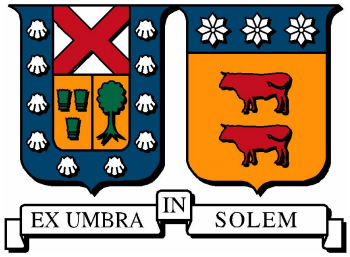
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTRÓNICA

VIÑA DEL MAR – CHILE




# "TRANSFORMACIÓN DEL SISTEMA OPERATIVO B.LU OS PARA APLICACIONES DE TIEMPO REAL"

MEMORIA PRESENTADA POR

NICOLÁS ESTEBAN SANZ YURASZECK

COMO REQUISITO PARCIAL PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO MENCIÓN COMPUTADORES

| | |
|---|---|
| PROFESOR GUÍA | : WOLFGANG FREUND |
| PROFESOR CORREFERENTE | : AGUSTÍN GONZÁLEZ |

ENERO – 2011

# Resumen

En este documento se plantea una solución para transformar a Sistema Operativo B.lu OS (BOS) de un sistema operativo de uso general, en un sistema operativo de tiempo real. Se comienza estudiando el funcionamiento interno de BOS, colocando especial énfasis en el núcleo y el planificador de tareas.

BOS es un sistema operativo de código abierto para sistemas embebidos. BOS está diseñado para trabajar sobre un microcontrolador de la familia *MSP430* y brinda al programador una capa de abstracción sobre este *hardware*. Como algoritmo de planificación de tareas, BOS cuenta con uno de tipo apropiativo y de *quantum* fijo.

También se estudian dos algoritmos de planificación de tiempo real, *Earliest deadline first* (*EDF*) y *Rate-monotonic scheduling* (*RMS*). Estos algoritmos de planificación del procesador se programaron y se agregaron a BOS.

Por último, se realizaron diversas pruebas y mediciones para evaluar el funcionamiento del sistema modificado, mejorando la latencia y el *Jitter* del sistema.

# Abstract

This document propose a solution to transform Sistema Operativo B.lu OS (BOS) from a general purpose operating system, into a real time operating sistem. Starts studing the internal operation of BOS, placing special enphasis to the kernel and the sheduling of tasks.

BOS is a open source operating system for embeded systems. BOS is designed to work on a *MSP430* and provides an abstraction layer over this hardware to the designer. As task scheduling algoritms, BOS has a preemptive task manager.

Also studied two real time scheduling algoritms, *Earliest deadline first* (*EDF*) y *Rate-monotonic scheduling* (*RMS*). This algoritms of processor are programed and added to BOS.

Finally, performed various tests and measurements to evaluate the performance of the modified system.

# Índice general

# Índice de figuras

# Índice de códigos fuente

# Capítulo 1

# Introducción

En la actualidad podemos encontrar sistemas de tiempo real en muchos aparatos que usamos en nuestra vida diaria, como pueden ser microondas, televisores, lavadoras automáticas o el sistema anti bloqueo de frenos en los autos, entre otros. Un sistema se considera de tiempo real si el correcto funcionamiento no solo depende de que la respuesta sea correcta, sino también del tiempo que demore en entregarse. Existen dos modelos clásicos, el primero se denomina sistema de tiempo real duro y se caracteriza porque los resultados deben cumplir con un tiempo dado y si se supera dicho tiempo el sistema falla. Por otro lado tenemos los sistemas de tiempo real suaves, los cuales también tienen el requerimiento de responder en un tiempo dado, pero si no se cumple, el sistema puede seguir funcionando pero de manera disminuida. Por ejemplo, mientras se despliega un video se omiten ciertos cuadros porque no se alcanzaron a decodificar.

## 1.1. Microcontrolador MSP430

*MSP430* es una familia de microprocesadores producidos por *Texas Instruments* [2]. Este microcontrolador es de bajo costo y está diseñado para un bajo consumo energético. Posee una arquitectura *RISC* de 16 bit y alcanzan un máximo de 25 [Mhz]. Además, incluye una serie de periféricos que pueden variar dependiendo del modelo tales como: oscilador interno, temporizadores que incluyen *PWM*, *watchdog*, puertos seriales asincrónicos y sincrónicos, ademas de conversores ADC de 10/12/14/16 bit.

## 1.2. Sistemas Operativos Embebidos de Tiempo Real

En las siguientes secciones se describirán varios sistemas operativos con capacidades de ejecutar tareas de tiempo real. Estos sistemas operativos resaltan por ser de código abierto y están liberados bajo licencias bastante permisivas a la hora de usarlos como base para algún desarrollo educacional o comercial.

### 1.2.1. FreeOSEK

FreeOSEK es un sistema operativo de tiempo real (*RTOS* por sus siglas en inglés) escalable para sistemas embebidos basado en las especificaciones OSEK-VDX [6]. FreeOSEK está construido de forma modular, de manera que se compilan solo los recursos necesarios, mejorando y optimizando el uso de memoria. Su código es abierto y libre bajo la licencia *GPL3* [5] con la excepción de enlace. En caso de distribuir una modificación, *GPL3* obliga a extender los términos de la licencia a todo el trabajo. La excepción de enlace permite a una aplicación, que enlace el código licenciado bajo

*GPL3*, ser distribuida bajo otros términos (privados o abiertos), pero cumpliendo con lo establecido en la licencia para el código licenciado.

### 1.2.2. Atomthreads

Atomthreads [4] es un *RTOS* gratis, liviano y portable para sistemas embebidos de tiempo real. Esta distribuido para uso comercial y educacional bajo la licencia *BSD* [7]. Está diseñado para sistemas que requieran de un *Scheduler* y las primitivas básicas que posee un *RTOS*. No incluye sistema de archivos, controladores o un *Stack* IP.

### 1.2.3. Rtems

Rtems [8] es un *RTOS* de código abierto. Actualmente está portado para un gran número de arquitecturas entre las que se pueden destacar ARM y MIPS. Dentro de sus principales características posee un *Scheduler* basado en prioridades, uno apropiativo, uno manejado por eventos y *RMS*.

# Capítulo 2

# Sistema Operativo B.lu OS (BOS)

BOS es un sistema operativo multitarea para microprocesadores de la familia *MSP430* y fue creado por Luca Bertossi en 2007 [1]. BOS cuenta con todo lo necesario para empezar a desarrollar aplicaciones sin preocuparse de escribir el código necesario para manejar el microcontrolador, los periféricos y el hilo de la aplicación ya que BOS posee rutinas de alto nivel para crear y comunicar tareas, además de diversos controladores para los periféricos más comunes (display LCD, teclado matricial, etc).

BOS está escrito en tres capas principales representadas por una capa de abstracción de *hardware* (HAL), otra referente al núcleo (KRN) y por último una dedicada a los controladores (DRV), más una capa paralela dedicada a las bibliotecas auxiliares (LIB) como se aprecia en la Figura 2.1. Las capas HAL, KRN y DRV representan las capas de abstracción de *hardware*, núcleo y controladores respectivamente. La capa superior de usuario no está estandarizada para darle la libertad al programador de crear una aplicación de la manera que más le acomode.

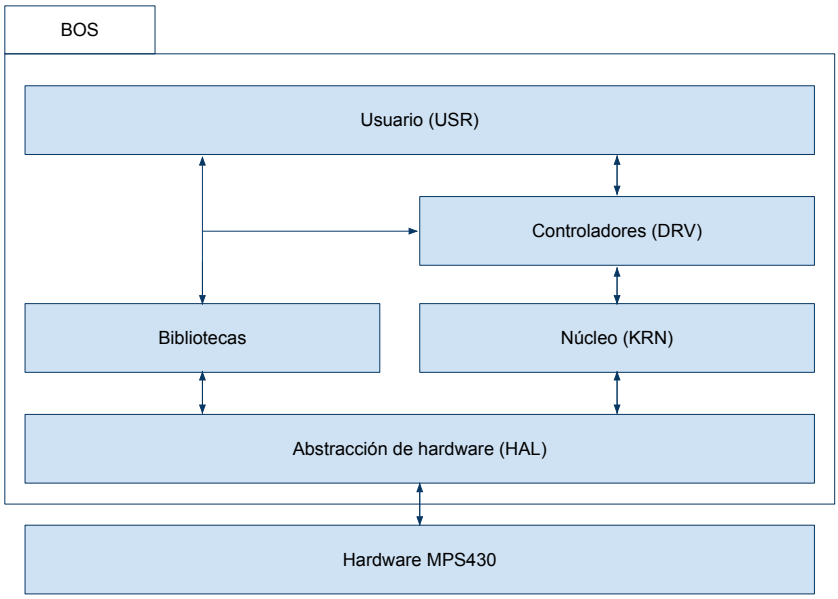



Figura 2.1: Abstracción de capas en BOS.

BOS se distribuye en forma de código fuente. Esta modalidad de distribución se debe a que BOS se compila junto con el código de la aplicación, de manera de optimizar el tamaño final de ésta. Además, BOS incluye sus propios makefiles, liberando al programador de la tarea de crear nuevos makefiles. Con el fin de reducir el tamaño final de la aplicación, BOS está construido de manera totalmente modular. A través de archivos de configuración se puede elegir los módulos y rutinas que se desea estén disponibles para la aplicación y así evitar compilar código que no se va a utilizar.

Un Sistema Operativo (OS por sus siglas en inglés) capaz de ejecutar múltiples tareas a la vez, no quiere decir que todas las tareas activas se estén ejecutando en la cpu al mismo tiempo. Para que un microcontrolador pueda ejecutar múltiples tareas a la vez, tendría que tener la misma cantidad de núcleos como tareas se requiera ejecutar simultáneamente. Por el contrario, un OS multitarea tiene la capacidad de administrar la cpu para permitir que todas las tareas puedan usarla por algún tiempo, creando la ilusión de que todas las tareas son ejecutadas simultáneamente.

Por último, cabe mencionar que BOS analiza los archivos de configuración, verificando que las rutinas disponibles no posean un llamado a alguna otra rutina que no se encuentre disponible, advirtiendo al usuario de la omisión de la declaración en los archivos de configuración.

## 2.1. Configuración de BOS

Como se mencionó con anterioridad, BOS está construido de manera modular con el fin de compilar lo mínimo necesario para que cada aplicación pueda ejecutarse utilizando la menor cantidad de memoria posible.

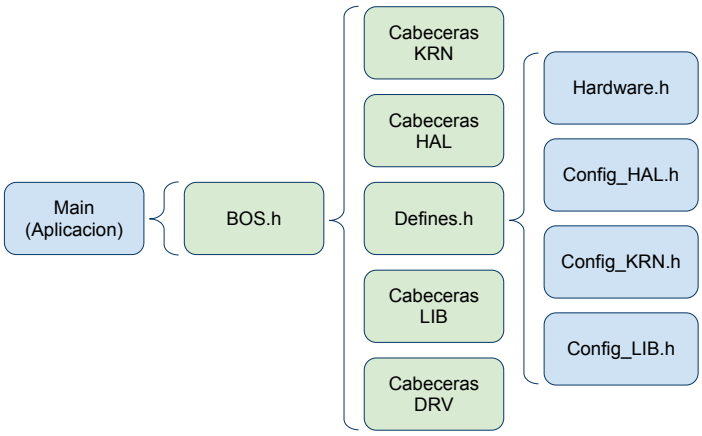



Figura 2.2: Estructura de la configuración de BOS.

Para configurar y definir las rutinas que estarán disponibles, BOS concentra todas las configuraciones en seis archivos de cabecera. La estructura con la que se incluye todo el sistema BOS en una aplicación se puede apreciar en la Figura 2.2. El usuario es responsable de la inclusión del archivo de cabecera *BOS.h* y de configurar *LIB_conf.h*, *HAL_conf.h*, *KRN_conf.h* y *Hardware.h*.

- BOS.h

  Es incluido desde la aplicación y garantiza que el archivo *Defines.h* y todas las capas de BOS sean incluidas.

- Config_HAL.h

  En este archivo se deben activar, a través de definiciones, las rutinas de la capa HAL (ver sección 2.3) y desactivar las que no sean necesarias para optimizar el uso de memoria.

- Config_KRN.h

  En este archivo se deben activar, a través de definiciones, las rutinas de la capa

KRN (ver 2.4) y desactivar las que no sean necesarias para optimizar el uso de memoria.

- Config_LIB.h

  En este archivo se deben activar, a través de definiciones, las rutinas de la capa LIB (ver 2.7) y desactivar las que no sean necesarias para optimizar el uso de memoria.

- Hardware.h

  Contiene las opciones para configurar los distintos relojes (ACLK,MCLK,SMCLK), el *quantum* y el temporizador utilizado por el *Scheduler* y las definiciones para disponer de leds para *debug*.

- Defines.h

  Es incluido en el archivo *BOS.h* y contiene todas las definiciones globales de BOS. Pertenece a BOS, y no se recomienda su modificación. En este archivo se incluyen los archivos de configuración de usuario (*Config_HAL.h* ,*Config_KRN.h* y *Config_LIB.h*).

