INSTITUTO POLITECNICO NACIONAL
COORDINACIÓN GENERAL DE POSGRADO E INVESTIGACIÓN

CARTA SESION DE DERECHOS

En la ciudad de México, Distrito Federal, el día **17** del mes de **Enero** del año **2006**, el (la) que suscribe **Manuel de Jesús Nango Méndez** Alumno del programa de **Maestria en Ciencias con especialidad en Ingeniería Eléctrica** con número de registro **B980859**, adscrito a la Sección de Estudios de Posgrado e Investigación de la Esime Unidad Zacatenco, manifiesta que es autor(a) intelectual del presente trabajo de Tesis Bajo la Dirección del **Dr. Raúl Ángel Cortés Mateos** y cede los derechos del trabajo Intitulado : **Diseño e Implementación de un Sistema Operativo de Tiempo Real para el DSP Microcontrolador TMS320F240**, al Instituto Politécnico Nacional para su difusión, con fines académicos y de Investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director de trabajo. Este puede ser obtenido escribiendo a la siguiente dirección: **mnango@hotmail.com** Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Manuel de Jesús Nango Méndez

Nombre y Firma

SIP-14

INSTITUTO POLITECNICO NACIONAL
SECRETARIA DE INVESTIGACION Y POSGRADO

ACTA DE REVISION DE TESIS

En la Ciudad de México, D. F. siendo las 14:00 horas del día 8 del mes de diciembre del 2005 se reunieron los miembros de la Comisión Revisora de Tesis designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de la E. S. I. M. E. para examinar la tesis de grado titulada:

"DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA OPERATIVO DE TIEMPO REAL PARA EL DSP MICROCONTROLADOR TMS320F240"

Presentada por el alumno:

NANGO	MÉNDEZ	MANUEL DE JESÚS
Apellido paterno	Apellido materno	Nombre(s)

Con registro: B 9 8 0 8 5 9

Aspirante al grado de:

MAESTRO EN CIENCIAS

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***SU APROBACION DE LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISION REVISORA

Director de tesis
Dr. Raúl Ángel Cortés Mateos

Presidente
Dr. David Romero Romero

Segundo Vocal
M. en C. Domitilo Libreros

Tercer Vocal
Dr. Antonio Osorio Cordero

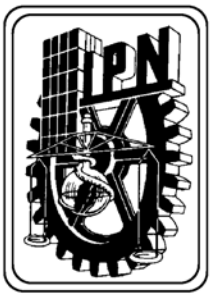
Secretario
Dr. Jaime Robles García

Suplente
M. en C. Jesús Reyes García

EL PRESIDENTE DEL COLEGIO

DR. JAIME ROBLES GARCIA

SECCION DE ESTUDIOS DE POSGRADO E INVESTIGACION



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE INGENIERÍA MECÁNICA Y ELÉCTRICA

SECCIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

Diseño e Implementación de un Sistema Operativo de Tiempo Real para el DSP-Microcontrolador TMS320F240

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN CIENCIAS CON ESPECIALIDAD EN INGENIERÍA ELÉCTRICA

PRESENTA

Manuel de Jesús Nango Méndez

MÉXICO D. F. **2005**

RESUMEN

En el presente trabajo, se presenta el diseño e implementación de un sistema operativo en tiempo real para el DSP-microcontrolador fabricado por Texas Instruments TMS320F240 basado en el núcleo de programación llamado μCOS-II creado por Jean J. Labrosse. Se desarrollan los códigos en lenguaje C y ensamblador para que funcione el sistema operativo, del cual no se encontró referencia hasta el momento. Se muestran algunos detalles de la arquitectura del TMS320F240, del compilador C, lenguaje ensamblador y μCOS-II necesarios para poder desarrollar este trabajo.

Se demuestra el funcionamiento del sistema operativo implementado, con el desarrollo de dos aplicaciones. La primera únicamente prende LED's en una secuencia de conteo binario con un retardo de tiempo, usando el servicio de retardo que proporciona el sistema operativo. La otra es una aplicación a ingeniería eléctrica, en la que se simula la sincronización de un generador al bus infinito, en esta aplicación se empleó además del servicio de retardo, el servicio de semáforo del sistema operativo. También, se realiza el análisis de la secuencia de ejecución de las dos aplicaciones con un diagrama de tiempo, para entender la filosofía de programación que se debe tener al emplear un sistema operativo en tiempo real para el desarrollo de aplicaciones.

ABSTRACT

Design and implementation of a real time operating system for the DSP-microcontrolller manufactured by Texas Instruments TMS320F240 based on the programming nucleus μCOS-II created by Jean J. Labrosse is presented. C language and assembler codes are developed to achieve that the operating system works, which there are not reference yet. Architecture of the TMS320F240, C compiler, assembly language and μCOS-II details are shown to be able to develop this work.

