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 FORMACIÓN COMPLEMENTARIA EN EL ÁREA DE SISTEMAS EMBEBIDOS Y TELECOMUNICACIONES

Trabajo recepcional que para obtener el grado de

MAESTRO EN DISEÑO ELECTRÓNICO

Presentan: Fernando Rodríguez Rivera

Director: Mtro. José Luis Chávez Hurtado

San Pedro Tlaquepaque, Jalisco. 27 de Septiembre de 2017.

Contenido

Introducción 1

1. Resumen de los proyectos realizados 2

1.1. PREEMPTION.. 2
1.1.1 Introducción 2
1.1.2 Antecedentes 2
1.1.3 Solución desarrollada 3
1.1.4 Análisis de resultados..4
1.1.5 Conclusiones 4
1.2. SAE J2716 IMPLEMENTATION ON XGATE 5
1.2.1 Introducción 5
1.2.2 Antecedentes 5
1.2.3 Solución desarrollada 6
1.2.4 Análisis de resultados..7
1.2.5 Conclusiones 7
1.3. CAN MULTICORE.. 8
1.3.1 Introducción 8
1.3.2 Antecedentes 8
1.3.3 Solución desarrollada 9
1.3.4 Análisis de resultados..9
1.3.5 Conclusiones 10

2. Conclusiones 11

Apéndices 12

A. Reporte: Preemtion..13

B. Reporte: SAE J2716 Implementation on XGATE.......................................20

C. reporte: CAN - Multicore...29

Introducción

El presente trabajo contiene un análisis de tres proyectos realizados en asignaturas correspondientes al área de concentración de Sistemas Embebidos y Telecomunicaciones. Por ser ésta el área en la que he desarrollado mi carrera profesional teniendo experiencia en sistemas de control y adquisición de datos, sistemas multimedia para dispositivos móviles y actualmente en sistemas embebidos para el sector automotriz.

Los sistemas embebidos en el área automotriz, mantienen una constante tendencia a proveer soluciones cada vez más complejas. Hace 30 años el panel de instrumentos de un automóvil común solo contaba con un micro controlador y un LCD necesarios para desplegar información básica como Velocidad, o Revoluciones del motor, posteriormente hace unos 20 años se introdujo el concepto de una red de comunicación interna en el carro, en automóviles provistos de una mayor cantidad de sensores y estaciones de procesamiento. No obstante en la actualidad la complejidad de los sistemas sigue en aumento, en un futuro muy cercano será común encontrar en los automóviles verdaderas estaciones de información que pueden procesar no solo la información propia del automóvil, además se deberá procesar la comunicación con otros vehículos, e interactuar con elementos del exterior para asistir la conducción ya sea parcial o totalmente. Esto implica soluciones más complejas donde ya no sólo se trata de procesadores comunicados mediante una red interna, ahora podremos encontrar dentro de los vehículos unidades de procesamiento de varios núcleos, mayores capacidades de almacenamiento, mayor número de sensores, diferentes tipos de redes internas de comunicación y conectividad con redes externas ya sea hacia otros vehículos o estaciones de información.

Las materias cursadas como parte del área de concentración de Sistemas Embebidos establecen las bases para resolver problemas actuales, pero también marcan un lineamiento hacia las tendencias de solución para aplicaciones cada vez más complejas y con mayores requerimientos de procesamiento, estabilidad y portabilidad.

1. Resumen de los proyectos realizados

Los proyectos que se mencionan en éste trabajo corresponden a las asignaturas de: Diseño de Sistemas Operativos para Ambientes Embebidos, Sistemas Embebidos, y Desarrollo de Software de Comunicaciones en Ambientes Embebidos, las cuales proporcionan un marco teórico necesario para poder aprender e implementar soluciones de software para ambientes embebidos.

1.1. Preemption

1.1.1 Introducción

En esta sección se describe el proyecto realizado en la asignatura de Diseño de Sistemas Operativos para Ambientes Embebidos, en donde se estudiaron e implementaron conceptos fundamentales de un sistema operativo de tiempo real (RTOS). El proyecto se conformó de cuatro prácticas que al final conformaron un sistema operativo básico tipo *preemptive* implementado en el microcontrolador MC9S12XEP100.

1.1.2 Antecedentes

Un sistema operativo es un software encargado de administrar los recursos tanto de hardware como de software de una computadora. En un ambiente embebido donde se tiene una unidad de procesamiento con recursos de hardware limitados y dedicados para una función específica, el sistema operativo debe ser capaz de proporcionar una respuesta en un tiempo determinado y con la máxima eficiencia en el uso de recursos.

Durante el curso se analizaron diferentes conceptos que permiten garantizar la respuesta y estabilidad de un sistema operativo en un ambiente embebido, como son: *Basic task, extended task, multitasking, task priority, priority inversión, deadlock, priority ceiling, shceduler, preepmtive, non-preemptive, colaborative, interrupts category 1, interrupts category 2.*

1.1.3 Solución desarrollada

Se implementó un Sistema Operativo básico *multitasking* bajo un esquema *preemptive*, con la capacidad de interrumpir tareas de menor prioridad y guardar su contexto de ejecución. La implementación del sistema se dividió en cuatro etapas las cuales se describen a continuación:

OS Tick: Corresponde al primer elemento del sistema operativo, el cual es la base para el control de tiempo del sistema. La implementación se desarrolló en un enfoque de capas de abstracción en donde cada capa provee cierta funcionalidad, de ésta forma las capas más bajas básicamente interactúan con el hardware, y las capas más altas tienen la capacidad de proveer el servicio.

Scheduler: Mecanismo encargado de calendarizar la ejecución de las tareas en un sistema operativo. En el proyecto se implementó un *Binary Progressive Scheduler Algorithm,* el cual consiste en un contador binario, en donde cada cierto tiempo incrementa la cuenta, si ésta cuenta coincide con una máscara en bits predefinida se activa la tarea asociada a dicha máscara. Por lo tanto el tiempo de activación para una tarea asociada a una máscara queda definido mediante la relación (1-1):

$$TaskRate = OsTickTime * (TaskMask + 1) \tag{1-1}$$

TaskManagement: Consiste en la capa de abstracción encargada de proveer los siguientes servicios:

ActivateTask: Activar una tarea.

TermianteTask: Terminar la ejecución de una tarea.

GetTaskID: Obtener el identificador de la tarea en ejecución.

GetTaskState: Obtener el estado de una tarea determinada.

En ésta capa también se implementa el *TaskControlBlock*¸ éste módulo se encarga desde leer la configuración de las propiedades de una tarea, hasta llevar el control de las tareas que deben ejecutarse, dependiendo de la prioridad y estado de las tareas.

Preemption: Corresponde a la implementación que permite la interrupción de una tarea en ejecución debido a la activación de una tarea de mayor prioridad, con la capacidad de guardar el contexto de ejecución de la tarea que se suspende, y posteriormente restaurar el contexto cuando la tarea reanuda su ejecución. Para soportar el cambio de contexto se manejó el concepto

de *task stack* en donde a cada tarea se le asigna un espacio limitado en memoria para guardar el estado actual de ejecución: variables locales y registros del cpu.

1.1.4 Análisis de resultados.

El sistema operativo se implementó para el microcontrolador MC9S12XEP100RMV1, utilizando la tarjeta de evaluación DEMO9S12XEP100, lo cual permitió utilizar puertos de salida para monitorear la activación y terminación de las diferentes tareas configuradas.

Mediante el uso de un osciloscopio se pudieron corroborar diferentes propiedades como: tiempo de activación, tiempo de ejecución, suspensión de tarea, tiempo de intercambio entre tareas (*task switch time*).

También fue posible identificar el tiempo de duración del *BackGround task*, tarea que se ejecuta cuando no hay más tareas pendientes por ejecutar.

1.1.5 Conclusiones

Durante la realización del proyecto fue posible identificar las partes esenciales de un sistema operativo y los retos que éste representa, como es el hecho de tener un *OS Tick* lo suficientemente rápido para proporcionar una alta resolución en el manejo de tiempos de las tareas. También el reto que representa la implementación de un *scheduler* y un *task control bock*, capaces de ser lo suficientemente rápidos y eficientes al momento de activar, ejecutar y terminar una tarea determinada. También fue posible identificar las implicaciones de diseño necesarias para soportar un cambio de contexto durante la ejecución de las tareas.

El sistema operativo implementado fue capaz de manejar tareas con periodos de ejecución del orden de microsegundos. Se corroboró la capacidad del sistema de respetar los tiempos de activación de las tareas según el algoritmo del *scheduler* y respetar la ejecución de las tareas en función de la prioridad previamente configurada para cada tarea.

1.2. SAE J2716 Implementation on XGATE

1.2.1 Introducción

En esta sección se describe el proyecto realizado en la asignatura de Sistemas Embebidos, el cuál consistió en implementar un sistema multitarea capaz de generar una trama de comunicación bajo el estándar SAE J2716. En éste proyecto se ponen en práctica varios temas vistos en clase como son: asignación dinámica de memoria, manejo de secciones críticas, sistema de arranque, multiprocesos y manejo de capas de abstracción.

1.2.2 Antecedentes

El protocolo SAE J2716 o también conocido como SENT, es un protocolo de un solo sentido, de voltaje asíncrono el cual requiere de solo tres líneas: una línea de estado (bajo < 0.5V, alto > 4.1V), línea de alimentación (5V) y línea de tierra.

Un mensaje SENT consiste de:

- Un pulso de calibración o sincronización de un periodo de 56 *ticks* de reloj.
- Un pulso de estatus y comunicación de 4 bits conformado de 12 a 27 *ticks*.