## 2.2. Verificación de Definiciones

BOS entrega la libertad al usuario de configurar qué módulos y rutinas estarán disponible como se mencionó en sección 2.1. Esta característica le brinda gran flexibilidad a BOS, pero puede generar problemas, por ejemplo, al momento de compilar debido la omisión en cuanto a la definición de rutinas o subrutinas. Para enfrentar este problema BOS cuenta con verificación de definiciones en los archivos de cabecera del sistema.

## 2.3. Abstracción de *Hardware* (HAL)

La capa HAL define todas las rutinas de manejo directo del *hardware*, como son el reloj, las interrupciones y los temporizadores. Las rutinas de esta capa son usadas por la capa KRN (ver 2.4) de manera directa para crear funciones de más alto nivel.

## 2.4. Núcleo (KRN)

La capa KRN implementa todos los servicios del núcleo como el manejo de tareas, semáforos, manejo avanzado de interrupciones, secciones críticas y eventos. Esta capa define e implementa todas las funciones de alto nivel usadas por las aplicaciones de usuario y los controladores.

Cabe mencionar que BOS no posee un administrador de memoria, esto se debe a que BOS está escrito para trabajar en microprocesadores con unos pocos *KiB* de memoria, obligando a declarar siempre el tamaño del *Stack* para cada tarea. Para poder optimizar cuanta memoria se le asigna a cada tarea, BOS pone a disposición del usuario un controlador de *debug*, con la capacidad de informar el número de bloques usados en cada *Stack* de las tareas.

BOS posee un planificador de tareas básico apropiativo de *quantum* fijo. Este planificador le asigna un tiempo predefinido de uso de CPU a cada tarea de manera equitativa, garantizando que el uso de la cpu sea parejo para todas las tareas en la fila de tareas activas.

En las siguientes secciones se describirán en mayor detalle los semáforos, mane-

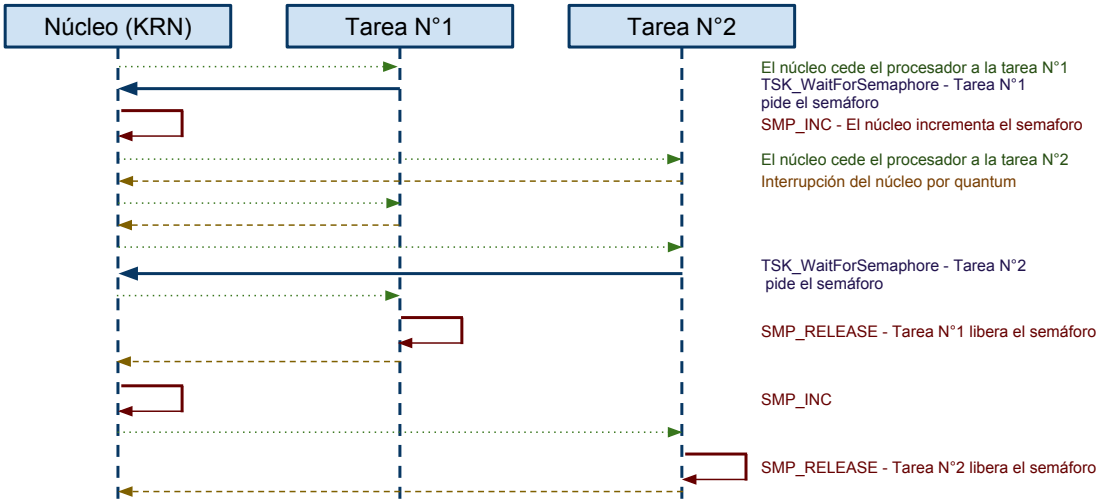



Figura 2.3: Ejemplo de diagrama temporal con uso de semáforo.

jo de interrupciones, eventos y el planificador de tareas, enfocándose en las rutinas y funcionamientos que ayudarán a comprender las modificaciones que se realizarán posteriormente a BOS.

### 2.4.1. Semáforos

En la Figura 2.3 se puede observar el uso del microcontrolador en una aplicación con dos tareas que intentan acceder a un mismo semáforo. En este caso la segunda tarea espera por la liberación del semáforo para poder continuar su ejecución. Además se encuentran destacadas todas las rutinas que se encuentran involucradas y se relacionan con los semáforos.

- TSK_ WaitForSemaphore

  Coloca la tarea desde la que es llamada en estado de espera por un semáforo y

provoca un cambio de contexto. En este caso corresponde a la Tarea 1 en primer lugar y luego a la Tarea 2.

- SMP_ INC

  Incrementa un semáforo en una unidad. Esta función es usada por el sistema para bloquear un semáforo y dar acceso a una sección de datos protegida.

- SMP_ RELEASE

  Decrementa un semáforo en una unidad. Esta función es usada por el usuario para desbloquear un semáforo que adquirió con una función de espera por semáforo.

- SMP_ RESET

  Deja un semáforo en cero, provocando que se desbloquee el semáforo.

### 2.4.2. Interrupciones

Las interrupciones en BOS son manejadas como un tipo de eventos, a diferencia de lo que se entiende por interrupciones en un microcontrolador, donde éstas son atendidas de manera inmediata por el procesador. Para entender mejor observemos un código para manejar un teclado matricial que se puede apreciar en el Código 2.1.

Lo primero es inicializar los datos de la interrupción, la función callback y el evento que se disparará como se aprecia en las lineas 7 y 8. Luego se debe registrar la interrupción indicando el tipo (puerto 2 en el ejemplo) y los datos de la interrupción previamente inicializados.

Luego la tarea hace una llamada a sistema con la función *TSK_ WaitForEvent* donde queda en espera a que se dispare el evento, que en este caso será cuando el puerto 2

Código 2.1: Ejemplo de tarea que utiliza interrupciones de puertos.

```
B_VOID UMN_KbdTask() {
  ISR_DATA   KbdPortInterrupt;   // Interrupt registration data
  EVENT    ISR2Event;            // Interrupt driven event
  ::
  ::
  EVT_RESET(ISR2Event);                          // Initialize interrupt driven event

  KbdPortInterrupt.Callback= UMN_Kbd_Callback;  // Setup interrupt callback
  KbdPortInterrupt.Event  = &ISR2Event;         // Setup interrupt event

  // Register interrupt for PORT2 with the same data
  ISR_Register(B_INT_PORT2, &KbdPortInterrupt);

  while(1)
  {
    TSK_WaitForEvent(&ISR2Event, TSK_WAIT_INFINITE);  // Wait for interrupt event

    EVT_RESET(ISR2Event);                          // Reset interrupt ervent
    ::
    ::
  }
}
```

del microcontrolador cambie de estado. Una vez que ocurre el evento, la tarea debe eliminar la ocurrencia del evento con *EVT_ RESET*.

### 2.4.3. Eventos

En la Figura 2.4 se puede observar el uso del microcontrolador en una aplicación con dos tareas, de las cuales una corresponde a *UMN_KbdTask*, tarea que fue mencionada en la sección 2.4.2. Esta tarea queda en espera de la ocurrencia de un evento y la Tarea 2 se ejecuta hasta la interrupción del puerto número dos. Luego el núcleo ejecuta una serie de instrucciones por causa de la interrupción del puerto. Estas instrucciones se ejecutan en un contexto distinto al de todas las tareas y en una sección critica del código.

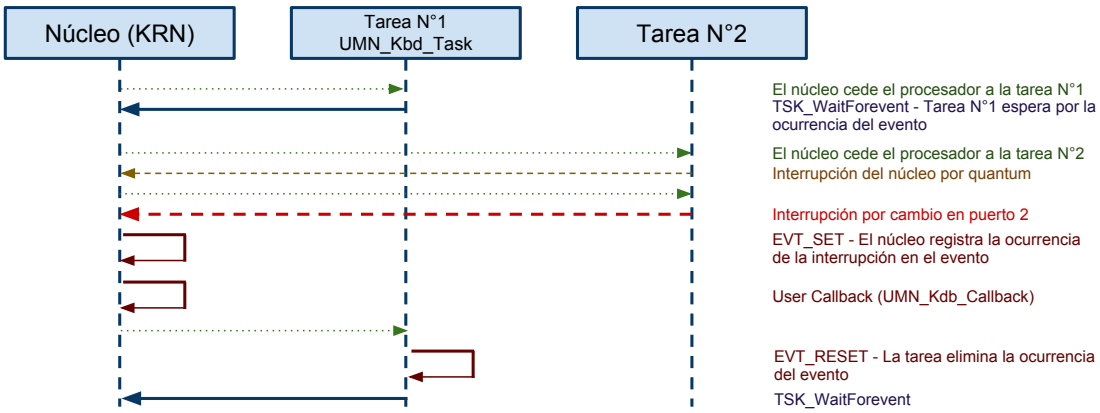



Figura 2.4: Ejemplo de diagrama temporal con uso de eventos.

- TSK_ WaitForEvent

  Coloca a la tarea que ejecuta esta rutina, en estado de espera por un evento y provoca un cambio de contexto. En el ejemplo de la Figura 2.4 corresponde a la Tarea 1.

- EVT_ SET

  Asigna estado *SET* a un evento, por ejemplo BOS llama a esta rutina cuando se genera una interrupción en el microcontrolador (puertos, USART, timers).

- EVT_ RESET

  Asigna estado *RESET* a un evento. Esta rutina debe ser llamada luego de continuar una tarea que haya hecho una llamada a sistema para esperar por un evento para indicarle a BOS que el evento ya fue atendido.

- UMN_ Kbd_Callback

  Esta función callback es definida por el usuario y debe ser lo más simple posible, ya que se ejecuta en una sección critica del código, bloqueando al *Scheduler* y deshabilitando las interrupciones del microcontrolador.

- ISR_Enable

  Macro que se encuentra disponible en el núcleo (KRN) de BOS, creada para hacer una llamada al núcleo y no a la capa HAL. Enlaza a la función *INT_Enable*.

- INT_ Enable

  Rutina encargada de habilitar o deshabilitar la interrupción que se le pase como parámetro.

### 2.4.4. Planificador

El Planificador decide qué tarea tiene el turno para utilizar la cpu, saca del estado *Waiting* a las tareas y crea el multitasking. Esta unidad se encarga de determinar en qué turno y durante cuánto tiempo, las distintas tareas de la aplicación se encuentran ejecutándose en la cpu del microcontrolador. Como se mencionó anteriormente, el algoritmo que utiliza el planificador de tareas es apropiativo y de *quantum* fijo. Esto quiere decir que cada tarea utiliza la cpu un tiempo equivalente al *quantum* y luego el planificador determina la siguiente tarea en la fila hasta el siguiente *quantum* donde se produce un cambio de contexto, lo que está ilustrado en la Figura 2.5. En esta figura se aprecia el uso de la cpu por cuatro tareas distintas, ejecutándose cada tarea por un tiempo igual y una después de otra. La lista de tareas usa el mismo orden en que aparecen en la Figura 2.5.

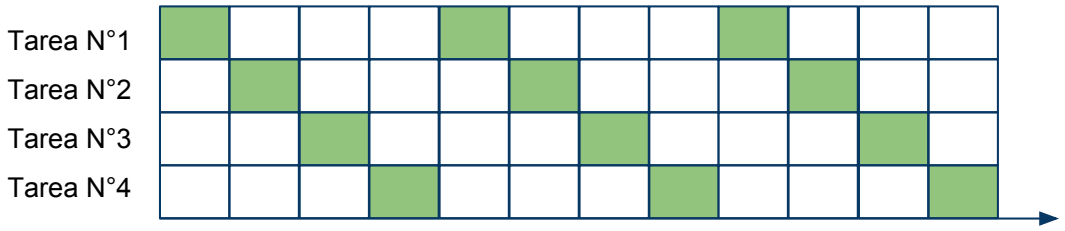



Figura 2.5: Ejemplo del planificador de tareas de BOS.

## 2.5. Diseño del Planificador en BOS

El Planificador posee una lista de tareas para administrar, un despachador (*Dispatcher*) y un algoritmo de planificación de tareas (*Scheduler*).

### 2.5.1. Tareas

BOS dispone de rutinas de alto nivel para administrar las tareas. La cantidad de tareas que se pueden crear está limitada a la disponibilidad de memoria *RAM* del microcontrolador. El *Stack* de cada tarea se guarda en la estructura que se aprecia en el Código 2.2 y el *Stack* de la tarea que se encuentra actualmente en ejecución, se guarda en la variable global *g_ Context*. Las tareas pueden tener cuatro estados predefinidos y que BOS es capaz de identificar.

- TSK_STATUS_ACTIVE (*Active*)

  En este estado la tarea puede estar ejecutándose en la cpu o en espera de su turno para usarla.

- TSK_STATUS_SLEEP

  La tarea está en estado de espera a que pase la cantidad de tiempo que se indicó.

Código 2.2: Estructura del *Stack* de las tareas.

```
typedef struct _TASK_CONTEXT_ {
  B_UINT16          SP;       // R1 = Stack Pointer
  B_UINT16          R[12];    // Registers from R4 to R15

  struct _TASK_CONTEXT_ *Next;    // Next task context to load

  // Task Data
  B_BYTE            State;    // Task state
  B_DWORD           StateData;  // task state dependant data

  #ifdef BOS_KRN_USE_TASKS_DEBUGDATA      // Debug information
    B_UINT16          StackEnd;
    B_UINT16          StackSize;
  #endif // BOS_KRN_USE_TASKS_DEBUGDATA
} TASK_CONTEXT;
```

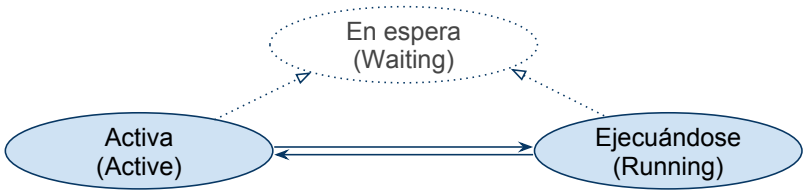



Figura 2.6: Estados de las tareas.

