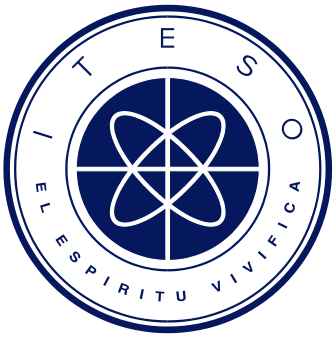
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE OCCIDENTE

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación el 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

ESPECIALIDAD EN SISTEMAS EMBEBIDOS



REPORTE DE FORMACIÓN COMPLEMENTARIA

Trabajo recepcional que para obtener el diploma de

ESPECIALISTA EN SISTEMAS EMBEBIDOS

Presenta: Juan Carlos Razo Esparza

Asesores:
Dr. Raúl Campos Rodríguez
Dr. Luis Enrique González Jiménez

Tlaquepaque, Jalisco. Marzo de 2017.

RECONOCIMIENTOS

Quiero agradecer al Consejo Nacional de Ciencia y Tecnología (CONCACYT) por su apoyo de la Beca No. 636287.

Gracias a la Especialidad en Sistemas Embebidos por brindarme la oportunidad de estudiar un posgrado en el ITESO esta área que me ha permitido desarrolla las habilidades para mi desarrollo profesional y personal.

Quisiera expresar mi sincero agradecimiento a mis asesores Dr. Raúl Campos Rodríguez y Dr. Luis Enrique González Jiménez por el apoyo que me ha brindado para lograr este trabajo.

Agradezco a los Profesores Abraham Tezmol por el material para el desarrollo de las prácticas reportadas en este trabajo.

Especialmente quiero agradecer al ITESO e Intel Tecnología de México por el equipamiento y las instalaciones para lograr cada actividad desarrollada en este programa. Un agradecimiento especial a todos mis profesores por su paciencia, motivación y conocimiento que recibí de ellos.

TABLA DE CONTENIDOS

LISTA DE ACRONIMOS Y ABREVIATURAS

HCS12x	Microcontroller family from Freescale Semiconductor
S12	Enhanced 32-bit processor included in the HCS12x MCU
XGATE	Freescale peripheral coprocessor that allows autonomous high-speed data processing and transfers between the MCU's peripherals and the internal RAM and I/O ports
MCU	Microcontroller Unit
PIT	Programmable Interrupt Timer
SENT	Single Edge Nibble Transmission
RTOS	Real-Time Operating System
AUTOSAR	Automotive Open System Architecture
LIN	Local Interconnect Network
MCU	Microcontroller Unit
FSM	Finite State Machine
CRC	Cyclic Redundancy Check
PWM	Pulse Width Modulation
RISC	Reduced Instruction Set Computing
BIOS	Basic Input/Output System
IO	Input/Output
CPU	Central Processing Unit
CAN	Controller Area Network bus
HAL	Hardware Abstraction Layer
ISR	Interrupt Service Routine
GPIO	General Purpose Input/Output
LIN	Local Interconnect Network
ECC	Error Correcting Code

1. INTRODUCCIÓN

Resumen: *Este capítulo presenta un resumen de este documento y los trabajos reportados.*

1.1. Introducción

El objetivo de este documento es describir el trabajo de formación complementaria realizado en el área de Sistemas Embebidos. Las materias de concentración y proyectos que se eligieron para este trabajo son los siguientes:

- **Sistemas Embebidos:**
    - Proyecto: PWM Generator on HCS12x Multicore Architecture
- **Desarrollo de software de comunicaciones para ambientes embebidos:**
    - Proyecto: CAN Multicore
- **Diseño de sistemas operativos para ambientes embebidos**
    - Proyecto: Binary Progression Scheduler for HCS12

La selección de estos proyectos representa una serie de conceptos y metodologías que reflejan las necesidades actuales de la industria. En primer lugar, se analiza una implementación multi-core de un generador de señal PWM en donde se cubre el concepto clave de la distribución de tareas primarias y secundarias entre diferentes elementos de procesamiento de un mismo MCU. El segundo proyecto hace nuevamente uso del concepto multi-core, y a la vez, nos pone en primer contacto con uno de los protocolos de comunicación más importantes, especialmente en la industria automotriz; el protocolo CAN. Finalmente, el último trabajo presentado en este documento cubre las bases de la implementación de “schedulers”. Concepto y metodología fundamental en el diseño e implementación de sistemas operativos de tiempo real.

2. RESUMEN DE LOS PROYECTOS

Resumen: *En este capítulo se presenta un resumen de los trabajos y proyectos finales seleccionados.*

2.1. PWM Generator on HCS12x Multicore Architecture

2.1.1. Introducción

El uso de microprocesadores multi-core en la industria de los sistemas embebidos se ha convertido en la norma durante los últimos años. El presente proyecto es una implementación de un generador PWM haciendo uso del microprocesador Freescale HCS12x. Dicho microprocesador empaqueta dos elementos de procesamiento en topología asimétrica. El procesador S12 es el core principal de ejecución, mientras que el procesador XGATE es un core RISC de menor desempeño y responsable de la ejecución de tareas secundarias.

El procesador S12 presenta una mayor capacidad operativa comparada con el procesador XGATE. Por esta razón, el procesador S12 realiza la tarea principal de generación de la señal PWM. El procesador XGATE es únicamente responsable de la tarea secundaria del manejo de puertos I/O.

2.1.2. Antecedentes

Por décadas, el uso de sistemas single-core ha sido, en cierta manera, suficiente para satisfacer las necesidades de la industria de los sistemas embebidos. Sin embargo, en recientes años, la complejidad y cantidad de tareas delegadas a elementos digitales de procesamiento de datos ha incrementado de manera considerable. En la actualidad, los requerimientos de tiempo de respuesta, procesamiento de grandes cantidades de datos y la alta demanda de comunicaciones; hacen imperativa la adopción de sistemas capaces de hacer una inteligente partición y distribución de tareas.

2.1.3. Desarrollo

Requerimientos generales

Generación de una señal PWM y manejo de puertos I/O mediante el uso de dos procesadores. Dos tareas, una primaria y una secundaria, deberán ser distribuidas de la siguiente manera:

- Procesador S12: Generación de señal PWM mediante el control de "duty cycle" de una señal oscilante.
- Procesador XGATE: Control y manejo de puertos I/O.

Requerimientos de la generación de señal PWM

- El procesador principal S12 deberá implementar la generación de una señal PWM mediante una tarea de 50ms.
- Deberá, además. Generar continuamente cambios en el "duty cycle" de la señal PWM de acuerdo a la siguiente tabla:

Time (seconds)	Duty Cycle (%)
0	0
1	100
2	0
5	Repeat from Time = 0s

- Deberá mantener una carga de trabajo en el procesador, no mayor al 5%.

Requerimientos del manejo de puertos I/O

- El procesador secundario XGATE deberá configurar un puerto de salida.
- Deberá hacer drive de una señal de frecuencia fija a 100 Hz.
- Usar el PIT (programable interrupt timer) como recurso de hardware para la generación de frecuencia base.

Implementación

Descripción funcional del módulo generador de señal PWM

El módulo generador de PWM genera continuamente valores numéricos correspondientes al "duty cycle" de la señal. Estos valores son comunicados al procesador secundario XGATE mediante una región de memoria compartida entre ambos elementos.

El generador es configurado mediante un módulo externo de configuración. Parámetros locales de configuración y estado son inicializados a partir de este módulo.

Descripción funcional del módulo de manejo de puertos I/O

El módulo de manejo de puertos I/O hace drive de la señal PWM a un puerto de salida configurable por el usuario. Se hace uso de un PIT para la generación de una señal de frecuencia fija a 100Hz y cumple con el correspondiente parámetro de "duty cycle" obtenido mediante la lectura de datos de la región de memoria compartida entre el procesador primario y secundario.

El módulo de manejo de puertos I/O es configurable mediante un módulo externo de configuración. Los parámetros configurables son la región de memoria compartida con el procesador primario, un valor de identificación de canal y un apuntador de función a la función de control de puertos I/O.

2.1.4. Analisis de Resultados

Los resultados obtenidos de esta implementación fueron satisfactorios. Después de resolver varios problemas técnicos, principalmente relacionados a la coherencia de memoria compartida entre los dos procesadores, el sistema dio los resultados esperados. La carga de trabajo del procesador principal se mantuvo alrededor del 2.5%, muy por debajo del 5% de acuerdo a los requerimientos del proyecto.

2.1.5. Conclusiones

El presente proyecto permitió familiarizarnos y lograr un entendimiento de los beneficios que brinda la metodología de sistemas multi-core. De igual manera, nos permitió entender y enfrentarnos a problemas que surgen al desarrollar dichos sistemas.

Una de las áreas críticas y propensas a fallas fue la coherencia de memoria compartida entre procesadores. Para evitar problemas en esta área, fue necesario el uso de instrucciones primitivas de ambos procesadores para asegurar que la ejecución de “regiones críticas” de código fuera accesible únicamente por un procesador a la vez y nunca permitir la situación en la que ambos procesadores ejecutan instrucciones de dicha región de manera concurrente.

En general, este proyecto nos permitió conocer las diferentes opciones disponibles para la comunicación entre procesadores. Esta implementación hace uso tanto de la comunicación mediante memoria compartida, como el uso de interrupciones entre procesadores.

Finalmente, el uso de módulos de configuración nos permitió realizar una implementación totalmente estructurada y modular debido en gran parte al uso de esta

metodología de configuración. Fue relativamente fácil incluir configuraciones para ambos módulos, lo cual permitió una fácil estructuración del proyecto.

