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

MAESTRÍA EN DISEÑO ELECTRÓNICO

REPORTE DE FORMACION COMPLEMENTARIA EN AREA DE CONCENTRACION EN SISTEMAS EMBEBIDOS

Trabajo recepcional que para obtener el grado de

MAESTRO EN DISEÑO ELECTRÓNICO

Presentan: Ing. José Antonio Maza Moreno

San Pedro Tlaquepaque, Jalisco. Enero 2017.

Contenido

Introducción

El objetivo principal del documento es dar a conocer los proyectos más sobresalientes que se desarrollaron en la Maestría de diseño electrónico en el área de sistemas embebidos y telecomunicaciones. Las materias de concentración y proyectos son los siguientes:

- **Diseño de Sistemas Operativos para Ambientes Embebidos:** Diseño e implementación de un sistema operativo para ambientes embebidos
- **Desarrollo de Software de Comunicaciones en Ambientes Embebidos:** Diseño de módulo de software de comunicación automotriz.
- **Ingeniería de Software en Ambientes Embebidos:** Desarrollo de documentación de las especificaciones de producto y de software para un módulo de control automotriz

Los proyectos anteriormente descritos me permitieron desarrollar nuevas habilidades técnicas en el área de electrónica y programación de sistemas embebidos usados principalmente en la industria automotriz. Cada una de estas materias me permitió conocer el aspecto teórico que fundamenta el área de sistemas embebidos y además me permitió enriquecer mi conocimiento al conocer algunas de las herramientas de desarrollo, estándares industriales y procesos de desarrollo que son utilizados en la industria.

La amplia experiencia en la teoría y la práctica de los maestros que impartieron las materias me permitió conocer diferentes facetas del desarrollo de software y cómo cada una de las capas de software en un producto interactúa con las otras.

En el área de comunicaciones el aprendizaje adquirido me permitió entender cómo las diferentes capas del modelo de comunicación OSI funcionan y son implementadas en diferentes protocolos de comunicación automotrices.

1. Resumen de los proyectos realizados

A continuación se presenta una síntesis de los tres proyectos realizados. Para mayor detalle referirse a los apéndices correspondientes a cada proyecto

1.1. Diseño de Sistemas Operativos para Ambientes Embebidos

1.1.1 Introducción

El objetivo de este proyecto fue desarrollar un sistema operativo para sistemas embebidos capaz de administrar los recursos del microcontrolador entre diferentes tareas configurables dentro del sistema permitiendo ejecutarlas de manera periódica.

1.1.2 Antecedentes

El desarrollo de este proyecto fue coordinado a lo largo del curso, el lenguaje de programación utilizado para desarrollar esa solución de software fue C, ya que es el lenguaje más utilizado en sistemas embebidos y por la numerosa cantidad de herramientas para desarrollar con este lenguaje que es un estándar industrial.

1.1.3 Solución desarrollada

Las diferentes etapas del proyecto se fueron desarrollando de manera gradual durante el curso, al inicio se definieron las estructuras básicas de control para el módulo, también fue importante conocer el hardware utilizado, ya que el módulo se encarga de controlar estos recursos. Una vez comprendidos estos puntos se comenzó con la implementación en la que se tuvieron que definir los algoritmos para poder asignar los recursos de las tareas así como los mecanismos para hacer el cambio entre las diferentes tareas sin que se perdiera la ejecución anterior.

1.1.4 Análisis de Resultados

Se realizaron diferentes experimentos y pruebas para comprobar que el módulo realizara las operaciones de manejo de recursos del procesador de manera correcta, para lo cual se definieron diferentes casos de prueba para validarlo. En el apéndice respectivo a este proyecto se pueden conocer más detalles.

1.1.5 Conclusiones

Se logró la implementación de un sistema de software modular basándose en la arquitectura de AUTOSAR lo que logró hacer este módulo más reusable y basado en estándares industriales. Este proyecto me permitió desarrollar diferentes habilidades teóricas y prácticas ya que tuve que comprender diferentes conceptos para el desarrollo del producto así como la parte práctica que consistió en usar estándares industriales como AUTOSAR y el uso de herramientas que son comúnmente usadas en el campo de los sistemas embebidos.

1.2. Diseño e Implementación de Software de Comunicaciones para Ambientes Embebidos

1.2.1 Introducción

En este proyecto se desarrolló un módulo de software que implementa el protocolo de comunicación automotriz LIN siguiendo los diferentes estándares que definen a este protocolo de comunicación.

1.2.2 Antecedentes

El protocolo de comunicación LIN es usado en la industria automotriz por su simplicidad y bajo costo de implementación, es ampliamente usado en redes que no requieren una alta cantidad de transferencia de datos y los datos transferidos no son críticos. Típicamente este tipo de módulos de software es comprado a otros fabricantes

1.2.3 Solución desarrollada

En este proyecto se definieron los bloques básicos que componen el módulo de software para poder operar en la tarjeta de desarrollo provista por la institución, la cual utiliza un microcontrolador orientado a la industria automotriz e industrial, además de implementar el hardware requerido para operar con el protocolo de comunicación LIN.

1.2.4 Análisis de Resultados

Para este proyecto se realizaron diferentes mediciones para verificar que todos los componentes de la trama de comunicación de LIN cumplieran con los requerimientos establecidos por el estándar de comunicación LIN 1.3 en cuanto a duración, tolerancia y contenido de la trama de comunicación. El encabezado de mensaje consiste de una interrupción usada para identificar el

inicio del marco y el campo de sincronización usado por el nodo esclavo para sincronización de reloj. El identificador (ID) consta de un ID de mensaje de seis bits y un campo paridad de dos bits. El ID indica una dirección específica de mensaje pero no el destino. Después de la recepción e interrupción del ID, un esclavo comienza la respuesta del mensaje, la cual consiste de uno a ocho bytes de datos y una suma de verificación de ocho bits. Las Detalles adicionales se pueden encontrar en el apéndice correspondiente a este proyecto

1.2.5 Conclusiones

En este proyecto se implementó un driver de LIN básico que permitiera la formación de una trama de datos como está definido por el Standard LIN y creando las interfaces necesarias de software para calcular los diferentes parámetros que componen al driver de LIN.

En este proyecto aprendí la estructura general de una trama de comunicación de LIN y sus posibles combinaciones así como su operación básica en una red usando este protocolo. También me ayudó a entender los diferentes tipos de errores especificados para LIN que pueden ser generados y el motivo de su aparición, además de cómo hacer el manejo de errores.

1.3. Diseño de módulo de control electrónico automotriz

1.3.1 Introducción

El módulo automotriz diseñado es el Multi-Function Switch Module, que es el módulo encargado de controlar las palancas indicadoras de direccionales y el control de luces del vehículo.

El proyecto consistió en diseñar y documentar las diferentes etapas del desarrollo del producto desde el análisis de requerimientos hasta la arquitectura de software para este módulo.

1.3.2 Antecedentes

En el departamento de electrónica se cuenta con diferentes tableros automotrices de aprendizaje. A lo largo de la materia cursada se seleccionó un módulo de este tablero de instrumentos y se procedió a hacer análisis de ingeniería inversa para poder desarrollar todas las etapas que implica el desarrollo de un producto de este tipo

1.3.3 Solución desarrollada

Para el desarrollo del proyecto se siguió la metodología V, que es un procedimiento comúnmente usado en el área de tecnologías de información y desarrollo de software.

Esta metodología permitió documentar todas las etapas que componen el desarrollo de este proyecto incluyendo la definición de los conceptos de operación, como debe de operar con los elementos externos del proyecto, la definición de requerimientos y la arquitectura del proyecto. Cada etapa del proceso se obtiene diferentes documentos que son utilizados por etapas posteriores del proyecto

1.3.4 Análisis de Resultados

En cada una de las etapas del desarrollo del proyecto se obtuvieron distintos documentos que permitieron tener una idea clara del objetivo y de los pasos necesarios para poder trabajar en las siguientes etapas del proyecto. Dichos proyectos se encuentran adjuntos en el apéndice de este documento.

1.3.5 Conclusiones

Este proyecto me permitió entender las diferentes etapas de desarrollo de un proyecto y como diferentes áreas de competencias interactúan y trabajan de manera coordinada bajo el esquema de la metodología V de desarrollo de productos.

Las ventajas de usar una metodología ya establecida es que al final de cada etapa se obtienen datos claros y precisos necesarios para continuar con la siguiente etapa, lo que tiende a minimizar riesgos, ya que permite identificar posibles problemas durante la etapa de planeación.

2. Conclusiones

La variedad de materias cursadas durante la maestría en diseño electrónico me permitió crecer en el aspecto profesional y académico. Los proyectos propuestos durante las asignaturas contribuyeron de manera importante a mi formación profesional ya que normalmente los maestros que impartieron la asignatura tienen una gran trayectoria académica y/o profesional y estuvieron en todo momento abiertos a compartir su experiencia profesional.

Los conocimientos adquiridos tienen gran importancia en el campo laboral de electrónica, y en lo particular en el área de sistemas embebidos, ya que muchos de estos conocimientos adquiridos los he usado en mi práctica profesional diaria y han enriquecido mis aportaciones en el aspecto profesional.

Los factores que contribuyeron a un aprendizaje más significativo del área de electrónica y sistemas embebidos fueron las instalaciones y el equipo que siempre estuvo disponible para poder realizar las diferentes asignaciones, así como la experiencia y actitud de los académicos que siempre realizaban buenas observaciones y contribuciones para mejorar el entendimiento de los temas estudiados.

Apéndices

A. REPORTE DE DISEÑO DE SISTEMA OPERATIVO PARA AMBIENTES EMBEBIDOS

Final Report

Operating Systems Design for Embedded Environments

Equipo 3
Manglio González Carrasco
Américo Lorenzana Gutiérrez
José Antonio Maza Moreno

07/05/2015

Embedded Systems Specialization Program

www.sistemasembebidos.iteso.mx/alumnos

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

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

Technical Report Number: ESE-O2014-001

Abstract: *This report covers the basic implementation of an Basic Preemptive Scheduler based on OSEK in demo9s12xep100 board from Freescale 16-bit MC9S12XE microcontroller*

Keywords: Scheduler, *OS System Tick, OSEK, Preemptive.*

1. TABLE OF CONTENTS

2. Table of Figures

3. Introduction

3.1. Os System Tick

The Os System Tick is the Timing Unit that the OS can use as a base time to calculate OS timer and delays if they are supported. In addition, it causes the scheduler to run and may cause a context switch – for example. The interval of time with which the timer is programmed to interrupt defines the unit of time handled by the kernel (time resolution).

In a typical embedded system a hardware timer is usually used and configured to generate an interrupt at a rate between 10 and 1000 Hz.

3.2. Scheduler

The scheduler is at the heart of every kernel. A scheduler provides the algorithms needed to determine which task executes when. Key concept related to scheduler activities are discussed in this section are summarized below:

- **Schedulable entities:** A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. A task is an independent thread of execution that contains a sequence of independently schedulable instructions. Note that message queues and semaphores are not schedulable entities. These items are inter-task communication objects used for synchronization and communication
- **Multitasking:** Is the ability of the operating system to handle multiple activities within set deadlines. A real-time kernel might have multiple tasks that it has to schedule to run. The scheduler must ensure that the appropriate task runs at the right time.
- **Context switching:** Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. A context switch occurs when the scheduler switches from one task to another.
- **Dispatcher:** The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel. When a task or ISR makes a system call, the flow of control passes to the kernel to execute one of the system routines provided by the kernel. When it is time to leave the kernel, the dispatcher is responsible for passing control to one of the tasks in the user's application.

- **Scheduling algorithms:** As mentioned earlier, the scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support different common scheduling algorithms, preemptive priority-based scheduling, round-robin scheduling and binary progression scheduling as in the implementation presented in subsequent chapters.

3.3. Task Scheduling

A task is an independent thread of execution that can compete with other concurrent tasks for processor execution time. Developers decompose applications into multiple concurrent tasks to optimize the handling of inputs and outputs within set time constraints.

A task is schedulable, which means, the task is able to compete for execution time on a system, based on a predefined scheduling algorithm. A task is defined by its distinct set of parameters and supporting data structures. Specifically, upon creation, each task has an associated name, a unique ID, a priority (if part of a preemptive scheduling plan), a task control block (TCB), a stack, and a task routine, as shown in Figure 3.1. Together, these components make up what is known as the task object.

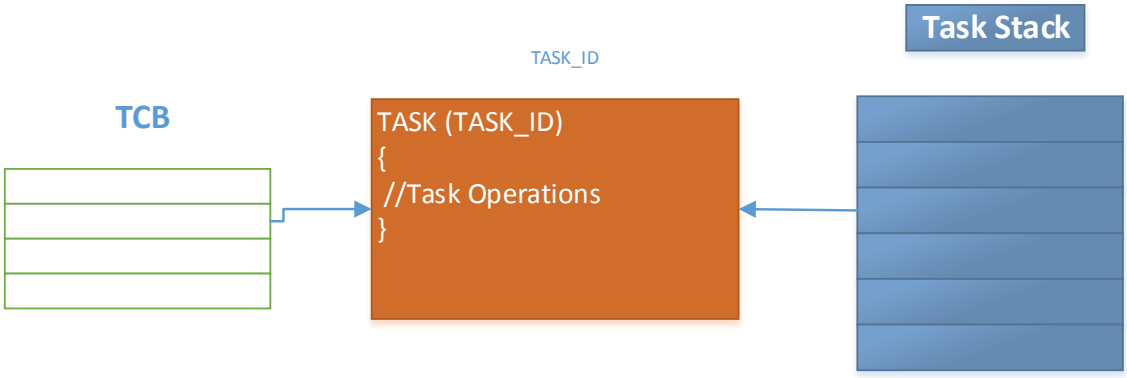


Figure 3.1 A task, its associated parameters, and supporting data structures.

When the kernel first starts, it creates its own set of system tasks and allocates the appropriate priority for each from a set of reserved priority levels. The reserved priority levels refer to the priorities used internally by the RTOS for its system tasks. An application should avoid using these priority levels for its tasks because running application tasks at such level may affect the overall system performance or behavior. For most RTOSes, these reserved priorities are not enforced. The kernel needs its system tasks and their reserved priority levels to operate. These priorities should not be modified. Examples of system tasks include:

- initialization or startup task—initializes the system and creates and starts system tasks,
- idle task—uses up processor idle cycles when no other activity is present,
- logging task—logs system messages,
- exception-handling task—handles exceptions, and
- debug agent task—allows debugging with a host debugger. Note that other system tasks might be created during initialization, depending on what other components are included with the kernel.

4. Requirements

Table 4.1 Initial Requirements

Component	Requirements
General	The following HW features need to be initialized before OS Tick is running: - PLL: Configuration must be modifiable with static configuration - Timer Resource: Required to generate the periodic ISR required by the OS Tick
General	Interrupt Vector Configuration has to be provided. This configurations will be defined in an Isr.c and Isr.h
General	The following conventions are to be taken from AUTOSAR specification release 4.1: - File Structure - MCU driver to support PLL initialization, clock distribution and if required startup code - GPT(General Purpose Timer) driver to support PIT initialization and timer control
General	Project Setup and configuration will ensure to have the following features – Interrupt Vector allocation – PLL Working at 48MHz with 4MHz to16MHz configurable XTAL – PIT Notification Enabled – PIT Dummy Callback (Incrementing counter and Pin Level Toggle can be provided here -> Scope purposes) – OS Tick Configuration - Configurable from PIT channel parameters – 781.25 us
MCU (Microcontroller Unit)	The following API's are to be provided: – void Mcu_Init(void); – void Mcu_InitClock(void); – void Mcu_DistributePllClock(void); AUTOSAR defined parameters are not required
MCU (Microcontroller Unit)	The following files need to be provided - Mcu_Cfg.h: Any MCU configuration must be placed within this file. Including PLL configuration, XTAL frequency and Bus Clock frequency - Mcu.c: Implementation of required functionality must be within this file - Mcu.h: This file will export any API required by MCU clients
GPT (General Purpose Timer)	The following API's are to be provided – void Gpt_Init(const Gpt_ConfigType* ConfigPtr); – void Gpt_StartTimer(Gpt_ChannelType Channel, Gpt_ValueType Value) ; – void Gpt_StopTimer(Gpt_ChannelType Channel) ; – void Gpt_EnableNotification(Gpt_ChannelType Channel) ; – void Gpt_DisableNotification(Gpt_ChannelType Channel) ;

GPT (General Purpose Timer)	The following files need to be provided – Gpt.h : This file will export any API required by MCU clients – Gpt.c: Implementation of required functionality must be within this file – Gpt_Cfg.h – Gpt_Cfg.c: This fill will hold the configuration structures for GPT driver configurable by the user. Parameters must be set in Pre-compile time configuration

Table 4.2 Scheduler Requirements

Component	Requirement
SchM - General	Scheduling refers to making a sequence of time execution decisions at specific intervals.
SchM - General	The decision that is made is based on a predictable algorithm.
SchM - General	An application that does not need its current allocation leaves the resource available for another application's use.
SchM - General	The underlying algorithm defines how the term “controlled” is interpreted. In some instances, the scheduling algorithm might guarantee that all applications have some access to the resource. The Scheduler is based on a binary counter incremented at a given time, this time is controlled by an interrupt, typically called OS Tick.
SchM - General	The Binary Progression Scheduler (BPS) manages the access to the CPU resources in a controlled way
SchM – Partitioning Mechanism	Partitioning is used to bind a task to a subset of the system's available resources.
SchM – Partitioning Mechanism	This binding guarantees that a known amount of resources is always available to the task.
SchM – Partitioning Mechanism	Those resources are taken by time-slicing the available processing time.
SchM – Partitioning Mechanism	Systems that use time-slicing take advantage of the CPU/Core utilization.

