATA(PIT_Channel->PIT_Channel) = 0x06;          /* Periodic Interrupt Timer
Channel, S12 Core handled, priority 6 */
  }
  /* Interrupt of PIT channel is enabled */
  PIT_WRITE_INTE(PIT_Channel->PIT_Channel, PIT_Channel->u8Interrupt_Enable);
  Channel_Status->u8Interruption_Enable = ENABLE;}
```

```
void vfnPIT_Start (const tPIT_Driver_Config* PIT_Driver)
{
  UINT8 u8Index;

  /* Disable Periodic Interrup Timer */
  PITCFLMT_PITE       =  DISABLED;

  for(u8Index = 0; u8Index < PIT_Driver->u8Number_of_Devices; u8Index++)
  {
     UINT8 u8IndexChannel;

     /* Load 8-bit microtimer load into the 8-bit micro timer down-counter */
     PIT_WRITE_CFLMT(MUX_TIMER0,ENABLE);
     PIT_WRITE_CFLMT(MUX_TIMER1,ENABLE);

     for (u8IndexChannel = 0; u8IndexChannel < PIT_Driver-
>ptrPIT_Device_Config[u8Index].u8Number_of_Channels; u8IndexChannel++)
     {
        /* Load 16-bit timer load into the 16-bit timer down-counter*/
        PIT_WRITE_FLT(PIT_Driver->ptrPIT_Device_Config[u8Index].ptrPIT_Channel_Config-
>PIT_Channel, ENABLE);
        /* Enable Periodic Interrup Timer */
        PITCFLMT_PITE       = ENABLE;
     }
  }
}
```

PIT DRIVER - PIT2.h file

```
/**********/
#ifndef __PIT_2_H        /*prevent duplicated includes*/
#define __PIT_2_H
/****** Include files**********/

/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"
/** PIT configuration definitions */
  #include "pit_config.h"
/*************** Declaration of module wide TYPEs ***********/
/*-- Register definitions ------------------------------------------------*/

#define PIT_WRITE_CE(NodeIndex, Value)      (PITCE |= (UINT8)Value << NodeIndex)
#define PIT_WRITE_MUX(NodeIndex, Value)     (PITMUX |= (UINT8)Value << NodeIndex)
#define PIT_WRITE_TF(NodeIndex, Value)      (PITTF |= (UINT8)Value << NodeIndex)
#define PIT_WRITE_INTE(NodeIndex, Value)    (PITINTE |= (UINT8)Value << NodeIndex)
#define PIT_WRITE_LD(NodeIndex, Value)      (*((&PITLD0) + (2u * NodeIndex)) = (UINT16)Value)
#define PIT_WRITE_MTLD(MuxIndex, Value)     (*((&PITMTLD0) + (1u * MuxIndex)) = (UINT8)
Value)
#define PIT_WRITE_CFLMT(MuxIndex, Enable)   (PITCFLMT |= (UINT8)Enable << MuxIndex)
#define PIT_WRITE_FLT(NodeIndex, Enable)    (PITFLT |= (UINT8)Enable << NodeIndex)
```

```
#define PIT_INT_CFDATA(NodeIndex)          (*((&INT_CFDATA5) - (NodeIndex)))
#define PIT_READ_TF(NodeIndex, Value)      (*(&PITTF) && ((UINT8)Value << NodeIndex))
/*---------------------------------- Status Structures ----------------------------------------------*/

typedef struct{
  enum tPIT_Channel PIT_Channel;
  UINT8 u8Interruption_Enable;
  UINT8 u8PIT_Enable;
  UINT8 u8Interruption_Flag;
  tCallbackFunction ptrCallbackFunction;
}tPIT_Channel_Status;

typedef struct{
  enum tPIT_Device PIT_Device;
  tPIT_Channel_Status * ptrPIT_Channel_Status;
  UINT8 u8Number_of_Channels;
}tPIT_Device_Status;

typedef struct{
  tPIT_Device_Status * ptrPIT_Device_Status;
  UINT8 u8Number_of_Devices;
}tPIT_Driver_Status;
```

```
/************************** Module Wide Functions ¨*********************************/
/* PIT Initialization*/
void vfnPIT_Init(const tPIT_Driver_Config* PIT_Driver);
/* PIT Countdown Start*/
void vfnPIT_Start (const tPIT_Driver_Config* PIT_Driver);

/*******************************************************************************/

#endif
```

PIT DRIVER - PIT_config.c file

```
/************* Include files**************/
/** PIT configuration definitions */
#include    "pit_config.h"