2.2. CAN Multicore

2.2.1. Introducción

El estándar de comunicaciones CAN (Controller Area Network) es sin duda uno de los estándares más utilizados en la industria automotriz. Es un protocolo basado en mensajes y diseñado para permitir la comunicación autónoma entre microcontroladores y dispositivos sin necesidades de un elemento central de control de comunicaciones.

En este proyecto se realiza la adaptación de un driver CAN single-core a una versión multi-core mucho más eficiente y estructurada, siguiendo la metodología de distribución de tareas primarias y secundarias entre diferentes elementos de procesamiento de un mismo MCU.

2.2.2. Antecedentes

Una de las tareas más importantes de un sistema embebido es la comunicación de datos entre diferentes dispositivos. Muchas veces, esta comunicación es crítica para el correcto funcionamiento y cumplimiento de requerimientos de tiempo de un sistema. Un ejemplo, en sistemas automotrices, es la comunicación entre sensores de impacto y los actuadores que activan las bolsas de aire. La comunicación entre estos dispositivos es altamente crítica y requiere del menor retraso posible. La metodología de distribución de tareas primarias y secundarias entre diferentes elementos de procesamiento, permite alcanzar comunicaciones de baja latencia en situaciones como esta.

2.2.3. Desarrollo

Requerimientos del driver CAN multi-core

- Adaptar el proyecto base driver CAN single-core y lograr una implementación multi-core que cumpla con los siguientes requerimientos:
    - Las tareas generadas a partir de interrupciones, deberán ser atendidas por el procesador secundario XGATE.
    - Las tareas generadas a partir de peticiones provenientes de la capa superiores de software HAL, deberán ser ejecutadas por el procesador primario S12.

- Realizar mediciones de tiempo de ejecución de las tareas ejecutadas por el sistema en configuración single-core. Hacer uso de un puerto I/O de salida para indicar el inicio y final de la ejecución de tareas. Usar un osciloscopio para las mediciones de tiempo y compilar las mediciones para posterior análisis.

- Realizar las correspondientes mediciones de tiempo de ejecución de tareas en la implementación multi-core.

- Hace análisis y comparación de las mediciones obtenidas en la ejecución de tareas en sistema single-core y multi-core.

- Hacer uso de semáforos para proveer protección y mantener coherencia de memoria entre los dos procesadores del sistema.

Implementación del driver CAN multi-core

La implementación multi-core del driver CAN se centra principalmente en reservar el uso del procesador primario S12 para únicamente dar servicio a peticiones del HAL (Hardware Abstraction Layer), mientras que el resto de las tareas asociadas

al servicio de interrupciones del controlador CAN, son atendidas por el procesador secundario XGATE.

Servicio de peticiones provenientes del HAL atendidas por el procesador HC12

Las peticiones generadas por el HAL son atendidas y almacenadas por el procesador S12 en una cola implementada en software. El mismo procesador primario es igualmente responsable de extraer y transmitir al HAL las transacciones de lectura recibidas por el controlador CAN y almacenadas en buffer de lectura.

Servicio de interrupciones del controlador CAN atendidas por el procesador XGATE

Las interrupciones de transmisión y recepción de mensajes del controlador CAN son atendidas por el procesador secundario XGATE. Cada una de estas interrupciones es atendida de manera diferente mediante la implementación de dos funciones de manejo de interrupción independientes.

2.2.4. Analisis de Resultados

La adaptación del módulo de comunicación CAN multi-core dio como resultado una implementación considerablemente más eficiente, comparada con la implementación single-core, en términos de tiempos de ejecución. En efecto, los resultados obtenidos concuerdan con las expectativas al migrar un sistema single-core a implementación multi-core.

2.2.5. Conclusiones

Como era esperado, la implementación multi-core del módulo CAN mostró una reducción en el tiempo de servicio de las peticiones del HAL atendidas por el

procesador primario. De igual manera, el servicio de interrupciones atendidas por el procesador XGATE mostro una reducción en tiempo de ejecución. La razón a esta reducción en tiempos de ejecución se debe a la distribución de tareas entre los dos procesadores. Tanto el procesador primario como el secundario atienden tareas específicas y, por lo tanto, se elimina la sobre carga de tareas a un solo procesador como ocurren en la implementación single-core.

La implementación de dos semáforos, uno para las tareas relacionadas a la transmisión y otro para las tareas relacionadas a la recepción mensajes, fue necesaria para evitar problemas de coherencia de memoria compartida entre los dos procesadores.

2.3. Binary Progression Scheduler for HCS12

2.3.1. Introducción

El presente proyecto documenta la implementación de un “scheduler” tipo progresión binaria. Este tipo de scheduler es uno de los más básicos y simples de implementar, y es idóneo en sistemas con MCU pequeños ya requiere únicamente de un timer de propósito general para la generación de un “tick” usado para generar una variable de conteo. La variable conteo genera un conteo binario donde cada uno de los bits de la variable provee la base de tiempo para diferentes tareas.

2.3.2. Antecedentes

Un sistema operativo de tiempo real requiere de una base de tiempo denominada como “OS tick”. En general, un sistema operativo deberá proveer la capacidad de activar y desactivar una variedad de tareas con diferentes periodos de tiempo. Esta generación de activación y desactivación de las diferentes tareas proviene del OS tick.

2.3.3. Desarrollo

Requerimientos

- El diseño del scheduler de progresión binaria deberá hace uso de un OS tick y una variable de conteo. Diferentes ticks de tarea deberán obtenerse a partir de la variable conteo y la combinación de mascara/offset para la selección de un bit de conteo especifico.
- El módulo scheduler deberá proveer una interfaz para soportar las siguientes funciones:
    - Inicialización del scheduler.

- Terminación o “de-initialization” del scheduler.
- Inicio del scheduler.
- Función callback del tick generado por el timer de propósito general.
- Implementación de una tarea background.
- Funciones callback para las diferentes tareas periódicas.

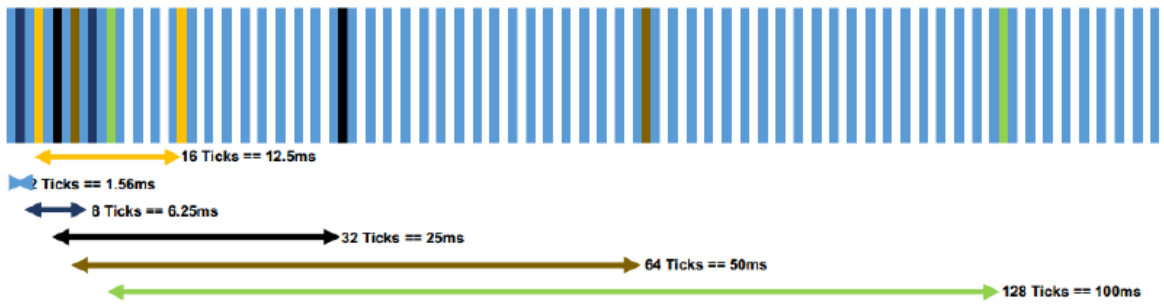
- El scheduler deberá ser diseñado de tal manera que permita ser portado a diferentes plataformas.
- El scheduler deberá activar y evitar la colisión de tareas con la periodicidad indicada a continuación:

1.56ms	6.25ms	12.5ms	25ms	50ms	100ms

Descripción Funcional

El microcontrolador HCS12x cuenta con timers de propósito general. Uno de estos timers es usado para la generación de un OS tick con periodo fijo de 781.25ps. Un tick es el término asignado a la unidad de tiempo que incrementa la variable de conteo del scheduler. Esta variable de conteo es usada para activación, a partir de los diferentes bits de su valor binario, de las diferentes tareas de acuerdo a los requerimientos establecidos.

La activación de las diferentes tareas es generada a partir de la variable de conteo. La siguiente imagen ilustra los puntos de activación de las tareas y sus respectivas periodicidades:



El desfasamiento relativo entre las diferentes tareas ocurre de la siguiente manera:

	1.56ms	6.25ms	12.5ms	25ms	50ms	100ms
Offset	0	1	3	5	7	11
Ticks	2	8	16	32	64	128

2.3.4. Analisis de Resultados

Los resultados obtenidos concuerdan con los requerimientos de generación de las diferentes tareas. Es interesante el hecho de que un solo recurso del MCU, un generador de tiempo de propósito general, permite lograr la activación de diferentes tareas periódicas.

Es importante mantener una tarea de background como parte del scheduler ya que permite una fácil inclusión de diferentes rutinas de diagnóstico y mantenimiento del sistema cuando estas sean necesarias, ya sea durante la implementación del sistema o durante la utilización final del mismo.

2.3.5. Conclusiones

La implementación de este proyecto se vio altamente beneficiada debido a la metodología de estructuración de software en capas. Fue posible hacer reuso de software previamente desarrollado en proyectos anteriores, específicamente el código relacionado a la generación del tick de tiempo base del scheduler.

Debido a la integración de la tarea background en el scheduler, fue posible añadir con facilidad rutinas de monitoreo de actividad. Estas rutinas permitieron hacer la medición y análisis de carga de CPU.

Este scheduler, aunque en cierta manera básico y simple, es la base para la implementación de sistemas mucho más complejos ya que la adición de nuevas tareas es relativamente sencilla.