SchM – Partitioning Mechanism	Keeping the CPU/Core occupied enhance the use of the MCU resources.
SchM – Partitioning Mechanism	A processor always have a task to execute even though all the other tasks are idle.
SchM – Partitioning Mechanism	When no tasks are executed the processor is running a Background Task
SchM – Masking Concept	A **mask** is a number defined by **mask=(2^n)-1** The mask is used to **mark** a task for execution. When the binary counter and the mask **matches** the task is executed. From the mask and the OS tick period we can obtain the task **rate**. Therefore the task rate is: **task rate = OS tick * (mask + 1)**
SchM – Offset Concept	A collision may occur between the tasks. If a collision is present some tasks will start being executed in a not desirable time. An offset is defined to allow the task execution being moved in different time slots. The offset can only be in the range already defined by the mask starting from the count of zero. With this approach the task collision is avoided.

Table 4.3 Task Scheduler Requirements

Component	Requirement
OsTask – Data Types	Used for all status information the OSEK API services offer. – All errors for API services start with E_, those reserved for the OS will begin with E_OS_.

	– Normal return value is E_OK which is associated with the value 0.
OsTask – Data Types	The following OS Error IDs must be present E_OS_ACCESS 1 E_OS_CALLEVEL 2 E_OS_ID 3 E_OS_LIMIT 4 E_OS_NOFUNC 5 E_OS_RESOURCE 6 E_OS_STATE 7 E_OS_VALUE 8
OsTask – Task Control Block	Task Control Block must be allocated in RAM Memory – To support this functionality, dynamic memory allocation services have to be used.

OsTask – Task Control Block	Task Control Block must contain at least the following parameters:
	• Task ID
	• Task priority
	• Task state
	• Stack information
	  ○ Start Address
	  ○ End Address
	• Deadline
	  ○ Relative
	  ○ Absolute

OsTask – Task Descriptor	Task Descriptor must be allocated in flash memory
OsTask – Task Descriptor	Task Descriptor provides the user a mean to define and configure the task behavior
OsTask – Task Descriptor	Task Descriptor must contain at least the following parameters: • Task ID • Priority • Mask • Offset • Stack Size • Relative Deadline • Callback function

OsTask – Task Priority	• Several tasks of different priorities can be in the ready state. • The task which has 'waited' the longest time, depending on its order of requesting, is shown at the bottom of each queue. • Once the processor has just processed and terminated a task, the scheduler selects the next task to be processed. • A tasks can only be processed after all tasks of higher priority have left the running and ready state. • Tasks are then removed from the queue either due to termination or due to transition into waiting state (for extended tasks).
OsTask – Dispatcher	The dispatcher should be as fast as possible
OsTask – Dispatcher	Unnecessary context switches should be avoided.
OsTask – Dispatcher	The following steps are necessary to determine the next task to be processed: • The dispatcher searches for all tasks in the ready state already located in the priority buffer. • From the set of tasks in the ready state, the dispatcher determines the set of tasks with the highest priority. • Within the set of tasks in the ready state and of highest priority, the dispatcher finds the oldest task and executes it.
OsTask – OS File Strucure	OS folder will be provided in Services layer containing the following files: • Os_TaskM.c/Os_TaskM.h: provide Task Manager services ActivateTask, TerminateTask, GetTaskID, GetTaskState • Os_TaskCfg.c/Os_TaskCfg.h: provide OS tasks configuration support, these files will replace SchM_Cfg.c/SchM_Cfg.h which will not be provided.

	– Os_Task.c/Os_Task.h: provide the configured tasks to the application replacing SchM_Task.c/SchM_Task.h • Os_Types.h: provide the required data definitions for Task management services support, will replace SchM_Types.h

5. Design

Since all the Os Tick system implementation must be based on AUTOSAR specification. Our current implementation must follow the same rough architecture structure.

Access to the microcontroller hardware is routed through the Microcontroller Abstraction Layer. The MCAL layer ensures a standard interface and controls the microcontroller peripherals. Standardized components are:

- MCU (Microcontroller Unit) driver: provides services for basic microcontroller initialization, power down functionality, reset and microcontroller specific functions
- WDG (Watchdog) driver: provides services for initialization, changing the operation mode and triggering the watchdog
- GPT (General Purpose Timer) driver: uses the hardware timer channels of the general-purpose timer unit

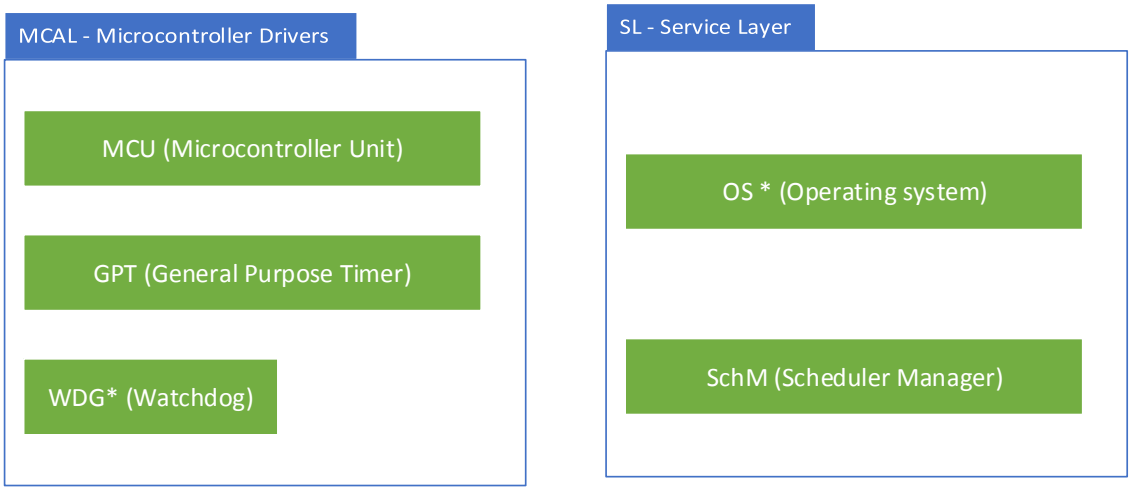


Figure 5.1 Architecture Diagram of AUTOSAR MCAL microcontroller drivers and OS Service Layer

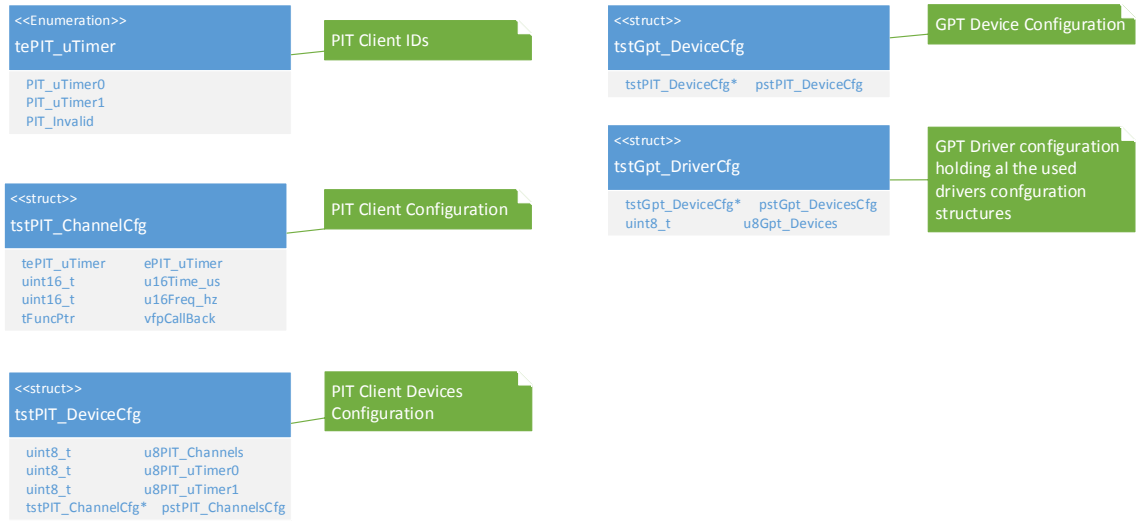


Figure 5.2 GPT Configuration Structures Definitions

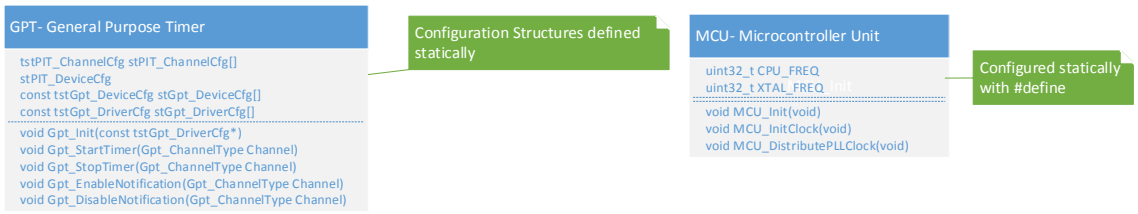


Figure 5.3 Class Diagram of MCU and GPT drivers

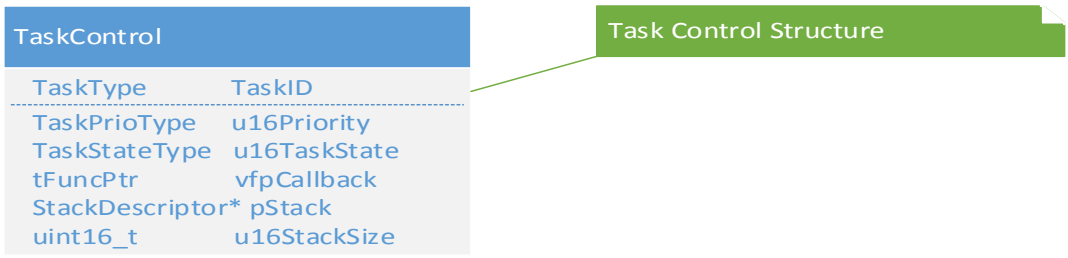


Figure 5.4 Os Configuration Structures Definitions

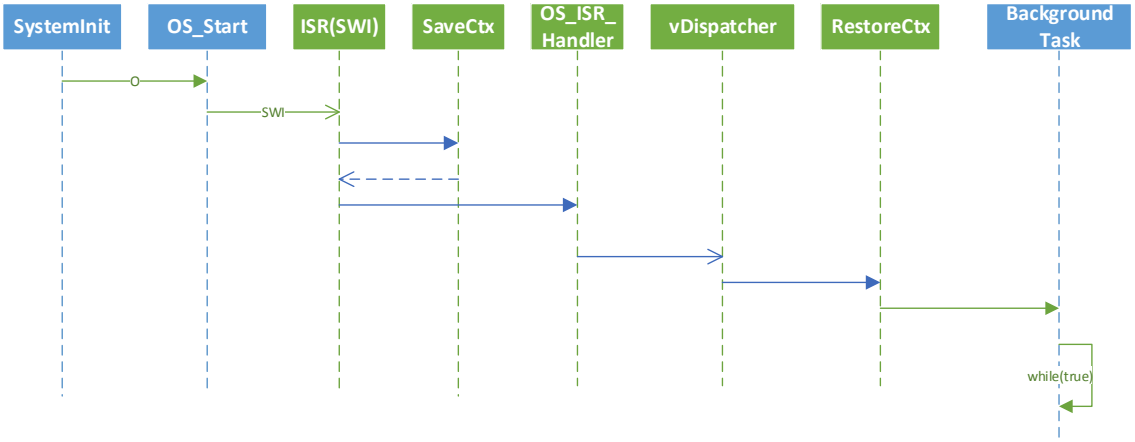


Figure 5.5 Os Initialization Sequence Diagram. At the end the default task is executed(Background Task). Blue Objects are executed normally, green objects are called whitin ISR context

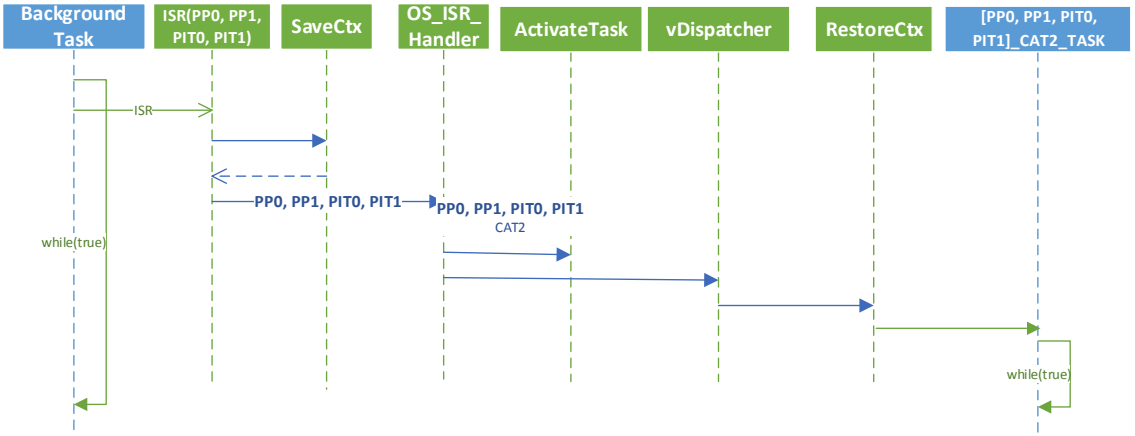


Figure 5.6 Os Context Swithc from Background Task Sequence Diagram. At the end the CAT2_task is executed since It has a higher priority. Blue Objects are executed normally, green objects are called whitin ISR context

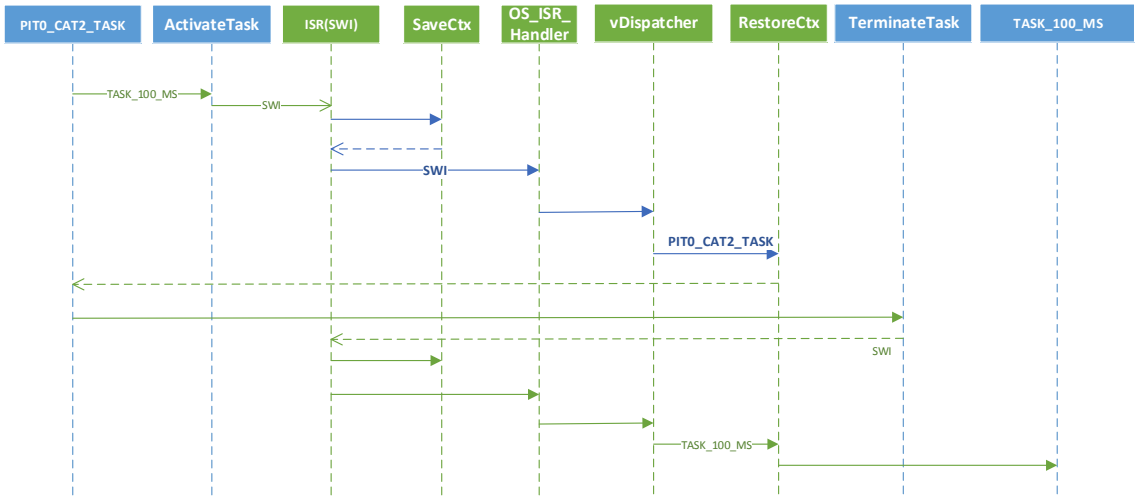


Figure 5.7 Os Task Priorities Handling Example. Assuming that PIT0 task is executin and has the highest priority. It Activates TASK_100_MS but the task activated is only enqueued since It has a lower priority. Once CAT2 task is terminated, TASK_100_MS task will be executed. Blue Objects are executed normally, green objects are called whitin ISR context

6. Results

Place a description of your solutions and (if available) some figures such as screen captures as evidence of your working project.