- TSK_STATUS_WAIT_EVT

  La tarea está en estado de espera a que suceda un evento.

- TSK_STATUS_WAIT_SMP

  La tarea está en estado de espera a que se libere un semáforo.

En la Figura 2.6 se muestra otra forma para describir los estados de una tarea. En el nuevo estado "En espera (*Waiting*)" se podría encontrar una tarea en estado *TSK_STATUS_SLEEP*, *TSK_STATUS_WAIT_EVT* y *TSK_STATUS_WAIT_SMP*. Una tarea en estado *TSK_STATUS_ACTIVE* por tanto podría estar en uno de los otros dos estados, en "Ejecutándose" (*Running*) o en "Activa" (*Ready*).

Código 2.3: Definición de la *ISR* del Planificador.

```
TASK_TIMER_ISR_DECL interrupt (TIMERA0_VECTOR) TSK_TimerInterrupt() __attribute__ ( ( naked ) )
```

### 2.5.2. Despachador (*Dispatcher*)

El *Dispatcher* se encarga de intercambiar entre contextos pertenecientes a las distintas tareas. En caso de que no existiera ninguna tarea en la cola *Ready*, existe una tarea *IDLE* siempre disponible y que consiste en un *loop* infinito, su existencia se debe a la necesidad de siempre tener una próxima tarea para ejecutar.

Como el planificador de BOS solo provoca un cambio de contexto en cada interrupción de su temporizador, por lo tanto el *Dispatcher* en BOS es una *ISR*. Utiliza uno de los temporizadores disponibles en la *MSP430*, el *Timer A* o el *Timer B* y se le da la libertad al usuario de elegir cual utilizar. Como se observa en el Código 2.3 la declaración de la *ISR* se hace con el atributo *naked*. Gracias a esta característica los registros no son empujados y la instrucción *reti* no es llamada cuando termina la *ISR*. Esto se debe a que las interrupciones modifican algunos registros, modificando el estado de la tarea que se encuentra en ejecución. La instrucción *reti* debe ser llamada al finalizar la rutina de manera explicita.

*TSK_Yield* es otro *Dispatcher* que se encuentra disponible. Es utilizado por el KRN cuando alguna tarea hace un llamado a sistema que tenga la necesidad de realizar un cambio de contexto al finalizar. Las rutinas que finalizan con un cambio de contexto utilizando *TSK_Yield* son:

- TSK_End

  Termina una tarea. Es llamada cuando una tarea retorna y ejecuta una limpieza.

- TSK_Destroy

  Destruye la tarea.

- TSK_Sleep

  Deja la tarea esperando el paso del tiempo.

- TSK_WaitForEvent

  Deja la tarea esperando la ocurrencia de un evento.

- TSK_WaitForSemaphore

  Deja la tarea esperando la liberación de un semáforo.

Para guardar y cargar contextos BOS dispone de una serie de rutinas que combinan código en *assembler*. Estas rutinas están escritas en *assembler* debido a que deben trabajar con los registros de la cpu. Sin entrar en detalle del funcionamiento interno de las rutinas, se describirán brevemente para poder comprender de mejor manera el *Dispatcher*:

- *PUSH_CONTEXT*

  Guarda el contexto del *Stack* apuntado por *g_Context*.

- *POP_CONTEXT*

  Carga el contexto del *Stack* apuntado por *g_Context*.

- *POP_END*

  Finaliza la carga retornando de la interrupción llamando a la instrucción *reti*.

- *MAIN_SAVE*

  Guarda el puntero al *Stack* (SP) del *Stack* principal.

Código 2.4: *ISR Dispatcher* de BOS.

```
TASK_TIMER_ISR_IMPL {

  PUSH_CONTEXT();       // Save current context
  MAIN_RESTORE();       // Restore main SP

  ::

  #if (defined(BOS_KRN_USE_TASKS_SLEEP) || defined(BOS_KRN_USE_TASKS_WAITEVT) || defined(↩
       BOS_KRN_USE_TASKS_WAITSEM))
    // Update task states
    TSK_UpdateStates();
  #endif

  TSK_Switch();         // Change context

  MAIN_SAVE();          // Save new main SP

  // Start new task
  POP_CONTEXT();
  POP_END();
}
```

- *MAIN_RESTORE*

  Carga el SP del *Stack* principal.

La manera en que se guarda un contexto varia en ambos casos. Para el primero es necesario guardar el contexto de la tarea y luego cargar el contexto principal, donde se ejecutan las instrucciones del núcleo para no afectar el *Stack* de las tareas. En el Código 2.4 se encuentra la *ISR* del planificador, donde la rutina *PUSH_CONTEXT* y *MAIN_RESTORE* son las encargadas de realizar el cambio de *Stack* de la tarea que se estaba ejecutando, con el *Stack* principal. Luego se llama al *Scheduler* con la función *TSK_Switch*, y por último se guarda el *Stack* principal y se carga el contexto de la tarea asignada por el *Scheduler*. De este último paso se encargan las rutinas *MAIN_SAVE*, *POP_CONTEXT* y *POP_END*.

Para el segundo caso, la rutina *TSK_Yield* solo debe guardar el contexto de la tarea

Código 2.5: TSK_ Yield *Dispatcher*.

```
B_VOID TSK_Yield()
{
  // Push r2 on stack pointer to simulate an interrupt
  asm("push r2");

  // Save current context
  PUSH_CONTEXT();

  // Change context
  TSK_Switch();

  // Start new context (and return with a "reti")
  POP_CONTEXT();
  POP_END();
}
```

en ejecución y cargar el contexto de la tarea elegida por el *Scheduler*. Como se muestra en el Código 2.5 y al igual que la *ISR*, *TSK_Yield* utiliza la rutina *PUSH_CONTEXT*, *POP_CONTEXT* y *POP_END* para guardar el contexto de la tarea en ejecución y cargar el contexto de la nueva tarea a ejecutar. Esta rutina además posee una linea escrita en *assembler* la que simula una interrupción de manera que la rutina *POP_END* tenga sentido.

### 2.5.3. Algoritmo de Planificación de Tareas (*Scheduler*)

BOS posee un *Scheduler* de tareas básico apropiativo de *quantum* fijo. Este *Scheduler* le asigna un tiempo predefinido de uso de cpu a cada tarea de manera equitativa, garantizando que el uso de la cpu sea parejo para todas las tareas en la fila de tareas activas.

En el Código 2.6 se encuentra la rutina del *Scheduler*, que consiste simplemente en buscar la siguiente tarea que se encuentre activa y en caso de no encontrarse ninguna

Código 2.6: Rutina del Planificador de BOS.

```
B_VOID TSK_Switch() {
  B_BYTE nC;

  // Go to next task
  g_Context = g_Context->Next;

  // If new task is not active, look for another valid task
  if ((g_Context->State != TSK_STATUS_ACTIVE) || (g_Context == &g_IdleContext))
  {
    // Search for a valid task (not idle)
    for (nC = 0; nC < g_TaskCount; nC++)
    {
      // Go to next task
      g_Context = g_Context->Next;

      // If it is found a valid task (ready to run and not idle)
      if ((g_Context->State == TSK_STATUS_ACTIVE) && (g_Context != &g_IdleContext))
        return;
    }

    // If no other task is ready, select idle as last (idle is always ready to run)
    g_Context = &g_IdleContext;
  }
}
```

se elige la tarea por omisión *IDLE*.

## 2.6. Controladores (DRV)

En esta capa se concentran todos los controladores que dispone BOS. El usuario no se encuentra limitado por los controladores que trae BOS y es libre de elegir, modificar y/o crear nuevos controladores para generar una capa estandarizada entre la aplicación de usuario y el *hardware* físico conectado al microcontrolador. BOS dipone de los siguientes controladores ya creados y probados:

- Internal Flash

- Debug
- Display LCD de matriz de puntos
- GPIO
- Teclado matricial 4x7
- SIPO 4094
- Puerto serial (UARTS)
- Bus multi I2C GPIO
- Temperatura (TMP175)

## 2.7. Bibliotecas (LIB)

Aunque LIB se menciona como una capa de BOS, se puede considerar como una capa paralela a la estructura de BOS como se aprecia en la Figura 2.1. Esta capa da soporte, por ejemplo, a funciones matemáticas avanzadas y a listas enlazadas.

# Capítulo 3

# Entorno de Programación

Como entorno de programación se utilizó *Ubuntu 10.04 LTS*, en donde se instaló el compilador cruzado *Mspgcc* para la familia de microprocesadores *MSP430*. Para descargar el programa a la *MSP430* se utilizó un programador *JTAG* paralelo y no un programador *JTAG-USB* ya que este último no está soportado en *Linux*. Por último como ambiente de desarrollo integrado (IDE por sus siglas en inglés) se eligió *Netbeans 6.8*.

## 3.1. *Mspgcc* en *Linux*

*Mspgcc* es un conjunto de herramientas para compilar y enlazar programas, además de incluir un debugger y otras utilidades necesarias para tener un completo ambiente de desarrollo para la familia de microprocesadores *MSP430*. *Mspgcc* puede ser utilizado en *Windows*, *Linux*, *BSD* y casi cualquier sistema *Unix*.

Para poder compilar BOS es necesario tener *Mspgcc* instalado y funcionando en el sistema. También es necesario tener instaladas y debidamente configuradas las bibliotecas que proporciona *Mspgcc*. Como mínimo las siguientes herramientas deben estar instaladas:

- msp430-gcc
- msp430-objdump
- msp430-objcopy
- msp430-size
- msp430-jtag

## 3.2. Usando *Netbeans 6.8* como IDE

La elección de *Netbeans 6.8* se debió a la necesidad de visualizar el código fuente de BOS completamente coloreado, esto incluye, además de las funciones y estamentos de control, a las macros. A pesar de que *Ubuntu 10.04 LTS* proporciona un editor de texto capaz de colorear distintos tipos de código fuente, carece de la capacidad de detectar macros. Por otro lado *Netbeans 6.8* es un completo y maduro IDE y aunque está desarrollado con un enfoque hacia *Java* también se le puede instalar soporte a *C*.

Si se configura *Netbeans 6.8* para que tenga conocimiento de las bibliotecas proporcionadas por *Mspgcc* y del directorio donde se encuentran los archivos de configuración de BOS, éste es capaz de auto detectar las rutinas y macros coloreando todo. Otra característica que se destaca es la capacidad de detectar el código fuente condicional, esto

es que distingue cuando un *#ifdef* es verdadero o falso, incluyendo o no el código correspondiente y creando un ambiente de trabajo óptimo para programar.

# Capítulo 4

# Sintonizando BOS

Para comenzar a trabajar con BOS hay que adaptarlo a nuestro nuevo ambiente de trabajo y nuevo *hardware*. En las siguientes secciones se describirá los cambios realizados a BOS para crear una base funcional donde comenzar a trabajar.

## 4.1. Sintonizado para Trabajar en Ambiente *Linux*

BOS está desarrollado en un ambiente *Windows*, el que presenta algunas diferencias con respecto a un ambiente *Linux*. La primera diferencia que se puede mencionar es como se representan las rutas de archivos en ambos ambientes. Para el caso de *Windows* se utiliza el carácter "/" como separador de carpetas, en cambio en *Linux* se utiliza el carácter "\". BOS proporciona un parche para realizar este cambio y aunque no se encuentra completo proporciona ejemplos que demuestran el procedimiento.

La segunda diferencia está en el carácter utilizado para indicar una nueva linea, en

*Windows* se utiliza *CRLF* (retorno de carro y relleno de linea) y en *Linux LF* (relleno de linea). El problema que presenta esta característica se observa al ver algún archivo del código fuente de BOS en cualquier editor de texto en *Ubuntu 10.04 LTS*, que agregará un salto de linea por cada salto que exista.

Además, BOS provee los makefiles necesarios para llevar a acabo la compilación. Como herramientas de soporte, BOS provee de dos herramientas necesarias para trabajar en el ambiente *Windows* (*listdir.exe* y *pwd.exe*). Estas herramientas no están creadas para trabajar en *Linux* y se hace necesario modificar los makefiles para utilizar comandos disponibles en esta plataforma. Al igual que la diferencia en la forma de representar las rutas de archivo, BOS dispone de ejemplos para arreglar esta diferencia en el mismo parche.

En la versión digital de este documento se adjunta un parche (0001-parche-adaptacion-linux) que genera todos estos cambios a los archivos de código fuente de BOS, de manera de tener una base sólida y cómoda para trabajar en ambiente *Linux*. Este parche se debe aplicar sobre BOS en su versión 1.1 posterior a la aplicación del parche proporcionado por BOS para adaptarlo a un ambiente *Linux*.