/* PIT Driver Configuration */
const tPIT_Channel_Config PIT_Channel_Config [] =
{
  {
    PIT_Channel2,       /*PIT Channel to be configured*/
    (UINT16) PIT_VALUE2,  /*PIT Timeout Value*/
    (UINT8) ENABLE,     /*Control Freeze*/
    (UINT8) MUX_TIMER1, /*PIT Multiplex select*/
    (UINT8) ENABLE,     /*PIT Enable*/
    (UINT8) ENABLE,     /*PIT Interrupt Enable*/
    (UINT8) __XGATE_CORE, /*CPU select*/
    (tCallbackFunction) NULL, /*Callback Function*/
  },
};
const tPIT_Device_Config PIT_Device_Config [] =
{ {
    PIT_Device0,                /*PIT Device to be configured*/
    (UINT8)PIT_MICROTIMER_DIV,  /*PIT Microtimer Base 0 Frequency*/
    (UINT8)PIT_MICROTIMER1_DIV, /*PIT Microtimer Base 1 Frequency*/
    &PIT_Channel_Config[0],     /*PIT Channel Configuration Structure*/
    sizeof(PIT_Channel_Config)/sizeof(PIT_Channel_Config[0]),     /* number of configured channels */
  },
};
```

```
const tPIT_Driver_Config PIT_Driver_Config [] =
{
  {
    &PIT_Device_Config[0],     /*PIT Device Configuration Structure*/
    sizeof(PIT_Device_Config)/sizeof(PIT_Device_Config[0]),         /* number of configured devices */
  },
};
```

PIT DRIVER - PIT_config.h file

```
/*****************/
#ifndef __PIT_CONFIG_H        /*prevent duplicated includes*/
#define __PIT_CONFIG_H

/************* Include files*************/
/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"

/** Used Modules */
/** PLL definitions and prototypes*/
  #include    "cnf_sent.h"
/*****************Declaration of module wide TYPES***********/
typedef enum{
   PIT_Device0,
   PIT_DeviceNull
}tPIT_Device;
```

```
typedef enum{
  PIT_Channel0,
  PIT_Channel1,
  PIT_Channel2,
  PIT_Channel3,
  PIT_Channel4,
  PIT_Channel5,
  PIT_Channel6,
  PIT_Channel7,
  PIT_ChannelNull
}tPIT_Channel;

/*------------------------------- Configuration Structures ----------------------------------------*/

typedef struct {
  enum tPIT_Channel PIT_Channel;
  UINT16 u8Timeout_Value;
  UINT8 u8Cntr_Frz;
  UINT8 u8Microtimer_Mux;
  UINT8 u8PIT_Enable;
  UINT8 u8Interrupt_Enable;
  UINT8 u8CPU_Select;
  tCallbackFunction ptrCallbackFunction;
}tPIT_Channel_Config;
```

```
typedef struct{
  enum tPIT_Device PIT_Device;
  UINT8 u8Microtimer_Base0;
  UINT8 u8Microtimer_Base1;
  const tPIT_Channel_Config * ptrPIT_Channel_Config;
  UINT8 u8Number_of_Channels;
}tPIT_Device_Config;

typedef struct {
  const tPIT_Device_Config * ptrPIT_Device_Config;
  UINT8 u8Number_of_Devices;
}tPIT_Driver_Config;



/*******Definition of  VARIABLEs - ***************/
extern const tPIT_Driver_Config PIT_Driver_Config [];
/**************Definition of module wide MACROS / #DEFINE-CONSTANTS *************/
#define    CONFIG_CHANNELS               1u
#define    CONFIG_DEVICES                1u

#define    MUX_TIMER0                    0u
#define    MUX_TIMER1                    1u
#define    BUS_FREQ                   24000000

/** Periodic Interrupt Timer definitions */
#define    PIT_MICROTIMER_PERIOD          24000000
```

```
/** Periodic Interrupt Timer macros */

#define    PIT_MICROTIMER_DIV              ( UINT8 ) ( ( BUS_FREQ /
PIT_MICROTIMER_PERIOD ) - 1 )



/~~~~~~~~~~~~ SENT DRIVER ~~~~~~~~~~~~~~~~~~~*/

/** Periodic Interrupt Timer definitions */

#define    PIT_TARGET_PERIOD_US                TICK_PERIOD


/** Periodic Interrupt Timer macros */

#define    PIT_MICROTIMER1_DIV             (UINT8) 8

#define    PIT_VALUE2                  (UINT8)(((PIT_TARGET_PERIOD_US *
(BUS_FREQ/1000000)) / (PIT_MICROTIMER1_DIV + 1))- 1)