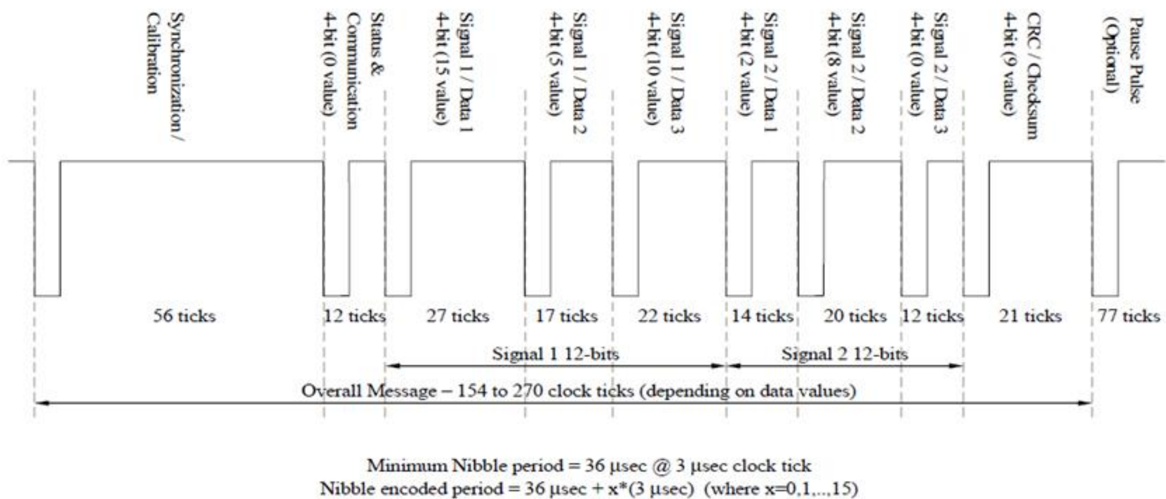


Fig. 1-1 Partes representativas de un mensaje SENT.

- Una secuencia de uno a seis *nibbles* de datos de 4 bits cada uno, representando los valores de la señal a comunicar.
- El número de *nibbles* es ajustado para cada aplicación dependiendo del esquema de codificación, pero puede variar entre aplicaciones.
- Un *checksum* conformado por un pulso de un *nibble* de 4 bits de 12 a 27 *ticks*.
- Un pulso opcional de pausa.
- El periodo de un *tick* de reloj puede durar entre 3 y 90 microsegundos.
- Un pulso bajo está conformado por cuatro *ticks*.

En la Fig. 1-2 se puede observar un ejemplo de un mensaje SENT, donde el *nibble* de status indica el número de *ticks* correspondientes a un cero lógico, posteriormente el valor que representa cada *nibble* de datos corresponde a la diferencia de la duración del *nibble* menos la duración de un cero lógico. Por ejemplo el primer dato de la trama de la Fig. 1-2, dura 27 *ticks*. Si le restamos 12 *ticks* correspondientes a un cero lógico, la diferencia son 15 *ticks*, es decir el *nibble* representa un valor lógico de 15.

1.2.3 Solución desarrollada

El proyecto se realizó utilizando una arquitectura basada en AUTOSAR en cuánto se refiere a la estructuración en capas y convenciones de nombramiento de interfaces.

Se implementó asignación dinámica de memoria, para el manejo de la configuración de los parámetros de operación del sistema, tales como: habilitación de pulso de pausa entre mensajes, pin de salida, longitud de mensaje en *nibbles*, etc.

Las interrupciones generadas por el PIT (*Programable Interrupt Timer)* se configuraron para ser atendidas en el procesador auxiliar (XGATE), el cual fue el encargado de generar los mensajes SENT, mientras tanto el procesador principal (S12) se encargara de ejecutar tareas adicionales en paralelo.

Para grabar el programa en el microcontrolador e iniciar su ejecución, se hizo uso del *boot loader* Omicron, provisto en clase. Para lo cual fue necesario incluir la librería de Omicron en el software del proyecto y configurar correctamente las secciones de memoria.

1.2.4 Análisis de resultados.

En un principio fue posible grabar el proyecto mediante el uso del *boot loader* Omicron, sin embargo como inevitablemente se tuvieron que hacer varios intentos de ejecución debido a la corrección de errores durante la implementación, se borró la memoria persistente y se procedió a grabar el programa utilizando directamente la herramienta de NXP (*Flashing tool).*

Para validar el funcionamiento del sistema, se habilitó un puerto de salida conectado a un led de la tarjeta de evaluación, el cual era encendido periódicamente en función de la tarea de mayor periodo, atendida por el *scheduler* en el procesador principal.

Para evaluar los mensajes SENT generados, se habilitó un puerto de salida en el cual se conectó un osciloscopio lo que permitió constatar la forma y contenido del mensaje.

El cálculo del tick originalmente se planeó generarlo acumulando interrupciones del PIT, sin embargo por cuestiones de practicidad se ajustó el PIT, de tal forma que cada interrupción correspondiera a un tick, quedando este configurado a 500us. Esta solución resulto ser suficiente para manejar tiempos bastante precisos en la generación de los nibbles que conforman el mensaje SENT.

1.2.5 Conclusiones

El desarrollo de una solución siguiendo una arquitectura como Autosar, al principio puede resultar demasiado trabajo por las normas que se tienen que respetar en cuanto a estructuración de archivos, manejo de interfaces y convenciones de nombramiento. Sin embargo éste esfuerzo finalmente obtiene sentido dado que se desarrolla una solución portable, capaz de crecer o integrarse con otras soluciones.

En cuanto al manejo de la memoria dinámica se asignaron diferentes regiones de memoria y se desarrollaron interfaces para acceder dichas regiones, teniendo así un mayor control del manejo de éste recurso.

El manejo de los dos procesadores S12 y XGATE, requirió del manejo de secciones protegidas mediante el uso de semáforos, lo cual garantizó la integridad de los datos compartidos entre ambos procesadores.

1.3. CAN Multicore

1.3.1 Introducción

En la asignatura de Sistemas de Comunicaciones para Ambientes Embebidos se analizaron los principales protocolos de comunicación ampliamente usados en la industria Automotriz, como son LIN, CAN, y CAN-FD. Durante la asignatura se estudió a detalle el protocolo CAN, abarcando las características eléctricas, implementación del protocolo en silicio e implementación en software de drivers e interfaces para realizar una comunicación por CAN.

El proyecto presentado en éste trabajo corresponde a la implementación en software de un driver de CAN en el microprocesador MC9S12XEP100RMV1.

1.3.2 Antecedentes

El protocolo CAN es uno de los principales protocolos utilizados en el área automotriz. El nombre CAN corresponde a *Controlled Area Network,* actualmente existen dos estándares: CAN 2.0A, conocido también como CAN 1.2, el cual soporta identificadores de 11 bits. Y el CAN 2.0B, el cual soporta identificadores de 11 y 29 bits, aumentando así la cantidad de mensajes que se pueden manejar en la red.

Invariablemente de la versión, las principales características del protocolo CAN son las siguientes:

- Acceso al bus por prioridad de mensaje.
- Sistema de arbitración para resolver conflictos de acceso al bus.
- Identificador único en los mensajes. El protocolo de CAN no maneja direcciones de nodos.
- Chequeo extensivo de errores: El protocolo de can maneja hasta cinco diferentes mecanismos de chequeo de errores, en donde cada nodo conectado participa.
- Consistencia de datos: Para que un mensaje sea considerado como válido debe ser aceptado por todos los nodos.

- Capacidad del protocolo determinada por capas superiores de comunicación: El protocolo CAN especifica únicamente la comunicación en capas inferiores. Las capacidades del sistema de comunicación quedan determinadas por la implementación en software de las capas superiores.

1.3.3 Solución desarrollada

El proyecto presentado en ésta materia consistió en desarrollar un driver de CAN el cual opera en el coprocesador (XGATE) del microcontrolador MC9S12XEP100RMV1. El driver de CAN implementado en éste proyecto abarca de manera las capas de red y transporte del modelo OSI. Donde la capa física queda determinada por el medio de comunicación y la capa de transporte es abstraída por el módulo de CAN implementado en silicio como parte del microcontrolador.

La solución de software consistió en implementar una interface de CAN (CANIF) encargada de comunicarse con el controlador de CAN (módulo del microcontrolador) y manejar el envío y recepción de datos mediante acceso a los registros de CAN y atención de las interrupciones de CAN.

1.3.4 Análisis de resultados.

El driver implementado fue capaz de soportar una velocidad de transmisión de 500 kbps. Fue de crucial importancia implementar una correcta atención a las diferentes interrupciones del controlador de CAN, tales que fueran capaces de atender los datos y señales recibidas del controlador en un tiempo mínimo de respuesta para evitar influenciar negativamente en el desempeño del driver.

La atención de las interrupciones se realizó en el coprocesador del microcontrolador (XGATE) lo cual requirió especial atención para tender adecuadamente:

- El manejo de datos dentro de secciones protegidas.
- Transporte de datos entre procesador y co-procesador.

Para validar la operación del driver se conformó un bus de CAN compuesto por dos nodos:

1. Tarjeta de evaluación DEMO9S12XEP100 la cual contiene un CAN *transceiver* conectado al microcontrolador en donde se implementó el driver.
2. Un CANCase VN1630, que funciona como un nodo en la red de CAN y es posible conectarlo a una PC permitiendo monitorear y enviar mensajes en el bus de CAN.

1.3.5 Conclusiones

El proyecto implementado en la asignatura de Sistemas de Comunicación para Ambientes Embebidos permitió visualizar elementos clave que deben considerarse durante el desarrollo de un driver, como son: manejo eficiente de la memoria, control de accesos concurrentes, manejo de interrupciones, abstracción de capas, etc.

El protocolo de comunicación CAN (*Controller Area Network*) es ampliamente usado en el sector automotriz por el bajo costo y alta confiabilidad. En una red de CAN todos los nodos conectados participan en la verificación de la integridad de los datos y en caso de que un nodo este causando errores recurrentes es capaz de "auto-silenciarse" para evitar afectar toda la red.

Una parte muy importante de la especificación de CAN es que ésta abarca solo las capas inferiores del modelo OSI. Ésta especificación se implementada en el silicio como un módulo del microcontrolador. Lo cual permite una alta eficiencia en el procesamiento y verificación de la integridad de los mensajes sin sobrecargar el procesador principal. Permitiendo a la aplicación de software enfocarse en las capas superiores de comunicación y poder implementar protocolos de mayor complejidad como un ISOTP.

2. Conclusiones

Los proyectos presentados en éste trabajo permitieron consolidar los conocimientos y desarrollar las habilidades necesarias para poder generar un buen diseño e implementación de los sistemas embebidos.