## 4.2. Sintonizado a Los Relojes

BOS está creado para trabajar con una placa de desarrollo construida por Luca Bertossi y la placa utilizada para esta memoria fue diseñada por el Departamento de Electrónica de la Universidad Técnica Federíco Santa María. Existen dos grandes diferencias entre ambas placas. En primer lugar la primera placa está construida con todo el *hardware* externo en ella (teclado, pantalla, etc) y la segunda dispone de acceso directo

a todos los pines de la *MSP430*, otorgándole la libertad al desarrollador de incluir el *hardware* deseado. La segunda diferencia se encuentra en los relojes utilizados, donde la placa de Luca Bertossi cuenta con un reloj de 8 [MHz] conectado a XT1 lo que difiere de la placa utilizada en este trabajo en la cual el mismo reloj está conectado a XT2, y en XT1 tiene conectados los componentes necesarios para que la *MSP430* genere un reloj de 32 [KHz]. Estas diferencias provocan que se deba modificar el código relacionado a los relojes en el archivo *HAL_ Clock.c* y *HAL_Timer.c*, para adaptarse a los nuevos tipos de relojes. Estas modificaciones se pueden apreciar en el Código 4.1 y Código 4.2.

Un efecto del cambio en la configuración de los relojes consiste en que se debe modificar el archivo de configuración *Hardware.h* para utilizar el nuevo escenario de los relojes. Esta modificación se puede observar en el Código 4.3.

## 4.3. Sintonizado a los Controladores

A causa de la diferencia de *hardware* entre la placa original de BOS y la utilizada en este trabajo, otra modificación que se hace necesaria corresponde a los controladores. No se modificaron ni probaron todos los controladores, debido a no poseer de la misma gama de componentes adosadas a la *MSP430*. Los controladores modificados se detallan a continuación y las modificaciones realizadas se encuentran en los anexos.

- Controlador *Display*

  Para manejar una pantalla BOS originalmente utiliza un circuito externo que convierte las señales de la *MSP430* de 3.3 [V] a 5 [V] de la pantalla. Para este trabajo no es necesario el circuito externo, por lo que se eliminó dicha sección de código. Además se hizo necesaria la modificación de los puertos y pines utilizados.

- Controlador *Keyboard*

  La diferencia de ambos teclados está principalmente en la cantidad de columnas que poseen. Originalmente el controlador es para un teclado matricial de 4x7 y el teclado usado es de 4x4. La otra modificación necesaria es el puerto que se utiliza para el teclado.

## 4.4. Sintonizando a los Ejemplos

Se modificaron todos los archivos *Hardware.h* de los ejemplos para utilizar la nueva configuración de relojes, ademas de los makefiles para utilizar los comandos *Linux*.

## 4.5. Modificación del *Scheduler*

En [9] se realizó varias modificaciones a BOS, principalmente a los controladores, pero la modificación más importante fue el desarrollo de un *Scheduler Round-Robin* con prioridad. Esta modificación también se realizó en este trabajo, estando disponible en el código fuente y parches de BOS.

Código 4.1: Modificación a *HAL_ Clock.c* para nueva configuración de relojes.

```
B_VOID CLK_Init() {
  // Setup clock
- // XT2OFF = Disable XT2
- // XTS    = Enable XT1 in "High Frequency" mode
- // DIVA_3 = Set ACLK to XT1 divided by 8 (B_CRYSTAL_FREQUENCY / 8)
- BCSCTL1 = XT2OFF | XTS | DIVA_3;
+ // ~XT2OFF  = Enable XT2
+ // DIVA_0 = Set ACLK to 32.768 KHz divided by 1 (B_CRYSTAL_FREQUENCY / 1)
+ BCSCTL1 &= ~XT2OFF;
+ BCSCTL1 |= DIVA_0;

  // Note : Oscillator is ready if "oscillator fault" flag
  //      can be cleared. Code must wait until flag can
  : :
  : :
  while ((IFG1 & OFIFG));

  // Setup clock
- // SELM_3 = Select MCLK from XT1 (B_CRYSTAL_FREQUENCY)
- // SELS   = Select SMCLK from XT1 (XT2 is unused)
- // DIVS_1 = Set SMCLK to XT1 divided by 2 (B_CRYSTAL_FREQUENCY / 2)
- BCSCTL2 = SELM_3 | SELS | DIVS_1;
+ // SELM_2 = Select MCLK from XT2 (B_CRYSTAL_FREQUENCY)
+ // SELS   = Select SMCLK from XT2
+ // DIVS_2 = Set SMCLK to XT2 divided by 4 (B_CRYSTAL_FREQUENCY / 4)
+ BCSCTL2 = SELM_2 | SELS | DIVS_2;

 }
```

Código 4.2: Modificación a *HAL_Timer.c* para nueva configuración de relojes.

```
TMR_Start(B_TMR_TYPE Type, B_BYTE nMillisec) {
      ::
      ::
      TACCR0 = TIMER_MILLISECOND * nMillisec;

      // Start timer
-     TACTL = TASSEL_1 | MC_1 | ID_3;
+     TACTL = TASSEL_2 | MC_1 | ID_2;
    break;

    case B_TMR_B:
    ::
    ::
      TBCCR0 = TIMER_MILLISECOND * nMillisec;

      // Start timer
-     TBCTL = TASSEL_1 | MC_1 | ID_3;
+     TBCTL = TBSSEL_2 | MC_1 | ID_2;
    break;

    // Unsupported timer
```

Código 4.3: Modificación a *Hardware.h* para nueva configuración de relojes.

```
// CPU core clock frequency in KHz

 // Clock configuration:
-//    ACLK   : XT1 / 8 = 1 MHz = 1000 KHz
-//    MCLK   : XT1 / 1 = 8 MHz = 8000 KHz (CPU core clock)
-//    SMCLK  : XT1 / 2 = 4 MHz = 4000 KHz
+//    ACLK   : XT1 = 32 KHz
+//    MCLK   : XT2 / 1 = 8 MHz = 8000 KHz (CPU core clock)
+//    SMCLK  : XT2 / 4 = 1 MHz = 2000 KHz

-#define CLOCK_FREQUENCY_ACLK 1000
+#define CLOCK_FREQUENCY_ACLK 32
 #define CLOCK_FREQUENCY_MCLK 8000
-#define CLOCK_FREQUENCY_SMCLK  4000
+#define CLOCK_FREQUENCY_SMCLK  2000
```

# Capítulo 5

# Extensión de BOS

En este capítulo se describirán las modificaciones que se realizaron a BOS para proporcionar soporte de tiempo real. Primero se explicarán algunas rutinas de apoyo, para continuar con la rutina que determina a qué tarea se le asigna la cpu, luego se modificó en qué momento se producen las interrupciones y por último se agregó soporte de tiempo real a los semáforos, interrupciones y eventos.

## 5.1. Rutinas de Apoyo

### 5.1.1. TSK_ Suspend

Esta rutina se encarga de poner la tarea desde donde se invoca en estado *Waiting*, hasta que el núcleo determine que debe pasar a estado *Ready*. Como se vio en 2.5.1 BOS posee cinco definiciones de estados para las tareas, de manera que se agregó un nuevo

Código 5.1: Rutina *TSK_ Suspend*.

```
B_VOID TSK_Suspend(){
  // Set sleep mode
  g_Context->State  = TSK_STATUS_SUSPEND;

  // Change task context
  TSK_Yield();
}
```

estado *TSK_ STATUS_ SUSPEND*. *TSK_ Suspend* por lo tanto se encarga de cambiar el estado de la tarea a suspendida y generar un cambio de contexto como se aprecia en el Código 5.1.

### 5.1.2. TSK_ UpdateTime

BOS posee una rutina que se encarga de actualizar los estados de las tareas (*TSK_ UpdateStatus*) y de manera similar se hace necesario disponer de una rutina que actualice el *deadline* relativo de cada tarea. Esta rutina se puede apreciar en el Código 5.2.

### 5.1.3. TSK_ SetNextInterrupt

BOS en su estado original solo genera un cambio de contexto debido a la interrupción del *timer* del *Scheduler* o en caso de que una tarea pase a estado *Waiting*. En un *RTOS* se debe configurar la interrupción de manera dinámica de manera de garantizar la ejecución de tiempo real de las tareas. Para dar soporte a esta necesidad se creó la rutina *TSK_ SetNextInterrupt*, que determina el tiempo restante para que cada una de las tareas que deba ser despertada de un *sleep* o un *suspend* y configura el *timer* con el tiempo adecuado, lo que se observa en el Código 5.3.

Código 5.2: Rutina *TSK_ UpdateTime*.

```
B_VOID TSK_UpdateTime(){
  TASK_CONTEXT *Current;

  // Look for all tasks
  if (g_KRN_ElapsedTime!=0){

    // Initialize variables
    Current = g_Head;

    while (1)
    {
      // Check task state
      if ((Current->State == TSK_STATUS_ACTIVE) && (Current->Deadline < TSK_NO_DL))
      {
        // disminuimos el deadline
        if ((Current->StateData > g_KRN_ElapsedTime))
          Current->StateData -= g_KRN_ElapsedTime;
        else
          Current->StateData = 0;

      } // if

      // If done, exit
      if (Current->Next == g_Head)
        break;

      // Go to next task
      Current = Current->Next;

    } // while(1)
  } // if (g_KRN_ElapsedTime!=0)
}
```

### 5.1.4. TMR_ GenerateInterrupt

Esta rutina se encarga de generar una interrupción por *software* del *timer* del *Scheduler* como se puede apreciar en el Código 5.4. Esta interrupción generada por *software* se realiza utilizando los registros secundarios del *timer*, de manera de poder identificar por *software* la causa de la interrupción.

### 5.1.5. KRN_ SetElapsedTime

Esta rutina se encarga de calcular el tiempo transcurrido desde el último evento de *Scheduler*. Lo primero que hace es evaluar si el *timer* ha terminado, y de ser así utiliza el valor al que fue configurado como referencia de tiempo. En caso contrario, calcula el tiempo transcurrido observando el registro del *timer*. Esto se puede apreciar en el Código 5.5.

## 5.2. Cambio de Contexto

Para realizar cambios de contexto en los momentos correctos se hizo necesario disponer de tiempos relativos de las tareas y tener la capacidad de configurar de manera correcta la siguiente interrupción del *timer*. Estas nuevas necesidades quedan resueltas por las rutinas mencionadas en 5.1. Teniendo esto en consideración y conociendo el funcionamiento de BOS (ver 2.5.2) se modificó la rutina de interrupción del *Scheduler*, así como *TSK_ Yield()* para que, además de actualizar los estados de las tareas y realizar un cambio de contexto, actualicen el tiempo relativo de las tareas y configuren correctamente la siguiente interrupción. Estas modificaciones se pueden apreciar en los Códigos 5.6 y 5.7.

A pesar de que ambas modificaciones son similares, difieren en cuándo se aplican estos cambios. En el primer caso se aplica para cada llamado a la rutina de interrupción y por el contrario para el segundo solo se aplica en caso de que la tarea que solicita el cambio de contexto sea periódica. Este comportamiento se propone dado que si la tarea que solicita un cambio de contexto es no periódica (prioridad más baja), entonces no existe una tarea periódica que esté en la fila *Active*, eliminándose así la necesidad de

actualizar todo.

## 5.3. Semáforos

Para los semáforos se modificó las rutinas *SMP_ RELEASE* y *SMP_ RESET* (ver 2.4.1). Estas modificaciones se ilustran en el Código 5.8.

## 5.4. Interrupciones y Eventos

Como las interrupciones en BOS son manejadas a través de eventos, solo se proporcionó soporte de tiempo real a los eventos. Como se vio en 2.4.3, los eventos están comandados por dos macros: *EVT_ SET* y *EVT_ RESET*, el primero es utilizado por el *kernel* para indicar la ocurrencia de un evento, y el segundo por el usuario para indicar que el evento ya fue atendido, por lo tanto solo se modificó *EVT_ SET* de manera que genere una interrupción y se produzca el cambio de contexto de manera adecuada. Estos cambios se pueden apreciar en el Código 5.9.

## 5.5. EDF

*Earliest Deadline Fisrst* (*EDF*) es un algoritmo de planificación de tareas con asignación de prioridad dinámica, dando preferencia a las tareas con un *deadline* más próximo. Una vez que ocurra un evento de planificación de tareas (se crea una nueva tarea, una tarea termina o simplemente una tarea pasa a estado *Ready*), se analiza la lista de

tareas *Ready* en busca de la tarea que posea su *deadline* más próximo para asignarle el uso de la cpu.

Se modificó la rutina *TSK_ Switch* donde se escribió un algoritmo de itineración que priorice las tareas evaluando su *deadline*Ėl código fuente propuesto se puede apreciar en el Código 5.10.

## 5.6. RMS

Al contrario que *EDF*, *Rate-Monotonic Scheduling* (*RMS*) asigna prioridades a las tareas de manera estática e igual a la frecuencia de la tarea. Esto quiere decir que mientras más pequeño el periodo de la tarea, más alta es su prioridad. Este algoritmo impone las siguientes restricciones sobre las tareas:

- Las tareas no comparten recursos (*hardware*, semáforos, entre otros).
- Las tareas que no sean periódicas no poseen *deadline*.
- El cambio de contexto no genera *overhead.*

Al igual que para *EDF* se modificó la rutina *TSK_ Switch*, pero se escribió un algoritmo de itineración que priorice las tareas evaluando su periodo (ver **??**). El código fuente propuesto se puede apreciar en el Código 5.11.