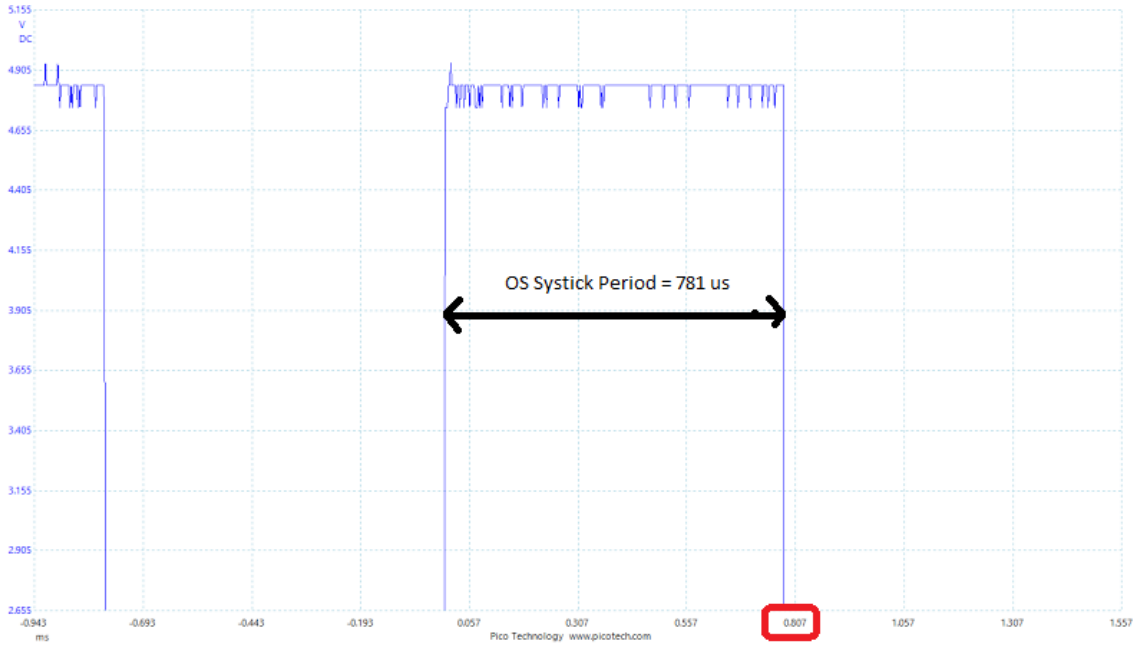


Figure 6.1 Measurement of OS Tick with test code developed to toggle a pin at each SysTick ISR

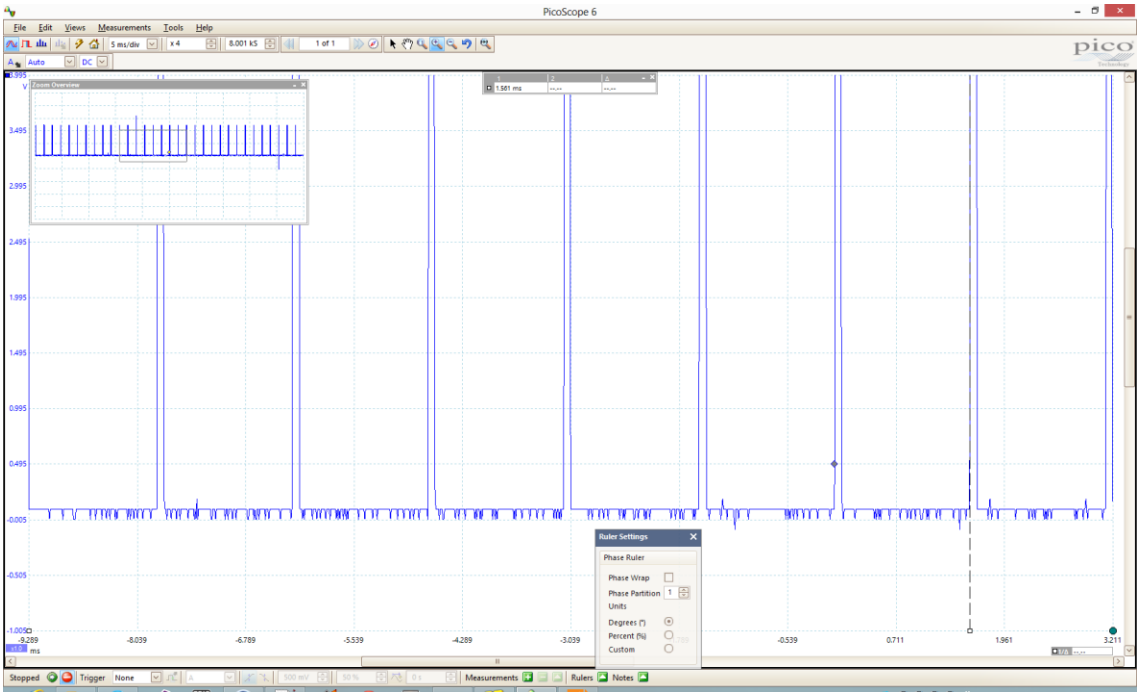


Figure 6.2 Task 1.56 ms setting a pin on beginning and resetting at the end of the task

```
void SchM_Task_1p56ms(void)
{
    uint32_t i = 0;

    PORTA = 0x01; //Setting D0 of Port A

    for(i = 0; i < 100; i++)
    {
        ;       //Do nothing
    }

    PORTA &= 0xFE; //Clearing D0 of Port A
}
```

Figure 6.3 Task 1.56ms Code Snippet

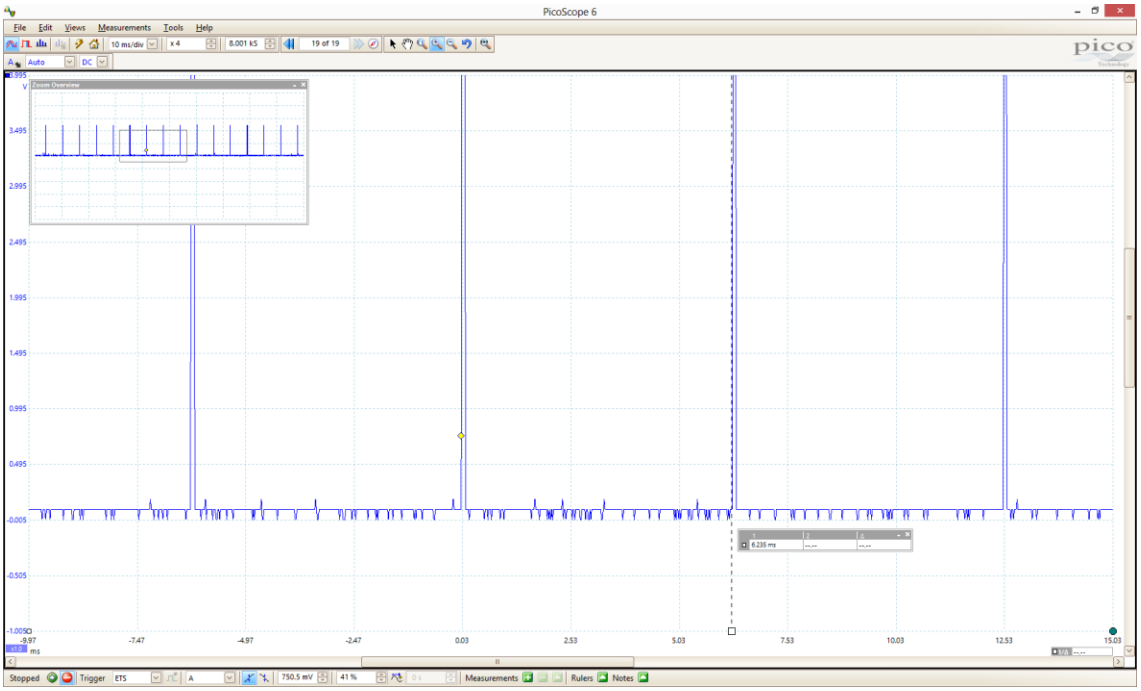


Figure 6.4 Task 1.56 ms setting a pin on beginning and resetting at the end of the task

```
void SchM_Task_6p25ms(void)
{
    uint32_t i = 0;

    PORTA = 0x02; //Setting D1 of Port A

    for(i = 0; i < 100; i++)
    {
        ;       //Do nothing
    }

    PORTA &= 0xFD; //Clearing of D1 of
Port A
}
```

Figure 6.5 Task 6.25ms Code Snippet

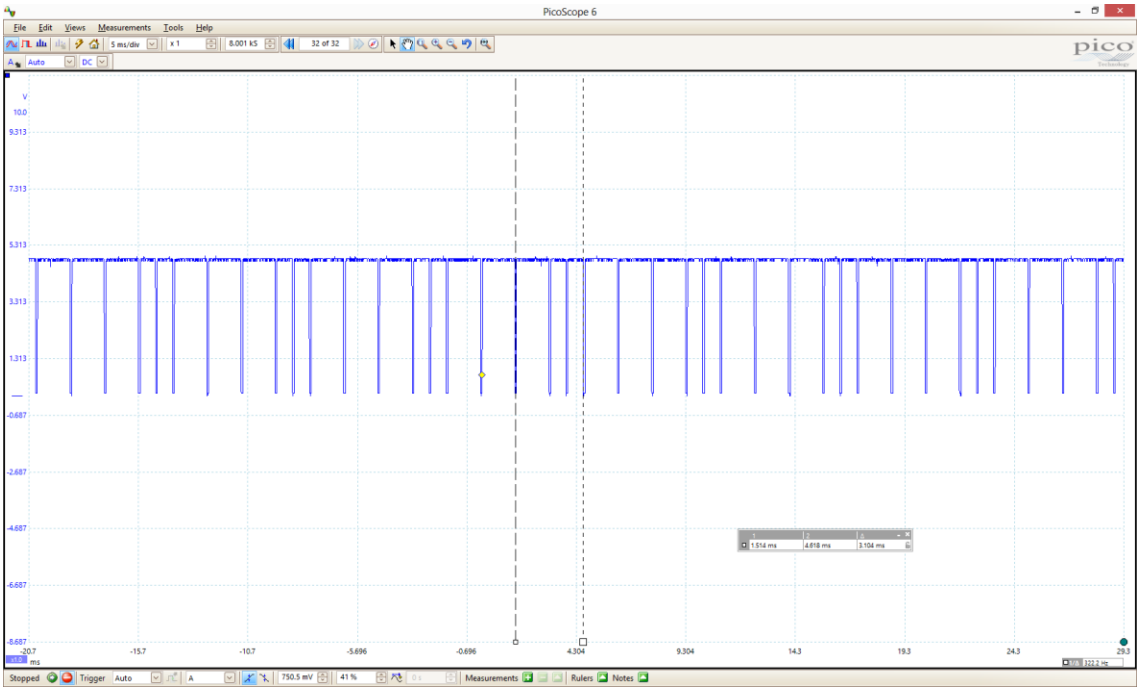


Figure 6.6 Background Task Setting pin when It enters. Any time a task start execution clear the same pin. In this image two tasks are being executed

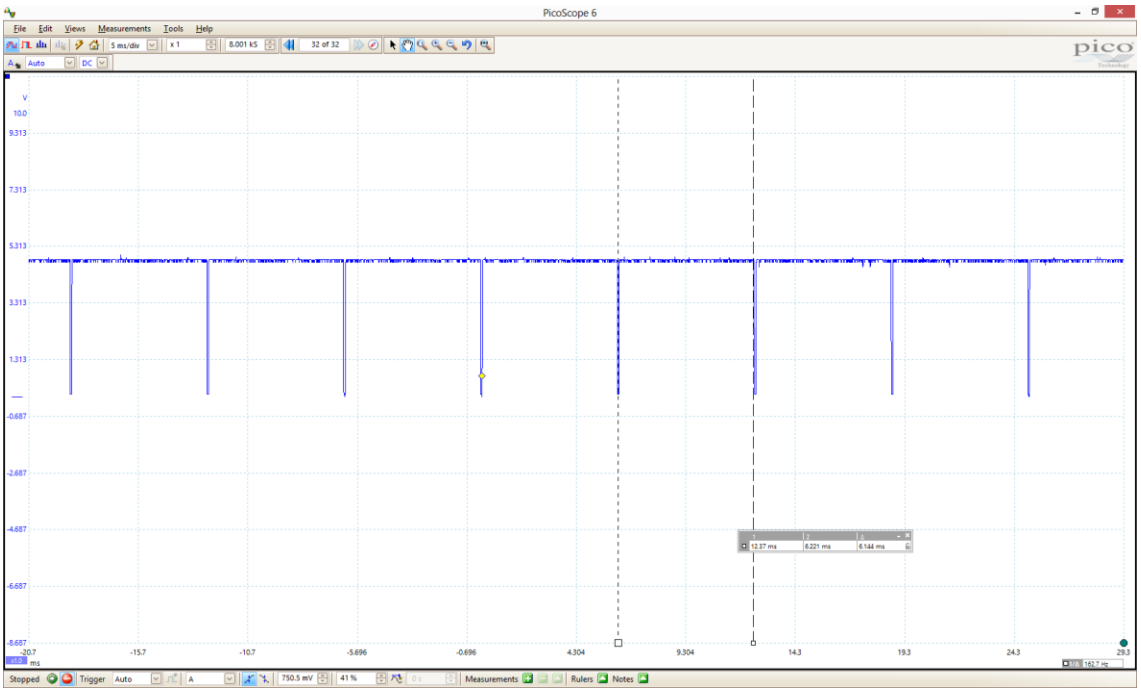


Figure 6.7 Background Task and Task6p25ms executing only

```
void SchM_Background(void)
{
    /* Loop forever */
    for(;;)
    {
        PORTA |= 0x08;  /* Set D3 of Port
A for "idling" measurements */
        _FEED_COP();    /* feeds the dog
*/
        //PORTA &= 0xF7; //Clearing of D3
of Port A
    }
}
```

Figure 6.8 Background Task Code Snippet

6.1. Results of Preemptive Scheduler

In order to measure and characterize the behavior of the preemptive scheduler, each task set and rest a pin to measure task execution time.

The tasks used in the testing on the scheduler are described in the following table:

Task Name	Port and bit used	Priority	Type	Description
Task_100ms	PORTB.0	4	TASK	100 ms task activated by CAT2_PIT0
Task_200ms	PORTB.1	5	TASK	200 ms task activated by CAT2_PIT1
OS_BackgroundTask	PORTA[0-3]	0	OS_TASK	Default Task to be executed when no task is scheduled
CAT2_PP0_Task	PORTB.2	9	CAT2_TASK	CAT2 task activated by PP0 ISR(Button)
CAT2_PP1_Task	PORTB.3	8	CAT2_TASK	CAT2 task activated by PP1 ISR(Button)
CAT2_PIT0_Task	PORTB.4	7	CAT2_TASK	CAT2 task activated by PIT0 ISR(Timer)
CAT2_PIT1_Task	PORTB.5	6	CAT2_TASK	CAT2 task activated by PIT1 ISR(Timer)

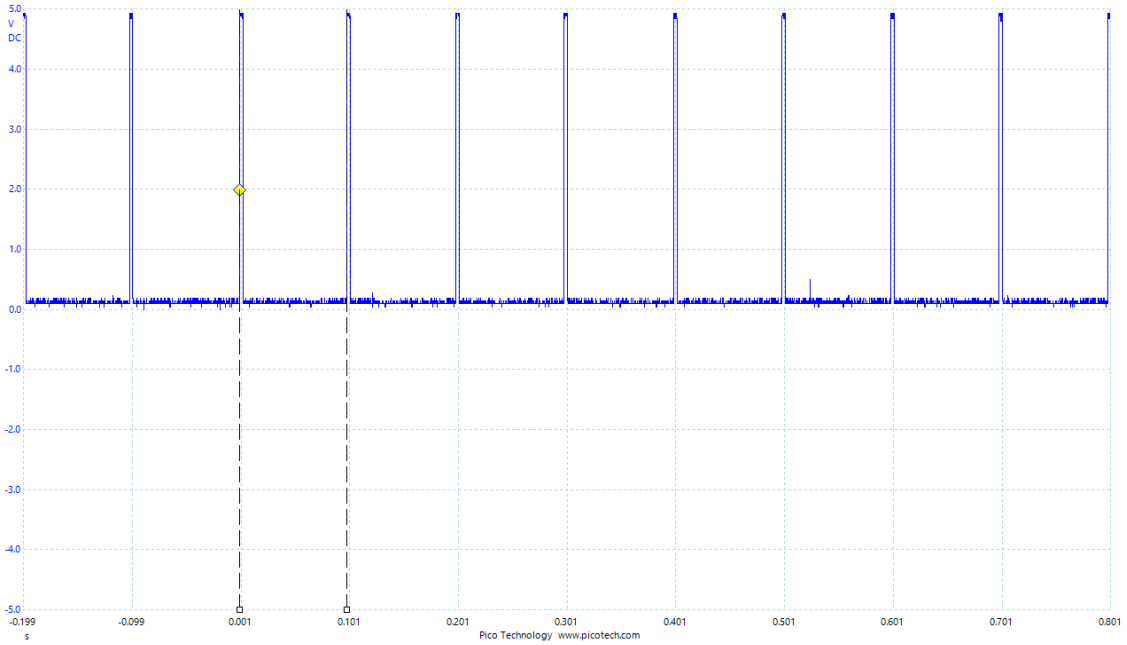


Figure 6.9 Task_100ms Task running at the requested 100 ms timer. This task is activated by CAT2 ISR PIT0

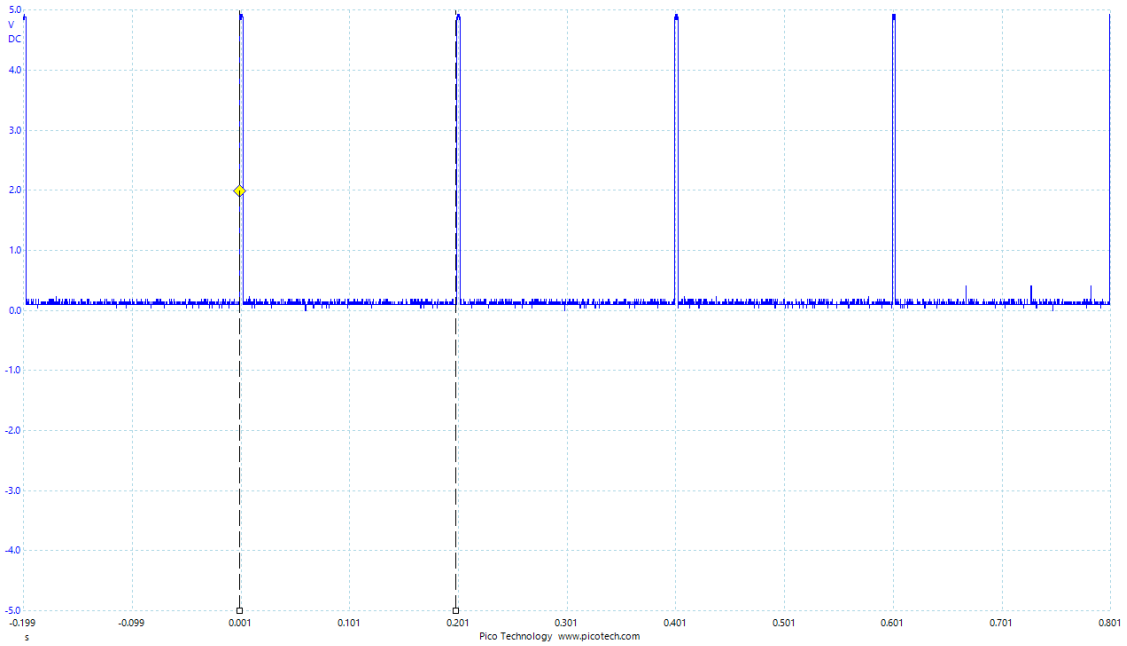


Figure 6.10 Task_200ms Task running at the requested 200 ms timer. This task is activated by CAT2 ISR PIT1