3. APENDICES

Resumen: *En este capítulo se presenta la recopilación de reportes de los proyectos presentados.*

3.1. PWM GENERATOR ON HCS12X MULTICORE

PWM Generator on HCS12x Multicore Architecture

Authors: **Juan C. Razo & Omar X. Avelar**

Abstract

The Freescale's HCS12x demo board (MC9S12XEP100) contains two processing cores that form an asymmetric dual core architecture. The first main execution core (HC12) and another one dedicated RISC core called the XGATE.

On this work, we will be making use of both cores to implement a generator and driver of PWM signals. The S12 core will be responsible of the PWM signals generation while the XGATE core performs the PWM signals driving to the IO ports.

Table of Contents

1. Design Specification

1.A. Overall Requirements

- Use both cores for the next tasks:
  - XGATE for PWM generation (PWM driving in this document).
  - S12 duty cycle change management (PWM generation in this document).
- Make use of coherency mechanisms to protect common data between both cores.
- Variable duty cycle from 0 to 100% in 1% steps.

1.B. S12 PWM Generator Requirements

- The S12 core shall implement a dummy PWM control application.
- It shall run on a 50ms task.
- It shall continuously change the duty cycle of the PWM as follows:

Time (seconds)	Duty Cycle (%)
0	0
1	100
2	0
5	Repeat from Time = 0s

- Do not increase CPU load significantly (less than 5%).

1.C. XGATE PWM Driver Requirements

1. Drive 1 user configurable port channel (i.e LED).
2. Fixed 100 Hz frequency.
3. Use PIT (programmable interrupt timer) as it's BIOS hardware resource for base frequency but maintains complete independence from it.

2. S12 PWM Generator Implementation

- **FUNCTIONAL DESCRIPTION**

The PWM generator module will basically generate a continuous variable numerical DC (duty cycle) value that will be written into a given PWM control signal. The generator has to be configured by an external configuration module and the local control/status structures will be initialized based on that configuration.

- **GENERATOR HIERARCHY**

The generator has been divided hierarchically into a generator and multiple possible signals for which an independent PWM DC value will be generated. The properties of each signal will be provided by the configuration module and the dynamic generation will be tracked by the internal control and status structures of the generator.

- **PWM SIGNAL PROPERTIES**

The signals generated by this module can be of two possible types: smooth transitioning PWM signals or rough transitioning PWM signals. The signal also requires the specification of pattern providing the different DC values to be generated over time and its associated pattern time base. All these properties shall be provided by the pattern configuration module.

- **CODE IMPLEMENTATION**

The PWM generator implementation adds the following files to the project:

- `pwm_generator.h` → PWM generator defs.
- `pwm_generator.c` → PWM generator implementation
- `cnf_pwm_generator.h` → Generator configurations defs.
- `cnf_pwm_generator.c` → Generator configurations.
- `pwm_common.h` → Header for shared resources between the two cores and common macros/defines.

The PWM generator implements the following functions:

- `vfnPWMGenerator_Init` → Initialization of per signal control/status structures from the configuration module and initialization of module's pointers to these structures. This function also includes the generation of first DC delta value for smooth transitions and first write of the PWM control variable for this signal.
- `VfnUpdateDelta` → This function will be called by the initialization and the overall periodic update function. It will update the DC delta value of the given signal in-

dex within the control/status structures array for both types of transitions based on the current and next values of the DC pattern.

- `VfnWritePWMControl` → This function will write to the output PWM control variable the current DC value of the given signal index within the control/status structures array.
- `VfnUpdatePWM` → This is the periodic PWM generation function. It shall be run by the scheduler periodically and it will dynamically provide the DC values of the pattern given by the configuration module.

2.A. PWM Generator Configuration

The PWM generator configuration is made up of two structures:

- **tPWMGenerator_config with the following elements:**
  - `pwm_generator_time_base` → Time base of the overall generator application. This is, how frequent the module has CPU available to run its duty cycle update function.
  - `pwm_number_of_signals` → Number of signals configured.
  - `ptr_pwm_signals_config` → Pointer to signals array of configurations.
- **tPWMSignal_config with the following elements:**
  - `pwm_signal_id` → ID for the current signal.
  - `pwm_pattern_length` → Number of points within the pattern array.
  - `pwm_dc_pattern` → Pointer to the pattern array that describes the PWM duty cycle over time.
  - `pwm_patter_time_base` → Time base of the pattern. This is, the time between points in the pattern in ms.
  - `pwm_transition_type` → Type of transition between points in the pattern. The transition can be smooth or rough.
  - `pwm_control_signal` → Pointer to the output memory location of the generated PWM numerical duty cycle value.

2.B. PWM Generator Control/Status

The PWM generator control and status is made up of two structures:

- **tPWMGenerator_status with the following elements:**
  - `u8Number_of_PWM_signals` → Number of signals configured.
  - `PWMSignal_status` → Pointer to signals array of control and status.
- **tPWMSignal_status with the following elements:**
  - `pwm_signal_id` → ID for the current signal.
  - `pwm_current_signal_time` → This variable keeps track of the current time between 0 and the maximum time required to sweep the whole DC pattern.
  - `pwm_max_pattern_time` → This keeps the total time required to sweep the entire DC pattern. This value is used when doing the rollback of the pwm_current_signal_time value.
  - `pwm_current_dc` → This is the current DC value for this signal.
  - `pwm_current_pattern_index` → This value keeps track the current position from the pattern array.
  - `pwm_dc_transition_steps` → This variable keeps the total number of transition between two points of the pattern array when doing smooth DC generation.
  - `pwm_dc_trantision_type` → This variable tells the type of transition whether smooth or rough, given by the configuration module.
  - `pwm_dc_delta` → This keeps the current transition DC delta value to be added or decremented to the current dc value. The delta is updated when moving across the DC pattern.

3. XGATE PWM Driver Implementation

The PWM driver will basically enable a PIT for generating the PWM signal with the specified 100MHz frequency and it will drive the correct PWM duty cycle that is continuously updated by the PWM generator on the shared memory region between the two cores. The driving of the PWM signal will be output to the IO port specified by the driver configuration module.

The hierarchy of the PWM driver is very similar to that of the PWM generator. It includes a single driver that has a variable number of channels behind.

The PWM driver includes a much simpler configuration compared to that of the PWM generator. It only includes the configuration of the input DC value shared variable with the S12 core, a channel identification value and a function pointer to the IO drive function.

3. XGATE PWM Driver Implementation

The PWM driver implementation adds three files to the **PWM Generator** folder:

- `pwm_driver.cxgate` → PWM Driver for the XGATE
- `cnf_pwm_driver.h` → Driver configurations defs
- `cnf_pwm_driver.cxgate` → Driver configurations

It also makes use of the common header file **pwm_common.h** for accessing the shared variables between S12 and XGATE cores and for accessing common macros/defines.

The following two functions are all that is needed for our **pwm.cxgate** PWM driver.

```
/* ISR's are CXGATE entry points for code execution */
#pragma CODE_SEG XGATE_CODE
void vfnPWM_SW_Generate_Pulse_Cb(void);
void interrupt vfnPWM_SW_Init_XGATE_Isr(void);
#pragma CODE_SEG DEFAULT
```

The only way for our big HCS12x core to synchronize and make it execute **vfnPWM_SW_Init_XGATE_Isr()** is to perform a software interrupt as seen in our **main.c**:

```
XGATE_SW_TRIGGER(PWM_SW_Init_XGATE, SOFTWARE_TRIGGER_ENABLE);
```

In here we have made use of the project's **XGATE_SW_TRIGGER(UINT8, UINT8)** function that let's us initiate a based interrupt by writing to the **XGSWT register**. After all this is in place we now have a way to let our XGATE perform and execute the setup steps for our PWM driver.

Now we analyze that due to the fixed 100 Hz requirement we will have to go and change the PIT1 target frequency to be a hundred times 100 Hz in order to achieve the 1% increment duty cycle as noted in Design Specification. The change was done this way since it is a non used configurable requirement.

```
/** Periodic Interrupt Timer definitions */
#define    PIT_MICROTIMER_PERIOD          24000000
#define    PIT_TARGET_FREQ                2000

#define    PIT_MICROTIMER_PERIOD1         200000
#define    PIT_TARGET_FREQ1               (100 * 100)
```

Now on our XGATE code in **pwm.cxgate** we must add a call to configure and route the PIT1 timer interrupt and choose the callback of our function that will eventually generate the ON/OFF pulses; **vfnPWM_SW_Generate_Pulse_Cb()**.

```
/* Periodic Interrupt Timer registering to interrupt our XGATE core */
vfnPIT1_Init(&vfnPWM_SW_Generate_Pulse_Cb, __XGATE_CORE);
```

4. S12 CPU load constraint

One of the design specifications was avoid loading the CPU with more than 5% of extra load when integrating the PWM generator implementation .

A quick demonstration below, will show that the implementation meets this specification:

1. Taking a window of 50ms, we measured the CPU time used by the most repetitive task of the scheduler, the task of 1ms, and the usage of the PWM generator task that runs every 50ms.