La implementación del Sistema Operativo *Multitasking*, permitió poner en práctica los fundamentos teóricos que aseguran que un sistema pueda responder en un tiempo determinado y mantener una funcionalidad estable y eficiente. En el ámbito laboral me toca trabajar con diferentes sistemas operativos de tiempo real para aplicaciones automotrices, con los conocimientos adquiridos en la materia de sistemas operativos para ambientes embebidos, tengo mejores elementos para valorar los elementos clave de un sistema operativo, sus prestaciones, sus limitaciones y así poder identificar los sistemas operativos más apropiados para determinadas aplicaciones.

La implementación del protocolo SENT, permitió abordar varios temas comunes en la mayoría de los sistemas embebidos como son: Estructuración de software por capas de abstracción, interfaces entre módulos que permiten la adaptabilidad a diferentes arquitecturas, el manejo de: memoria, puertos, interrupciones y registros. El uso de un *bootloader* o sistema de arranque. Manejo de librerías, manejo de una arquitectura *multicore* y las implicaciones que esto representa para asegurar la integridad de los datos. Consideraciones especiales del lenguaje C para la programación para un ambiente embebido, el rol que juega el compilador y las sentencias especiales del lenguaje que permiten mejorar la eficiencia de un sistema durante su ejecución en un ambiente embebido. Tener claridad en éstos conceptos permite entender y aplicar mejor los estándares de programación como lo es MISRA para el sector automotriz.

La implementación del driver de CAN fue una consolidación de varios temas. En primer lugar se pusieron en práctica los diferentes conceptos para el desarrollo de software para un ambiente embebido. En segundo lugar se conoció a detalle el estándar de CAN, un protocolo ampliamente usado en la industria automotriz. Y en tercer lugar y no menos importante fue posible estructurar un sistema que requirió el conocimiento de otras disciplinas como el diseño de sistemas digitales y conceptos de electrónica analógica.

Apéndices

A. REPORTE: PREEMTION

Preemption Report

Nr. : 1.01

Responsible:
Victor Huidobro
Fernando Rodriguez Rivera

1. Introduction

Currently the system is capable, to activate tasks periodically based on a Binary Progressive Algorithm. The system is also capable to manage the tasks Activation, Execution and Termination using the Task Control Block. However there is a big disadvantage that only one task can be executed at a time, it means that if a higher priority task is "Ready" it should wait until the currently running tasks is terminated, before it can be executed.

In order to provide a system capable to switch between tasks depending on its priorities, it is necessary to implement the "Preemption" concept. This introduces the following considerations:

If a task with higher priority is activated, the currently running tasks should be preempted, in order to execute the higher priority task, and after the higher priority task finishes the previous running task execution is resumed.

Preempt a task means stop its execution and store its execution context, so the execution could be resumed later with the execution context restored.

2. Main Requirements

Dispatcher

› Dispatcher shall only be called at the end of the following system services:
– ActivateTask
– TerminateTask
– End of CAT 2 ISR's.
› If higher priority task than the current running task is ready for execution, the dispatcher shall switch tasks context and run the new task.
› Previous task shall be queued in its corresponding priority buffer.
› Task states shall be changed accordingly.

Context Switch

› SaveContext and RestoreContext interfaces shall be provided to support Context Switch mechanism.
– They both shall have as parameter a pointer to the target stack.
› SaveContext shall be called immediately starting CAT 2 ISR's with a defined address to store current stack as parameter, this stack will be referred as MainStack.
› When the dispatcher detects that a Context Switch is required, it will move the context stored in MainStack to the required task stack.

Task Stack

› Task Stack shall be allocated with Memory Allocation interface
› Each task shall contain its own stack including the Background Task
› Stack size will depend on the project memory model

Task Stack Frame

› The following registers are to be saved:
– CCR
– A
– B
– IX
– IY
– PC
– SP
– If banked memory used -> PPAGE.

Task Descriptor

› Project shall support besides periodic tasks, event driven tasks.
– New configuration element shall be added to define if the task is periodic or event driven task.

3. Design considerations

3.1 Saving – Restoring Task Context

In order to save/restore the current task execution context, we rely on the Hardware Interruption mechanism. When an Interruption occurs the hardware stores the CPU Registers in the Stack. Then when the "RTI" instruction is executed (Returning from interruption) the hardware pulls from the stack the CPU registers information previously stored.

This way when an Interruption occurs we just take CPU registers information form the stack and we store this information in a specific memory region associated with the task (task stack). Then the interruption is handled, after that, the Dispatcher is called which itself restores the task context, by reading the "task stack" and stores this information in the main stack. Finally the "RTI" instruction is executed and the hardware restores the CPU registers from what we just write in the main stack.

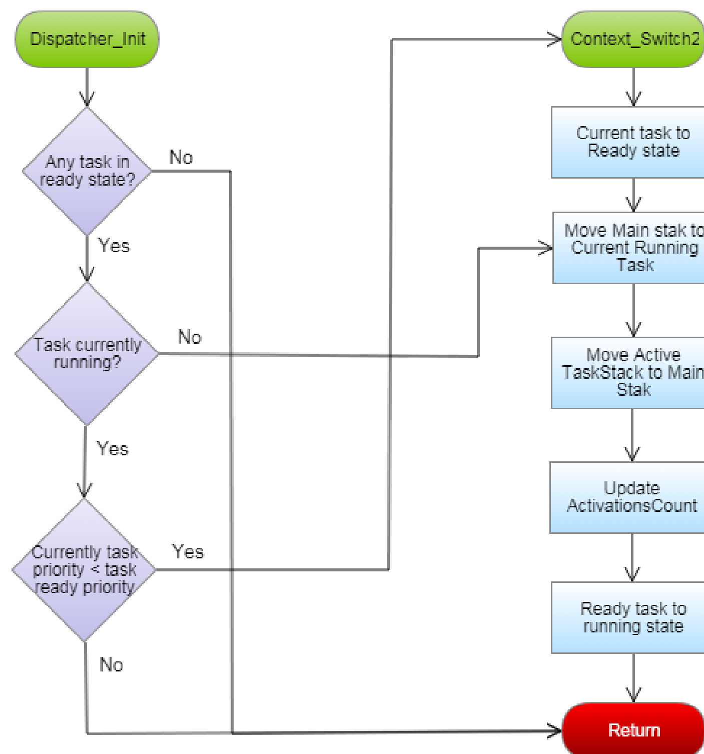


Figure 1: Save/Restore context flow diagram

3.2 Saving – Restoring Task Context among tasks

It is also possible that one task may activate another higher priority task, then the current task should be preempted. For this case we use a "Software Interruption" so the hardware stores the CPU Registers into the main stack and then we can continue using the same mechanism to retrieve and restore the task context.

4. Implementation

4.1 Save Context Instructions

```
#define SaveContext()                                                   \
{                                                                       \
      extern volatile void* currentTCB_Stack;                           \
      extern void Os_vWrapperDispatcherFunction(void);                  \
      __asm( "ldaa 0x30" );  /* 0x30 = PPAGE */      \
      __asm( "psha" );                                                  \
      __asm( "des"  );                                                  \
      __asm( "ldx currentTCB_Stack" );                                  \
      __asm( "movw 0,SP,0,X" );                                         \
      __asm( "movw 2,SP,2,X" );                                         \
      __asm( "movw 4,SP,4,X" );                                         \
      __asm( "movw 6,SP,6,X" );                                         \
      __asm( "movw 8,SP,8,X" );                                         \
      __asm( "movw 10,SP,10,X" );                                       \
}
```

4.2 Restore Context Instructions

```
#define RestoreContext()                                                \
{                                                                       \
      extern volatile void* currentTCB_Stack;                           \
                                                                        \
      __asm( "ldx currentTCB_Stack" );                                  \
      __asm( "movw 0,X,0,SP" );                                         \
      __asm( "movw 2,X,2,SP" );                                         \
      __asm( "movw 4,X,4,SP" );                                         \
      __asm( "movw 6,X,6,SP" );                                         \
      __asm( "movw 8,X,8,SP" );                                         \
      __asm( "movw 10,X,10,SP" );                                       \
      __asm( "ins"  );                                                  \
      __asm( "pula" );                                                  \
            __asm( "staa 0x30" ); /* 0x30 = PPAGE */                    \
            __asm( "rti" );                                             \
}
```

4.3 OS Interrupt service

The Os interrupt cat 2, service has the following structure:

```
void OS_Entry_Mcu_vPIT_Channel0_Isr(void)
{
  SaveContext();
  INTERRUPT_DISABLE();
  Mcu_vHandle_PIT_Interrupt(PIT0);
  DispatcherService();
}
```

4.4 Example of a Task Main Function

```
TASK(vTask_4)
{
  vOS_Task_Work(OS_TASK_4_WORK_LOAD, PORTB_PB4_MASK);
  (void)TerminateTask();
  DispatcherService();
}
```

4.5 Example of a Task Main Function which activates another Task

```
TASK(vTask_3)
{
  vOS_Task_Work(OS_TASK_3_WORK_LOAD, PORTB_PB3_MASK);

  (void)ActivateTask(OS_Task_4);
  __asm( "swi" );

  vOS_Task_Work(OS_TASK_3_WORK_LOAD, PORTB_PB3_MASK);

  (void)TerminateTask();
  DispatcherService();
}
```

5. Operation

Under previous design considerations and configuring task 3 to perform certain work load, then it activates Task 4 (higher priority), so task 4 gets executed. When task 4 finishes then Task 3 resumes and should perform a second work load. Obtaining the following graph.