Código 5.3: Rutina *TSK_ SetNextInerrupt*.

```
B_VOID TSK_SetNextInterrupt(){
  TASK_CONTEXT  *Current, *ndTask;

  // Initialize current
  Current = g_Context->Next;

  // Initialice next default task
  ndTask = &g_IdleContext;

  // Look for all tasks to fin the next
  while (1)
  {
    // Check task type
    if ((Current->Deadline!=TSK_NO_DL) || (Current->Period!=TSK_NO_PERIOD)){
        // Check task state
      #if defined(BOS_KRN_USE_TASKS_SLEEP)
        if((Current->State == TSK_STATUS_SUSPEND) || (Current->State == TSK_STATUS_SLEEP))
      #else
        if(Current->State == TSK_STATUS_SUSPEND)
      #endif
      {
        if ((Current->StateData < ndTask->StateData))
            ndTask = Current;
      } // if
    } // if

    // If done, exit
    if (Current->Next == g_Context)
      break;

    // Go to next task
    Current = Current->Next;

  } // while(1)

  if(ndTask!=&g_IdleContext){
    if ((ndTask->StateData)>TSK_TIME_MAX)
      g_KRN_NextInterruption = (B_BYTE) TSK_TIME_MAX;
    else if ((ndTask->StateData)<TSK_TIME_MIN)
      g_KRN_NextInterruption = TSK_TIME_MIN;
    else
      g_KRN_NextInterruption = ndTask->StateData;
  } else {
    g_KRN_NextInterruption = (B_BYTE) TSK_TIME_MAX;
  }
  // Set the new time for the sheduler interruption
  TASK_TIMER_UPDATE(g_KRN_NextInterruption);
}
```

Código 5.4: Rutina *TMR_ GenerateInterrupt*.

```
B_UINT16 TMR_GenerateInterrupt(B_TMR_TYPE Type){
  switch (Type)
  {
    case B_TMR_A:
      // Set Interrupt
      TACTL |= TAIFG;

    break;

    case B_TMR_B:
      // Set Interrupt
      TBCTL |= TBIFG;
    break;

    // Unsupported timer
    default:
    return B_EC_UNSUPPORTED;
  }
  // All Ok
  return B_EC_SUCCESS;
}
```

Código 5.5: Rutina *KRN_ SetElapsedTime*.

```
B_VOID TSK_SetElapsedTime(){
  B_WORD TARTime = TAR;
  if (TMR_IsEnded(TASK_TIMER)==B_EC_SUCCESS) g_KRN_ElapsedTime = g_KRN_NextInterruption;
  else {
    g_KRN_ElapsedTime = (TARTime+g_KRN_RemainTime)/TIMER_MILLISECOND;
    g_KRN_RemainTime = (TARTime+g_KRN_RemainTime)%TIMER_MILLISECOND;
  }
}
```

Código 5.6: Rutina de interrupción del *Scheduler* modificada.

```
TASK_TIMER_ISR_IMPL
{
  // Save current context
  PUSH_CONTEXT();

  // Restore main SP
  MAIN_RESTORE();

  // Note : Now SP is set to original CPU stack. This way task switch operations
  //       are performed out of every task stacks.

  #if (defined(BOS_KRN_USE_TASKS_RMS) || defined(BOS_KRN_USE_TASKS_EDF))
    // Get global elapsed time
    TSK_SetElapsedTime();

    // Update task deadlines
    TSK_UpdateTime();

    // Update task states
    TSK_UpdateStates();
  #else
    #if (defined(BOS_KRN_USE_TASKS_SLEEP) || defined(BOS_KRN_USE_TASKS_WAITEVT) || defined(↩
          BOS_KRN_USE_TASKS_WAITSEM))
      // Update task states
      TSK_UpdateStates();
    #endif
  #endif // BOS_KRN_USE_TASKS_EDF

  // Change context
  TSK_Switch();

  #if (defined(BOS_KRN_USE_TASKS_RMS) || defined(BOS_KRN_USE_TASKS_EDF))
    // Set the next interrupt
    TSK_SetNextInterrupt();
  #endif

  // Save new main SP
  MAIN_SAVE();

  // Start new task
  POP_CONTEXT();
  POP_END();
}
```

Código 5.7: TSK_ Yield modificado.

```
B_VOID TSK_Yield(){
  // Push r2 on stack pointer to simulate an interrupt
  asm("push r2");

  // Save current context
  PUSH_CONTEXT();

  // Restore main SP
  MAIN_RESTORE();

  if(g_Context->Period == TSK_NO_PERIOD || g_Context->Deadline == TSK_NO_DL)
    TSK_Switch();
  else {
    // Get global elapsed time
    TSK_SetElapsedTime();

    // Update the task relative time
    TSK_UpdateTime();

    // Update Task status
    TSK_UpdateStates();

    // Change context
    TSK_Switch();

    // Set the next timer interrupt
    TSK_SetNextInterrupt();
  } // else

  // Save new main SP
  MAIN_SAVE();

  // Start new context (and return with a "reti")
  POP_CONTEXT();
  POP_END();
}
```

Código 5.8: Modificación del semáforo en *KRN_ Semaphore.h*

```
#define SMP_RELEASE(Sem) ENTER_CRITICAL_SECTION(); Sem.Count--; TMR_GenerateInterrupt(TASK_TIMER); ↩
    EXIT_CRITICAL_SECTION();
#define SMP_RESET(Sem) ENTER_CRITICAL_SECTION(); Sem.Count = 0; TMR_GenerateInterrupt(TASK_TIMER); ↩
    EXIT_CRITICAL_SECTION();
```

Código 5.9: Modificación a eventos en *KRN_ Event.h*

```
#define EVT_SET(Event) {ENTER_CRITICAL_SECTION(); Event.State = EVENT_STATE_SET; TMR_GenerateInterrupt(↩
    TASK_TIMER); EXIT_CRITICAL_SECTION();}
```

Código 5.10: *Scheduler* EDF.

```
// Go to next task
g_Context = g_Context->Next;

// Default next task
Current = &g_IdleContext;

// Search for a valid task (not idle)
for (nC = 0; nC < g_TaskCount; nC++)
{
  if ((g_Context->State == TSK_STATUS_ACTIVE) && (g_Context != &g_IdleContext))
    if (g_Context->StateData < Current->StateData)
      Current = g_Context;

  // Go to next task
  g_Context = g_Context->Next;
}

g_Context = Current;
```

Código 5.11: *Scheduler* RMS.

```
// Default next task
  CurrentNext = &g_IdleContext;

  // Go to next task
  Original = g_Context;

  // Search for a valid task /comentado:(periodic) whith less period
  //for (nC = 0; nC < g_TaskCount; nC++)

  while(1)
  {
    if ((g_Context->State == TSK_STATUS_ACTIVE) && (g_Context != &g_IdleContext))
    {
      //if (g_Context->Period < TSK_NO_PERIOD)
      //{
        // Less period => more priority
        if (g_Context->Period < CurrentNext->Period)
          CurrentNext = g_Context;
      //}
    }

    // If done, exit
    if (g_Context->Next == Original)
      break;

    // Go to next task
    g_Context = g_Context->Next;
  }
  g_Context = CurrentNext;
```

# Capítulo 6

# Evaluación Experimental

Para poner a prueba los algoritmos resueltos se medirá el *Jitter* [10] y la latencia [11], además de proponer un caso que permita evaluar el funcionamiento de *EDF* y *RMS* bajo situaciones de alta demanda. Para poder tomar muestras se utilizará el analizador lógico *Agilent Technologies 1693A*, que posee una precisión de picosegundos. Para el caso del *Jitter* se utilizó un espacio muestral mayor a 4.000 eventos.

## 6.1. Caso de Prueba

Se propone un programa que lleva a BOS a una utilización de cpu cercana al 100 %. Para lograr esto se plantea el uso de tres tareas periódicas con periodos de 30, 40 y 60 [ms]. Estas tareas permanecen en un ciclo *for* por aproximadamente 15, 15 y 5 [ms] respectivamente. En la Figura 6.1 se puede apreciar como BOS, usando *EDF*, soporta la exigencia de las tres tareas y respeta el periodo de 60 [ms] de la tercera tarea (M1 to

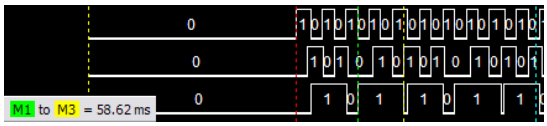



Figura 6.1: Caso en que *EDF* funciona.


Figura 6.2: Caso en que *RMS* falla.

M3 = 59). Esto se puede apreciar ya que las tareas periódicas mantienen en estado 0 un pin de salida al estar activa y en 1 al estar en espera. Por el contrario, si se configura BOS para utilizar *RMS*, en la Figura 6.2 se aprecia como la tercera tarea se ejecuta tarde (M1 to M3 = 69)

## 6.2. Medición del Jitter

Para medir el *Jitter* se prepararon escenarios de prueba con una sola tarea periódica, otro de tres tareas periódicas y por último se agregó una cuarta tarea no periódica que hace uso de semáforos y eventos, todas ejecutándose junto a BOS configurado para utilizar *RMS* y *EDF*. En los tres casos se evaluó el *Jitter* periódico y *Jitter* acumulado para tareas con periodos de 10, 20 y 40 [ms]. Las definiciones de *Jitter* periódico y acumulado las podemos ver a continuación, donde $T_0$ representa el tiempo del primer evento, $T_n$ la ocurrencia del evento n y p representa el periodo de la tarea analizada.

*Jitter* periódico: $$J_{per} = T_n - T_{n-1}$$

*Jitter* acumulado: $$J_{acu} = T_n - T_0 - n * p$$

Los resultados obtenidos para *Jitter* periódico de las tareas utilizando *EDF* se pueden apreciar en las Figuras 6.3, 6.4, 6.5, 6.9, 6.10, 6.11, 6.15, 6.16 y 6.17. Y utilizando *RMS* se pueden apreciar los resultados en las Figuras 6.21, 6.22, 6.23, 6.27, 6.28, 6.29, 6.33,

6.34 y 6.35. Además, se determinó el *Jitter* acumulado del sistema para el caso en que se encuentran corriendo tres tareas periódicas en BOS más la tarea no periódica, que se puede apreciar en la Figura 6.39 para el caso de *EDF* y en la Figura 6.40 para el caso de *RMS*.

Los resultados obtenidos para *Jitter* acumulado utilizando *EDF* se pueden apreciar en las Figuras 6.6, 6.7, 6.8, 6.12, 6.13, 6.14, 6.18, 6.19 y 6.20. Y utilizando *RMS* se pueden apreciar los resultados en las Figuras 6.24, 6.25, 6.26, 6.30, 6.31, 6.32, 6.36, 6.37 y 6.38.

## 6.3. Medición de la Latencia

Para medir la latencia se preparó BOS para que posea una sola tarea ejecutándose y además se configuró el *timer* que queda libre para generar una señal *PWM* con periodo de 10, 20, y 40 [ms], además de realizar el experimento para *EDF* y *RMS*. La tarea en ejecución responde a cantos de subida de la *PWM* y cambia el estado de salida de un *pin*. De esta manera podemos observar el tiempo transcurrido entre el canto de subida de la *PWM* y el cambio de estado del *pin*. Los resultados obtenidos para el caso de *EDF* se pueden apreciar en las Figuras 6.41, 6.42 y 6.43. Los resultados obtenidos para el caso de *RMS* se pueden apreciar en las Figuras 6.44, 6.45 y 6.46.

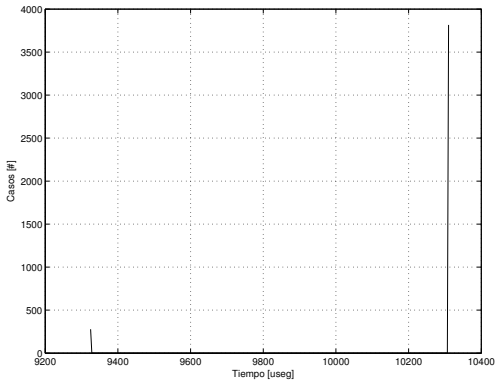



Figura 6.3: *Jitter* periódico para tarea con periodo de 10 [ms] en *EDF*.

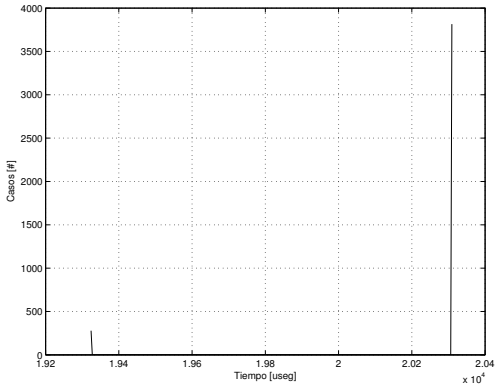



Figura 6.4: *Jitter* periódico para tarea con periodo de 20 [ms] en *EDF*.