2. The CPU load time used by the PWM generator was 14.6us as shown in *Figure 1*.

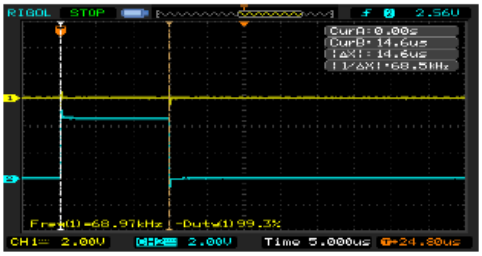


Figure 1: CPU time usage by PWM generator task

3. The CPU load time of a single 1ms task was 11.6us as shown in *Figure 2*.

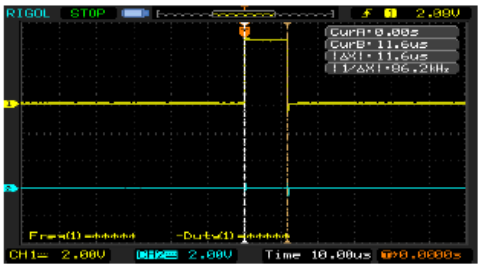


Figure 2: CPU tiem usage by 1ms task

4. Since we have 50 executions of this task, as shown in *Figure 3*, we end up with an accumulated CPU time usage of 580us only for the 1ms task.

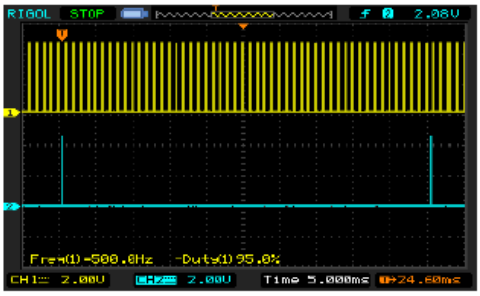


Figure 3: 50ms CPU usage window showing 1ms task (yellow) and PWM generator task (blue)

5. The percentage of CPU usage for the PWM generator task is ~2.5% when compared to the usage by the 1ms task. By adding the rest of the tasks that run within this 50ms window, the percentage of CPU usage by the PWM generator can only be less than the given approximation so far. Given the above information, we decided that this is enough evidence to confirm that the design requirement is meet.

5. Conclusions

This work allowed us to realize about the hooks available by this specific microcontroller, that makes possible a robust and efficient implementation of a multi-core application. It also allowed us to understand how important is to ensure the coherency of shared memory regions between two or more cores of a system.

Another important learning from this implementation was the understanding of the available options for communication between cores. We were able to make use of both, the shared memory regions method and also the software interrupts between cores.

We were glad to see how structured and modular our implementation turned out by including the learning of previous works about configuration modules. We easily included configuration for both modules of the application and that helped a lot to achieve an easy to scale and well structured implementation.

In general, this work was much easier to implement than previous ones but at the same time it was still fun and very interesting. One of the challenges that we faced was the dealing with pointers for the XGATE core since it required the use of pragma statements to define the proper data segment to be used.

References

[1] Embedded Systems Coursework, Fall 2014, by Abraham Tezmol. ITESO.

[2] MC9S12XEP100 Reference Manual, 2013.

3.2. CAN MULTICORE

CAN Multicore

Authors: **Juan C. Razo & Omar X. Avelar**

Abstract

This document describes the implementation and process of adapting a single-core CAN driver to a multi-core version of it. There is a base single-core implementation that works reliably for the S12 core of the demo board DEMO9S12XEP100. The purpose of this work is to achieve an adaptation of the same CAN driver but, improving its implementation by splitting the functionality between two cores. The S12 will be used for handling HAL requests while a second core, the XGATE (also available with the same demo board), will be used for handling/servicing HW interrupts.

Table of Contents

1. Assignment

Using as a base the provided working implementation of a CAN driver for MSCAN on S12XEP100 board, proved an implementation for the following requirements:

1.A. Multicore CAN driver requirements

a) Perform function execution time measurements for all Runtime functions in the single Core implementation. Record these measurements.
   a) Use a GPIO to indicate start and end of measurement. Use an oscilloscope to perform these measurements. **(See section 2)**

b) Complete the analysis presented in "CAN Multicore.xlsx" **(See section 3.C)**

c) Follow the 8 steps presented in file "Multicore for Communications" to implement "Task Parallelism – on CAN LLD" **(See sections 2 & 3)**

d) Adapt the provided source code to meet the following functional multicore requirements:
   a) Runtime functions that are triggered by events (ISR) shall be executed from XGATE core
   b) Runtime functions that are executed upon upper layer request shall be executed from S12 core **(See sections 2 & 3)**

e) Perform function execution time measurements for all Runtime functions in the dual Core implementation. Record these measurements.
   a) Use a GPIO to indicate start and end of measurement. Use an oscilloscope to perform these measurements. **(See section 2)**

f) Contrast your measurements from points a) and f). What are your conclusions? **(See sections 2 & 4)**

g) How many semaphores (as minimum) are needed per device (CAN controller) to provide multicore protection to this driver? **(See section 4)**

h) If more semaphores are used to provide multicore protection to this driver, would this result in functionality improvements? Yes/No, Why? **(See section 4)**

i) If less semaphores are used to implement multicore protection to this driver, what would be the impact in the functionality? **(See section 4)**

2. Analysis of implementation

The requirement for the multi-core implementation of the driver is to reserve the use of the S12 core for attending HAL requests while the rest of the functionality that has to do with servicing the MSCAN controller interrupts will be handled by the XGATE core.

In this section we summarize the functionality implemented by each core as well as a runtime function execution comparison between the single-core and multi-core driver implementations.

2.A. Service of HAL requests by the S12 core

The HAL requests serviced by the S12:

- En-queue of TX frames for later transmission by the MSCAN controller **(u8CAN_enqueueFrameforTx)**.

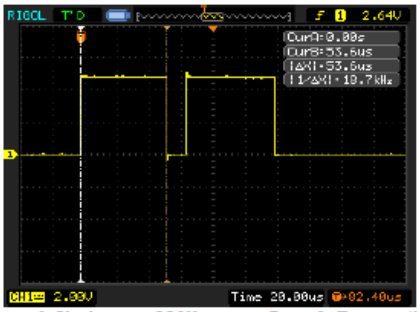


Figure 1: Single-core u8CAN_enqueueFrameforTx execution

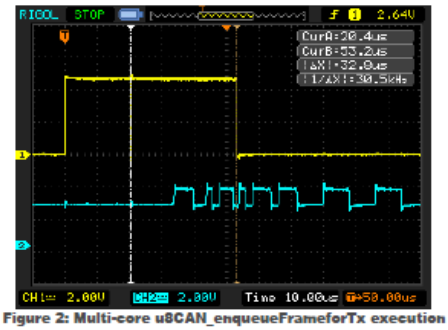


Figure 2: Multi-core u8CAN_enqueueFrameforTx execution

- Retrieval of RX messages received by the MSCAN controller and available in its RX buffers (**vfnCAN_GetRxMessages**).

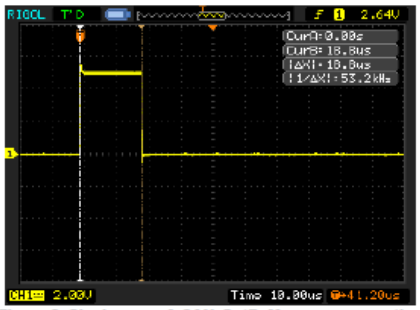


Figure 3: Single-core vfnCAN_GetRxMessages execution

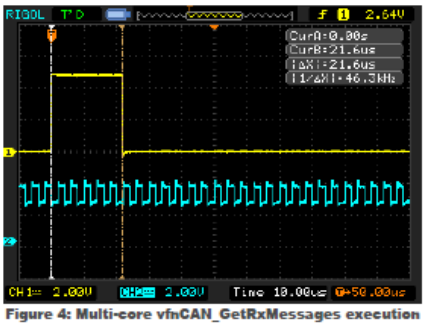


Figure 4: Multi-core vfnCAN_GetRxMessages execution

- Return of current TX buffer depth (**u8CAN_TxQueueDepth**).
  Not used by this implementation.

- Return of current RX buffer depth (**u8CAN_RxFifoDepth**).

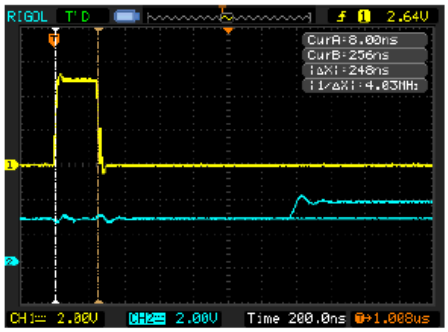


Figure 5: Single-core & Multi-core u8CAN_RxFifoDepth execution

2.B. Service of MSCAN ISRs by the XGATE core

The service of the two MSCAN RX and TX ISRs will be handled by XGATE. Each ISR will make a call to the corresponding handler.

- TX ISR (**vfnCAN_A_TxFrame_Isr**).

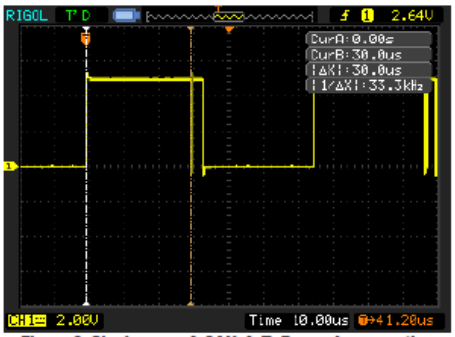


Figure 6: Single-core vfnCAN_A_TxFrame_Isr execution

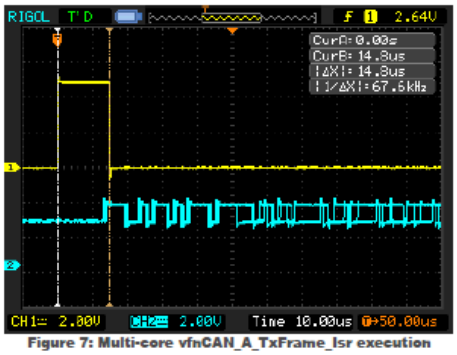


Figure 7: Multi-core vfnCAN_A_TxFrame_Isr execution

2.B. Service of MSCAN ISRs by the XGATE core

- RX ISR (**vfnCAN_A_RxFrame_Isr**).