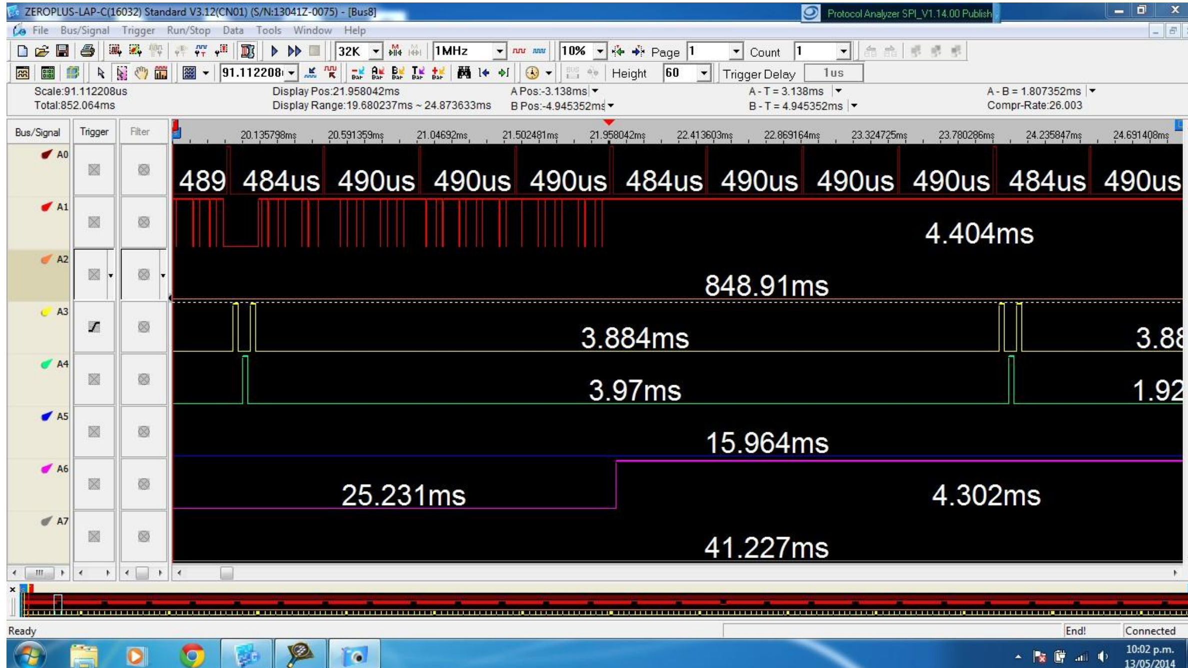


Figure 2: Preemption operation

Where:

Graph A1 represents Background Task. Priority 0.

Graph A3 represents Task 3. Priority 4.

Graph A4 represents Task 4. Priority 5.

Graph A5 represents Task 5 (Not Activated during this period). Priority 4.

Graph A6 represents Task 6. Priority 3.

B. REPORTE: SAE J2716 IMPLEMENTATION ON XGATE

Integrantes del equipo:
CHAVEZ RODRIGUEZ HECTOR IVAN
GARCIA PALOMERA JESUS
RODRIGUEZ RIVERA FERNANDO

Turn Non-preemtive system into Cooperative.

A continuación se describen los cambios generales realizados para implementar el sistema Cooperativo.

Archivo: tasksCfg.h (nuevo)

En éste archivo se definen los diferentes tipos de datos a utilizar para determinar las tareas y propiedades de las tareas que se manejan en el sistema. Así como también se definen enumeraciones para especificar los diferentes Task ids, status y priorirades.

```
#ifndef _TASKSCFG_H_
#define _TASKSCFG_H_

/** Tasks definitions */
#include "tasks.h"
#include "typedefs.h"

/*
** ====================================================================
**     Basic definitios for current
** ====================================================================
*/
#define    CONFIGURED_TASKS    8  /* Number of configured TASKS */

typedef enum
{
  TASK_EvDv_1 = 0, // Event driven 1
  TASK_EvDv_2 = 1, // Event driven 1
  TASK_1ms    = 2, // 1ms
  TASK_2ms_A  = 3, // 2ms A
  TASK_2ms_B  = 4, // 2ms B
  TASK_10ms   = 5, // 10ms
  TASK_50ms   = 6, // 50ms
  TASK_100ms  = 7, // 100ms
  TASK_LIMIT  = 8
}teTask_Id;

typedef enum
{
  SUSPEND = 0,
  READY   = 1,
  RUNNING = 2
}teTask_Status;

typedef enum
{
  PRIO_1 = 1,
  PRIO_2 = 2,
  PRIO_3 = 3,
  PRIO_4 = 4,
  PRIO_5 = 5,
  PRIO_6 = 6,
}teTask_Priority;

typedef struct
{
  teTask_Id tId;
  teTask_Status tStatus;
  teTask_Priority tPriority;
  tCallbackFunction tCallback;
} tsTaskType;
```

```
extern tsTaskType tTaskDescriptor[];


#endif /*_TASKSCFG_H_*/
```

Archivo: tasksCfg.c (nuevo)

En éste archivo se define el arreglo principal (task descriptor) el cual contienen la configuración de tareas y sus propiedades que deberá soportar el sistema.

```
#include "tasksCfg.h"

tsTaskType tTaskDescriptor[]=
{
  /* Tasks Configuration - Tasks ordered by Priority :) */
        /* Task 1 - Event Driven task 1*/
    {
      TASK_EvDv_1,   /* Id       */
      SUSPEND,       /* Status   */
      PRIO_6,        /* Priority */
      vfnTask_EvDr_1 /* Callback */
    },
        /* Task 2 - Event Driven task 2 */
    {
      TASK_EvDv_2,   /* Id       */
      SUSPEND,       /* Status   */
      PRIO_6,        /* Priority */
      vfnTask_EvDr_2 /* Callback */
    },
        /* Task 3 - 1ms */
    {
      TASK_1ms,      /* Id       */
      SUSPEND,       /* Status   */
      PRIO_5,        /* Priority */
      vfnTask_1ms    /* Callback */
    },
        /* Task 4 - 2ms_a */
    {
      TASK_2ms_A,    /* Id       */
      SUSPEND,       /* Status   */
      PRIO_4,        /* Priority */
      vfnTask_2ms_A  /* Callback */
    },
        /* Task 5 - 2ms_b */
    {
      TASK_2ms_B,    /* Id       */
      SUSPEND,       /* Status   */
      PRIO_4,        /* Priority */
      vfnTask_2ms_B  /* Callback */
    },
        /* Task 6 - 10ms */
    {
      TASK_10ms,     /* Id       */
      SUSPEND,       /* Status   */
      PRIO_3,        /* Priority */
      vfnTask_10ms   /* Callback */
    },
        /* Task 7 - 50ms */
    {
      TASK_50ms,     /* Id       */
      SUSPEND,       /* Status   */
      PRIO_2,        /* Priority */
```

```
      vfnTask_50ms   /* Callback */
    },
        /* Task 8 - 100ms */
    {
      TASK_100ms,    /* Id       */
      SUSPEND,       /* Status   */
      PRIO_1,        /* Priority */
      vfnTask_100ms  /* Callback */
    },

};
```

Archivo: tasks.h

Se remueven los MACROS que originalmente definían los tasks callbacks.
Se agregan declaraciones externas de funciones las cuales corresponden a los tasks callbacks.

```
/*****************************************************************************************************
* Declaration of module wide FUNCTIONS
*****************************************************************************************************/
extern void vfnTask_EvDr_1(void);
extern void vfnTask_EvDr_2(void);
extern void vfnTask_1ms(void);
extern void vfnTask_2ms_A(void);
extern void vfnTask_2ms_B(void);
extern void vfnTask_10ms(void);
extern void vfnTask_50ms(void);
extern void vfnTask_100ms(void);

#endif /*__TASKS_H */
```

Archivo: tasks.c (nuevo)

Se definen los callbacks correspondientes a cada task.

```
void vfnTask_EvDr_1(void)
{
  //vfnPinToggle(); //Maybe we can change this function to toggle i.e. PA03 only
  vfnTask_Activate(TASK_EvDv_1);
}

void vfnTask_EvDr_2(void)
{
  vfnTask_Deactivate(TASK_EvDv_2);
}

void vfnTask_1ms(void)
{
  vfnTask_Deactivate(TASK_1ms);
}

void vfnTask_2ms_A(void)
{
  vfnTask_Deactivate(TASK_2ms_A);
}

void vfnTask_2ms_B(void)
{
  vfnTask_Deactivate(TASK_2ms_B);
```

```
}

void vfnTask_10ms(void)
{
  vfnTask_Deactivate(TASK_10ms);
}

void vfnTask_50ms(void)
{
  vfnTask_Deactivate(TASK_50ms);
}

void vfnTask_100ms(void)
{
  vfnPinToggle();
  vfnCOPWatchdog_Reset();
  vfnTask_Deactivate(TASK_100ms);
}
```

Archivo: scheduler.c / scheduler.h

Se agregaron las siguientes interfaces:

void vfnTask_Activate(teTask_Id tTaskId)	Interfaz utilizada para realizar la activación de una tarea.
void vfnTask_ Deactivate (teTask_Id tTaskId)	Interfaz utilizada para desactivar una tarea.

Se modificaron las siguientes interfaces:

void vfnTask_Scheduler(void)
Se cambió el contenido de ésta función, para hacer un barrido sobre el arreglo de las tareas para identificar basado en el status y prioridad la tarea que debe ser ejecutada. Para sencillez del programa las tareas en el arreglo están inicializadas por prioridad.