Operation of the implemented operating system is demonstrated, with the development of two applications. The first one, LEDs are lighted in a sequence of binary counter with a time delay, using the delay service that provides the operating system. The other one, is an application to electric engineering, which simulates generator to the infinite bus synchronization, in this application, it was used delay and semaphore service of the operating system. Also, the execution sequence analysis of this applications are done with a time's diagram to understand the programming philosophy that should be had when using a real time operating system in an application.

INDICE

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE CÓDIGOS

CAPÍTULO 1

INTRODUCCIÓN

Actualmente hay cientos de sistemas operativos de tiempo real en venta, en el apéndice E se recopiló varios de ellos. Estos están disponibles para microprocesadores de 8, 16, 32 e incluso 64 bits [2]. Algunos de ellos son paquetes de desarrollo completos que incluyen un núcleo de programación, un manejador de entrada-salida, sistemas de ventaneo, un sistema de archivo, soporte de red, bibliotecas de interfaz de lenguaje, depuradores y soporte de diversos compiladores. El costo de un sistema operativo en tiempo real varia entre 100 hasta 50,000 dólares [2].

Los sistemas operativos en tiempo real no soportan un procesador específico, por lo que se debe "rediseñar" el sistema operativo para poder ser usado en el procesador que se necesite.

La presente tesis trata el diseño de un sistema operativo de tiempo real específico para el microcontrolador-procesador de señales digitales TMS320F240 fabricado por Texas Instruments, basado en el sistema operativo de tiempo real μCOS-II. Este sistema operativo actualmente es comercial, pero inicialmente era un sistema operativo con fines didácticos y se encuentra aplicado en productos comerciales [3] tales como: teléfonos celulares (INFEA, usando procesadores Hitachi), sistemas de radio comunicación (Microwave Radio Communications, sobre procesadores MC683xx), terminal de negocios (Monitor Business

Machines), etc. Hasta el momento, no se encontró referencia en la cual se haya hecho el presente trabajo, en [3] están publicados los trabajos realizados en diversos procesadores de varios fabricantes. Los procesadores fabricados por Texas Instruments en loscuales se ha trabajado son los siguientes: *MSP-430, TMS320-C31, TMS320-C40, TMS320-C2400, TMS320-C5409, TMS320-C6201, TMS320-C6711, TMS470 (ARM7), TMS320VC5470, TMS320C5509/5510, TMS320C6205 y TMS320C6414.*

1.1 OBJETIVO

Desarrollar e implementar un sistema operativo en tiempo real para el microcontrolador-procesador de señales digitales TMS320F240 mediante las herramientas de programación (compilador C, depurador, ensamblador y el módulo de evaluación) que proporciona Texas Instruments. Y comprobar el funcionamiento del sistema operativo desarrollando una aplicación de tiempo real.

1.2 JUSTIFICACIÓN

Proporcionar una herramienta para el desarrollo de aplicaciones en tiempo real en los sistemas eléctricos. Especialmente, donde la implementación se complique al involucrar además de un sistema de control complejo, algún protocolo de comunicación y una interfaz gráfica para el usuario.

1.3 ESTRUCTURA DE LA TESIS

Capítulo 1, Introducción. Explica la forma "tradicional" de programar un procesador que forma parte de un sistema de control digital y la alternativa de programar el

procesador usando un sistema operativo de tiempo real. También, se describe el estado del arte, el objetivo, la justificación y la estructura de la tesis.

Capítulo 2, Conceptos Sobre Sistemas Operativos de Tiempo Real. Presenta conceptos básicos sobre sistemas operativos de tiempo real (multitareas, núcleo de programación, planificador, servicios de los núcleos de programación) y características del núcleo de programación μCOS-II (sistema de tareas, planificador, manejo de interrupciones, sistema de tiempo, servicios y manejo básico de tarea).

Capítulo 3, Diseño e Implementación del Sistema Operativo Basado en μC0S-II para el TMS320F240. Describe las características, el diseño y la implementación de los archivos necesarios para que el núcleo de programación μCOS-II pueda ser usado en el TMS320F240.

Capítulo 4, Prueba de Funcionamiento del Sistema Operativo. Discute el desarrollo de dos aplicaciones con las que se comprueba el funcionamiento del sistema operativo. Una prende LED's en una secuencia de conteo binario con un retardo de tiempo y la otra es una aplicación a ingeniería eléctrica, en la que se simula la sincronización de un generador al bus infinito.