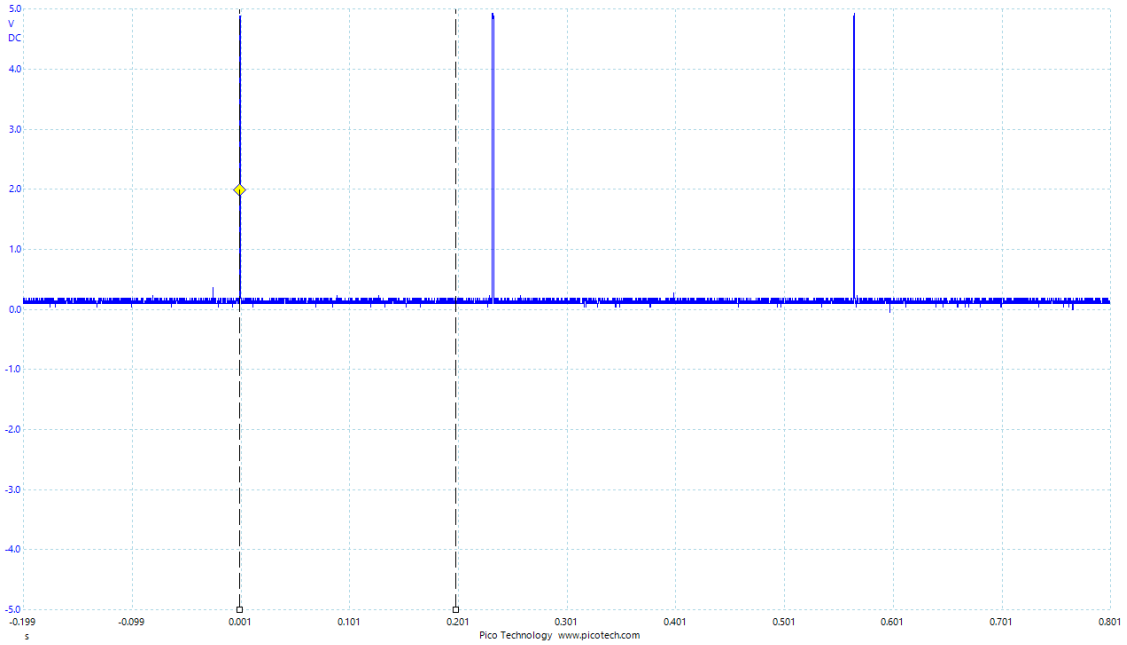


Figure 6.11 CAT2_PP0_Task Task called at irregular time intervals. This task is activated by CAT2 ISR PP0

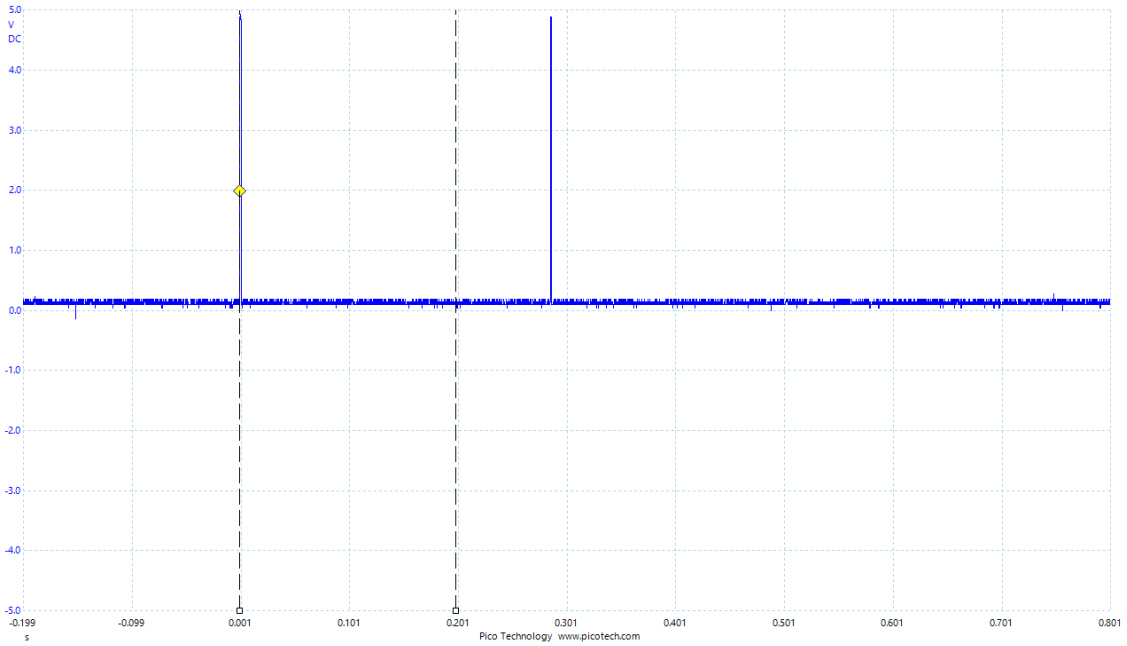


Figure 6.12 CAT2_PP1_Task Task called at irregular time intervals. This task is activated by CAT2 ISR PP1

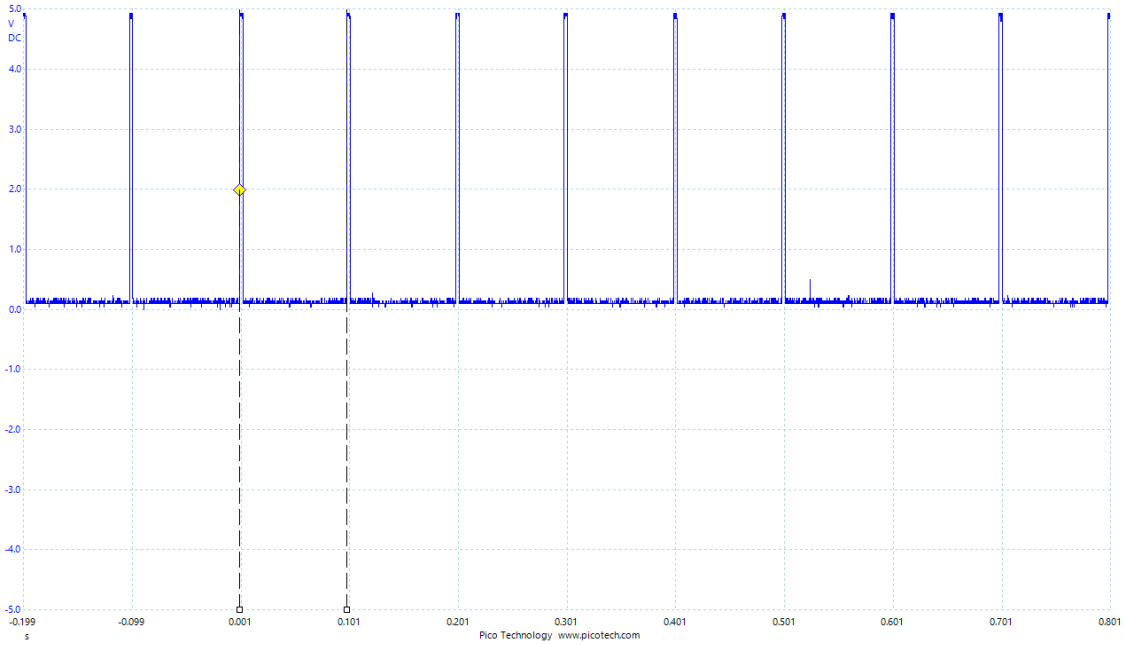


Figure 6.13 CAT2 ISR PIT0 task called by Dispatcher after ISR adds to the dispatcher

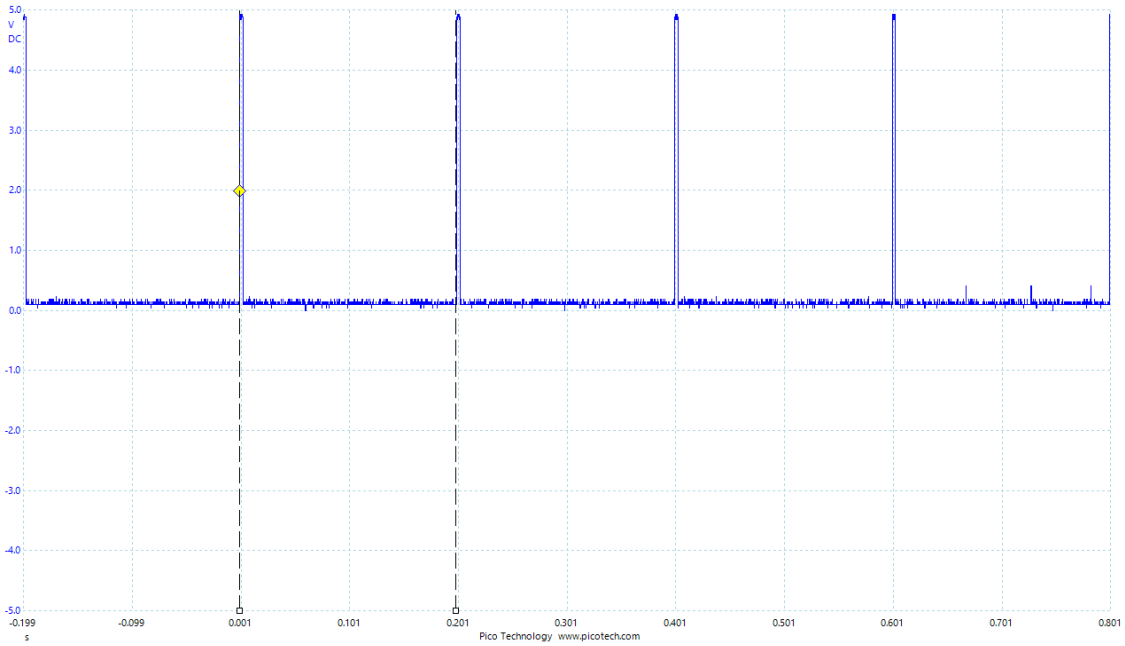


Figure 6.14 CAT2 ISR PIT1 task called by Dispatcher after ISR adds to the dispatcher

7. Conclusions

First we learned the main purpose of the OS system tick and its role in a typical RTOS system which is in simple terms the system's heartbeat. This rate is application specific and depends on the desired resolution of this time source in order to achieve the Real Time requirements imposed to the system, however, we must be careful if a higher tick rate is used since It will impose a higher overhead to the system.

Another point of consideration when implementing scheduler is the load that each task impose onto the system. We should always keep in mind that the maximum allowed execution time must not exceed the period of the fastest task (normally), otherwise the scheduler will behave erratically if no policy to handle such situations is included.
We were able to verify that point just by increasing the number of cycles of both for-loops, when the number reached the same or above value of the fastest task (1.65 ms), the system behavior was random and defining it was not possible.

In this final practice because of the requirement of made it preemptive and support CAT2 task required to develop knowledge on the architecture used by the system (Freescale HCS12) and use the mechanisms provided by the instruction set and architecture to manipulate the different registers which control the cpu code instruction execution and manipulate it according to the preemptive algorithm implemented in order to modify the normal flow of execution according to the priorities defined and the requirements of the RTOS itself. We also developed and understand the data structures required to implement this kind of systems and the common problems that appear in this kind of RTOS design which we had to implement in the code of this basic preemptive system. The nature of an environment is that multiple units of execution (tasks) share and contend for the same set of resources. As such, resource sharing requires careful coordination to ensure that each task can eventually acquire the needed resource or resources to continue execution. In a preemptive multitasking environment, resource sharing is a function of task priority. The higher the priority of a task, the more important the task is. Higher priority tasks have precedence over lower priority tasks when accessing shared resources. Therefore, resource sharing cannot violate this rule. In the current implementation the only resource available is the CPU.

B. REPROTE DE DISEÑO DE MÓDULO DE SOFTWARE DE COMUNICACIÓN LIN AUTOMOTRIZ.

Instituto Tecnológico de Estudios Superiores de Occidente

Maestría en Diseño Electrónico

Diseño de Software de Comunicaciones para Ambientes Embebidos

Reporte Final. Software de Comunicaciones usando LIN

Fernando Rodriguez Rivera
José Antonio Maza Moreno
Saúl Alfonso Nuñez Corona

1. Introduction

2. LIN (Local Interconnect Network) is a serial network protocol used for communication between components in vehicles. The need for a cheap serial network arose as the technologies and the facilities implemented in the car grew, while the CAN bus was too expensive to implement for every component in the car. European car manufacturers started using different serial communication topologies, which led to compatibility problems.

The LIN bus is an inexpensive serial communications protocol, which effectively supports remote application within a car's network. It is particularly intended for mechatronic nodes in distributed automotive applications, but is equally suited to industrial applications. It is intended to complement the existing CAN network leading to hierarchial networks within cars.
The protocol for the Local Interconnect Network (LIN) is based on the Volcano-Lite technology developed by the Volvo spin-out company Volcano Communications Technology (VCT). Since other car corporations also were interested in a more cost effective alternative to CAN, the LIN syndicate was created. In the middle of 1999 the first LIN protocol (1.0) was released by this syndicate. The protocol was updated twice in 2000. In November 2002 LIN 1.3 was released with changes mainly made in the physical layer. The latest version LIN 2.0 was released in 2003. With LIN 2.0 came some major changes and also some new features like diagnostics. The changes were mainly aimed at simplifying use of off-the-shelves slave nodes.

3. LIN Pysical Properties

The LIN-bus transceiver is a modified version of the transceiver used by the ISO 9141 standard. The bus is bidirectional and connected to the node transceiver, and also via a termination resistor and a diode to Vbat of the node (Figure 1).

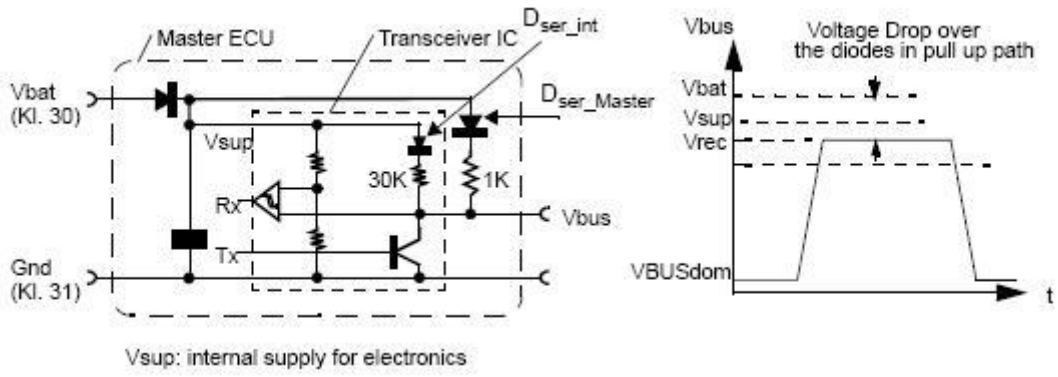


Figure 1: Description of a transceiver. (from the LIN 2.0 spec)

On the bus a logical low level (0) is dominant and a logical high level (1) is recessive.

Voltage supply (Vsup) for an ECU should be between 7 V and 18 V. The limits for how the level of the bus is interpreted are shown in figure 2.

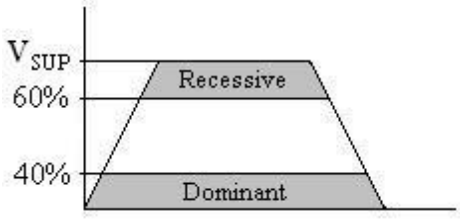


Figure 2: Determination of the logical level on the bus.

3.1. Data Transmission

The LIN network is described by a LDF (LIN Description File) which contains information about frames and signals. This file is used for creation of software in both master and slave.

The master node controls and make sure that the data frames are sent with the right interval and periodicity and that every frame gets enough time space on the bus. This scheduling is based on a LCF (LIN Configuration File) which is downloaded to the master node software.

All data is sent in a frame which contains a header, a response and some response space so the slave will have time to answer. Every frame is sent in a frame slot determined by the LCF.

Messages are created when the master node sends a frame containing a header. The slave node(s) then fills the frame with data depending on the header sent from the master.

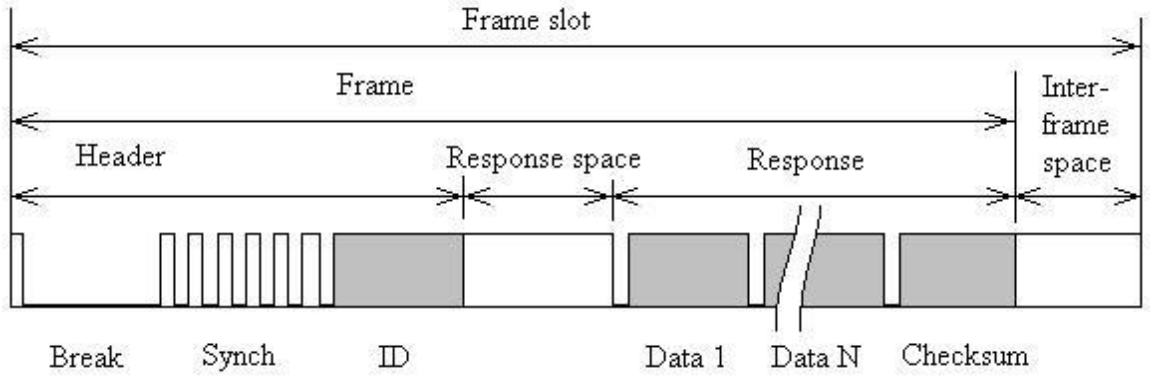


Figure 3: Example of LIN frame.