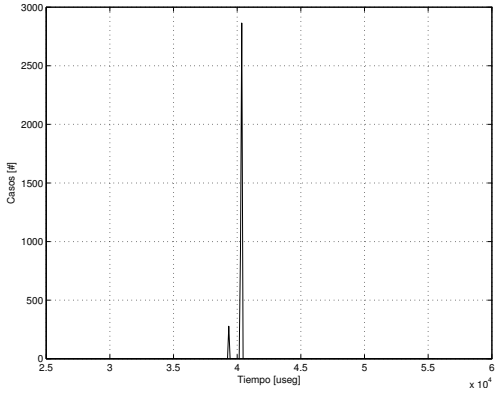



Figura 6.5: *Jitter* periódico para tarea con periodo de 40 [ms] en *EDF*.

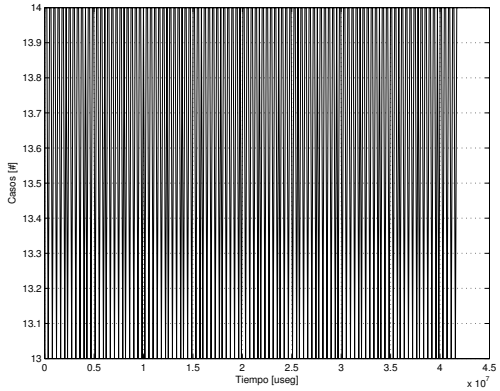



Figura 6.6: *Jitter* acumulado para tarea con periodo de 10 [ms] en *EDF*.

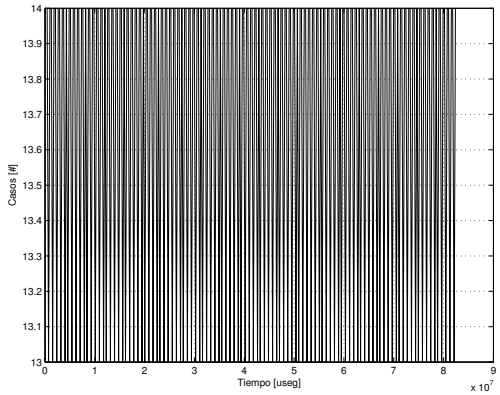



Figura 6.7: *Jitter* acumulado para tarea con periodo de 20 [ms] en *EDF*.

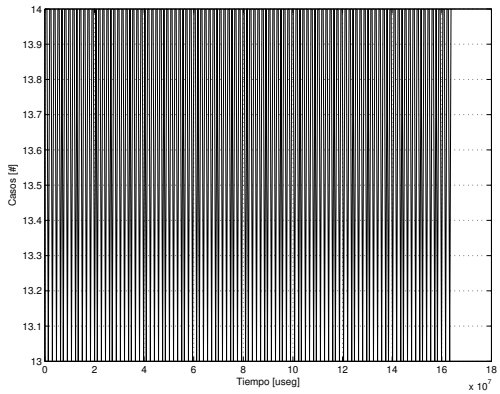



Figura 6.8: *Jitter* acumulado para tarea con periodo de 40 [ms] en *EDF*.

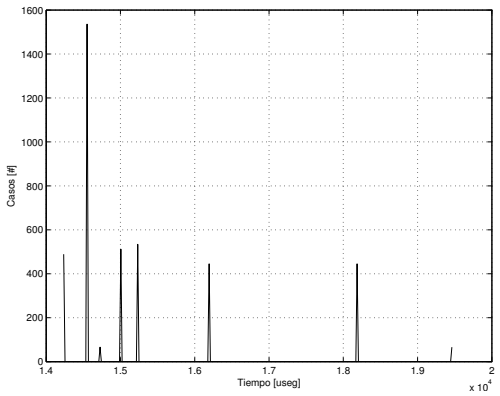



Figura 6.9: *Jitter* periódico para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas en *EDF*.

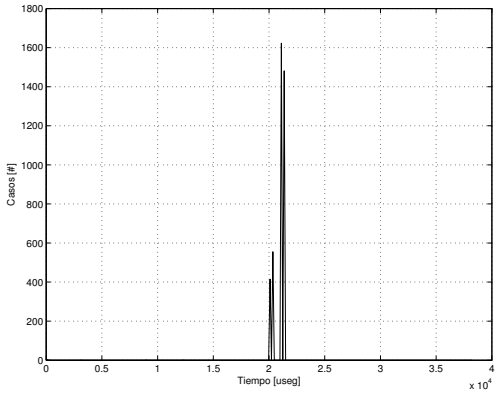



Figura 6.10: *Jitter* periódico para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas en *EDF*.

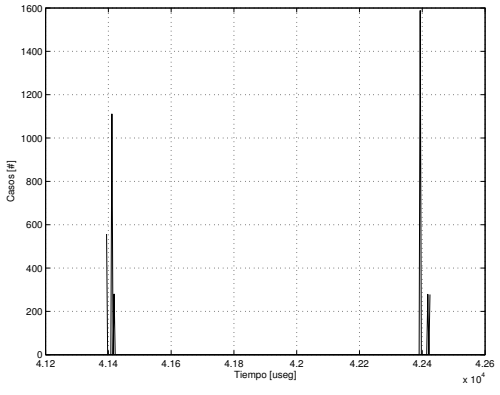



Figura 6.11: *Jitter* periódico para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas en *EDF*.

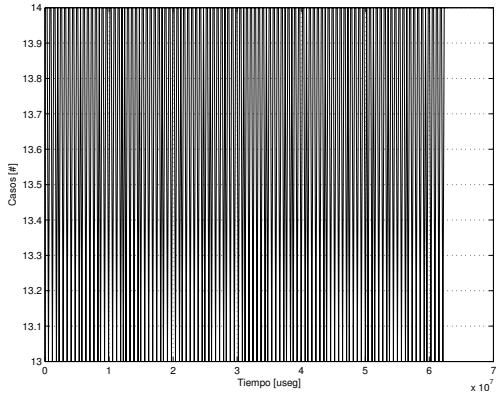



Figura 6.12: *Jitter* acumulado para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas en *EDF*.

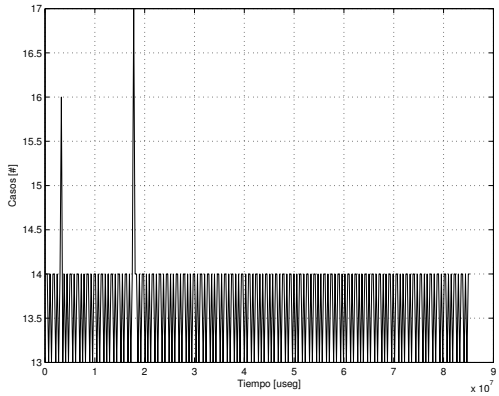



Figura 6.13: *Jitter* acumulado para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas en *EDF*.

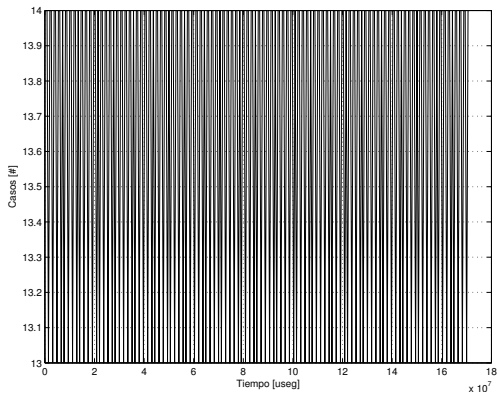



Figura 6.14: *Jitter* acumulado para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas en *EDF*.

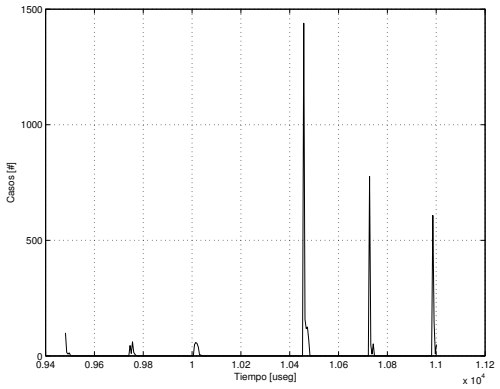



Figura 6.15: *Jitter* periódico para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

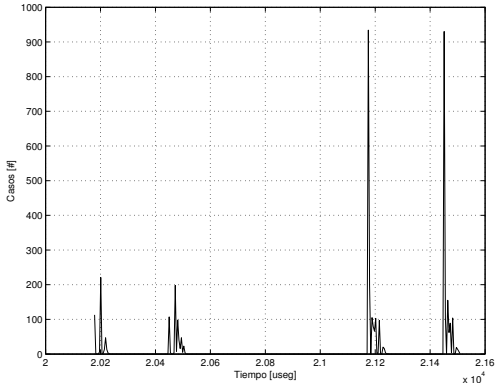



Figura 6.16: *Jitter* periódico para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

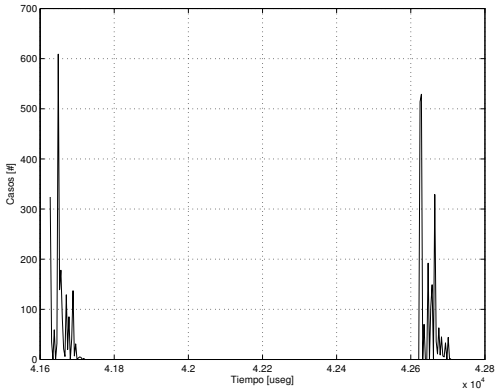



Figura 6.17: *Jitter* periódico para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

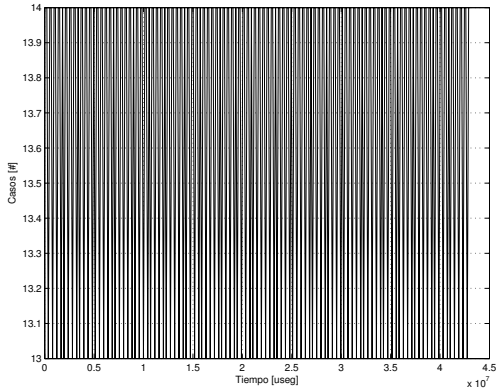



Figura 6.18: *Jitter* acumulado para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

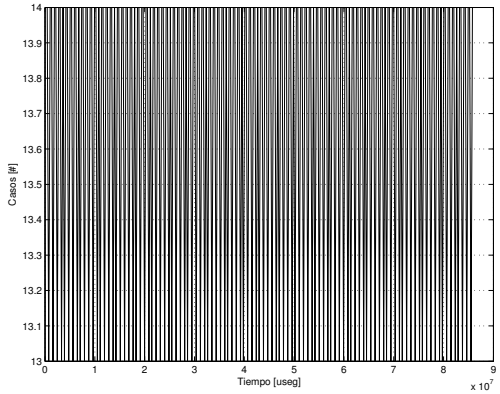



Figura 6.19: *Jitter* acumulado para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

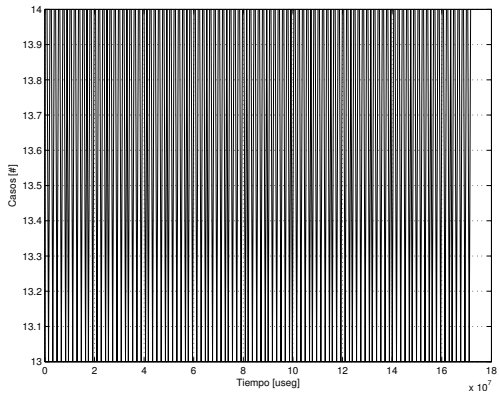



Figura 6.20: *Jitter* acumulado para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *EDF*.

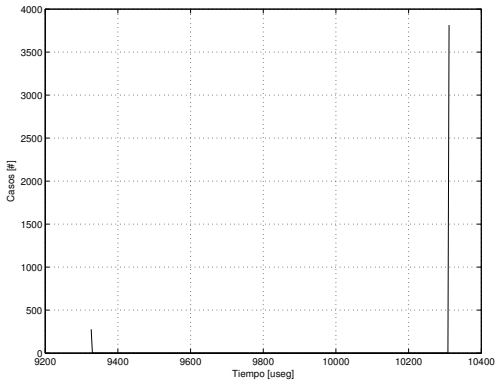



Figura 6.21: *Jitter* periódico para tarea con periodo de 10 [ms] en *RMS*.

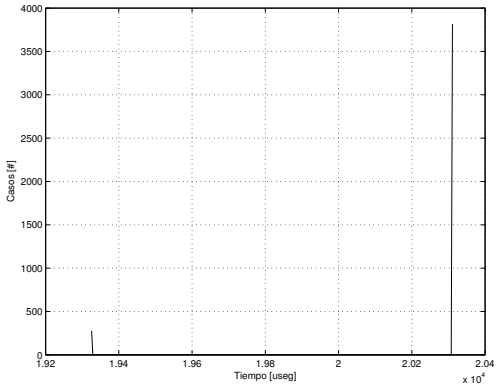



Figura 6.22: *Jitter* periódico para tarea con periodo de 20 [ms] en *RMS*.