Capítulo 5, Conclusiones, Aportaciones y trabajos futuros. Expone cada punto del nombre del capítulo.

CAPÍTULO 2

CONCEPTOS SOBRE SISTEMAS OPERATIVOS DE TIEMPO REAL

2.1 SISTEMAS EN TIEMPO REAL

Un Sistema en Tiempo Real (STR) es aquel sistema donde el tiempo en que se produce su salida es significante. Donde generalmente la entrada corresponde a algún instante τ del mundo físico y la salida tiene relación con ese mismo instante más un tiempo adicional: $\tau+\varepsilon$; el retraso transcurrido entre la entrada y la salida ε, debe ser lo suficientemente pequeño, encontrándose acotado dentro de un intervalo $(0,\delta]$, con $\varepsilon \in (0,\delta]$ y δ un plazo de tiempo; para considerarse una respuesta puntual [1].

Un sistema en tiempo real común son los sistema de control digital como el mostrado en la figura 2.1. La computadora digital tiene básicamente las siguientes funciones:

- Leer la señal de entrada.
- Procesar la señal de entrada (comparar la señal de entrada con el valor de referencia y aplicar el algoritmo de control).
- Dar la señal de control.

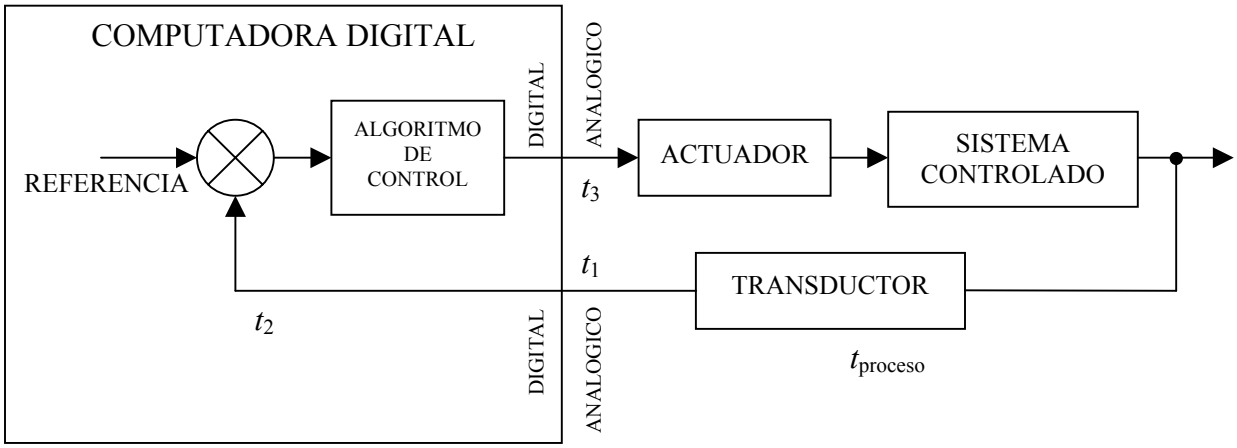


Figura 2.1 Diagrama de bloques de un sistema de control digital.

Estas funciones se llevan a cabo mediante la ejecución de un programa (serie de instrucciones) asociada a la computadora digital. Al programa se le considera un sistema de tiempo real debido a su dependencia del tiempo, es decir que el tiempo que le toma a la computadora digital en realizar sus funciones por medio del programa, debe ser menor ó igual al tiempo del proceso, tiempo en el cual el sistema controlado responde adecuadamente a la señal de control. Y solo si se cumple esta condición se asegura el funcionamiento del sistema. En la figura 1.1.

$$t_1 + t_2 + t_3 \leq t_{proceso} \qquad (1)$$

donde:

t_1, es el tiempo que tarda la conversión analógica digital (lectura de la señal).

t_2, la duración del procesamiento de la señal de entrada (aplicación del algoritmo).

t_3, el lapso de conversión digital analógica (dar la señal de control)

$t_{proceso}$, el tiempo del proceso.

2.2 PROGRAMACIÓN DE LOS SISTEMAS EN TIEMPO REAL

El diseño "tradicional" de un sistema de tiempo real como el mostrado en la figura 2.1, es la uniprogramación, que consiste en un ciclo infinito que llama a módulos de aplicación que se ejecutan secuencialmente para realizar las operaciones deseadas y rutinas de servicios de interrupción (ISR) que se encargan de los eventos asíncronos (eventos que se desconoce el momento en que ocurren); la uniprogramación se muestra en la figura 2.2, donde cada rectángulo representa código ejecutable que está en la memoria de la computadora digital. Este esquema de diseño resuelve sistemas de tiempo real simples, es decir que el sistema se puede ver como un módulo completo. Sin embargo, con la uniprogramación se dificulta el diseño cuando se necesita desarrollar sistemas de control que involucren además de un algoritmo de control complejo, una interfaz gráfica con el usuario y algún protocolo de comunicación.