#endif /* __PIT_CONFIG_H */
```

GPIO DRIVER - Gpio.cxgate file

```
/***************** Include files******************/

/** Configuration Options */
      #include     "configuration.h"
/** S12X derivative information */
      #include     __MCU_DERIVATIVE
/** Variable types */
      #include     "typedefs.h"

  #include  "xgate_config.h"

  #include "cnf_gpio.h"
  #include "gpio2.h"


#pragma DATA_SEG XGATE_DATA
 tGPIO_Driver_Config* Driver_Configuration;
#pragma DATA_SEG DEFAULT


/******** Definitions of module wide FUNCTIONs **************/


#pragma CODE_SEG XGATE_CODE

void GPIO_Init(tGPIO_Driver_Config* GPIO_Driver_Config){
  UINT8 u8Index;

 Driver_Configuration = GPIO_Driver_Config;

 for(u8Index = 0; u8Index<Driver_Configuration->u8GPIO_No_Of_Devices;u8Index++)
  {
       GPIO_Port_Config(Driver_Configuration-
>GPIO_Device_Config[u8Index].Config_PORT);
  }
}

#pragma CODE_SEG DEFAULT


#pragma CODE_SEG XGATE_CODE

void GPIO_Port_Config(enum GPIO_Port IO_PORT)
{
    UINT8 u8Index_1;
    UINT8 u8Index_2;

for(u8Index_1 = 0; u8Index_1<Driver_Configuration-
>u8GPIO_No_Of_Devices;u8Index_1++)
  {
    if( IO_PORT == Driver_Configuration-
>GPIO_Device_Config[u8Index_1].Config_PORT)
     {
```

```
          for(u8Index_2 = 0; u8Index_2<Driver_Configuration-
>GPIO_Device_Config[u8Index_1].u8GPIO_No_Of_Channels ; u8Index_2++)
          {
              *(Register_GPIO_Config[IO_PORT].PORT_DDR) |=
(Driver_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].IO_State)<<(Driver
_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].u8Channel);
              *(Register_GPIO_Config[IO_PORT].PORT_DR)  |=
(Driver_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].Logic_State)<<(Dri
ver_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].u8Channel);
          }
     }
  }
}

#pragma CODE_SEG DEFAULT



/*This function Toogles a certain bit of a certain PORT */
/*To be used as a part of the PWM Driver*/

#pragma CODE_SEG XGATE_CODE

void GPIO_Toggle_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{
     UINT8 Temp;

      Temp = (*(Register_GPIO_Config[Config_PORT].PORT_DR))&(1<<u8Channel);
      if(Temp)
      {
        *(Register_GPIO_Config[Config_PORT].PORT_DR) &= ~Temp;
      }

      else
      {

        *(Register_GPIO_Config[Config_PORT].PORT_DR) |= (1<<u8Channel);
      }

}

#pragma CODE_SEG DEFAULT

#pragma CODE_SEG XGATE_CODE

void GPIO_Set_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{


   *(Register_GPIO_Config[Config_PORT].PORT_DR) |= (1<<u8Channel);
```

```
}

#pragma CODE_SEG DEFAULT


#pragma CODE_SEG XGATE_CODE
void GPIO_Clear_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{
   UINT8 u8Temp;

    u8Temp = *(Register_GPIO_Config[Config_PORT].PORT_DR);

   *(Register_GPIO_Config[Config_PORT].PORT_DR) = (~(1<<u8Channel))&u8Temp;

}

#pragma CODE_SEG DEFAULT
```

GPIO DRIVER - Gpio2.h file

```
/*****************/
#ifndef __GPIO2_H        /*prevent duplicated includes*/
#define __GPIO2_H

/****************** Include files**************/
/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"

  #include "cnf_gpio.h"

  #define SET_PORTB_PTB0(val)   PORTB_PB0 = val
   #define SET_PORTB_PTB1(val)   PORTB_PB1 = val

/*************************** Module Wide Functions ¨*********************************/

#pragma CODE_SEG XGATE_CODE

void GPIO_Init(tGPIO_Driver_Config* GPIO_Driver_Config);

#pragma CODE_SEG DEFAULT
```

```
#pragma CODE_SEG XGATE_CODE

void GPIO_Port_Config(enum GPIO_Port);

#pragma CODE_SEG DEFAULT


#pragma CODE_SEG XGATE_CODE

void GPIO_Toggle_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel);


#pragma CODE_SEG DEFAULT

#pragma CODE_SEG XGATE_CODE

void GPIO_Set_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel);