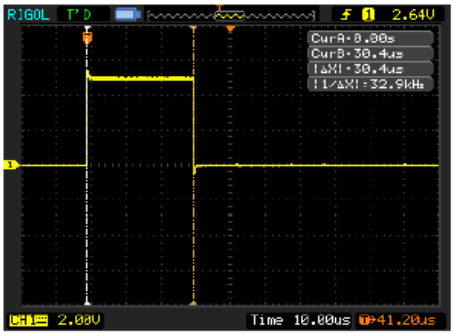


Figure 8: Single-core vfnCAN_A_RxFrame_Isr execution

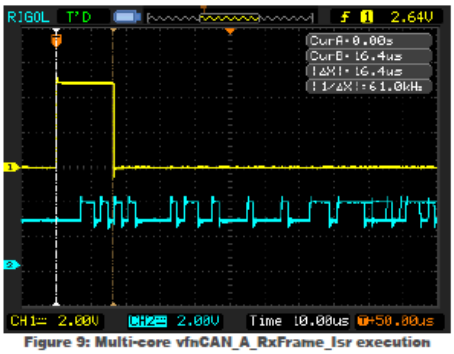


Figure 9: Multi-core vfnCAN_A_RxFrame_Isr execution

- Handler for TX ISR (**vfnCAN_Tx_handler**).

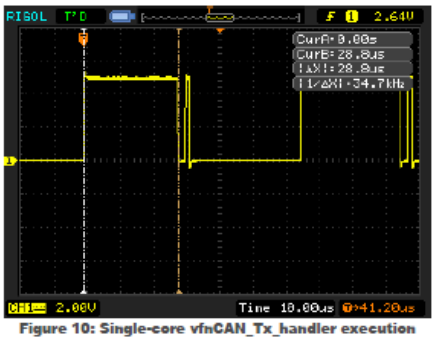


Figure 10: Single-core vfnCAN_Tx_handler execution

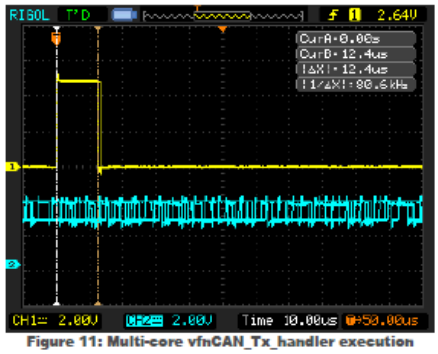


Figure 11: Multi-core vfnCAN_Tx_handler execution

- Handler for RX ISR (**vfnCAN_Rx_handler**).

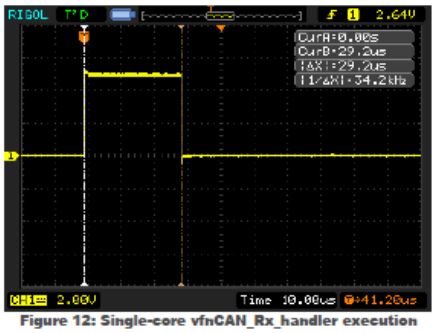


Figure 12: Single-core vfnCAN_Rx_handler execution

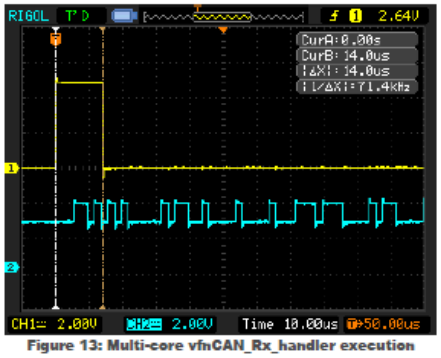


Figure 13: Multi-core vfnCAN_Rx_handler execution

3. Implementation details

In this section we present a summary of the most relevant details of implementation that made possible a successfully working multicore MSCAN driver.

3.A. S12 and HAL service

The effort for getting the S12 to handle the HAL requests was minimal since the base project was already a working implementation for this same core. A couple of things had to be adjusted specific to this core:

1. Remove the MSCAN ISR handling from this core. It involved removing the ISR from its interrupt vector and changing the destination of the MSCAN interrupts to the XGATE core instead of the S12.
2. Add the appropriate memory protection for the locations shared between S12 and the XGATE cores.

3.B. XGATE and ISR service

The effort required to get the XGATE correctly servicing the MSCAN ISRs, took us a little more of work. Below is the list of issues that had to be solved in order to have the XGATE doing what it was assigned to do:

- **XGATE C file size limitation:** The first problem faced was the limitation from the CodeWarrior IDE (student edition) of having a maximum of 512 bytes of code per C file. Once we found that this limitation was only on the file size, we were able to implement the functions for TX and RX interrupts in two different files (one for TX ISR/Handler and another for RX ISR/Handler).

- **Shared pointers and address translation between S12/XGATE local memory maps:** Another problem arose when sharing pointer variables between the two cores. The S12 and XGATE have their own local memory address maps and this requires to have an address translation for the pointers so we can make reference to the same physical memory location.

  The specific problem had to do with a pointer to a structure and its pointer members defined in the S12 code segment and accessible to the XGATE through a header file and a shared pointer within the shared data segment as shown in *CodeSnippet 1*.

```
tMSCAN_DeviceStatus *mscan; //S12 mscan pointer

#pragma DATA_SEG SHARED_DATA
/* pointer to driver global status structure
(dynamically allocated) */
tMSCAN_DeviceStatus *mscan4Xgate; //mscan pointer
for the XGATE core
#pragma DATA_SEG DEFAULT
```

CodeSnippet 1: Shared mscan4Xgate pointer

Since this pointer points to an address within the S12 address map, we had to convert the S12 pointed address 0x3400 into the corresponding XGATE address 0xF400 by using the address translation tool that comes with the CodeWarrior IDE as shown in *Figure 14*.

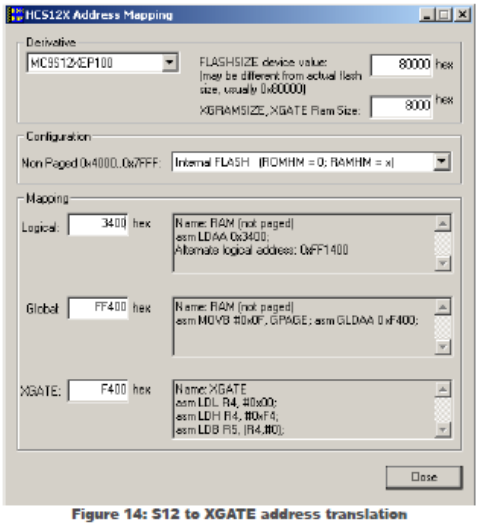


Figure 14: S12 to XGATE address translation

With the known offset between the S12 and XGATE address maps of 0xC000, we just went and added the adjustment to all the shared pointers defined by the S12 and used by the XGATE. For example for the **mscan4Xgate** pointer used by the XGATE, we did the adjustment shown in *CodeSnippet 2* below from the S12 **mscan** structure by adding the 0xC000 offset to pointer used by XGATE (mscan4Xgate).

```
#define XGATE_MEM_OFFSET        0xC000
...
mscan = (tMSCAN_DeviceStatus *)
MemAlloc_NearReserve( u8numberOfDevices *
sizeof(tMSCAN_DeviceStatus));

mscan4Xgate = ((tMSCAN_DeviceStatus*)((UINT16)mscan
+ (UINT16)XGATE_MEM_OFFSET));
```

CodeSnippet 2: mscan S12 pointer adjustment into shared mscan4Xgate pointer

- **Pointers arithmetic and XGATE natural 2 byte memory alignment:** The last problem faced when enabling the XGATE part of the driver was when doing pointers arithmetic with those pointers defined in the S12 code region. Since the XGATE has a natural 2 byte memory alignment and the S12 does not, we were seeing different pointer arithmetic results between the two cores.

  The solution to this was to enforce a 2 byte memory alignment for the data typed defined by the S12 and shared between the two cores. We achieved this by adding a pragma linker directive for enabling the 2-byte memory alignment specified in the prm file for the heap used by the memory allocator and where the shared structures end up existing physically (*Figure 15 and Figure 16).*

```
#pragma align 2
typedef struct
{
    tMSCAN_Message          *rx_buffer;
    tMSCAN_Message          *tx_buffer;
    tMSCAN_Message          *rx_write;
    tMSCAN_Message          *rx_read;
    tMSCAN_Message          *rx_end;
    tMSCAN_Message          *tx_write;
    tMSCAN_Message          *tx_read;
    tMSCAN_Message          *tx_end;
    enum tMSCAN_Device       device;
    UINT8                    rx_fifo_depth;
    UINT8                    rx_fifo_current_depth;
    UINT8                    tx_queue_depth;
    UINT8                    tx_queue_current_depth;
}tMSCAN_DeviceStatus;
#pragma align off
```

Figure 15: Use of pragma align linker directive to enforce memory alignment as defined in the prm file

```
RAM       = READ_WRITE    0x2000 TO   0x33FF  ALIGN 2[1:1];
RAM_HEAP  = READ_WRITE    0x3400 TO   0x3BFF  ALIGN 2[1:1];
CODE_RAM  = READ_WRITE    0x3C00 TO   0x3FFF  ALIGN 2[1:1];
```

Figure 16: 2-byte alignment in the prm file for the memory allocation heap region

After this memory alignment in the S12 heap was enabled, the pointers arithmetic for the two cores was trowing the same results and the problem got solved.