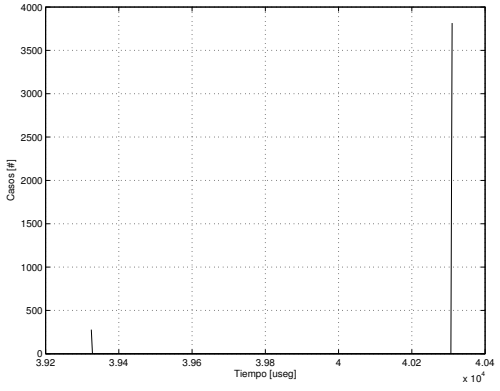



Figura 6.23: *Jitter* periódico para tarea con periodo de 40 [ms] en *RMS*.

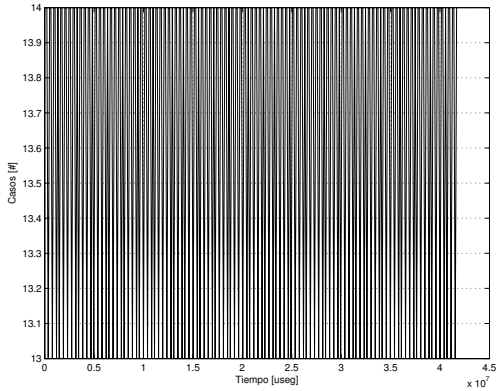



Figura 6.24: *Jitter* acumulado para tarea con periodo de 10 [ms] en *RMS*.

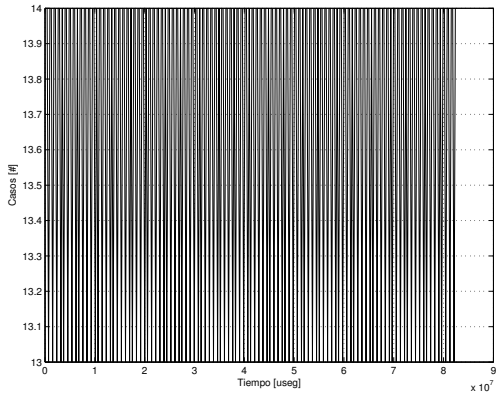



Figura 6.25: *Jitter* acumulado para tarea con periodo de 20 [ms] en *RMS*.

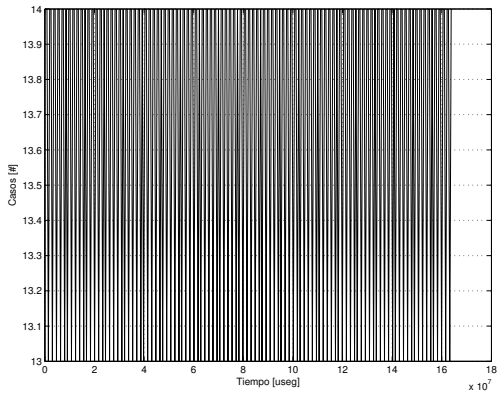



Figura 6.26: *Jitter* acumulado para tarea con periodo de 40 [ms] en *RMS*.

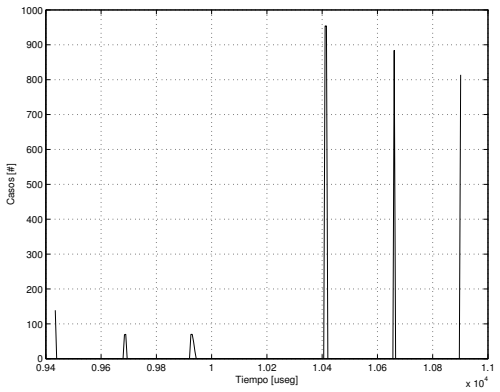



Figura 6.27: *Jitter* periódico para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas en *RMS*.

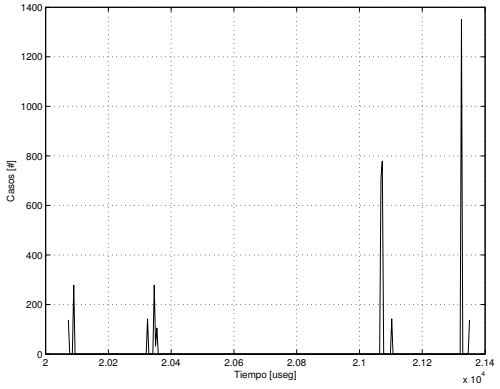



Figura 6.28: *Jitter* periódico para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas en *RMS*.

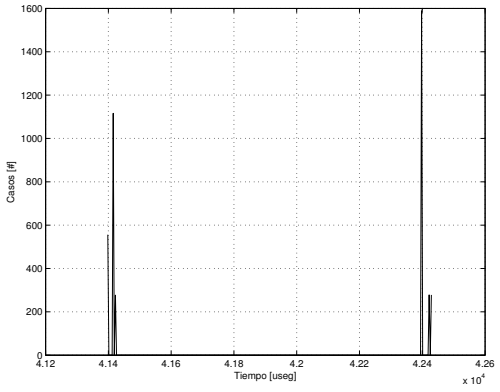



Figura 6.29: *Jitter* periódico para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas en *RMS*.

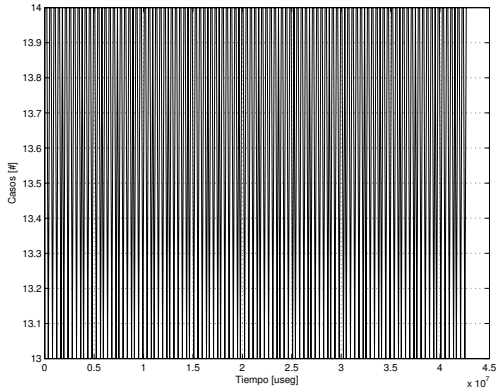



Figura 6.30: *Jitter* acumulado para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas en *RMS*.

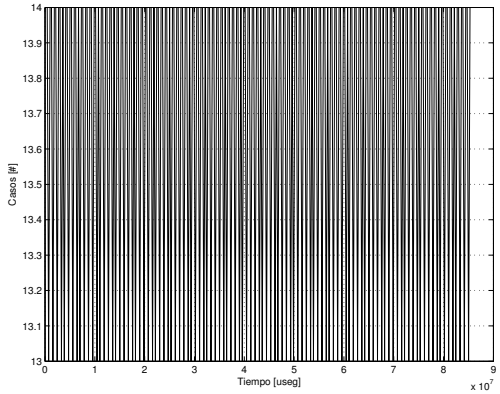



Figura 6.31: *Jitter* acumulado para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas en *RMS*.

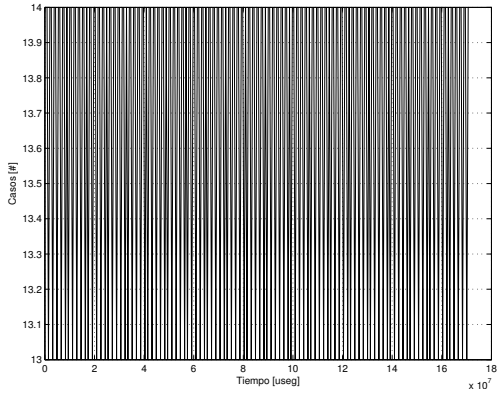



Figura 6.32: *Jitter* acumulado para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas en *RMS*.

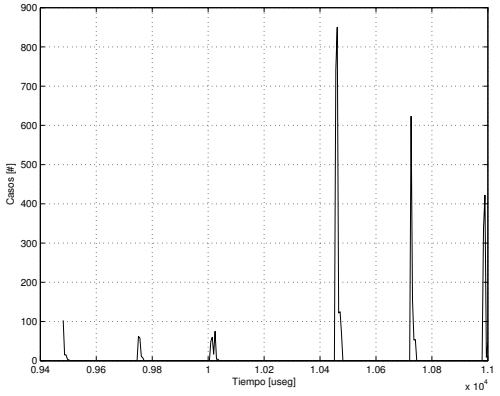



Figura 6.33: *Jitter* periódico para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

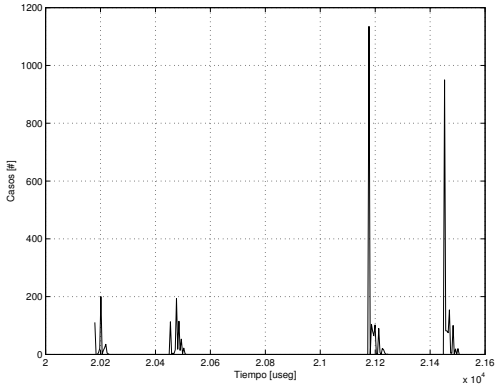



Figura 6.34: *Jitter* periódico para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

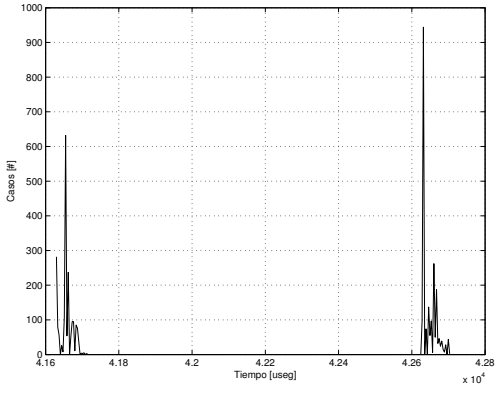



Figura 6.35: *Jitter* periódico para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

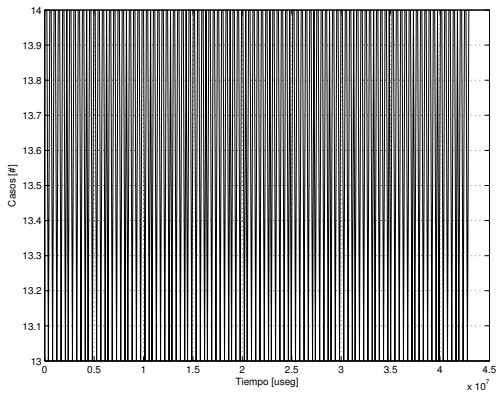



Figura 6.36: *Jitter* acumulado para tarea con periodo de 10 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

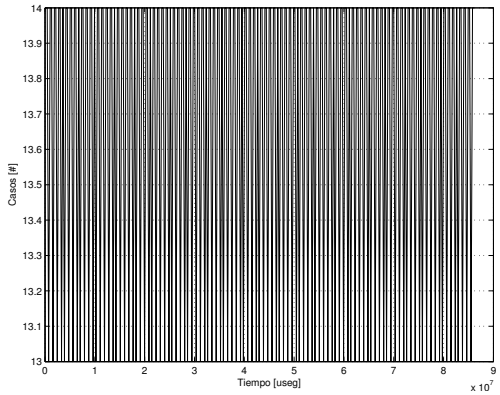



Figura 6.37: *Jitter* acumulado para tarea con periodo de 20 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

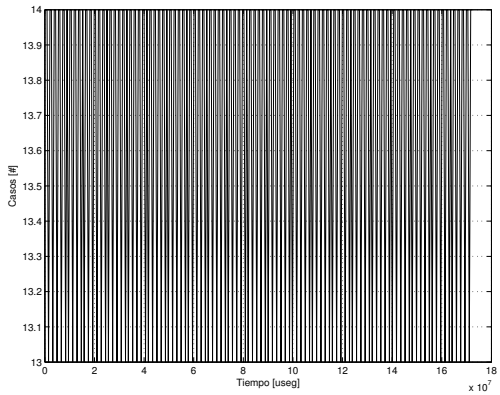



Figura 6.38: *Jitter* acumulado para tarea con periodo de 40 [ms] junto a otras 2 tareas periódicas y 1 no periódica en *RMS*.

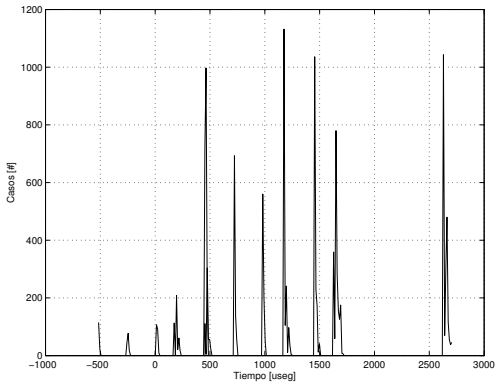



Figura 6.39: *Jitter* periódico del sistema en *EDF*.

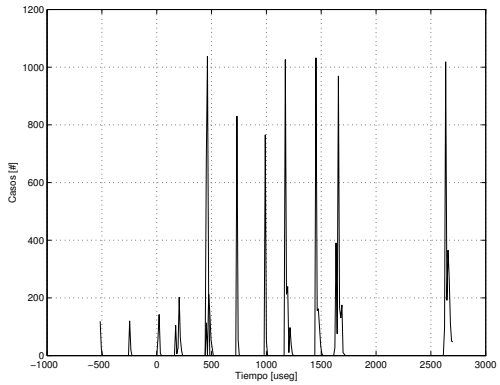



Figura 6.40: *Jitter* periódico del sistema en *RMS*.