Original Code (Non-preemtive)

```
void vfnTask_Scheduler( void  )
{

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    /*  1ms execution thread - used to derive two execution threads:
*/
    /*  a) 1ms thread (high priority tasks)
*/
    /*  b) 100ms thread (lowest priority tasks)
*/
    /*  As any other thread on this scheduler, all tasks must be executed in
<=500us*/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
    if( ( gu8Scheduler_Thread_ID == TASKS_1_MS )
         || ( gu8Scheduler_Thread_ID == TASKS_100_MS ) )
    {
        /* Make a copy of scheduled task ID */
        gu8Scheduler_Thread_ID_Backup = gu8Scheduler_Thread_ID;

        EXECUTE_1MS_TASKS ()
        if( gu8Scheduler_Thread_ID == TASKS_100_MS )
        {
            EXECUTE_100MS_TASKS ()
```

New code (Cooperative)

```
void vfnTask_Scheduler( void  )
{
  UINT8 uTaskId = 0;

  /* Has scheduler been started */
  if(gu8Scheduler_Status == TASK_SCHEDULER_RUNNING)
  {
    /* No task is being currently running */
    if(gtScheduler_Running_ThreadId == TASK_LIMIT)
    {
      /* Look for Hihest Priority Ready Task and execute it!!!*/
      /* Very basic Scheduler, tasks are already sort by priority */
      for(uTaskId = 0; uTaskId < TASK_LIMIT; uTaskId++)
      {
        if(tTaskDescriptor[uTaskId].tStatus == READY)
        {
          ptTaskType = &tTaskDescriptor[uTaskId];
          gtScheduler_Running_ThreadId = ptTaskType->tId;
          ptTaskType->tStatus = RUNNING;
          ptTaskType->tCallback();
          break;
        }
      }
    }
```

```
        }
        /* Verify that thread execution took less than 500 us */
        if( gu8Scheduler_Thread_ID_Backup == gu8Scheduler_Thread_ID )
        {
             /* In case execution of all thread took less than 500us */
            gu8Scheduler_Thread_ID = NO_TASK_PENDING;
        }
        else
        {
            gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_1MS;
        }
    }
    else
    {

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        /*  2ms execution thread - used to derive two execution threads:
*/
        /*  a) 2ms group A thread (high priority tasks)
*/
        /*  b) 50ms thread (second lowest priority tasks)
*/
        /*  As any other thread on this scheduler, all tasks must be executed in
<=500us*/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
        if( ( gu8Scheduler_Thread_ID == TASKS_2_MS_A )
             || ( gu8Scheduler_Thread_ID == TASKS_50_MS ) )
        {
            /* Make a copy of scheduled task ID */
            gu8Scheduler_Thread_ID_Backup = gu8Scheduler_Thread_ID;

            EXECUTE_2MS_A_TASKS ()
            if( gu8Scheduler_Thread_ID == TASKS_50_MS )
            {
                EXECUTE_50MS_TASKS ()
            }
            /* Verify that thread execution took less than 500 us */
            if( gu8Scheduler_Thread_ID_Backup == gu8Scheduler_Thread_ID )
            {
                 /* In case execution of all thread took less than 500us */
                gu8Scheduler_Thread_ID = NO_TASK_PENDING;
            }
            else
            {
                gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_2MS_A;
            }
        }
        else
        {

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            /*  2ms execution thread - used to derive two execution threads:
*/
            /*  a) 2ms group B thread (high priority tasks)
*/
            /*  b) 10ms thread (medium priority tasks)
*/
            /*  As any other thread on this scheduler, all tasks must be executed
in <=500us*/

/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
            if( ( gu8Scheduler_Thread_ID == TASKS_2_MS_B )
                 || ( gu8Scheduler_Thread_ID == TASKS_10_MS ) )
            {
                /* Make a copy of scheduled task ID */
                gu8Scheduler_Thread_ID_Backup = gu8Scheduler_Thread_ID;

                EXECUTE_2MS_B_TASKS ()
                if( gu8Scheduler_Thread_ID == TASKS_10_MS )
                {
                    EXECUTE_10MS_TASKS ()
                }
                 /* Verify that thread execution took less than 500 us */
                if( gu8Scheduler_Thread_ID_Backup == gu8Scheduler_Thread_ID )
                {
                    /* In case execution of all thread took less than 500us */
                    gu8Scheduler_Thread_ID = NO_TASK_PENDING;
                }
                else
                {
                    gu8Scheduler_Status = TASK_SCHEDULER_OVERLOAD_2MS_B;
                }
            }
        }
    }
}
```

```
  }
}
```

vfnScheduler_Callback (void)

La modificación en éste archivo consistió en reemplazar la asignación del Tread_Id por una llamada la interfaz vfnTask_Activate. También se removieron algunos "esle" blocks debido a que con el algoritmo original cada vez que se ejecutaba una tarea secundaria, la primaria no se ejecutaba.

Original Code (Non-preemtive)		New code (Cooperative)
`void vfnScheduler_Callback( void )`	=	`void vfnScheduler_Callback( void )`
`{`	< >	`{`

`/*-- Update scheduler control variable --*/`
`gu8Scheduler_Counter++;`
 	
`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
`/* 1ms execution thread - used to derive two execution threads: */`
`/* a) 1ms thread (highest priority tasks) */`
`/* b) 100ms thread (lowest priority tasks) */`
`/* As any other thread on this scheduling scheme, */`
`/* all tasks must be executed in <= 500us */`
`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
`if( ( gu8Scheduler_Counter & 0x01u ) == 0x01u )`
`{`
`u8_100ms_Counter++;`
`/*-- Allow 100 ms periodic tasks to be executed --*/`
`if( u8_100ms_Counter >= 100u )`
`{`
	=	`/*-- Update scheduler control variable --*/`
		`gu8Scheduler_Counter++;`
		
		`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
		`/* 1ms execution thread - used to derive two execution threads: */`
		`/* a) 1ms thread (highest priority tasks) */`
		`/* b) 100ms thread (lowest priority tasks) */`
		`/* As any other thread on this scheduling scheme, */`
		`/* all tasks must be executed in <= 500us */`
		`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
		`if( ( gu8Scheduler_Counter & 0x01u ) == 0x01u )`
		`{`
		`u8_100ms_Counter++;`
		`/*-- Allow 100 ms periodic tasks to be executed --*/`
		`if( u8_100ms_Counter >= 100u )`
		`{`

`gu8Scheduler_Thread_ID = TASKS_100_MS;`	< >	`//Activate task 100ms`
		
		`vfnTask_Activate(TASK_100ms);`

`u8_100ms_Counter = 0u;` `}`	=	`u8_100ms_Counter = 0u;` `}`
	- +	
`/*-- Allow 1 ms periodic tasks to be executed --*/`	=	`/*-- Allow 1 ms periodic tasks to be executed --*/`

`else`
 	
`{`
`gu8Scheduler_Thread_ID = TASKS_1_MS;`
`}`
	< >	`// Activate task 1ms`
		
		`vfnTask_Activate(TASK_1ms);`

`}`
`else`
`{`
`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
`/* 2ms execution thread - used to derive two execution threads: */`
`/* a) 2ms group A thread (high priority tasks) */`
`/* b) 50ms thread (second lowest priority tasks) */`
`/* As any other thread on this scheduling scheme, */`
`/* all tasks must be executed in <= 500us */`
`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
`if( ( gu8Scheduler_Counter & 0x02u ) == 0x02u )`
`{`
`u8_50ms_Counter++;`
`/*-- Allow 50 ms periodic tasks to be executed --*/`
`if( u8_50ms_Counter >= 25u )`
`{`
	=	`}`
		`else`
		`{`
		`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
		`/* 2ms execution thread - used to derive two execution threads: */`
		`/* a) 2ms group A thread (high priority tasks) */`
		`/* b) 50ms thread (second lowest priority tasks) */`
		`/* As any other thread on this scheduling scheme, */`
		`/* all tasks must be executed in <= 500us */`
		`/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/`
		`if( ( gu8Scheduler_Counter & 0x02u ) == 0x02u )`
		`{`
		`u8_50ms_Counter++;`
		`/*-- Allow 50 ms periodic tasks to be executed --*/`
		`if( u8_50ms_Counter >= 25u )`
		`{`

`gu8Scheduler_Thread_ID = TASKS_50_MS;`	< >	`// Activate task 50 ms`
		
		`vfnTask_Activate(TASK_50ms);`

`u8_50ms_Counter = 0u;` `}`	=	`u8_50ms_Counter = 0u;` `}`
	- +	

/*-- Allow 2 ms group A periodic tasks to be executed --*/	=	/*-- Allow 2 ms group A periodic tasks to be executed --*/

else
 	
{
gu8Scheduler_Thread_ID = TASKS_2_MS_A;
}
	<
	>
		// Activate task 2 ms A
		
		vfnTask_Activate(TASK_2ms_A);

}
else
{
/*~~~~~~
~~*/
/* 2ms execution thread - used to derive two
execution threads: */
/* a) 2ms group B thread (high priority
tasks) */
/* b) 10ms thread (medium priority
tasks) */
/* As any other thread on this scheduling
scheme, */
/* all tasks must be executed in <=
500us */
/*~~~~~~
~~*/
if((gu8Scheduler_Counter & 0x03u) == 0x00u)
{
u8_10ms_Counter++;
/*-- Allow 10 ms periodic tasks to be executed
--*/
if(u8_10ms_Counter >= 5u)
{
	=	}
		else
		{
		/*~~~~~~
		~~*/
		/* 2ms execution thread - used to derive two
		execution threads: */
		/* a) 2ms group B thread (high priority
		tasks) */
		/* b) 10ms thread (medium priority
		tasks) */
		/* As any other thread on this scheduling
		scheme, */
		/* all tasks must be executed in <=
		500us */
		/*~~~~~~
		~~*/
		if((gu8Scheduler_Counter & 0x03u) == 0x00u)
		{
		u8_10ms_Counter++;
		/*-- Allow 10 ms periodic tasks to be executed
		--*/
		if(u8_10ms_Counter >= 5u)
		{

gu8Scheduler_Thread_ID = TASKS_10_MS;	<
	>
		// Activate task 10 ms
		
		vfnTask_Activate(TASK_10ms);

u8_10ms_Counter = 0u; }	=	u8_10ms_Counter = 0u; }
	- +	
/*-- Allow 2 ms group B periodic tasks to be executed --*/	=	/*-- Allow 2 ms group B periodic tasks to be executed --*/

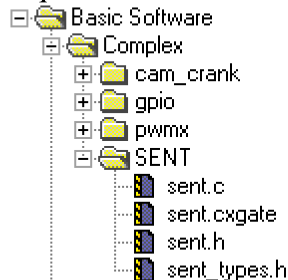
else
 	
{
gu8Scheduler_Thread_ID = TASKS_2_MS_B;
}
	<
	>
		// Activate task 2 ms B
		
		vfnTask_Activate(TASK_2ms_B);

} } } } /** **************************************/	=	} } } } /** **************************************/

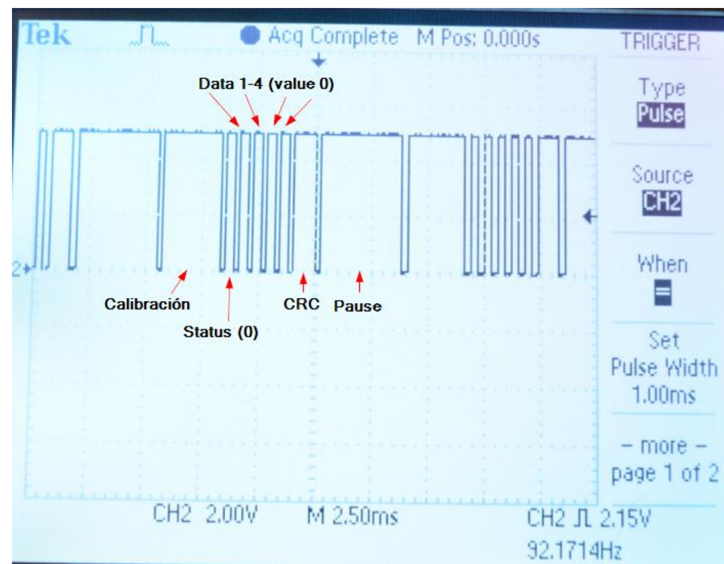
Implementación protocolo SENT.