There are three different ways of transmitting frames on the bus: unconditional, event triggered, and sporadic frames.

Unconditional Frames

This is the "normal" type of LIN communication. The master sends a frame header in a scheduled frame slot and the designated slave node fills the frame with data.

Event Triggered Frames

The purpose of this method is to receive as much information from slave nodes without overloading the bus with frames. An event triggered frame can be filled with data from more than one slave node. A slave only updates the data in an event triggered frame when the value has changed. If more than one slave wants to update data in the frame a collision occurs. The master should then send unconditional frames to each of the slaves starting with the one with the highest priority.

Sporadic Frames

This method provides some dynamic behavior to the otherwise static LIN protocol. The header of a sporadic frame is only sent from the master when it knows that a signal has been updated in a slave node. Usually the master fills the data bytes of the frame itself and the slave nodes will be the receivers of the information.

3.1.1 Definition of a Byte Field

The protocol is byte oriented which means that data is sent one byte at a time. One byte field contains a start bit (dominant), 8 data bits and a stop bit (recessive). The data bits are sent LSB

first (least significant bit first). Data transmission can be divided into a master task and a slave task.

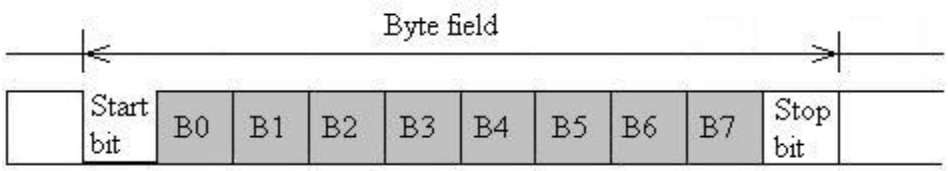


Figure 4: Structure of a byte field.

3.1.2 The Master Task

The frame (header) that is sent by the master contains three parts; synch break, synch byte and an ID-field. Each part begins with a start bit and ends with a stop bit.

Synch break marks the start of a message and has to be at least 13 dominant bits long including start bit. Synch break ends with a “break delimiter” which should be at least one recessive bit.

Figure 5: Break field.

Synch byte is sent to decide the time between two falling edges and thereby determine the transmission rate which the master uses. The bit pattern is 0x55 (01010101, max number of edges). This is especially usable for compatibility with off-the-shelves slave nodes.

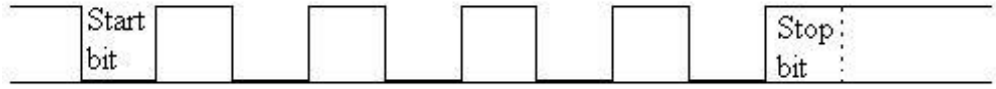


Figure 6: Synch byte field.

The ID field contains a 6 bits long identifier and two parity bits. The 6 bit identifier contains information about sender and receiver and the number of bytes which is expected in the response. The parity bits are calculated as followed: parity P0 is the result of logic “or” between ID0, ID1, ID2 and ID4. Parity P1 is the inverted result of logic “or” between ID1, ID3, ID4 and ID5.

Figure 7: ID field.

ID range		Frame length
0-31	0x00 – 0x1f	2
32-47	0x20 – 0x2f	4
48-63	0x30 – 0x3f	8

Figure 9: Frame length depending on ID.

The response (data field) from the slave can be 2, 4 or 8 bytes long depending on the two MSB (Most Significant Byte) of the identifier sent by the master. This ability came with LIN 2.0, older versions have a static length of 8 bytes.

Figure 8: The response data field.

3.1.3 The Slave Task

The slave waits for synch break and then the synchronization between master and slave begins on synch byte. Depending on the identifier sent from the master the slave will either receive or transmit or do nothing at all. A slave that should transmit sends the number of bytes which the master has requested and then ends the transmission with a checksum field.

There are two different kinds of checksum. The classic checksum is used in LIN 1.3 and consists of the inverted eight bit sum of all (8) data bytes in a message. The new checksum used in LIN 2.0 also incorporates the protected identifier in the checksum calculation. The inverted eight bit sum is not the same as modulo-256. Every time the sum is greater than 256, then 255 is subtracted. Ex: 240+32=272 à 272-255=17 and so on…

To save power the slave nodes will be put in a sleep mode after 4 seconds of bus inactivity or if the master has sent a sleep command. Wakeup from sleep mode is done by a dominant level on the bus which all nodes can create.

4. LIN Software Driver Development

In the image below, the typical flow diagram of the LIN communication stack is shown

4.1.

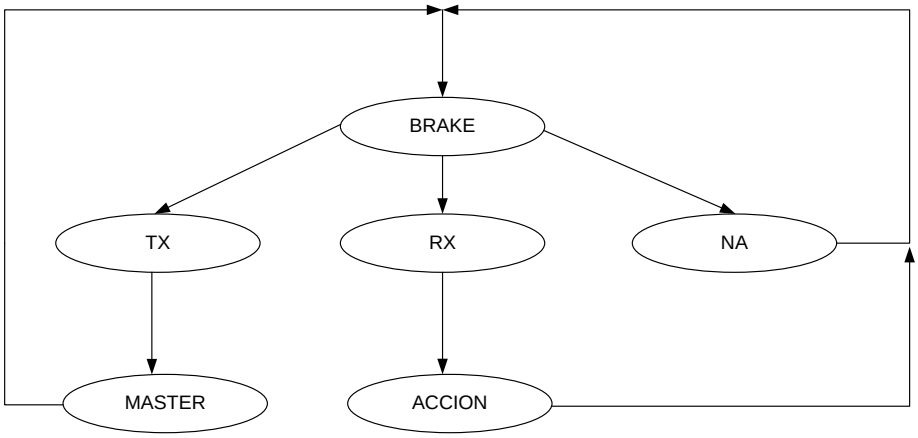


Figure 9. LIN Flow Diagram

In the diagram below, the Flow diagram of the MASTER/SLAVE interaction of two LIN nodes is described

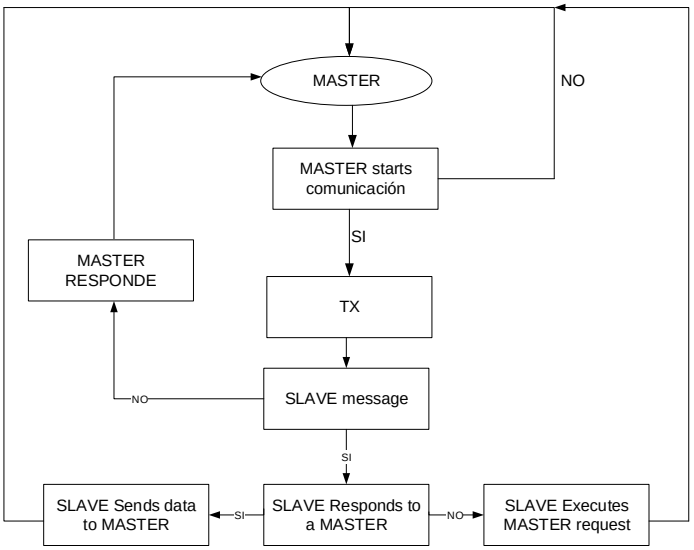


Figure 10. Flow diagram of information request from MASTER to SLAVE.

The flow diagram that the SLAVE must follow is shown in the figure 11.

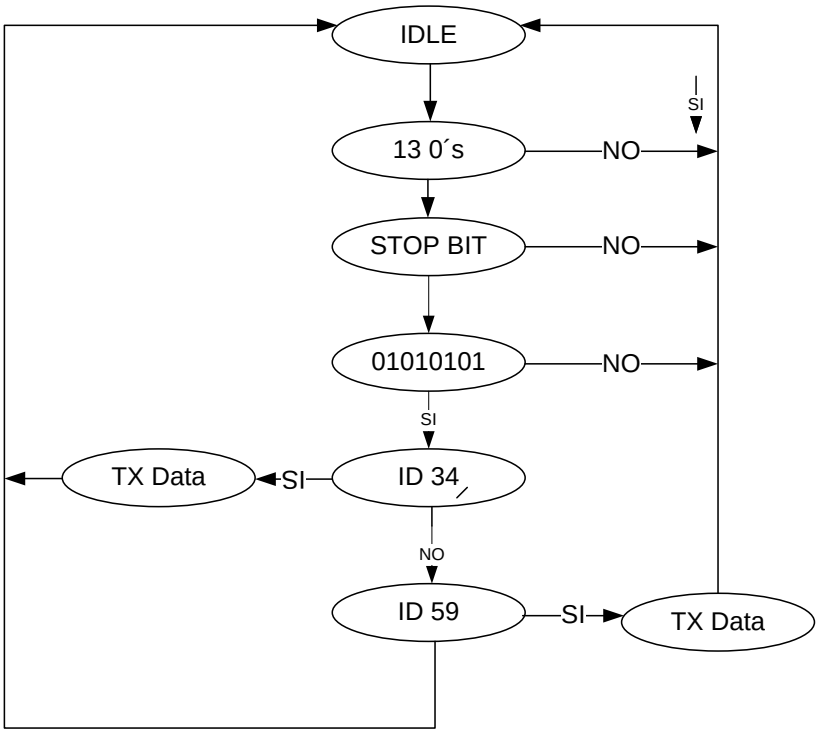


Figura 11. SLAVE flow diagram to verify LIN message

4.2. Data Structures and Source code of LIN Driver

```
typedef struct
{
  UINT16 RearFogLampInd : 1;
  UINT16 PositionLampInd: 1;
  UINT16 FrontFogLampInd: 1;
  UINT16 IgnitionKeyPos : 5;
  UINT16 SLVFuncIllum   : 4;
  UINT16 SLVSymbolIllum : 4;
       }Frame1_Packet;
```

Figure 12. Data structure for data transmission defined as a BitField to optimize memory usage and facilitate memory access

```
typedef struct
{
   UINT32 SLVSWPartNo  :  8;
   UINT32 SLVHWPartNoB0: 16;
   UINT32 SLVHWPartNoB1:  8;
}Frame2_Packet;
```

Figure 13. Frame 2 data structure defined to transmit 4 bytes of payload.

```
typedef struct{
   UINT32 FanIdealSpeed: 16;
   UINT32 FanMeasSpeed : 16;
   UINT32 WaterTemp    : 32;
}Frame3_Packet;
```

Figure 14. Frame 2 data structure BitField using 8 bytes as payload which is the máximum allowed by LIN standard

```
typedef struct{
  UINT8 identifier;
  UINT8 data_field[MAX_DATA];
  UINT8 size;
}lin_message;
```

Figure 15. Internal data structure for LIN message handling to be transmitted.

```
if(u32_TaskCounter % 40 )
{
      FLIP_LED(D2);
      msg_send.identifier = FRAME3_ID;                  //set ID
      msg_send.size = 8;
      msg_send.data_field[0] = 0xAA;    //set LIN data
      msg_send.data_field[1] = 0x99;    //set LIN data
      msg_send.data_field[2] = 0x88;    //set LIN data
      msg_send.data_field[3] = 0x77;    //set LIN data
      msg_send.data_field[4] = 0x66;    //set LIN data
      msg_send.data_field[5] = 0x55;    //set LIN data
      msg_send.data_field[6] = 0x44;    //set LIN data
      msg_send.data_field[7] = 0x33;    //set LIN data
      linStatus = LINCheckState(LIN_0);
      if( linStatus == LIN_IDLE )
            (void)LINSendMsg(LIN_0,  //LIN Channel
            TRUE,   //Master Mode = TRUE, Slave Mode = FALSE
            TRUE,   //Send Data = TRUE, Not Send Data = FALSE
            msg_send//Message to be Sent
            );   //send LIN mesage
}
```

Figura 16. Example of emsagge transmission from the master with response from the SLAVE.

```
void LINInit(UINT8 lin_num);
UINT8 LINSendMsg(UINT8 lin_num, UINT8 master, UINT8 send_data,
lin_message msg);
UINT8 LINGetMsg(UINT8 lin_num, UINT8 get_data, struct  message *msg);
lin_state LINCheckState(UINT8 lin_num);
```

Figure 17. LIN Driver APIs provided to facilitate data transmission

```
UINT8 LINCalcParity(UINT8 id)
{

  UINT8 parity, p0,p1;

  parity=id;
  p0=(BIT(parity,0)^BIT(parity,1)^BIT(parity,2)^BIT(parity,4))<<6;
  p1=(~(BIT(parity,1)^BIT(parity,3)^BIT(parity,4)^BIT(parity,5)))<<7;
  parity|=(p0|p1);
  return parity;
}
```

Figure 18. Algorithm used to calculate the parity byte of hte LIN frame

```
UINT8 LINCalcLIN_CHECKSUM(UINT8 *data, UINT8 size)
{
  unsigned int sum = 0;
  UINT8 i;
  for(i = 0; i < size; i++)
  {
    sum += data[i];
    if(sum&0xFF00)
      // Add carry
      sum = (sum&0x00FF) + 1;
  }
  sum ^= 0x00FF;
  return((UINT8)sum);
}
```

Figure 19. Checksum algorithm which allows to calculate checksums base don classical mode and enhanced LIN modes of operation

5. LIN Error Handling

This document describes how the LIN errors shall be handled based on the HS12 hardware features.

5.1. Preliminary considerations

As a first step we shall consider the different levels of specifications: An **AUTOSAR LIN driver** should be developed, capable to implement the **LIN Protocol**, which shall work on a **HS12 Hardware**. Now if we focus on the error handling topic, then we shall consider:

1. AUTOSAR LIN Driver possible status, and remark those that are generated by an error condition.
2. LIN Protocol error detections specification.
3. HS12 Hardware support to detect Serial Communication Interface errors.

LIN related status and errors specifications

LIN AUTOSAR Lin_StatusType values:
(Autosar Specification of LIN Driver V1.5.0)
LIN_NOT_OK
LIN_TX_OK
LIN_TX_BUSY
LIN_TX_HEADER_ERROR
- Mismatch between sent and read back data
- Identifier parity error
- Physical bus error

LIN_TX_ERROR
- Mismatch between sent and read back data
- Physical bus error

LIN_RX_OK
LIN_RX_BUSY
LIN_RX_ERROR
- Framing error
- Overrun error
- Checksum error
- Short response

LIN_RX_NO_RESPONSE
- No response byte has been received so far

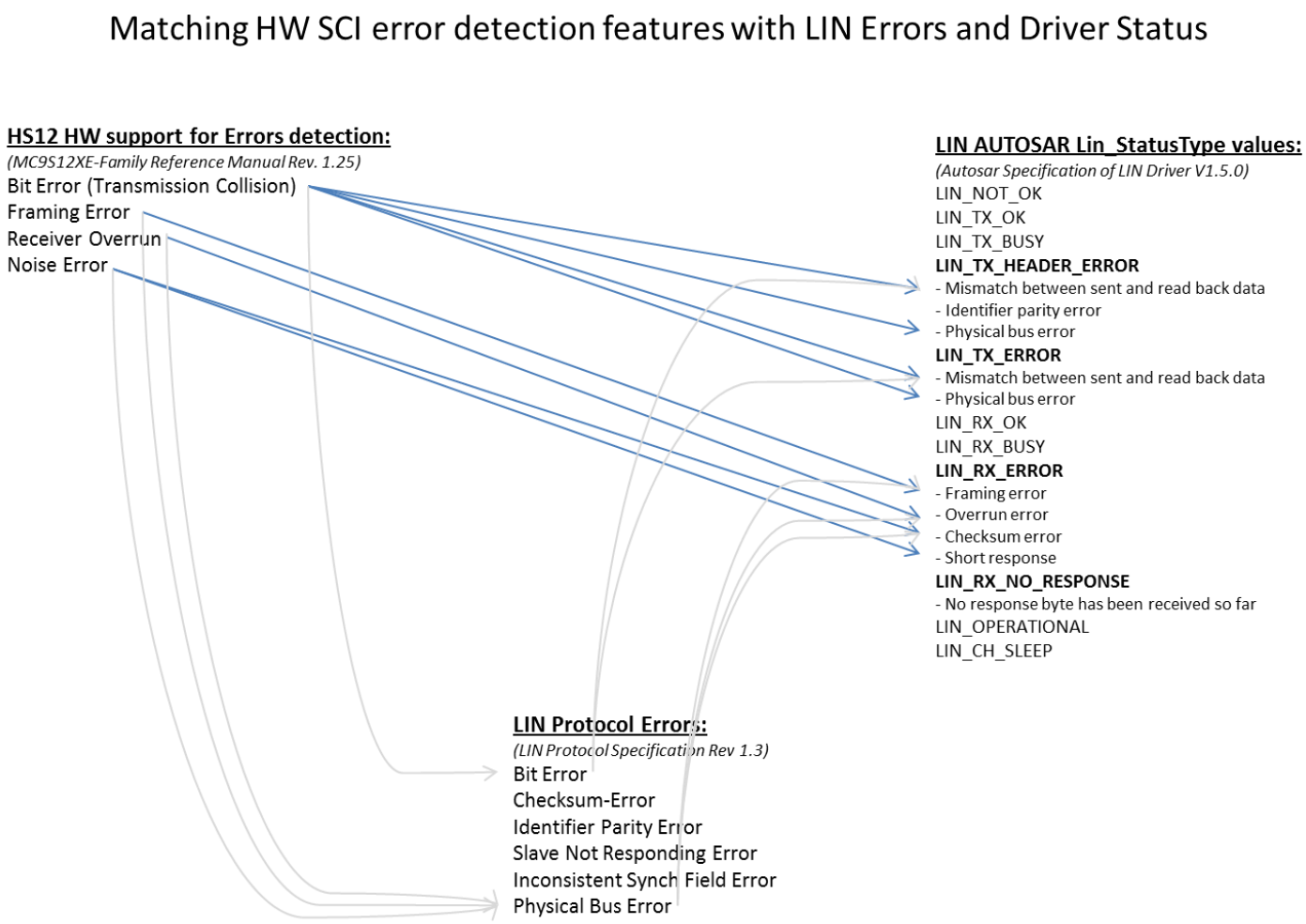
LIN_OPERATIONAL
LIN_CH_SLEEP

LIN Protocol Errors:
(LIN Protocol Specification Rev 1.3)
Bit Error
Checksum-Error
Identifier Parity Error
Slave Not Responding Error
Inconsistent Synch Field Error
Physical Bus Error