3.C. Memory resources protection between S12 and XGATE cores

Since we are dealing with an implementation of a multi-core system, caution and proper protection mechanisms must be employed in order to avoid undesired behavior.

This section presents an analysis of the resources accessed by each of the functions that composes the multicore MSCAN driver implementation. With this analysis it was identified what memory resources needed to implement protection for avoiding concurrent access and thus data corruption. The protection mechanism was the use of hardware semaphores provided by the XGATE core.

According to *Table 1*, the resources that represent a potential data corruption scenario, are the following:

- mscan[device].tx_queue_current_depth
- mscan[device].rx_fifo_current_depth
- TIER register

The implementation of the hardware semaphore for protecting the above data, only makes use of three macros for getting, setting and releasing the semaphore as shown in *CodeSnippet 3*. It is important to note that protecting resources at the variable/register access level, ensures that the lock of the semaphore will not last for more than 10us which is the general recommendation [1].

```
* Acquire lock to access mscan which is shared
between xgate and s12 */
do{SET_SEM(CAN_SEMAPHORE0);
}while (!TST_SEM(CAN_SEMAPHORE0));

  mscan[device].rx_fifo_current_depth--;

/* Release the hardware semaphore */
REL_SEM(CAN_SEMAPHORE0);
```

CodeSnippet 3: Use of hardware semaphore for protecting memory resources

3.C. Memory resources protection between S12 and XGATE cores

Function Name	S12/XGATE Core	Register Resource	Memory Resource	RD/WR Access
u8CAN_enqueueFrameforTx	S12	TIER (WR)	mscan[device].tx_queue_current_depth	RD
	S12		mscan[device].tx_write->ID	WR
	S12		mscan[device].tx_write->Data[u8Index]	WR
	S12		mscan[device].tx_write->Length	WR
	S12		mscan[device].tx_write->id_type	WR
	S12		mscan[device].tx_queue_current_depth	WR
	S12		mscan[device].tx_write	WR
	S12		mscan[device].tx_end	RD
	S12		mscan[device].tx_buffer	RD
u8CAN_TxQueueDepth	S12		mscan[device].tx_queue_current_depth	RD
vfnCAN_Tx_handler	XGATE	TFLG(RD/WR)	mscan[device].tx_queue_current_depth	RD
	XGATE	TBSEL(RD/WR)	mscan[device].tx_read->id_type	RD
	XGATE	TXIDR0(WR)	mscan[device].tx_read->ID	RD
	XGATE	TXDSR0(WR)	mscan[device].tx_read->Length	RD
	XGATE	TXDLR(RD)	mscan[device].tx_read->Data[u8Index]	RD
	XGATE	TXTBPR(WR)	mscan[device].tx_queue_current_depth	WR
	XGATE	TIER(WR)	mscan[device].tx_read	WR
	XGATE		mscan[device].tx_end	RD
	XGATE		mscan[device].tx_buffer	RD
u8CAN_RxFifoDepth	S12		mscan[device].rx_fifo_current_depth	RD
vfnCAN_Rx_handler	XGATE	RFLG(RD)	mscan[device].rx_write->filter_id	WR
	XGATE	IDAC(RD)	mscan[device].rx_write->ID	WR
	XGATE	RXIDR0(RD)	mscan[device].rx_write->Length	RD
	XGATE	RXIDR1(RD)	mscan[device].rx_write->Data[u8Index]	WR
	XGATE	RXIDR2(RD)	mscan[device].rx_write->NewMessage	WR
	XGATE	RXIDR3(RD)	mscan[device].rx_fifo_current_depth	WR
	XGATE	RXDLR(RD)	mscan[device].rx_write	WR
	XGATE	RXDSR0(RD)	mscan[device].rx_end	RD
	XGATE		mscan[device].rx_buffer	RD
vfnCAN_GetRxMessages	S12		mscan[device].rx_fifo_current_depth	RD
	S12		mscan[device].rx_read	RD
	S12		mscan[device].rx_read->NewMessage	WR
	S12		mscan[device].rx_fifo_current_depth	WR
	S12		mscan[device].rx_end	RD
	S12		mscan[device].rx_buffer	RD

Table 1: *Shows the access to memory and registers from the different functions of the driver.*

Highlighted resources must be protected: ___, ___, ___

4. Conclusions

As conclusion to this work, we are giving an answer to the initial questions from the requirements list.

Contrast your measurements from points a) and f). What are your conclusions? We noticed a reduced execution time for the HAL services that the S12 core executes, as well as for the execution time of the ISR + Handler that gets serviced by the XGATE core. This was actually the expectation since there is less overhead on the S12 for not having to take care of the ISRs and the XGATE does not incurs on too much overhead as well by only having to attend ISRs and nothing else.

How many semaphores (as minimum) are needed per device (CAN controller) to provide multicore protection to this driver? Our way of seeing things tell us to use as minimum two semaphores, one for the TX related functions and the other for the RX. This will allow the transmission and reception of frames to function independently without possibility of locking between each other.

If more semaphores are used to provide multicore protection to this driver, would this result in functionality improvements? Yes/No, Why? We think that it will actually bring some drawbacks to the driver since having more semaphores than the needed, will lead to sections of code where we will stall waiting for the semaphore to be released and that would signify a performance loss.

If less semaphores are used to implement multicore protection to this driver, what would be the impact in the functionality? Not using the minimum amount of semaphores could lead to sections of code unnecessarily waiting for a semaphore release even when the two protected sections does not actually need to be protected by the same semaphore. The worst case though, will lead to data corruption in the memory resources shared across the two cores of the system.

This work allowed us to understand the advantages of implementing multi-core solutions and of coruse, it allowed us to face some of the common and probably highly time-consuming issues that could frustrate the proper implementation of this type of solutions.

We also had the chance to work closely with the XGATE core and better understand the differences between this core and the S12. We learned that the XGATE is actually optimized for doing what we implemented in this driver, the service of interrupts.

Another big learning was the understanding of the coherency type of problems and the fact of coming out with a solution to scenarios where multi-core access to a shared resource may lead to undesired behavior due to data corruption in the system. The understanding of semaphores and their implementation was vital for the completion of this driver.

References

[1] Embedded Networks in Multi-Core Environments Part 1 by Abraham Tezmol, 2015.

[2] Controller Area Network by Marci Di Natale, Scuola Superiore S. Anna - Pisa, Italy.

3.3. BINARY PROGRESSION SCHEDULER

EMBBEDED SYSTEMS SPECIALIZATION PROGRAMM

Binary Progression Scheduler for HCS12

Operating Systems Design for Embedded Environments

Juan C. Razo
Email: se551985@iteso.mx

Omar Avelar
Email: se146413@iteso.mx

04/02/2015

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: *The document takes into the implementation of doing a simple counter based scheduler. It is called the binary progression scheduler (BPS) and it makes use of alternating binary values and masks in order to decide where the activation of the task happens.*

Keywords: *Scheduler, GPT, BPS, Binary Progression, Tasks, Tick, Interrupts, HCS12.*

1. Table of Contents

2. Table of Figures

3. Introduction

A binary progression scheduler shall be implemented and explained in the following pages. This type of scheduler is easy to implement for small MCU systems as it just requires a basic general purpose timer that we can use to increase a counter variable. The counter variable shall be seen as a binary counter whose bit transition to different states, and depending on the mask of the actual timer we shall activate different tasks.

4. Requirements

a) The design is based on the OS tick, mask and offset concepts.
b) The scheduler module shall provide interface to support the following purposes:

- Scheduler Initialization. `SchM_Init(SchM_TaskConfigType *SchM_Config)`
- Scheduler De-initialization. `SchM_DeInit(void)`
- Scheduler Start. `SchM_Start(void)`
- OS tick callback function. `SchM_OsTick(void)`
- Background Task. `SchM_Background(void)`
- Scheduler task callback functions.
  - Callback functions shall be referred as per the task period `SchM_Task_##period(void)` – E.g. `SchM_Task_1p56ms(void)`

c) The scheduler is designed in a manner that it can be portable between different platforms.

Scheduler Module Shall be located at BSW and Services layer from AUTOSAR.

The following task rates shall be implemented minimizing activation collisions.

1.56ms	6.25ms	12.5ms	25ms	50ms	100ms

5. Functional Description

Our HCS12 microcontroller has an already configured OS tick from the previous lab experiment, it is enabled with a period of **781.25ps**. A tick in this terminology corresponds to a unit in time and our internal counter variable shall increment each timer interrupt hence a tick is **781.25ps**. Figure 1 is a close-up to the first 4 task groups and represent how they will behave over time, where each unit is a tick and a colored bar means the activation of the task.

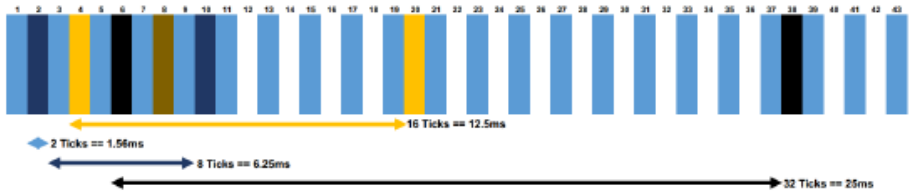


Figure 1 - First four task activation time diagram.