#pragma CODE_SEG DEFAULT

#pragma CODE_SEG XGATE_CODE

void GPIO_Clear_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel);

#pragma CODE_SEG DEFAULT
/*****************************************************************************/

#endif
```

GPIO DRIVER - Cnf_gpio.h file

```
/***************/
#ifndef __CNF__GPIO_H        /*prevent duplicated includes*/
#define __CNF__GPIO_H
/**************Include files***************/
/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"

/** Used Modules */

typedef enum{
  GPIO_INPUT,
  GPIO_OUTPUT,
  GPIO_NULL
}GPIO_IO_State;

typedef enum{
  OUTPUT_LOW,
  OUTPUT_HIGH,
  INPUT_NULL
}GPIO_Logic_State;
```

```
typedef enum{
  PORT_A,
  PORT_B,
  PORT_C,
  PORT_D,
  PORT_E
}GPIO_Port;

typedef struct{
  UINT8 u8Channel;
  enum GPIO_IO_State IO_State;
  enum GPIO_Logic_State Logic_State;
}tGPIO_Channel_Config;

typedef struct{
enum GPIO_Port Config_PORT;
tGPIO_Channel_Config* GPIO_Channel_Config;
UINT8 u8GPIO_No_Of_Channels;
}tGPIO_Device_Config;

typedef struct{
tGPIO_Device_Config* GPIO_Device_Config;
UINT8 u8GPIO_No_Of_Devices;
}tGPIO_Driver_Config;
```

```
/**************************Status Structures************************************/

typedef struct{
   enum GPIO_IO_State IO_State;
   enum GPIO_Logic_State Logic_State;
   UINT8 u8Channel;
}tGPIO_Channel_Status;

typedef struct{
  tGPIO_Channel_Status* GPIO_Channel_Status;
  UINT8 GPIO_No_Of_Channels;
  enum GPIO_Port Config_PORT;
}tGPIO_Device_Status;

typedef struct{
tGPIO_Device_Status* GPIO_Device_Status;
UINT8 GPIO_No_Of_Devices;
}tGPIO_Driver_Status;
```

```
/**************************Register Structures***********************************/
typedef struct{
  UINT8* PORT_DR;  /*Data Register - reserved for Output - 1 for HIGH - 0 for LOW */
  UINT8* PORT_DDR; /*Data Direction Register - 1 for Output - 0 for Input */
}GPIO_Register_Config;

/**************************Configuration MACROS***********************************/

#define PIN0 0u
#define PIN1 1u
#define PIN2 2u
#define PIN3 3u
#define PIN4 4u
#define PIN5 5u
#define PIN6 6u
#define PIN7 7u
/**************************** Making public the configuration **********************/
  #pragma CONST_SEG XGATE_CONST
extern tGPIO_Driver_Config GPIO_Driver_Config[];
 #pragma CONST_SEG DEFAULT
#pragma CONST_SEG XGATE_CONST
extern const GPIO_Register_Config Register_GPIO_Config[];
#pragma CONST_SEG DEFAULT

/*******************************************************************************/

#endif /* __CNF__GPIO_H */
```

GPIO DRIVER - Gpio.cxgate file

```
/***********Include files*****************/
/** Configuration Options */
        #include        "configuration.h"
/** S12X derivative information */
        #include        __MCU_DERIVATIVE
/** Variable types */
        #include        "typedefs.h"
 #include  "xgate_config.h"
 #include "cnf_gpio.h"
 #include "gpio2.h"
#pragma DATA_SEG XGATE_DATA
 tGPIO_Driver_Config* Driver_Configuration;
#pragma DATA_SEG DEFAULT



/***************Definitions of module wide FUNCTIONs ***************/
#pragma CODE_SEG XGATE_CODE
void GPIO_Init(tGPIO_Driver_Config* GPIO_Driver_Config){
  UINT8 u8Index;
Driver_Configuration = GPIO_Driver_Config;
for(u8Index = 0; u8Index<Driver_Configuration->u8GPIO_No_Of_Devices;u8Index++)
  {
     GPIO_Port_Config(Driver_Configuration->GPIO_Device_Config[u8Index].Config_PORT);
  }
}

#pragma CODE_SEG DEFAULT
```

```
#pragma CODE_SEG XGATE_CODE

void GPIO_Port_Config(enum GPIO_Port IO_PORT)

{

   UINT8 u8Index_1;

   UINT8 u8Index_2;


for(u8Index_1 = 0; u8Index_1<Driver_Configuration->u8GPIO_No_Of_Devices;u8Index_1++)