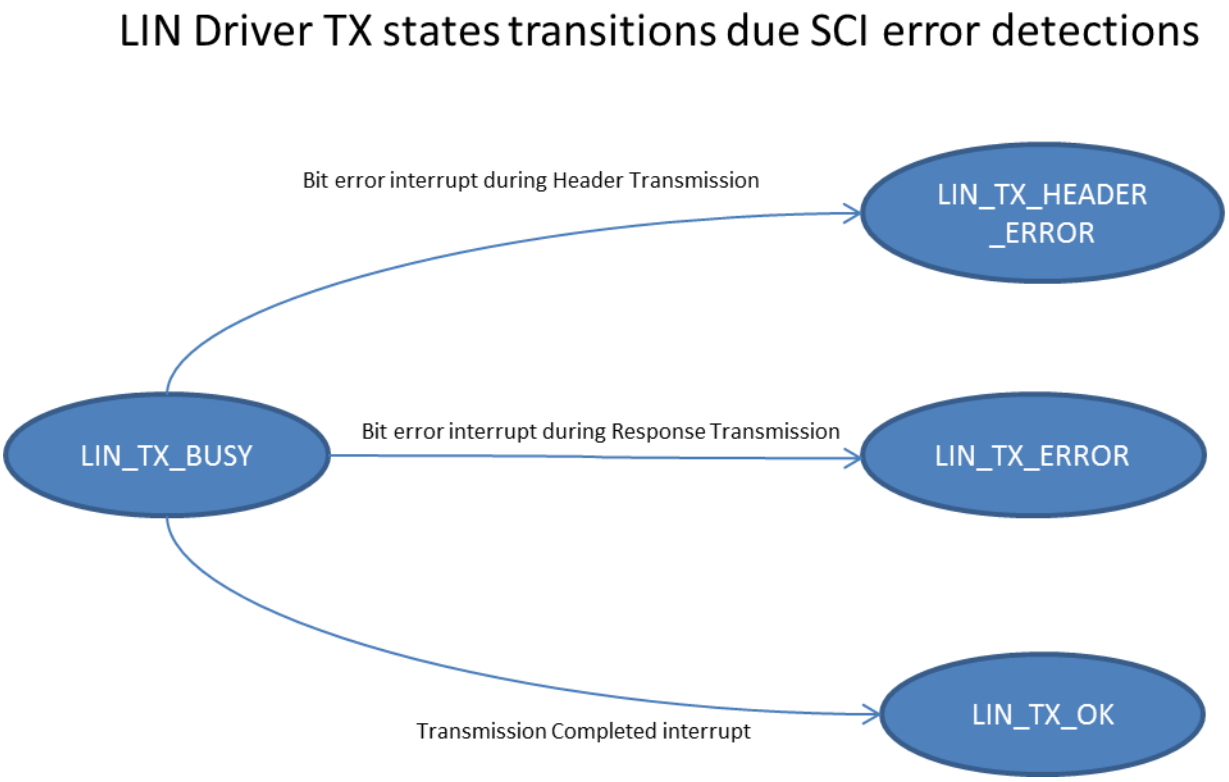
HS12 HW support for Errors detection:
(MC9S12XE-Family Reference Manual Rev. 1.25)
Bit Error (Transmission Collision)
Receiver Overrun
Noise Error
Framing Error

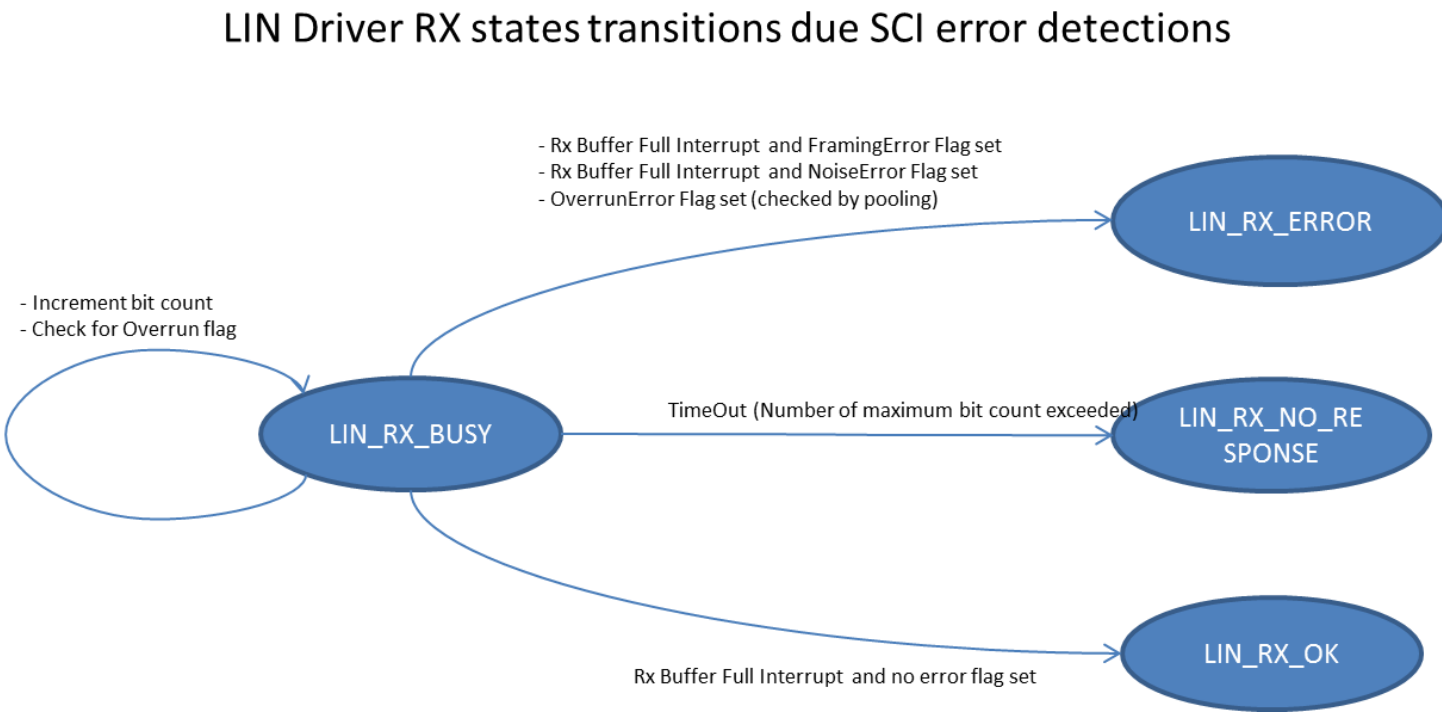
5.2. Linking HW resources with errors and status

Now if se rewrite the previous diagram in a reverse orders we could identify which HW resources can be used to detect specified LIN errors which would lead to some LIN driver status:



5.3. In the following sub-state-machines we can specify the LIN Driver status transitions due SCI Error detections:




LIN Driver RX states transitions due SCI error detections
- Rx Buffer Full Interrupt and FramingError Flag set
- Rx Buffer Full Interrupt and NoiseError Flag set
- OverrunError Flag set (checked by pooling)
LIN_RX_ERROR
- Increment bit count
- Check for Overrun flag
LIN_RX_BUSY
TimeOut (Number of maximum bit count exceeded)
LIN_RX_NO_RESPONSE
LIN_RX_OK
Rx Buffer Full Interrupt and no error flag set

5.4. Considerations in operation of Hardware Error Detection

5.4.1 Bit Error

- Bit error circuitry configuration is performed at register **SCIACR2:BERRM[1:0]** where operation is set as follows:

BERRM1	BERRM0	
0	0	Bit error detect circuit disable
0	1	Sampling at 9th time tick
1	0	Sampling at 13th time tick
1	1	Reserved

- Bit error Interrupt is enable by setting register **SCIACR1[BERRIE]**
- Bit error Interrupt flag is located at register **SCIASR1[BERRIF]** and is set when a Bit error condition occurs. The flag is cleared by writhing "1" to it.

Note: In order to Read/Write previous register Auxiliary Memory flag should be set: AMAP=1.

5.4.2 Overrun error Flag

- This flag is located at register **SCISR1[OR]** and is set when an overrun condition occurs. It means when a new frame arrives to the shifting registers but previous data has not been read yet.
- This flag is cleared by reading SCISR1 with OR set, and then read SCIDRL.

5.4.3 Noise error Flag

- This flag is located at register **SCISR1[NF]** and is set when a noise is detected in the channel.
- This flag is set on the same cycle of RDRF (Data Rx Full).
- This flag is not set if OR is set.
- This flag is cleared by reading SCISR1 with NF set, and then read SCIDRL.

5.4.4 Framing error Flag

- This flag is located at register **SCISR1[FE]** and is set when a logic 0 is accepted as a stop bit.
- This flag is set on the same cycle of RDRF (Data Rx Full).
- This flag is not set if OR is set.
- When this flag is set, it inhibits further data reception, until cleared.

- This flag is cleared by reading SCISR1 with FE set, and then read SCIDRL.

5.5. LIN: Driver Análisis and results

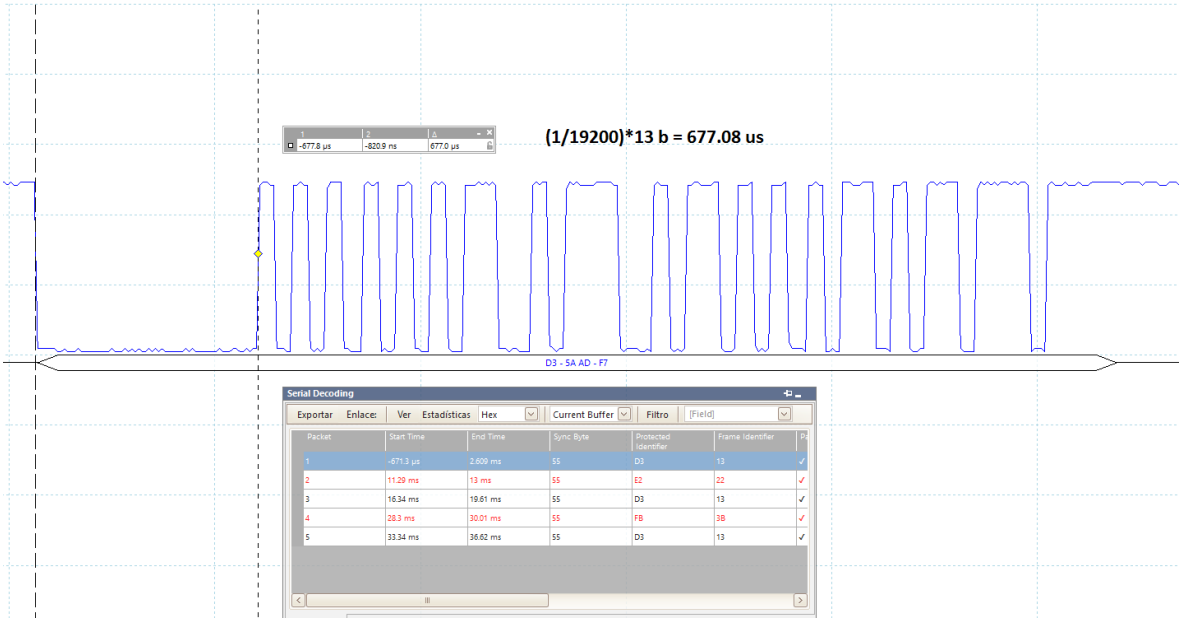


Figure 20. In this figure It is shown BREAK signal with a length of 13 bit measured with a transfer speed of 19200 bps

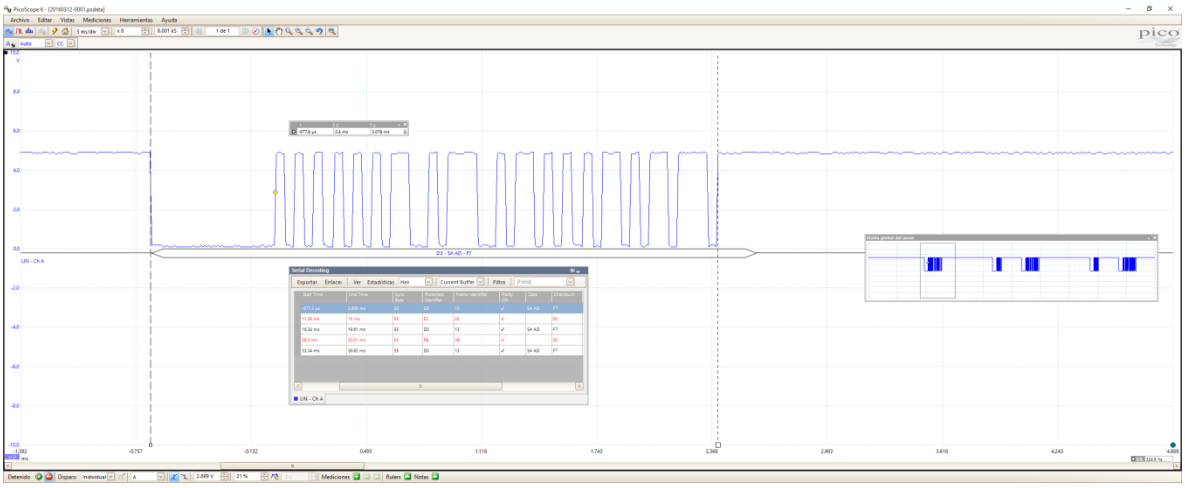


Figure 21. Transmission example of Frame 1 with MASTER response.

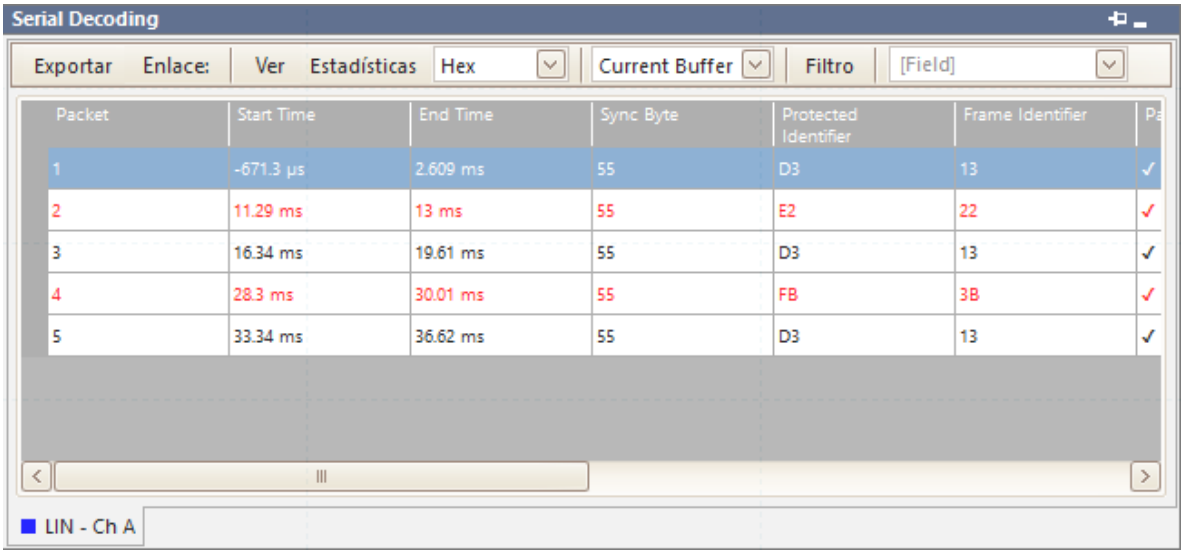


Figure 22. Data traces decoded by the oscilloscope using LIN protocol.

In the figure 22 the table show the decoded data from the oscilloscope using LIN protocol. It can be observed the PID and the ID match with the frame IDs defined i nthe driver stack besides showing the SYNCH byte transmitted properly.