## 6.4. Análisis de Resultados

### 6.4.1. Casos

Si se observan las Figuras 6.1 y 6.2, se puede apreciar como para el caso de *EDF*, BOS puede determinar de manera correcta la siguiente tarea en utilizar el procesador, pero para el caso de *RMS* la tercera tarea, que tiene un periodo de 60 [ms], BOS no es capaz de asignarle la cpu a tiempo, provocando el fallo del sistema. Cabe destacar que el caso de *EDF* fue examinado utilizando el *software Matlab 2008* para comprobar que más de 2000 ciclos de trabajo de las tres tareas funcionaran de manera correcta.

### 6.4.2. Jitter

Al observar las Figuras 6.3, 6.4, 6.5, 6.9, 6.10, 6.11, 6.15, 6.16, 6.17, 6.21, 6.22, 6.23, 6.27, 6.28, 6.29, 6.33, 6.34 y 6.35 se puede concluir que la diferencia entre los

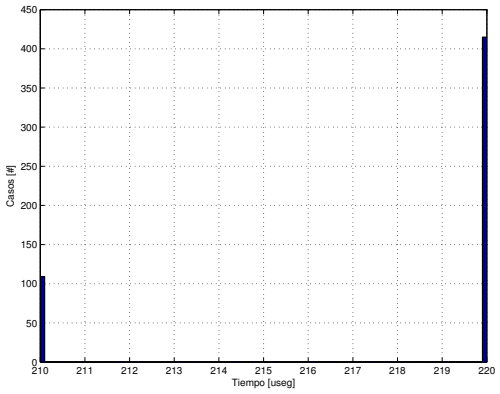



Figura 6.41: Latencia en *EDF* para *PWM* de periodo 10 [ms].

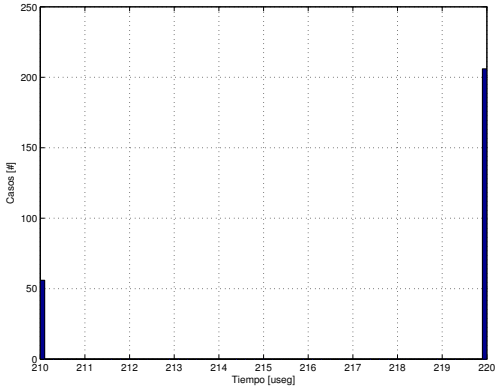



Figura 6.42: Latencia en *EDF* para *PWM* de periodo 20 [ms].

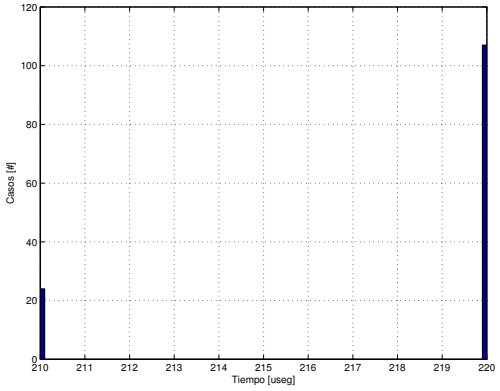



Figura 6.43: Latencia en *EDF* para *PWM* de periodo 40 [ms].

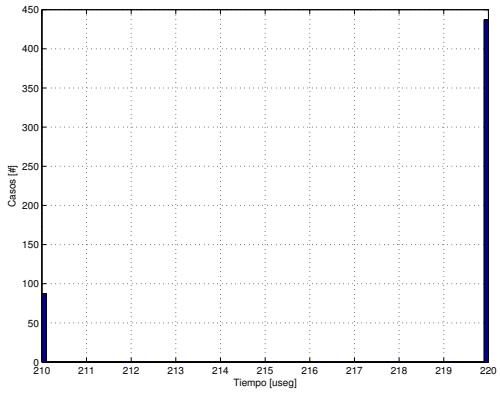



Figura 6.44: Latencia en *RMS* para *PWM* de periodo 10 [ms].

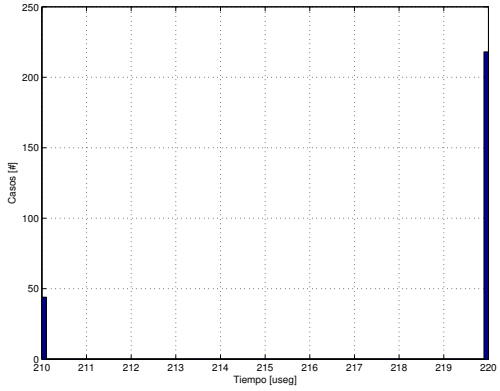



Figura 6.45: Latencia en *RMS* para *PWM* de periodo 20 [ms].

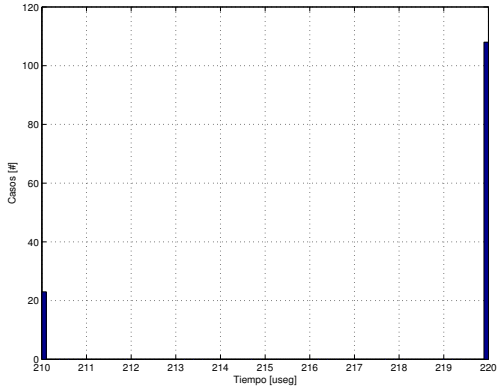



Figura 6.46: Latencia en *RMS* para *PWM* de periodo 40 [ms].

algoritmos propuestos para *EDF* y *RMS* no difieren mayormente en los resultados de *Jitter* periódico. En general BOS posee un *Jitter* periódico menor a los 700 [us] para el caso de que se esté ejecutando una sola tarea. Por otro lado si se consideran tres tareas periódicas simultaneas se puede observar un *Jitter* menor a 1 [ms] para la tarea con periodo de 10 [ms], menor a 2 [ms] para las tareas de 20 [ms] y menor a 2,5 [ms] para tareas con periodo de 40 [ms]. Por ultimo, para el caso en que existan tres tareas periódicas y una tarea no periódica corriendo en BOS, el *Jitter* periódico aumenta en 100 [us] para la tarea de periodo 10 [ms], en 200 [us] para la tarea de periodo 20 [ms] y en 300 [us] para la tarea de periodo 40 [ms] en comparación al caso de que se incluya la tarea no periódica.

Al considerar el *Jitter* acumulado los resultados no son tan alentadores, y se puede observar una tendencia de aumento en el *Jitter* acumulado a medida que el número de casos aumenta. La razón de este aumento en el *Jitter* acumulado se debe a que la solución propuesta considera tiempos relativos de las tareas y no tiempos absolutos. Se eligió esta directriz de diseño basados en que el funcionamiento original de BOS es de esta forma y con la intención de mantener la estructura de BOS. Además para poder determinar cuanto tiempo ha transcurrido, se debió detener el *timer* para leer su estado durante el proceso de *Scheduling*, provocando un aumento en el *Jitter* acumulado debido a que el *timer* no sigue corriendo mientras se determina a que tarea le corresponde la cpu.

### 6.4.3. Latencia

La latencia como se puede observar en las Figuras 6.41, 6.42, 6.43, 6.44, 6.45 y 6.46 es independiente de la utilización de *EDF* o *RMS* y también del periodo de la

*PWM* utilizada. Podemos concluir entonces que la latencia de BOS como un *RTOS* es menor a los 220 [us].

# Capítulo 7

# Conclusiones y Trabajos Futuros

BOS es una gran herramienta disponible para trabajar con la *MSP430*, muy portable y estable y que facilita enormemente la programación de programas multitarea. Además dispone de varios controladores y ejemplos de utilización, lo que facilita aun más el desarrollo de nuevos controladores y aplicaciones.

Trabajar con la *MSP430* en ambiente *Linux* cada día se hace más simple con el desarrollo de *msp-gcc*, lo que permite que BOS, luego de pequeñas modificaciones, es capaz de ser compilado y cargado en ambientes *Linux*. Además, disponer de herramientas como *Netbeans 6.8*, facilitan bastante la programación en este ambiente. El único inconveniente de este ambiente consiste en que el programador *JTAG USB* no se puede utilizar en *Linux*, lo que dificulta el desarrollo, pero no lo imposibilita.

Los algoritmos implementados para *EDF* y *RMS* demostraron gran robustez al poder seguir ejecutando las tareas programadas por periodos extensos de tiempo. Por otro lado, si bien estos algoritmos presentan un *Jitter* periódico bastante pequeño, el

*Jitter* acumulado demuestra la necesidad de mejorar el sistema para utilizar un tiempo absoluto. Aunque el *Jitter* no sea óptimo, igual presenta mejoras con respecto a BOS en su estado original, ya que en él, el *Jitter* depende del *quantum* del *Scheduler* y si una tarea requiere un periodo intermedio a este, su *Jitter* aumenta considerablemente y en los algoritmos propuestos el *Jitter* solo depende del periodo de la tarea y de cuantas otras tareas se ejecutan.

Con respecto a la latencia, se puede concluir que es bastante pequeña, cercana a los 200 [us], que si bien es mayor que la latencia que presenta la *MSP430* sin la utilización de BOS, es considerablemente menor que la latencia presente al utilizar BOS en su estado original.

## 7.1. Trabajos Futuros

En primer lugar cabe mencionar como trabajo futuro el modificar la solución propuesta para manejar tiempos absolutos en vez de tiempos relativos, de manera de disminuir considerablemente el *Jitter* acumulado que se observó. Además disponer de un tiempo absoluto provocará una disminución del *Jitter* periódico dado que no será necesario actualizar constantemente los tiempos relativos de cada tarea, evitando así recorrer la lista de tareas repetidamente. En los anexos digitales se deja disponible una rama de desarrollo que afronta este problema y aunque no está completa, sirve de guía para comprender las modificaciones necesarias para comenzar a enfrentar este problema.

Otra optimización que se puede realizar consiste en agregar una lista de tareas activas, de manera de recorrer una lista más acotada y eliminar la necesidad de evaluar el estado de éstas. Además, para el caso de *RMS* se puede crear una lista ordenada con las

tareas de tiempo real, de manera de elegir la primera tarea que se encuentre activa, sin requerir recorrer toda la lista sino solo hasta la primera tarea activa que se encuentre.

Una consideración a tener en cuenta es, en el caso de existir por ejemplo varias tareas no periódicas en conjunto con una tarea periódica con periodo muy pequeño, la tarea periódica no presentaría problemas a diferencia de las tareas no periódicas podrían sufrir de inanición. El problema se presenta dado que la lista de tareas se recorre a partir de la tarea que acaba de ceder la cpu y si se da el caso que esté en el siguiente orden: (1) tarea tiempo real, (2) tarea no periódica 1, (3) tarea no periódica 2, puede darse el caso que la tarea (3) no se llegue a ejecutar dado que mientras se ejecuta la tarea (2) BOS le cede la cpu a la tarea (1) y luego de que esta libera la cpu, BOS se la asigna a la tarea (2) nuevamente y dado que la tarea (1) tiene un periodo pequeño, ésta le quita la cpu a la tarea (2) constantemente, entonces la tarea (3) nunca alcanza a ejecutarse.

Existe algo que BOS no considera con respecto a la *MSP430* y es la capacidad de ésta de trabajar en modo bajo consumo. El *hardware* en que montaron el proyecto BOS original solo considera relojes de alta frecuencia por lo que no se hizo necesario preocuparse de esta característica. Una de las características principales de la *MSP430* es el hecho de que tiene un bajo consumo eléctrico, que viene dado por el uso de relojes de baja frecuencia y se le brinda al usuario la posibilidad de determinar con qué reloj trabajar e incluso de cambiar de reloj durante el funcionamiento según los requerimientos del usuario.

Por último, cabe mencionar que el *WDT* no se puede utilizar y ha quedado inhabilitado con los nuevos algoritmos de *Scheduler*, por lo que una posible mejora podría ser habilitar esta función.

# Bibliografía


[1] Luca Bertossi, Sistema Operativo B.lu OS, http://blubos.sourceforge.net/index.htm, 2007.

[2] Texas Instriments, MSP430, www.ti.com

[3] FreeOSEK Project, FreeOSEK, http://opensek.sourceforge.net/, 2009.

[4] Atomthreads: Open Source RTOS, Atomthreads, http://atomthreads.com/, 2010.

[5] Free Software Foundation Inc, GPLv3, http://www.gnu.org/licenses/gpl-3.0.html, 2007.

[6] OSEK VDX, Continental Automotive GmbH, http://portal.osek-vdx.org/, 1993.

[7] BSD License, The Regents of the University of California, http://www.xfree86.org/3.3.6/COPYRIGHT2.html#6, 1993.

[8] RTEMS Operating System, OAR Corporation, http://www.rtems.com/, 2010.

[9] Tomás Calderon, Introducción de un Sistema Operativo de Código Abierto para Microcontroladores de la Familia MSP430, Universidad Técnica Federico Santa María, 2009.

[10] Fundacion Wikimedia, Wikipedia, http://es.wikipedia.org/wiki/Jitter

[11] Fundacion Wikimedia, Wikipedia, http://es.wikipedia.org/wiki/Latencia

[12] Fengxiang Zhang - Alan Burns,Schedulability Analysis for Real-Time Systems, IEEE, 2009.

[13] Dong Shuzhen, Xu Qiwen and Zhan Naijun, A Formal Proof of the Rate Monotonic Scheduler, The United Nations University, 2000.