1. Proyecto originalmente se podía grabar con el booloader, pero como inevitablemente tuvimos que flashear la tarjeta, se perdió esta funcionalidad y generamos el proyecto para ser flasheado directamente con el CodeWarrior

2. La lógica del protocolo sent se encuentra en los siguientes archivos, tratando de seguir la arquitectura AUTOSAR.



El mensaje se genera mediante el XGATE, produciendo una salida como se muestra a continuación:



3. El cálculo del tick originalmente se planeó acumulando interrupciones del PIT, pero por cuestiones de tiempo se ajustó PIT de tal forma que cada interrupción corresponde a un Tick de Sent, estando configurado éste a 50us.

4. Se implementó el driver del ADC pero la interrupción correspondiente no se está llamando periódicamente, no tuvimos tiempo de solucionar éste problema.

C. REPORTE: CAN - MULTICORE

Fernando Rodríguez Rivera
José Antonio Maza Moreno
Saúl Alfonso Núñez Corona

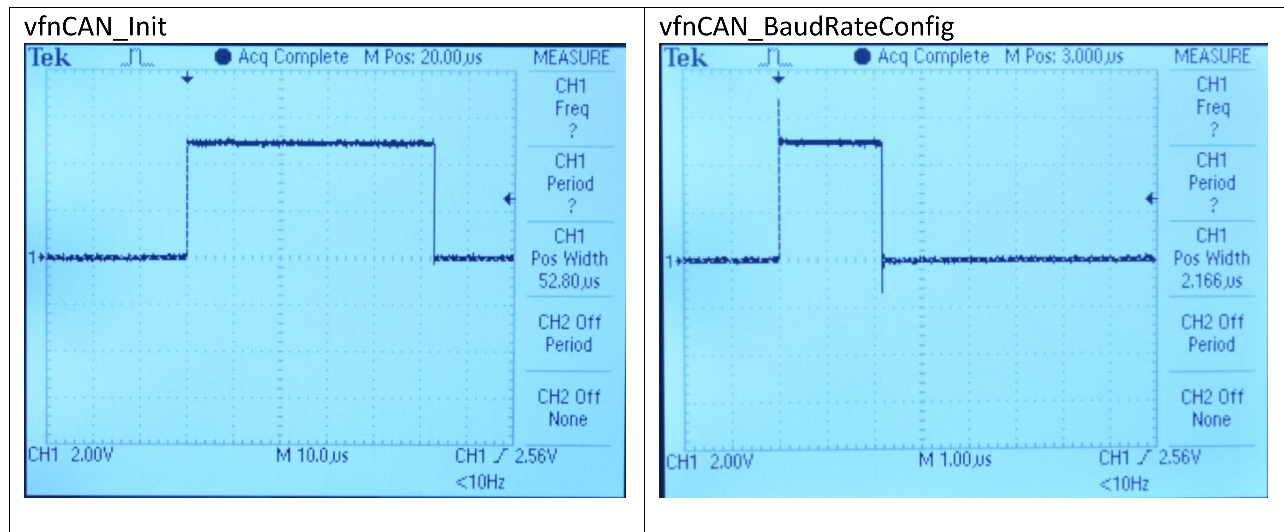
1. Análisis de Funciones del driver de CAN original.

2. La medición de tiempo de ejecución de las funciones en el driver original de CAN se puede ver en la tabla 1.

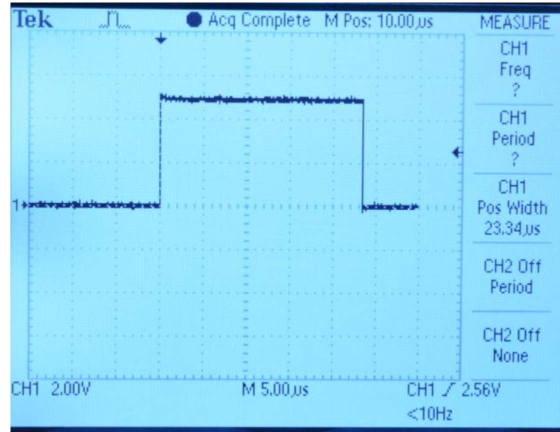
	Function	Private / Public	Activation	Return	Synch / Asyinch	Init / Runtime	Time (us)	Need to Change
	vfnCAN_Init	Public	User	void	Synch	Init	52.80	No
	vfnCAN_BaudRateConfig	Private	User	void	Synch	Init	2.17	No
	vfnCAN_AcceptanceFiltersInit	Private	User	void	Synch	Init	23.34	No
	u8CAN_enqueueFrameforTx	Public	User	uint8	Synch	Runtime	52.00	No
	u8CAN_TxQueueDepth *	Public	User	uint8	Synch	Runtime	1.63	No
	vfnCAN_Tx_handler	Public	Hw Trigger	void	Synch	Runtime	28.70	Yes
	u8CAN_RxFifoDepth	Public	Hw Trigger	uint8	Synch	Runtime	1.58	No
	vfnCAN_Rx_handler	Private	Hw Trigger	void	Synch	Runtime	24.24	Yes
mscan.c	vfnCAN_GetRxMessages	Public	Hw Trigger	void	Synch	Runtime	10.38	No
	vfnCAN_A_RxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	25.08	Yes
	vfnCAN_A_TxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	29.50	Yes
	vfnCAN_B_RxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
	vfnCAN_B_TxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
	vfnCAN_C_RxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
	vfnCAN_C_TxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
	vfnCAN_D_RxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
	vfnCAN_D_TxFrame_Isr	Public	Hw Trigger	void	Synch	Runtime	-	Yes
hal_can.c	vfnCAN_Periodic_Tx_Queuing	Public	User	void	Synch	Runtime	Max: 128.70 Min: 2.46	No
	vfnCAN_Periodic_Rx	Public	User	void	Synch	Runtime	Max: 12.96 Min: 1.71	No

Tabla 1. Medición de tiempo

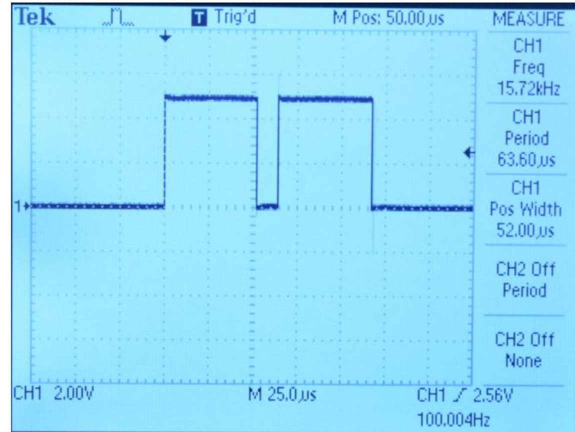
3. Medición de las señales en el Osciloscopio:



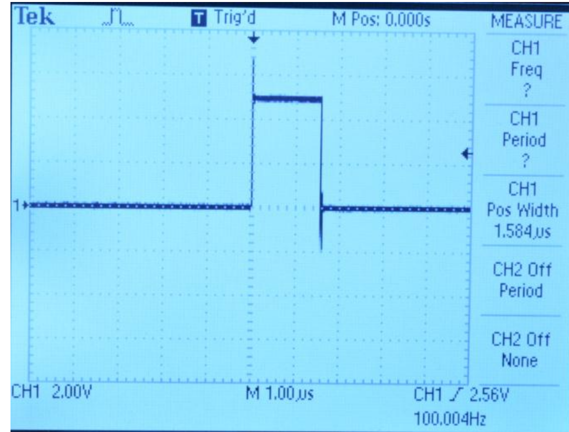
vfnCAN_AcceptanceFiltersInit



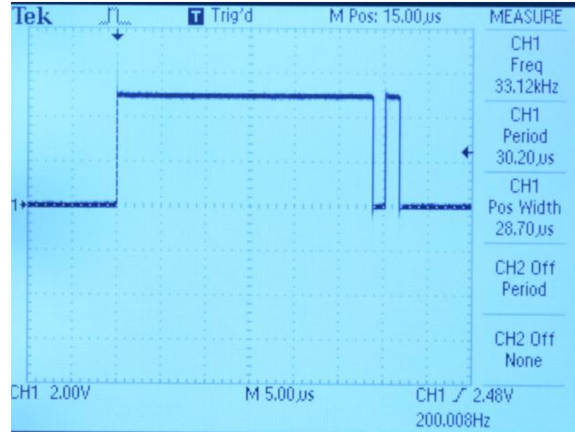
u8CAN_enqueueFrameforTx



u8CAN_TxQueueDepth



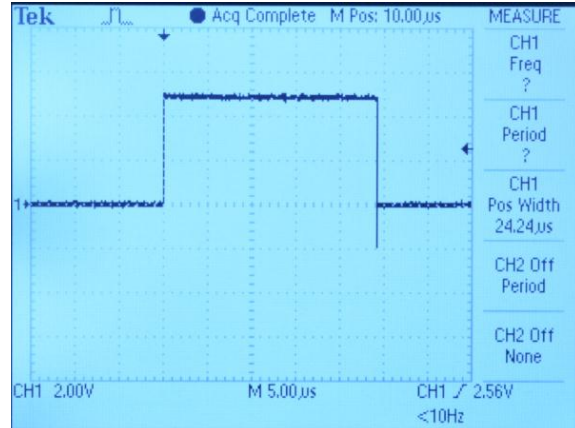
vfnCAN_Tx_handler



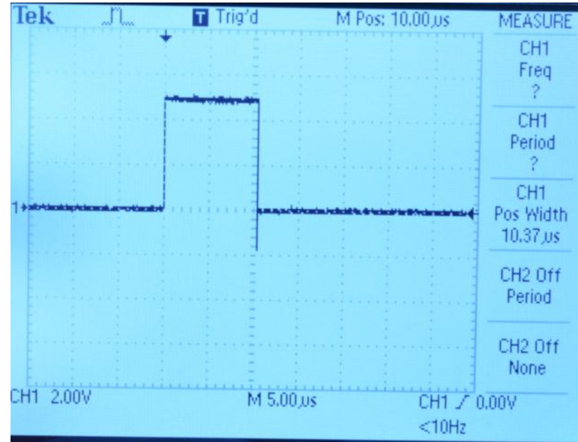
u8CAN_RxFifoDepth

-- picture not taken ☹--

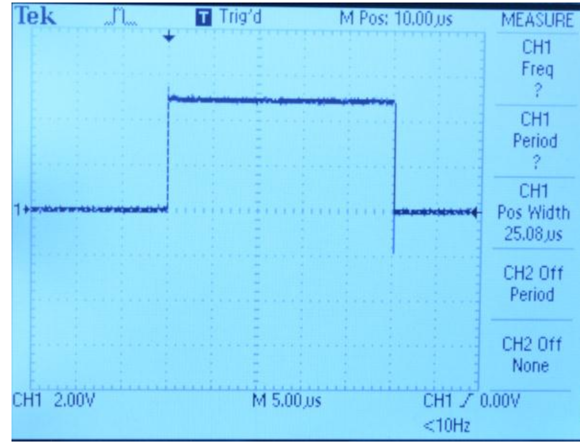
vfnCAN_Rx_handler



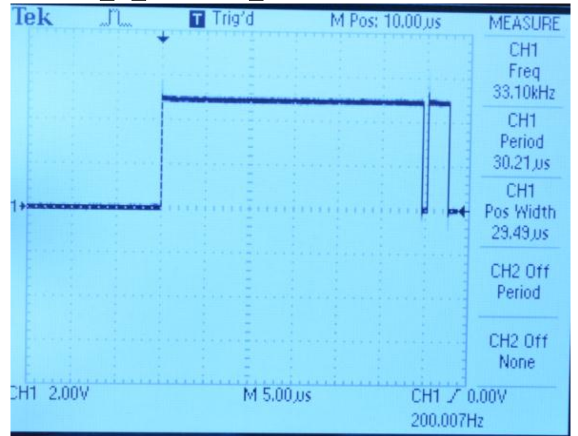
vfnCAN_GetRxMessages



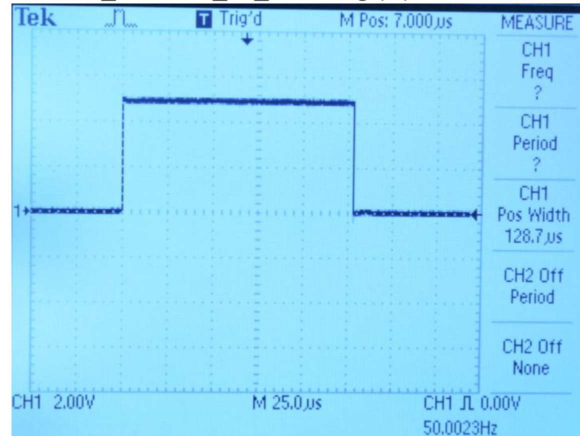
vfnCAN_A_RxFrame_Isr



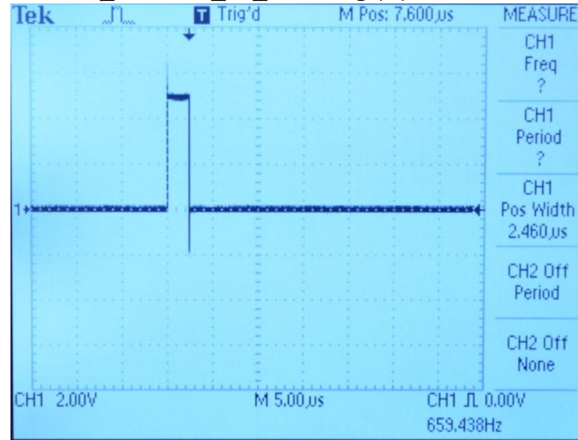
vfnCAN_A_TxFrame_Isr



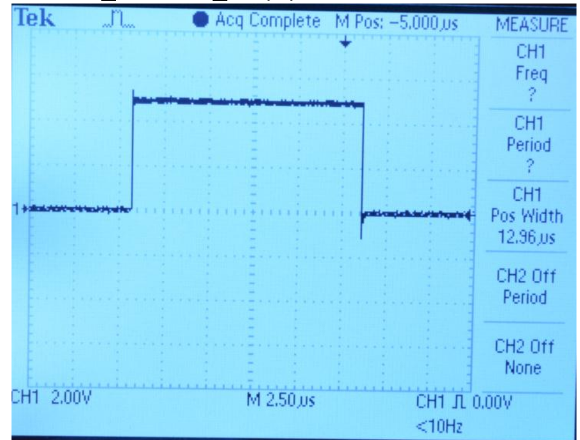
vfnCAN_Periodic_Tx_Queuing (A)

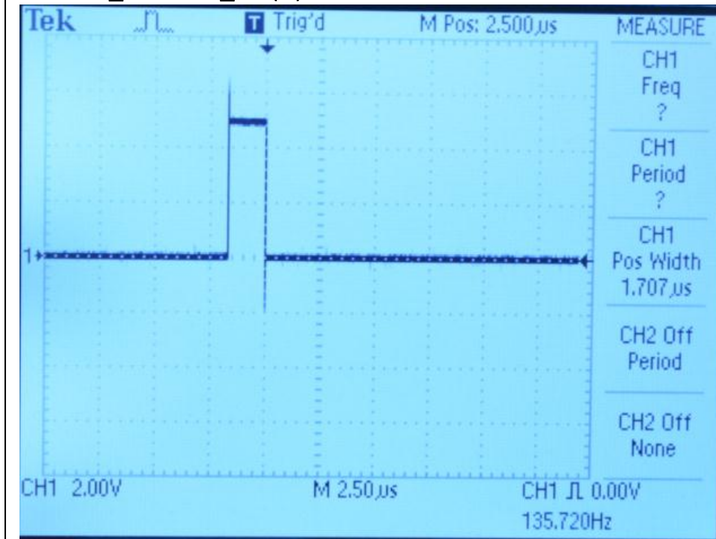


vfnCAN_Periodic_Tx_Queuing (B)



vfnCAN_Periodic_Rx (A)



vfnCAN_Periodic_Rx (B)	
	

4. Diseño Multicore

Para la implementación multicore, ser continuó con el análisis CAN Multicore.xlsx, en donde se determinó que las funciones que se deberían ejecutar en el **XGATE** son **vfnCAN_Tx_handler** y **vfnCAN_Rx_handler**, también se determinó las secciones que deberían ser protegidas para garantizar la coherencia de los datos.

Los resultados del análisis realizado se pueden ver en la tabla 2.

Function	Data	R/W access	Critical portions of code	Mechanism	Comentarios	Registers	R/W access	Critical portions of code	Mechanism
u8CAN_enqueueFrameforTx	mscan[device].tx_queue_current_depth	read	Yes	Semaphore	Se escribe en un método y core diferentes.	TIER	write	Yes	Semaphore
	mscan[device].tx_write->ID	write	No		Se leen en un solo método y no tenemos un sistema preemtive y la función solo se ejecuta en un core.				
	mscan[device].tx_write->Data[u8Index]	write	No						
	mscan[device].tx_write->Length	write	No						
	mscan[device].tx_write->id_type	write	No						
	mscan[device].tx_queue_current_depth	write	Yes		Se leen en un método y core diferentes.				
	mscan[device].tx_write	write	No		Llamado desde una sola tarea.				
	mscan[device].tx_end	read	No		Su valor no se modifica durante la ejecución.				
	mscan[device].tx_buffer	read	No		El apuntador no se modifica durante la ejecución.				
u8CAN_TxQueueDepth	mscan[device].tx_queue_current_depth	read	Yes	Semaphore	En teoría se debe proteger porque el valor se escribe en otro core. Aunque actualmente esta función nunca se llama.			Yes	Semaphore
vfnCAN_Tx_handler	mscan[device].tx_queue_current_depth	read	Yes	Semaphore	Se escribe un método y core diferente.	TFLG	read	Yes	Semaphore
	mscan[device].tx_read->id_type	read	No		Se leen en un solo método y no tenemos un sistema preemtive y la función solo se ejecuta en un core.	TBSEL	write		
	mscan[device].tx_read->ID	read	No			TBSEL	read		
	mscan[device].tx_read->Length	read	No			TXIDR0	write		
	mscan[device].tx_read->Data[u8Index]	read	No			TXDSR0	write		
	mscan[device].tx_queue_current_depth	write	Yes	Semaphore	Se en un método diferente y en un core diferente.	TXDLR	read		
	mscan[device].tx_read	write	No		Llamado desde una sola tarea.	TXTBPR	write		
	mscan[device].tx_end	read	No		Su valor no se modifica durante la ejecución.	TFLG	write		
	mscan[device].tx_buffer	read	No		El apuntador no se modifica durante la ejecución.	TIER	write		
u8CAN_RxFifoDepth	mscan[device].rx_fifo_current_depth	read	Yes	Semaphore	Se lee y escribe en diferentes métodos y diferentes cores.			yes	
vfnCAN_Rx_handler	mscan[device].rx_write->filter_id	write	No		Se leen en un solo método y no tenemos un sistema preemtive y la función solo se ejecuta en un core.	RFLG	read	yes	
	mscan[device].rx_write->ID	wrie	No			IDAC	read		
	mscan[device].rx_write->Length	write	No			RXIDR1	read		
	mscan[device].rx_write->Length	read	No			RXIDR0	read		
	mscan[device].rx_write->Data[u8Index]	write	No			RXIDR2	read		
	mscan[device].rx_write->NewMessage	write	No			RXIDR3	read		
	mscan[device].rx_fifo_current_depth	write	Yes		Se lee y escribe en diferentes métodos y diferentes cores.	RXDLR	read		
	mscan[device].rx_write	write	No		Llamado desde una sola tarea.	RXDSR0	read		
	mscan[device].rx_end	read	No		Su valor no se modifica durante la ejecución.	RFLG	read		
	mscan[device].rx_buffer	read	No		El apuntador no se modifica durante la ejecución.				
vfnCAN_GetRxMessages	mscan[device].rx_fifo_current_depth	read	yes		Se lee y escribe en diferentes métodos y diferentes cores.				
	mscan[device].rx_read	read	No		Llamado desde una sola tarea.				
	mscan[device].rx_read->NewMessage	write	No						
	mscan[device].rx_fifo_current_depth	write	yes		Se lee y escribe en diferentes métodos y diferentes cores.				
	mscan[device].rx_read	write	Yes		El valor que se le asigna proviene de un una variable modificada en otro core.				
	mscan[device].rx_end	read	No		Su valor no se modifica durante la ejecución.				
	mscan[device].rx_buffer	read	No		El apuntador no se modifica durante la ejecución.				
vfnCAN_A_RxFrame_Isr			No		Se utiliza solo un vector de interrupción por interrupción				
vfnCAN_A_TxFrame_Isr			No						