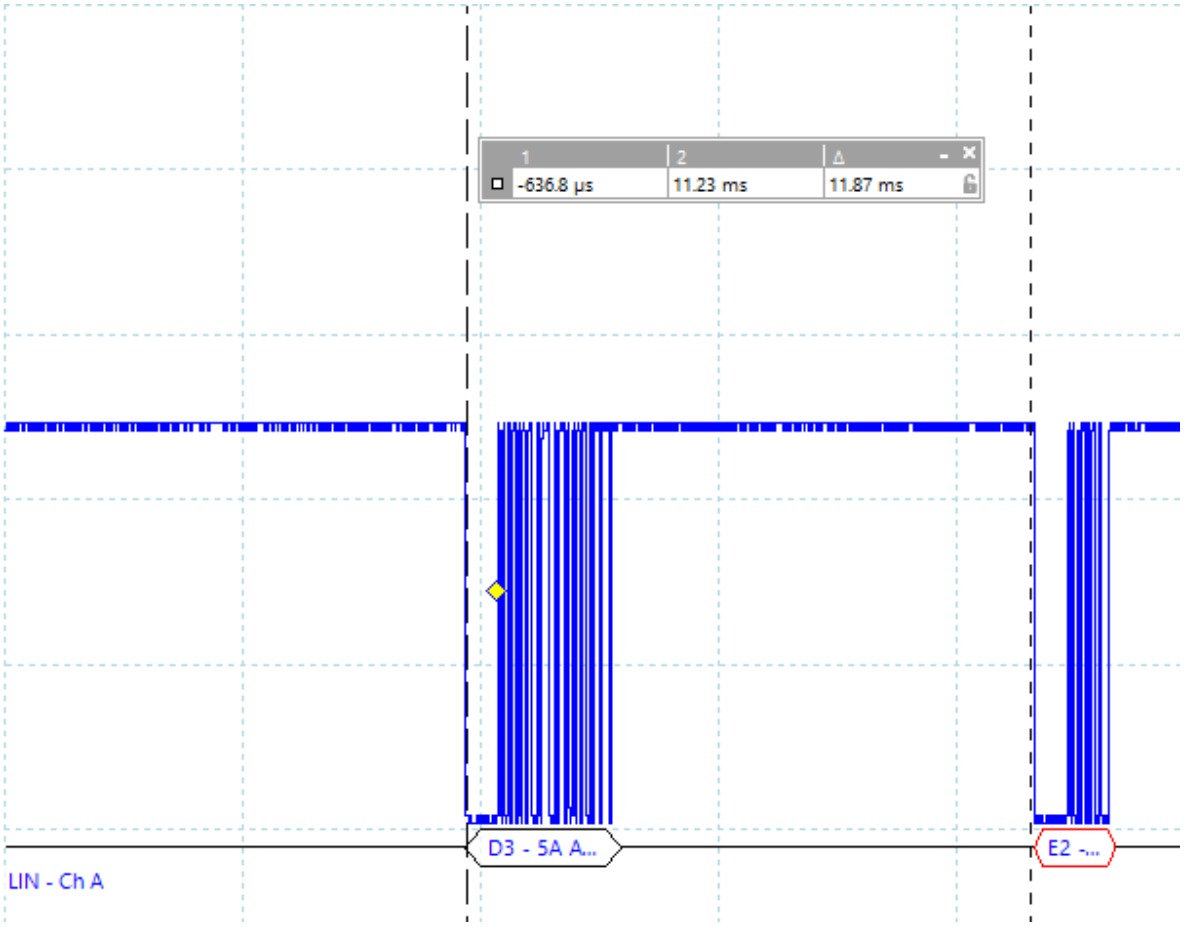


Figure 23. Inter frame message space of 10 ms between different LIN frames

6. Conclusions

The present owrk discusses the implementation of a basic LIN driver using the SC peripherial from the Freescale Board that acomplished the following functionality:
Messsage assembling of standard LIN communication frames creating the basic software infrastructure required to calcule the different parameters that the LIN frame is composed such ase PID, BREAK signal, and SYNCH byte as required by the LIN standard

This work allowed me to understand the LIN protocol and its structure and alos the basic operating principles of a network using this protocol.

It also helped me to understan the different error codes that can be generated and the reason of those as well as the posible erro handling on these failure scnearios

7. Bibliography

[1] Introducción al Bus de Red Local de Interconexión http://www.ni.com/white-paper/9733/es/
[2] Course notes, 02 Specification SCI_Driver SWS_V1.0.pdf
[3] Course notes, 03 ECU Abstraction Layer Comm Protocol.pdf

C. REPROTE DE DISEÑO DE MÓDULO DE SOFTWARE DE COMUNICACIÓN LIN SAUTOMOTRIZ MULTI-FUNCTION SWITCH.

Instituto Tecnológico de Estudios Superiores de Occidente

Maestría en Diseño Electrónico

Ingeniería de Software

REPORTE FINAL. DISEÑO DE MÓDULO DE SOFTWARE DE COMUNICACIÓN LIN SAUTOMOTRIZ MULTI-FUNCTION SWITCH.

Fernando Rodriguez Rivera
José Antonio Maza Moreno
Saúl Alfonso Nuñez Corona

Introduction

Typically, vehicle components such as a climate control system, radio, compact disk player, and vehicle information displays are controlled independently by separate control devices mounted on an instrument panel and/or console. This requires a significant amount of packaging space in the instrument panel and/or console, and involves multiple components requiring significant engineering, manufacturing and assembly costs. Additionally, control keys of the various control devices are generally spaced across the instrument panel and/or console so that the driver must sometimes search for multiple control keys in order to control the various vehicle components.
The Multi-function switch controller is an automotive module used in different modern vehicles which typically offer the following functionality:

- Allow multiple vehicle functions (wipers, lights) to be controller without increasing the number of switches on the steering wheel, enabling drivers to select and operate more vehicle functions while keeping their hands on the wheel.
- Offer interior styling functionality.
- Offers a Human Machine Interface to interact with the vehicle driver and avoid distraction during their operation.

The scope of this project is to define the minimum set of system and software requirements of a multi-function switch as well to define the basic software architecture required by this system to operate and interact with other vehicle modules.

System Requirements

In systems engineering and software engineering, **requirements analysis** encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product or project, taking account of the possibly conflicting requirements of the various stakeholders, *analyzing, documenting, validating and managing* software or system requirements.
Requirements analysis is critical to the success or failure of a systems or software project. The requirements should be documented, actionable, measurable, testable, traceable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:[1]

Customer Requirements

Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:
Operational distribution or deployment: Where will the system be used?
Mission profile or scenario: How will the system accomplish its mission objective?
Performance and related parameters: What are the critical system parameters to accomplish the mission?
Utilization environments: How are the various system components to be used?
Effectiveness requirements: How effective or efficient must the system be in performing its mission?
Operational life cycle: How long will the system be in use by the user?
Environment: What environments will the system be expected to operate in an effective manner?

Architectural Requirements

Architectural requirements explain what has to be done by identifying the necessary systems architecture of a system.

Structural Requirements

Structural requirements explain what has to be done by identifying the necessary structure of a system.

Behavioral Requirements

Behavioral requirements explain what has to be done by identifying the necessary behavior of a system.

Functional Requirements

Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the toplevel functions for functional analysis.[1]

Non-functional Requirements

Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

Core Functionality and Ancillary Functionality Requirements

Murali Chemuturi defined requirements into Core Functionality and Ancillary Functionality requirements. Core Functionality requirements are those without fulfilling which the product cannot be useful at all. Ancillary Functionality requirements are those that are supportive to Core Functionality. The product can continue to work even if some or all of the Ancillary Functionality requirements are fulfilled but with some side effects. Security, safety, user friendliness and so on are examples of Ancillary Functionality requirements.[5]

Performance Requirements

The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.

Design Requirements

The "build to," "code to," and "buy to" requirements for products and "how to execute" requirements for processes expressed in technical data packages and technical manuals.

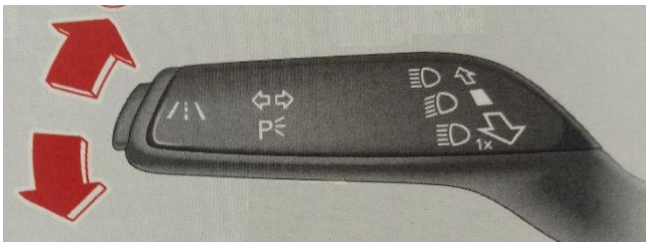
Derived Requirements

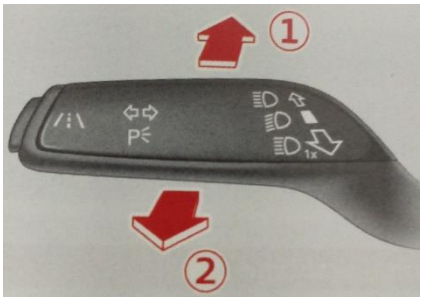
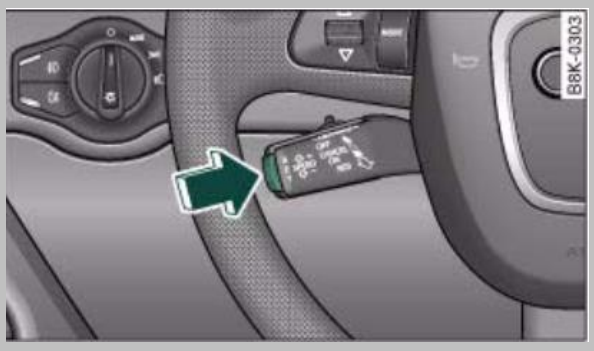
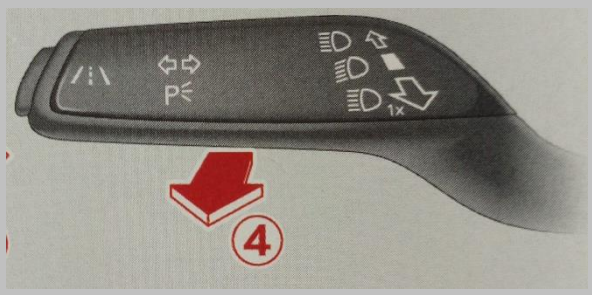
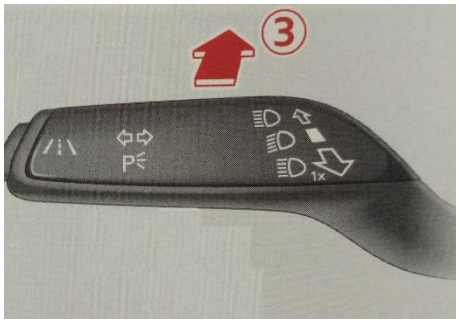
Requirements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.

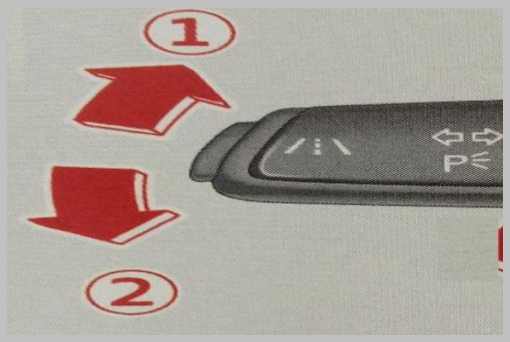
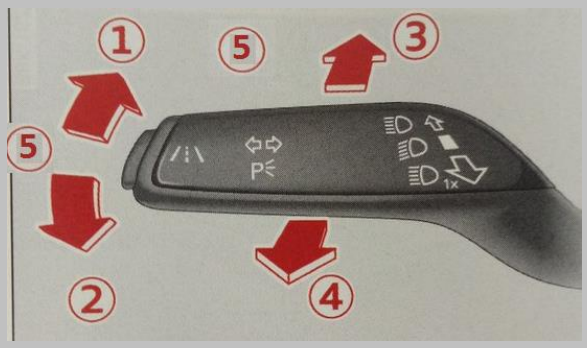
Allocated Requirements

A requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements. Example: A 100-pound item that consists of two subsystems might result in weight requirements of 70 pounds and 30 pounds for the two lower-level items.

The following table shows the system requirements defined for the multi-function control switch

DOORS Id	*Section*	*Requirements*
FR001	**Light_Control**	The Light Control lever **shall** have 2 directional movements on Y axis direction 

FR002		The Light Control lever **shall** have 3 directional movements on Z axis direction 
FR003	**Light_Control**	The Light Control lever **shall** be positioned at the left side of the Control Unit Dashboard. 
FR004		The Light Control lever **shall** activate the light horn with negative Z axis movements of the 
FR005	**Light_Control**	The Light Control lever **shall** activate the high lights with positive Z axis movements of the 

FR006	**Light_Control**	The Light Control lever **shall** activate the directional lights using the Y axis 
FR007	**Light_Control**	The light Control lever **shall** have 1 stop position in push direction within the Z axis. 
FR008	**Light_Control**	The light Control lever **shall** have the following functionality as described in the table **FR00** referenced in the image 

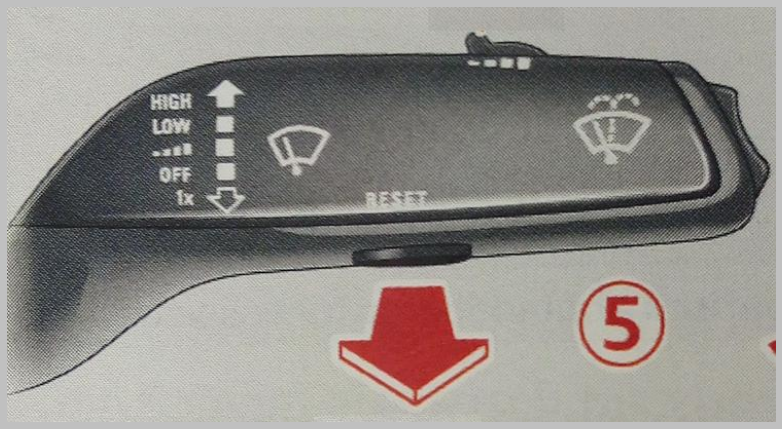
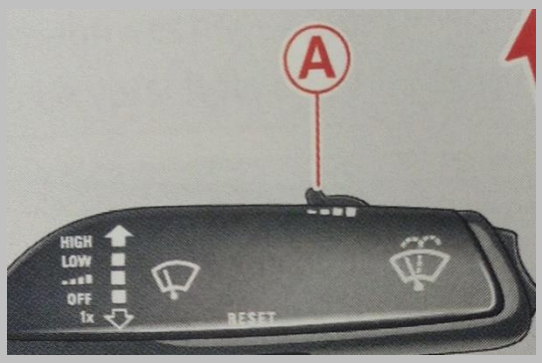
FR009

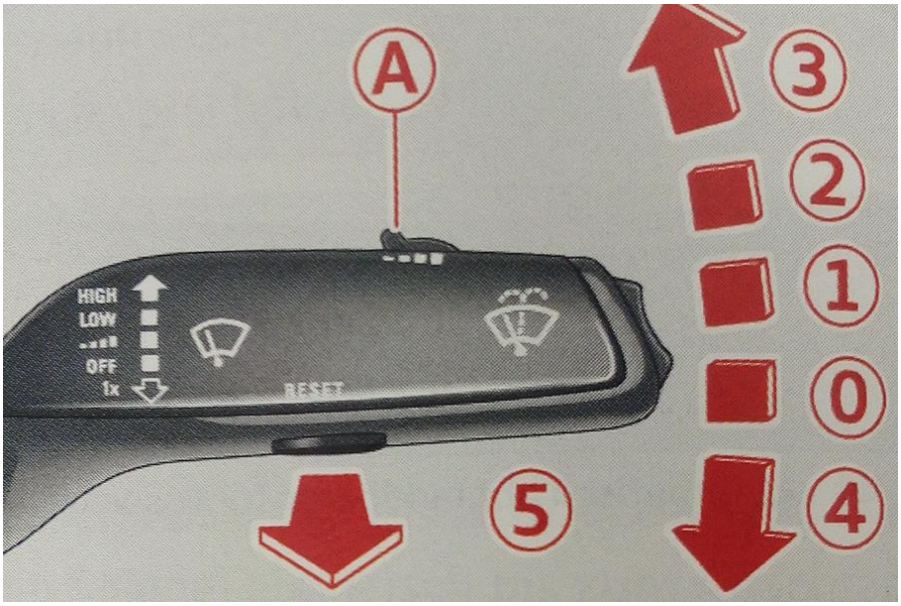
Light Lever Functionality

Positi	Function	Description
1	Right Turn Signal	Lever position to indicate right. A signal is send to indicate this function is used while this Position is active
2	Left Turn Signal	Lever position to indicate left.. A signal is send to indicate this function is used while this Position is active
3	Main Beam Headlights	Press the lever forward to switch on the main beams. A signal is send to indicate this function is used while this Position is active
4	Headlight Flasher	Pull the lever towards the steering wheel to operate the flasher. A signal is send to indicate this function is used while this Position is active
5	Off Position	Nothing is processed in the central Lever state

FR010 **Wiper_Control**

The Wiper Control lever **shall** have a 5 different directional movements on Y axis including

FR011	**Wiper_Control**	The Wiper Control lever **shall** have a 1 directional movement on Z axis. 
FR012	**Wiper_Control**	The light control lever **shall** include a selector for "Rain Sensor Sensitivity" with 4 positions 

FR013	**Wiper_Control**	The lWiper Control lever **shall** have the following functionality as described in the table **FR** referenced in the image 

FR014		

Wiper Lever Functionality

osi	Function	Description
0	Off Position	All the wiper functionality is turned off. Not
1	Intermitent Wipe	Move the lever to position 1 to operate the "Intermittent Wiper functionality. A signal s be sent to indicate this function is used whil this Position is active along with the Rain Sensor Sensitivity Selected
2	Slow Wiper Speed	Move the lever to position 2 to operate the "Slow Wiper Speed" functionality. A signal is send to indicate this function is used while t Position is active
3	Fast Wiper Speed	Move the lever to position 3 to operate the "Fast Wiper Speed" functionality. A signal is send to indicate this function is used while this Position is active
4	Brief Wipe	Move the lever to position 2 to operate the brief wipe functionality. A signal is send to indicate this function is used while this Posit is active
5	Auto Wash and Wiper	Release the lever again. The washer will sto and the wipers will keep running for approximately 4 seconds. The number of wi cycles varies according to the length of time lever is pulled. A signal is send to indicate th function is used while this Position is active. signal is send to indicate this function is use while this Position is active
A	Rain Sensor	The rain sensor will only function in the intermittent wipe position 1. The intermittent wipe function is activated automatically when it starts to rain. It has 4 levels of sensitivity