Zooming out a little we can now have a representation in time of the last two tasks relative to the first picture.

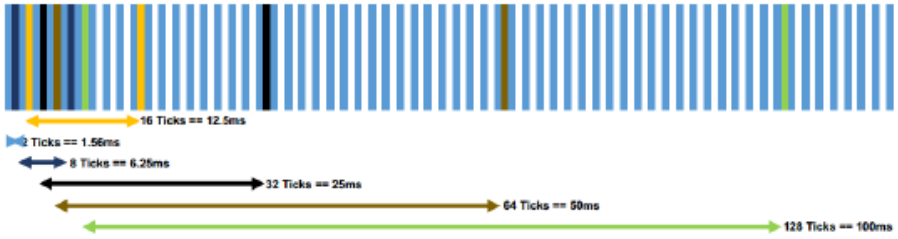


Figure 2- Overall time diagram of the six tasks.

You can see from Figure 1 and Figure 2 respectively that the following relative offsets between tasks are as follow:

	1.56ms	6.25ms	12.5ms	25ms	50ms	100ms
Offset	0	1	3	5	7	11
Ticks	2	8	16	32	64	128

Notice how the 6.25ms task is occupying the tick slot number 10, and hence we cannot make use of the 100ms to start with offset 9 but we pick 11.

The C code and module organization of the scheduler is as shown in Figure 3 – we have put a "**Services**" folder under "**BSW**".

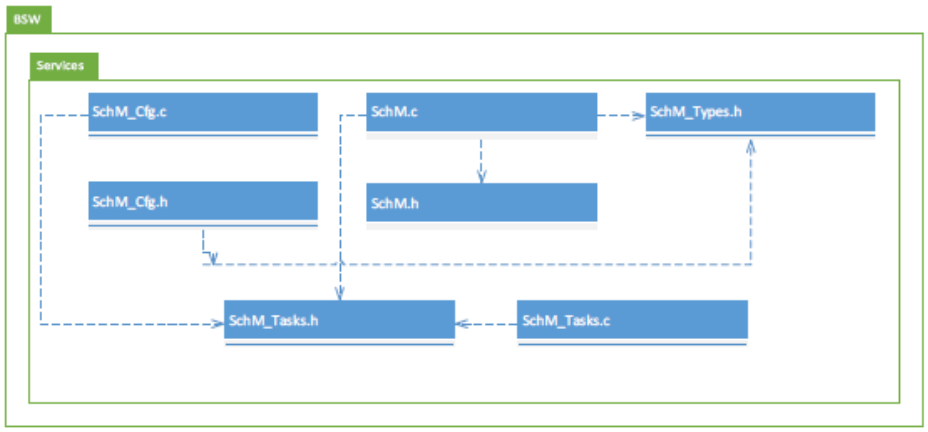


Figure 3 - Module Organization.

SchM_Cfg.c

Here the user would go and modify the array of structs defined by **SchM_PeriodicTasks[]**. Where the callback, the activation mask, offset, real mask comparison (based from mask + offset to compare against the timer, to avoid adding the variables each time), and the **SCHM_TASK_STATE_***.

SchM_Tasks.c
The body of each task (**SchM_Task_##period(void)**) that is set in our scheduler configurations live here. Here we have pin toggles to observe the behavior of the scheduler and actual work functions can live here.

SchM.c
Here we interface the global **u16_SchM_OsTickCounter** variable that is increased by our OS tick interrupt with the masks. We also provide the initialization and de-initialization API that will enable or disable our scheduler module. It is important to note that the **SchM_Start(void)** function sets our module to **SCHM_STATE_RUNNING** and starts executing our infinite **SchM_Background(void)** task.

Another very important aspect here and where the task activation and dispatching happens is inside **SchM_OsTick(void)** who in the end calls **SchM_UpdateActiveTasks(void)** and then **SchM_RunTasks(void)** in a serialized manner.

6. Results

We were able to check the final PLL frequency based on the output from the debugger provided with the CodeWarrior IDE. The figure below (Figure 4) shows the message printed during initialization of the device.

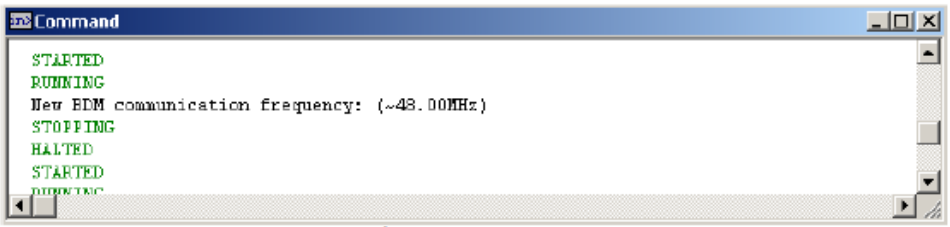


Figure 4 - PLL Frequency

The check of the PIT interrupt was done with an oscilloscope and making use of an external pin of the microcontroller. In this particular case, the pins from PORTA were used to toggle at each interrupt within a callback called inside of the PIT ISR. Since we are toggling the PORTA pin during each interrupt, the expectation is to see a waveform with double the calculated period ~1.570ms = ~785us (Figure 5).

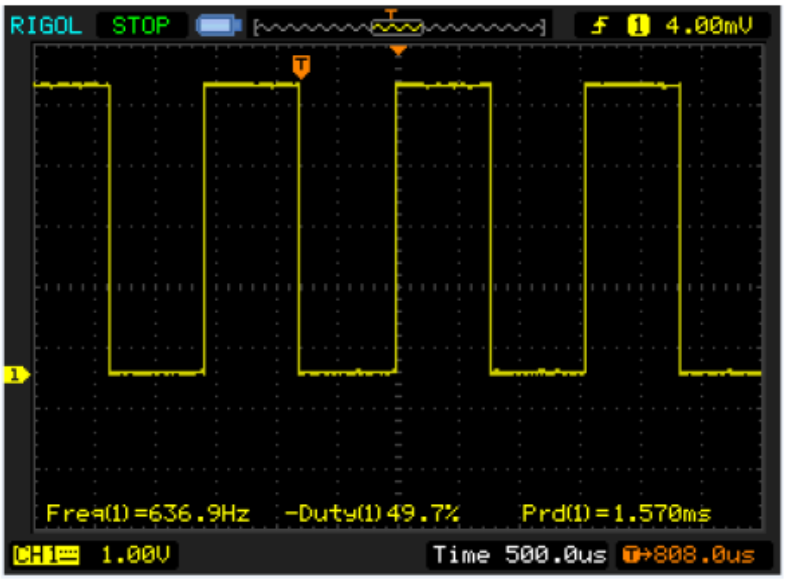


Figure 5 - Double the PIT interrupt period

6.1. Task Periodicity Measurements

Measurement to the tasks periodicity was done with an oscilloscope as shown in the Figures below:

- 1.56ms task:

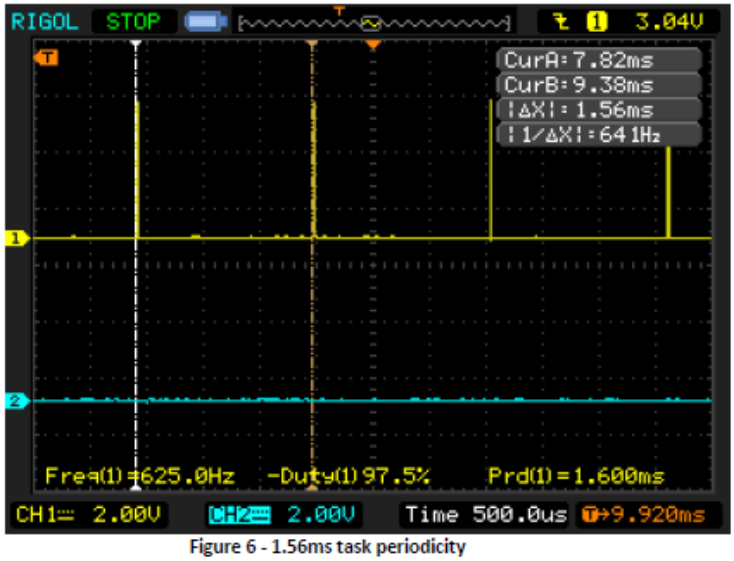


Figure 6 - 1.56ms task periodicity

- 6.25ms task:

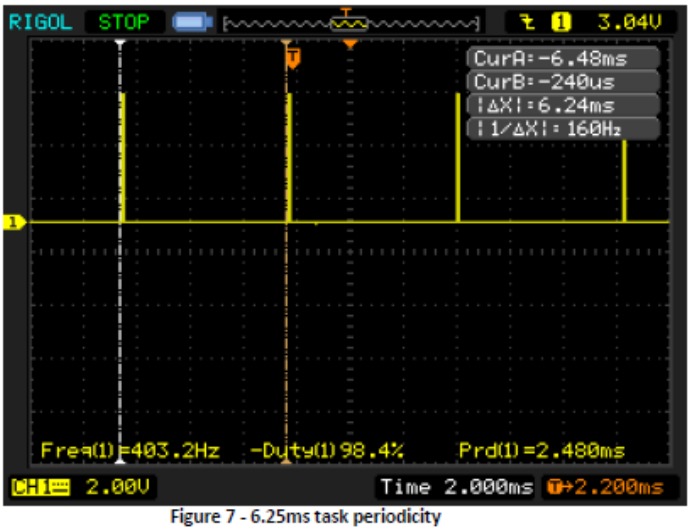


Figure 7 - 6.25ms task periodicity

- 12.5ms task:

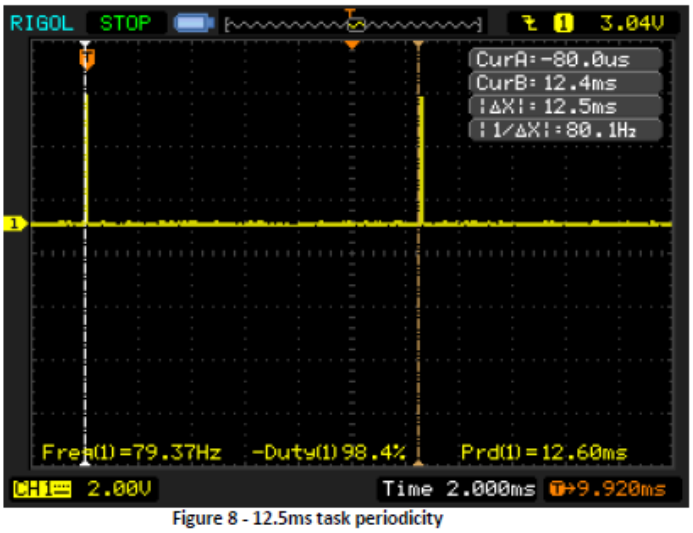


Figure 8 - 12.5ms task periodicity

- 25.0ms task:

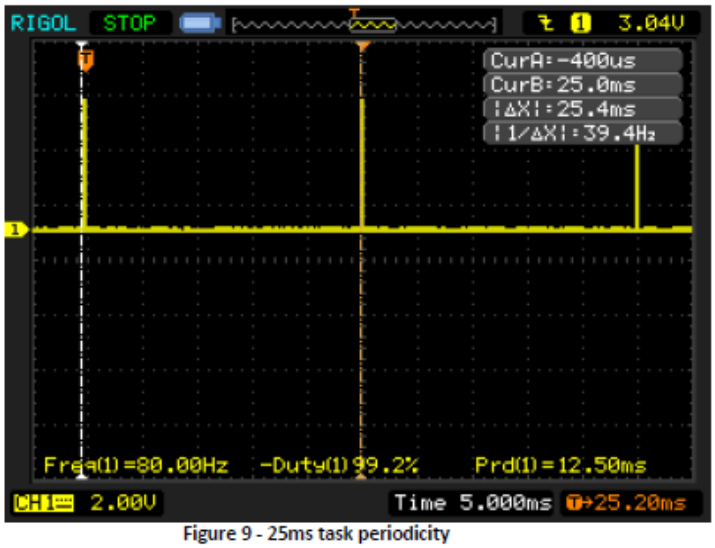


Figure 9 - 25ms task periodicity

- 50.0ms task:

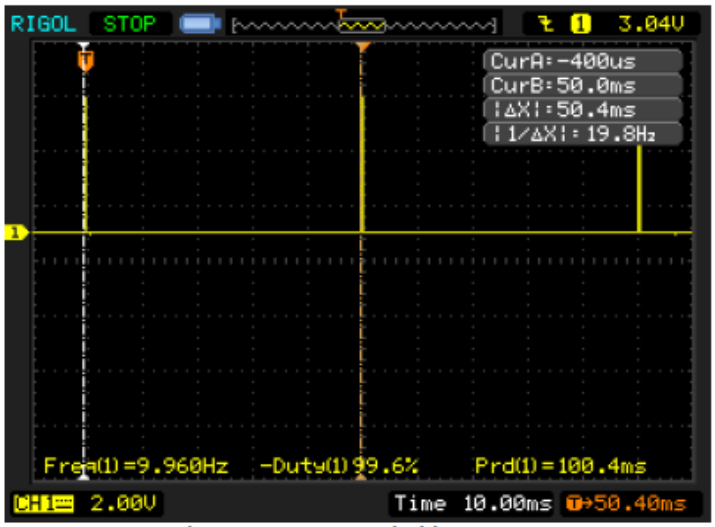


Figure 10 - 50ms task periodicity

- **100.0ms task:**

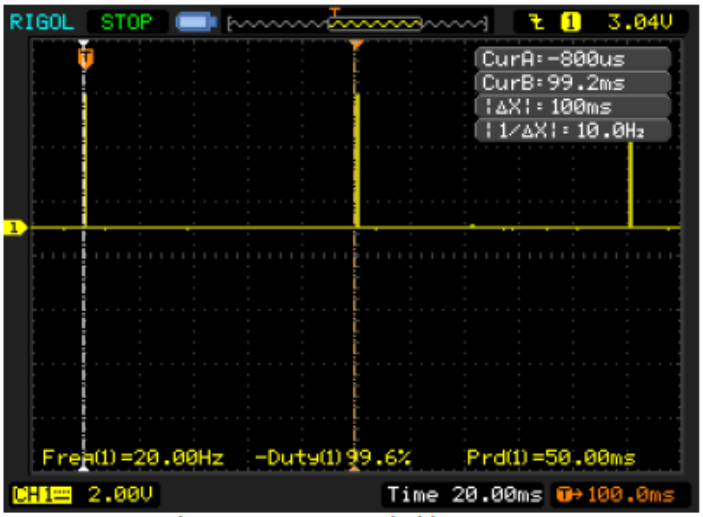


Figure 11 - 100ms task periodicity

6.2. CPU Load Measurements

The CPU load was measured with an oscilloscope. The only task being measured to get the CPU load was the background task. A pin of the board is set to 1 whenever we execute the background task and so, the duty cycle measurement of the background execution task will give us the idle percentage of the CPU.

Figure 12 shows the percentage of CPU idle state when the periodic tasks doesn't make much use of the CPU. We see a 90% of time as the background task execution and thus 10% of CPU load.

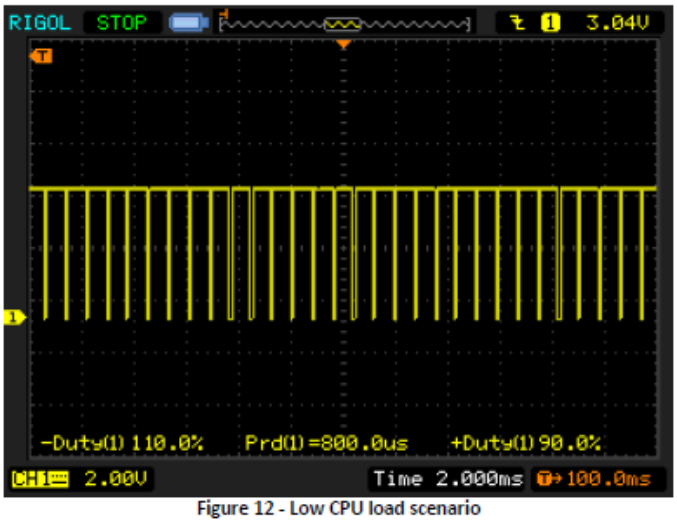


Figure 12 - Low CPU load scenario

Figure 13 shows a higher CPU load scenario. In this case the periodic tasks make use of the CPU for a longer period of time as compared with the previous low CPU load scenario.

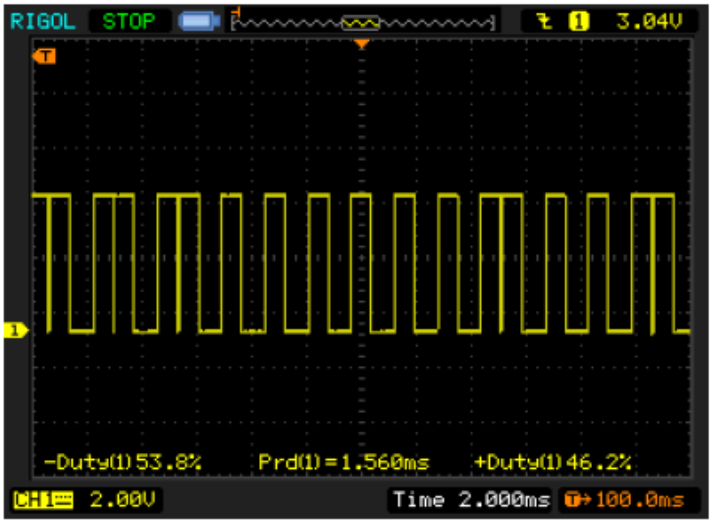


Figure 13 - Higher CPU load scenario

7. Conclusions

We keep reinforcing the benefits of having a layered and module structured project -- in here it did not require much changes but just additions to get the scheduler infrastructure up and running.

We noticed that having a background task running is beneficial and flexible for future scenarios for starters it makes it easy to measure the CPU utilization across tasks on our system and also allows for performing diagnostics or other maintenance tasks in a real system while making use of the CPU utilization -- this kind of maintenance task to optimize our CPU utilization to be close to a 100% would be true for RTOS systems where we do not have power constraints, in power constrained scenarios we would need a different strategy here to go into the low power states.

Having a scheduler is a core part of a more complex software system as we want to make use of computing resources efficiently. By starting an easy to do implementation we can see why more complex scheduling systems exist as we might hit limitations such as per-task execution time, no pre-emption and context switching depending on the goal of the project. But it is a tradeoff on microcontroller execution overhead, implementation/development efforts and code complexity.