 {

   if( IO_PORT == Driver_Configuration->GPIO_Device_Config[u8Index_1].Config_PORT)

    {

        for(u8Index_2 = 0; u8Index_2<Driver_Configuration-
>GPIO_Device_Config[u8Index_1].u8GPIO_No_Of_Channels ; u8Index_2++)

        {

           *(Register_GPIO_Config[IO_PORT].PORT_DDR) |=  (Driver_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].IO_State)<<(Driver_Configura
tion->GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].u8Channel);

           *(Register_GPIO_Config[IO_PORT].PORT_DR)  |=  (Driver_Configuration-
>GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].Logic_State)<<(Driver_Config
uration->GPIO_Device_Config[u8Index_1].GPIO_Channel_Config[u8Index_2].u8Channel);

        }

    }

 }

}


#pragma CODE_SEG DEFAULT
```

```
/*This function Toogles a certain bit of a certain PORT */
/*To be used as a part of the PWM Driver*/

#pragma CODE_SEG XGATE_CODE

void GPIO_Toggle_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{
    UINT8 Temp;
     Temp = (*(Register_GPIO_Config[Config_PORT].PORT_DR))&(1<<u8Channel);
     if(Temp)
     {
       *(Register_GPIO_Config[Config_PORT].PORT_DR) &= ~Temp;
     }
     else
     {
       *(Register_GPIO_Config[Config_PORT].PORT_DR) |= (1<<u8Channel);
     }
}

#pragma CODE_SEG DEFAULT
#pragma CODE_SEG XGATE_CODE
void GPIO_Set_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{
   *(Register_GPIO_Config[Config_PORT].PORT_DR) |= (1<<u8Channel);
}

#pragma CODE_SEG DEFAULT
```

```
#pragma CODE_SEG XGATE_CODE
void GPIO_Clear_Pin(enum GPIO_Port Config_PORT ,UINT8 u8Channel)
{
  UINT8 u8Temp;
   u8Temp = *(Register_GPIO_Config[Config_PORT].PORT_DR);
  *(Register_GPIO_Config[Config_PORT].PORT_DR) = (~(1<<u8Channel))&u8Temp;
}

#pragma CODE_SEG DEFAULT
```

GPIO DRIVER - Cnf_gpio.cxgate

```
/********Include files***************/
/** Core Modules */
/** Configuration Options */
  #include    "configuration.h"
/** S12X derivative information */
  #include    __MCU_DERIVATIVE
/** Variable types and common definitions */
  #include    "typedefs.h"

/** Configuration Types**/
  #include "cnf_gpio.h"
  #include "gpio2.h"
```

```
/** Used Modules */
/* GPIO Register Structure*/
#pragma CONST_SEG XGATE_CONST
const GPIO_Register_Config Register_GPIO_Config[] =
{
  { &((UINT8)PORTA),
    &((UINT8)DDRA)
  },
  { &((UINT8)PORTB),
    &((UINT8)DDRB)
  },
  { &((UINT8)PORTC),
    &((UINT8)DDRC)
  },

  { &((UINT8)PORTD),
    &((UINT8)DDRD)
  },

  { &((UINT8)PORTE),
    &((UINT8)DDRB)
  },
};

  #pragma CONST_SEG DEFAULT
```

```
 /*Channels defined for PORTB device*/
  #pragma CONST_SEG XGATE_CONST
tGPIO_Channel_Config GPIO_PORTB_Config_Channel[] = {
  {
    PIN0,
    GPIO_OUTPUT,
    OUTPUT_LOW
   },
   {
    PIN1,
    GPIO_OUTPUT,
    OUTPUT_LOW
   },
};
 #pragma CONST_SEG DEFAULT
 /*Definition of Devices */
  #pragma CONST_SEG XGATE_CONST
tGPIO_Device_Config GPIO_Config_Device[] =
{
    {
      PORT_B,
      &GPIO_PORTB_Config_Channel[0],
      (sizeof(GPIO_PORTB_Config_Channel)/sizeof(GPIO_PORTB_Config_Channel[0]))
    },
};
   #pragma CONST_SEG DEFAULT
```

```
/*Definition of the Driver */
#pragma CONST_SEG XGATE_CONST
tGPIO_Driver_Config GPIO_Driver_Config[] =
{
   {
     &GPIO_Config_Device[0],
     (sizeof(GPIO_Config_Device)/sizeof(GPIO_Config_Device[0]))
   },
};
#pragma CONST_SEG DEFAULT
```