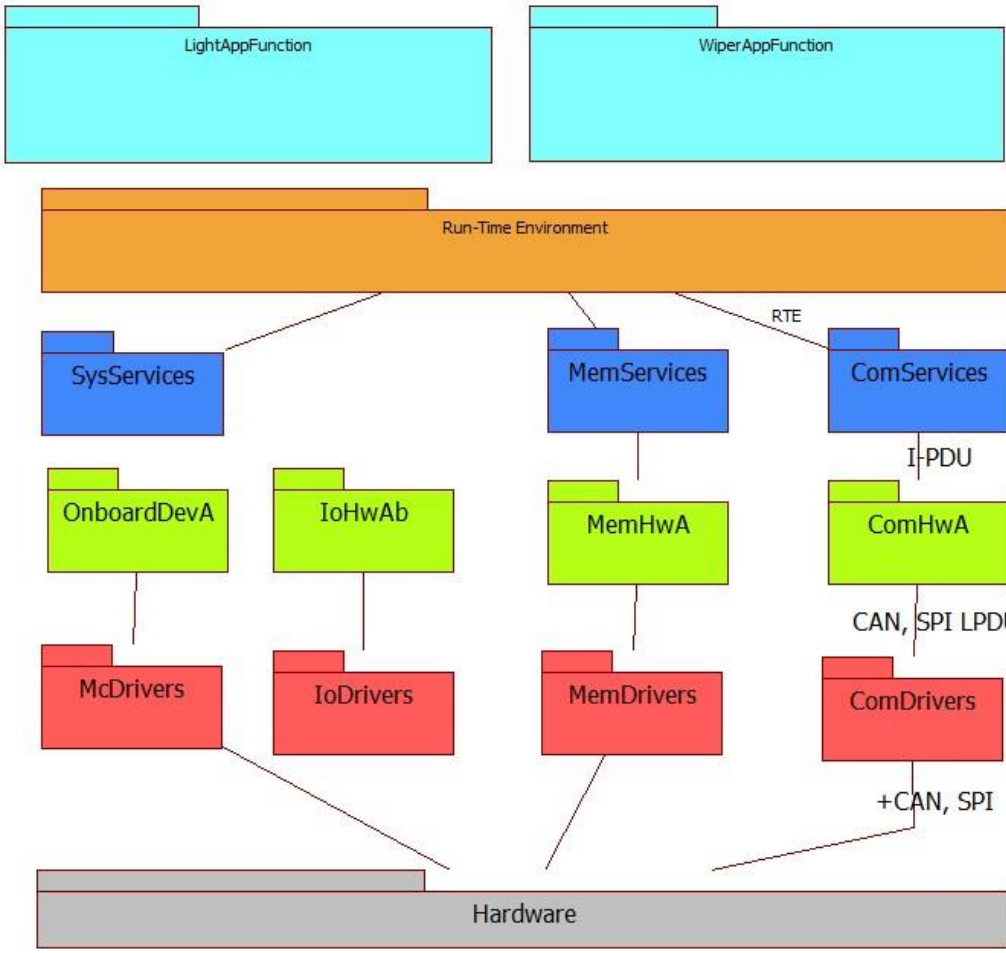
FR014		
FR015	**Wiper Control**	The Wiper Control lever **shall** be positioned at the right side of the Control Unit Dashboard 
FR0016	**CAN Protocol**	Data **shall** be transmitted on a cental CAN bus
FR0017	**CAN Protocol**	Specially formatted CAN bus messages **shall** be put on the bus communicating the differe Wiper Control and Light Control levers
FR0018	**CAN Protocol**	In case the CAN module enters Bus Off It **shall** create a DTC and transmit it to the ystem again
FR0019	**CAN Protocol**	The transitions of the module from Bus Off to Bus Active **shall** be controlled by the applica

FR0020	**CAN Protocol**	The application **shall return** from Bus Off State once 128 ocurrencies of recesive state hav monitored by hte CAN Module
FR021	**Specification Requirements**	The software **shall** be based on AUTOSAR 3.0
FR022	**General**	The system **shall** reset itself if It hangs after 1 minute
FR023	**General**	The system **shall** report the light and wiper status functionality periodically while the modul

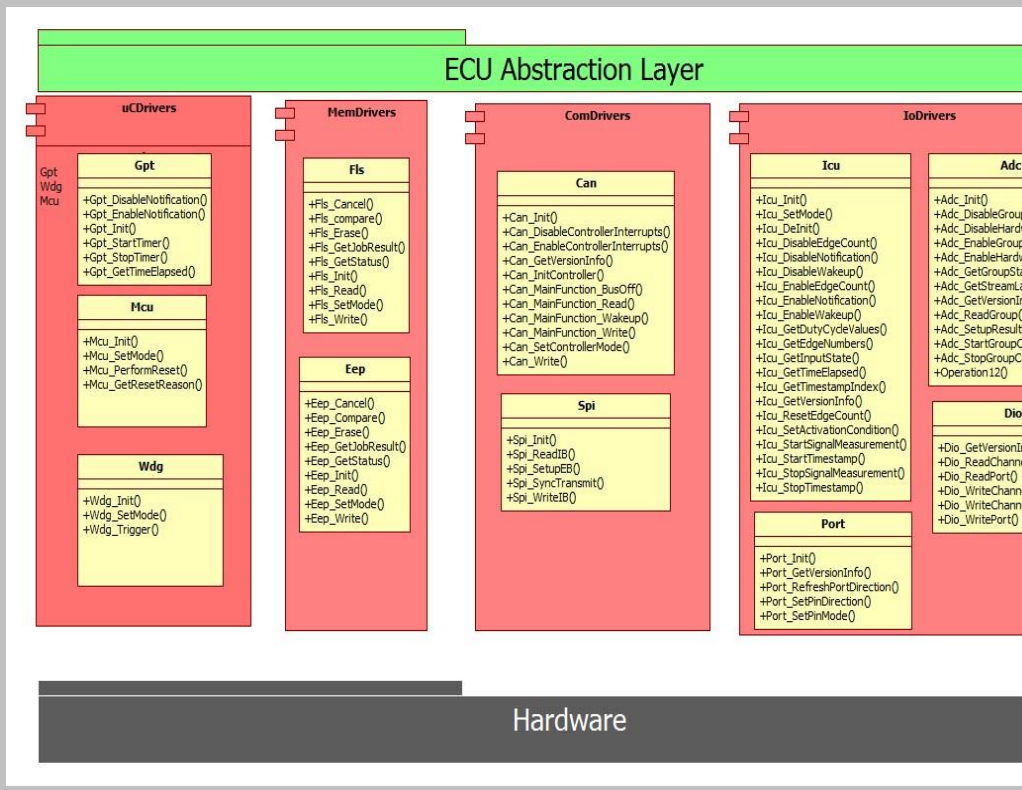
Software Requirements

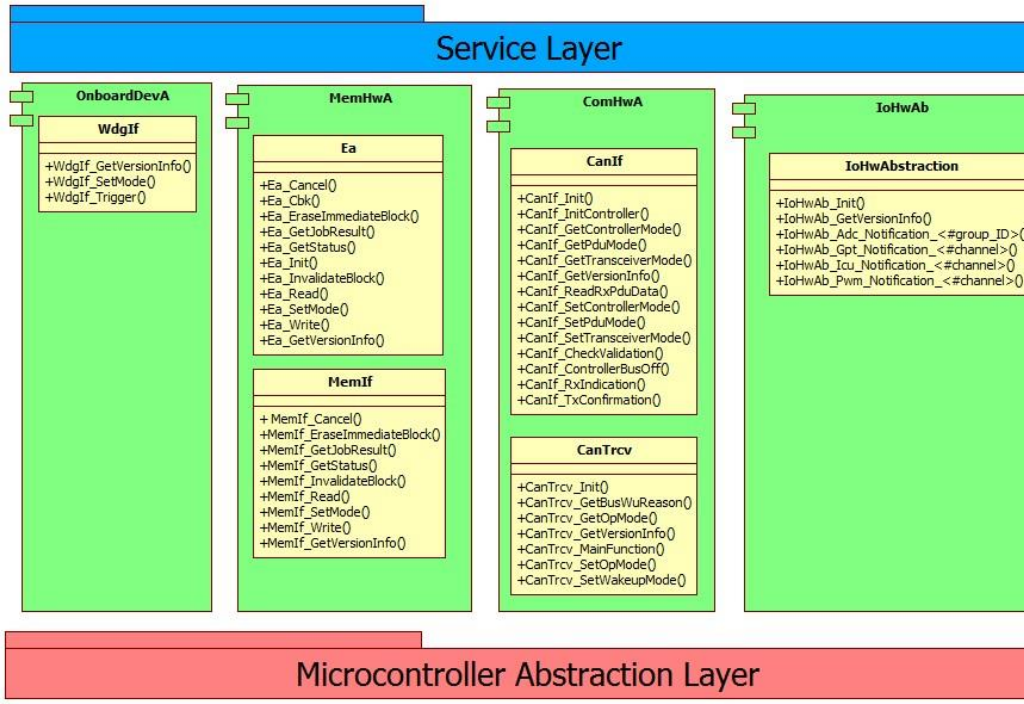
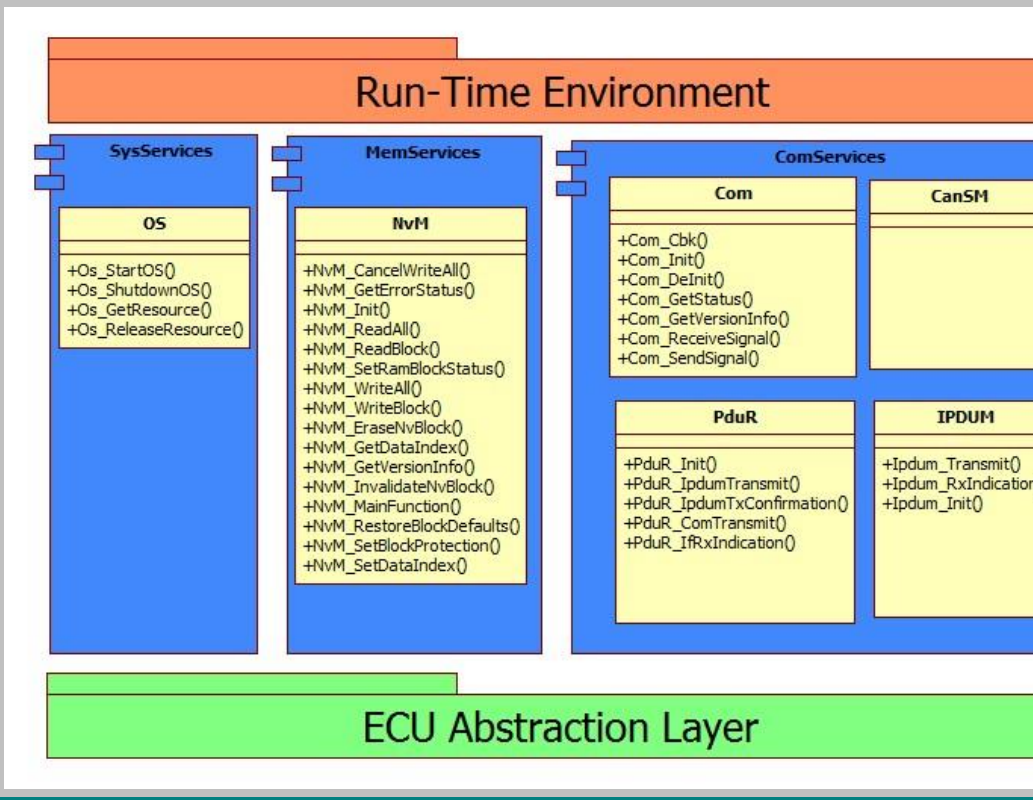
The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design. A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design

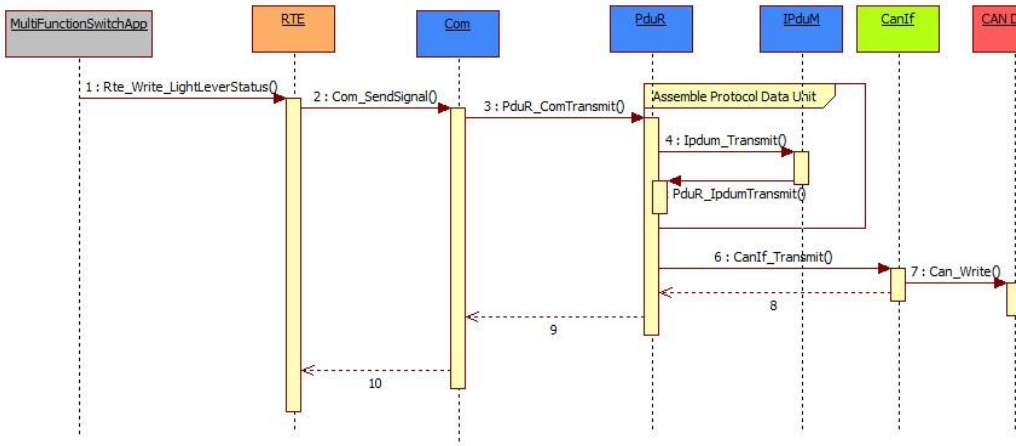
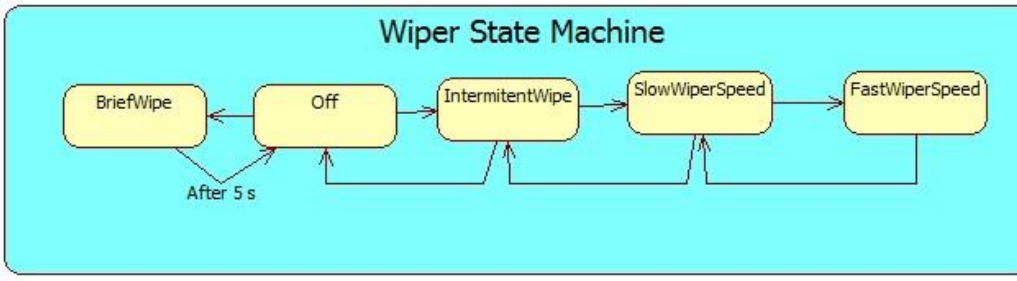
DOORS Id	***Section***	***Requirements***

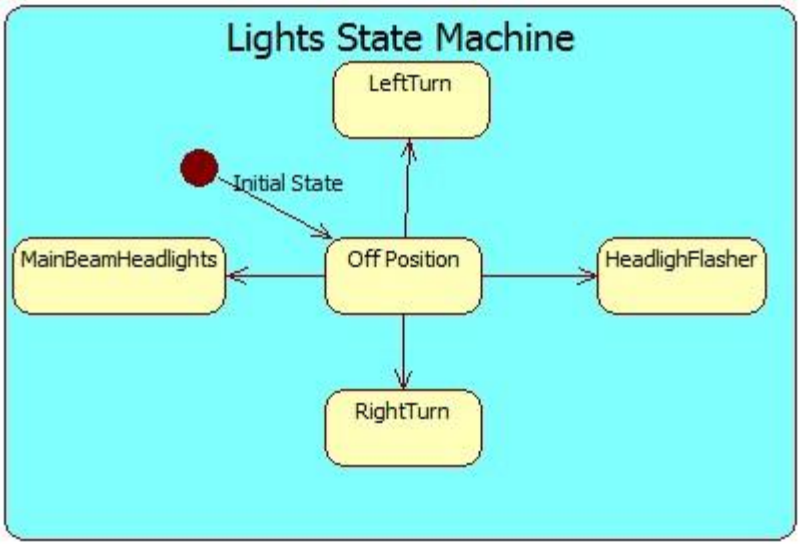
SR001	**Software Architecture**	The system **shall** be implemented using AUTOSAR 3.0 standard 

SR002	**Software Architecture**	The Autosar **MCAL**(Microcontroller Abstraction Layer) Layer **shall** be composed of the fo



SR003	**Software Architecture**	The Autosar ECUAL(ECU Abstraction Layer) **shall** be composed of the following module 
SR004	**Sofware Architecture**	The Autosar SL(ServiceLayer) **shall** be composed of the following modules 

SR005	**Software Architecture**	The CAN communcation stack **shall** follow the sequence flow defined in the diagram for t 
SR006	**Software Architecture**	The CAN communcation stack **shall** follow the sequence flow defined in the diagram for
SR007	**Software Architecture**	The Os System Used **shall** be Autosar compliant with version 3.0 of the standard
SR008	**Software Architecture**	The CAN communication stack **shall** be implemented following the Autsoar 3.0 standard
SR009	**Software Architecture**	The System **shall** detect if the system is hanged by the usage of Wathdog functionality de module which is part of the Software Architecture
SR010	**Software Architecture**	The ystem **shall** reset after 1 minute if the Watchdog functionality has not been signaled period
SR011	**Software Architecture**	The Wiper Functionality **shall** follow State Machine described in the figure below 

SR012	**Software Architecture**	The Lights Functionality **shall** follow State Machine described in the figure below 
SR013	**CAN Communication**	The module shall periodically report the status of the Wipe and Light Functions
SR014	**CAN Communication**	The module shall transmit with CAN Id 0X2C1
SR015	**CAN Communication**	The CAN frame shall have 3 bytes of Payload(data)
SR016	**CAN Communication**	The CAN frame payload shall be structured as follows(describe payload structure
		Describe each field of the CAN frame

Conclusions

Requirements analysis is the first stage in the systems engineering process and software development process.
Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design.

Systems engineering model of Specification and Levels of Development. During system development a series of specifications are generated to describe the system at different levels of detail. These program unique specifications form the core of the configuration baselines.

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them.