Tabla 2. CAN Multicore

5. Implementación

Básicamente se decidió atender el RX_Handler y el TX_Handler en el XGATE, por lo tanto se generó el archivo mscan.cxgate donde se implementaron dichas funciones.

Los archivos modificados del código Base se pueden ver en la siguiente imagen.

LIN_project | MIPS_Single_Cycle_PipeLine | Omicron_CAN | Omicron_CAN_B

File Status
Working Copy
Branches
fer_rr
master
maza
Tags
Remotes
origin
fer_rr
master
maza
Stashes
On master: TX on

All Branches | Show Remote Branches | Date Order | Jump to:

Graph	Description	Date	Author	Commit
	Uncommitted changes	16 may 2016 21:44	*	*
	origin/master master Structures without enum types and with padding	14 may 2016 18:56	Fernando RR <fer_	e39cd40
	Adding padding in objects construction	14 may 2016 17:50	Fernando RR <fer_	8da1571
	Se agrega volatile y se mueve definicion a XGATE	14 may 2016 17:48	Antonio Maza <m	90206e4
	origin/fer_rr fer_rr TX Handler on XGATE, wiht MemAlloc, build - but don't send.	14 may 2016 17:00	Fernando RR <fer_	9c3ca8a
	Agregar padding Configuracion apra XGATE o S12	14 may 2016 16:53	Antonio Maza <m	7e1268a
	Merge branch 'master' into fer_rr	14 may 2016 16:30	Fernando RR <fer_	b934109
	TX on XGATE without MemAlloc - builds but not sending	14 may 2016 15:51	Fernando RR <fer_	b8ed3fe
	Add TX Handler in XGATE	14 may 2016 15:10	Fernando RR <fer_	caabd88
	mscan global variable as Shared Data	14 may 2016 14:23	Fernando RR <fer_	ea4aef1

Sorted by file name | Search

Displaying all changes between
1ec4cd845180ba6e137943a387d21f62c8b61def and
e39cd408f0109a759dccdf097b1568c55ddb2a37

?	Filename	Path
+	.gitignore	Sources/omicro...can_comm_handler
	C_Layout.hwl	Sources/omicro...can_comm_handler
	Omicron_ASW.mcp	
	CWSettingsWindows.stg	Omicron_ASW_Data
−	TargetDataWindows.tdt	Omicron_ASW_Data\SofTec_HCS12
+	SofTec_HCS12.hwl	
	SofTec_HCS12.ini	
	cnf_mscan.c	Sources\omicron_sw_layers\bios\can
	cnf_mscan.h	Sources\omicron_sw_layers\bios\can
	mscan.c	Sources\omicron_sw_layers\bios\can
+	mscan.cxgate	Sources\omicron_sw_layers\bios\can
	mscan.h	Sources\omicron_sw_layers\bios\can
	int_vectors.c	Sources\omicron_sw_layers\bios\int
	int_vectors.h	Sources\omicron_sw_layers\bios\int
	io.c	Sources\omicron_sw_layers\bios\io
	io.h	Sources\omicron_sw_layers\bios\io
	xgate_vectors.cxgate	Sources/omicron_sw_layers/bio...xgate
	hal_can.c	Sources/omicron...can_comm_handler

```
13 + /** IO Interface */
14 + #include "io.h"
15 +
16 + /* MSCAN acceptance filter configuration */
17 + #pragma DATA_SEG SHARED_DATA
18 + volatile tMSCAN_Message aCANMsgBuffer[CAN_DEVICE_COUNT][CAN_RX_BUFFER_DEPTH];
19 + volatile tMSCAN_Message aCANFifoDepth[CAN_DEVICE_COUNT][CAN_TX_QUEUE_DEPTH];
20 + volatile tMSCAN_DeviceStatus aCANDeviceStatus[CAN_DEVICE_COUNT];
21 + #pragma DATA_SEG DEFAULT
22 +
23 + /*****************************************************************************/
24 + /**
25 + * \brief    CAN transmission handler.
26 + * \         Should a CAN message has already been transmitted, queue the next one
27 + * \author   Abraham Tezmol
28 + * \param    enum tMSCAN_Device
29 + * \return   void
30 + */
31 + #pragma CODE_SEG XGATE_CODE
32 + void vfnCAN_Tx_handler_XGATE(enum tMSCAN_Device device)
33 + {
34 +     UINT8 u8Index;
35 +     /* Transmission buffer */
36 +     UINT8 u8TxBuffer = 0;
37 +     static UINT8 vDummyPointer = 0;
38 +     volatile tMSCAN_DeviceStatus *mscan = NULL;
39 +
40 +     //FUNC_ENTER_HOOK;
41 +
42 +     mscan = aCANDeviceStatus;
43 +
44 +     /* Verify if there exist pending messages in the Tx queue */
45 +     //TODO. CS tx_queue_current_depth
46 +     if (mscan[device].tx_queue_current_depth)
47 +     {
48 +        /* Check for Tx buffer availability on all three hw buffers */
49 +         while(!MSCAN_READ_TFLG(device))
50 +         {
51 +             /* Wait until one of them goes empty */;
52 +         }
53 +
54 +         /* Select lowest empty buffer */
55 +         MSCAN_WRITE_TBSEL(device, MSCAN_READ_TFLG(device));
56 +         /* Backup selected buffer */
```

File Status | Log / History | Search
11 | fer_rr Atlassian

6. Resutlados:

En cuanto al desarrollo de la presente práctica, algo que nos desconcertó fue el problema para leer los datos desde el XGATE. Este problema no lo pudimos superar y no nos fue posible completar la práctica.

a) Se evitó el uso del MemAlloc para generar los mensajes de TX generados durante la inicialización. En cambio se generaron variables en el scope global pero dentro de los pragmas “SHARED_DATA”. Lo cual parecía no generar ningún problema, sin embargo al intentar acceder estos arreglos desde el XGATE generaba una excepción.

b) Otra alternativa fue Mover la página de memoria que utiliza el MemAlloc a la sección “SHARE_DATA”, y entonces usar MemAlloc para generar los mensajes de TX durante la inicialización. No obstante se tenía el mismo problema durante la ejecución al momento de tratar acceder estos datos desde el XGATE.

c) Se identificó que las Enumeraciones declaradas dentro de las estructuras eran manejadas por el compilador de forma diferente para el S12 como para el XGATE, en donde a un dato tipo enumeración le asignaba diferente número de bytes. Por lo tanto se reemplazaron los tipos “enumeración” por tipos UINT8, y además se agregaron algunos paddings, para tener structuras alineadas a 16bits (necesario por el XGATE). Pero aun así seguíamos sin poder acceder los datos desde el XGATE.

d) Como última alternativa generé un arreglo unidimencional dentro de pragmas “SHARED_DATA” en el S12 y lo declare como externo igual dentro de pragmas “SHARED_DATA” en el XGATE, y efectivamente pude acceder los datos. Pero haciendo lo mismo con los arreglos de los mensajes TX, simplemente no funcionó. Estábamos considerando se debiera a un problema con los arreglos bidimensionales. Pero desafortunadamente ya no nos alcanzó el tiempo para continuar.

En resumen esto fue lo que se realizó respecto a la presente práctica:

- Hicimos el análisis del código base.
- Tomamos lecturas del tiempo de ejecución del programa original.
- Identificamos las funciones que deberían ejecutarse en el XGTE. (En archivo de Excel e implementado en código)
- Identificamos las secciones que deberían protegerse (comentado en código)
- Pero al momento de ejecutar el programa no pudimos superar el problema de acceso de datos desde el XGATE.

7. Conclusiones

La implementación de un Driver en Multicore, permite una operación asíncrona donde las tareas se pueden distribuir entre el CPU y el XGATE. Una ventaja importante de tener un driver asíncrono es que los módulos de capas superiores pueden llamar la interface del driver y no tener que esperar hasta que se complete la transmisión o recepción. Liberando así el CPU y pasando parte del trabajo al XGATE.

Sin embargo una desventaja podría ser la complicación para mantener la coherencia de los datos. Mantener dicha coherencia implica el uso de mecanismos como mutex o semáforos que pueden agregar overhead al sistema o tiempos de espera prolongados.

En general poder decidir si un driver multicore es mejor que un single-core creo que depende de cómo las capas superiores intentan utilizar el driver. Si la naturaleza de operación del sistema se basa en eventos o es completamente secuencial.