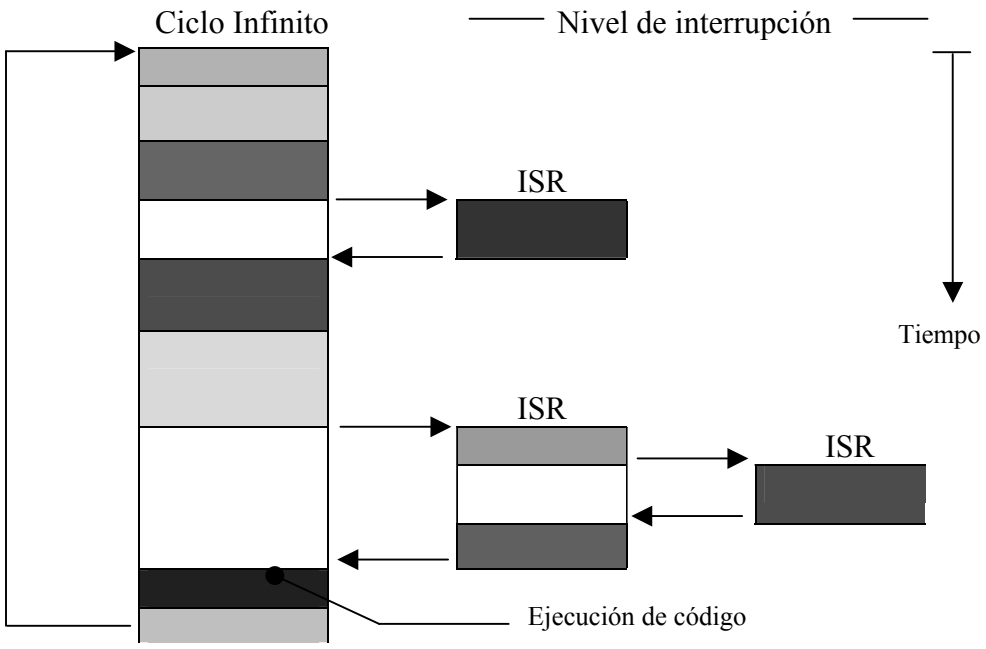


Figura 2.2 Diseño "tradicional" de un sistema de tiempo real.

Una herramienta para facilitar el diseño de sistemas de tiempo real complejos y aprovechar al máximo un procesador digital es el sistema operativo de tiempo real ó núcleo multitarea de tiempo real. Se trata de un programa base que divide las aplicaciones en múltiples módulos independientes llamadas tareas ó procesos y administra el tiempo de ejecución de estas de la manera más eficiente posible. La figura 1.3 muestra un sistema de tiempo real empleando el sistema operativo, donde el círculo de mayor tamaño representa el programa base (sistema operativo) que decide qué módulo de programa se va a ejecutar ó continuar su ejecución, ya sea la tarea 1, 2 ó 3; las flechas bidireccionales indican la interacción que existe entre los códigos, cuando una tarea deja de ejecutarse parte del código del sistema operativo se ejecuta para decidir la tarea que se ejecutará a continuación.

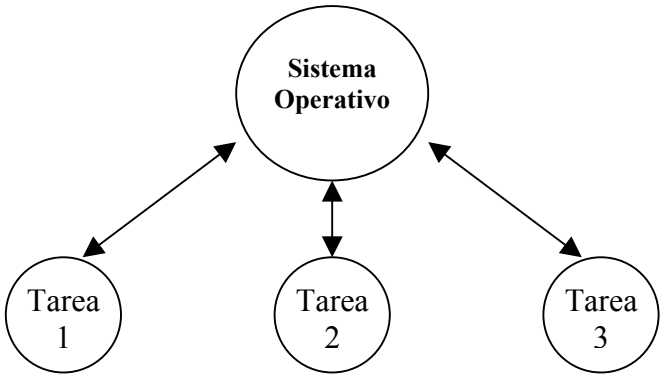


Figura 2.3 Sistema de tiempo real usando un sistema operativo.

2.3 MULTITAREAS

Este concepto se refiere al proceso de compartir la unidad de procesamiento central (CPU) de una computadora digital entre módulos independientes de programas llamadas tareas; en la computadora se encuentran activas diversas tareas y un programa control decide cuándo va a ejecutarse alguna de ellas y con qué recursos (dispositivos de entrada y salida ó variables). Este tipo de sistema da como resultado un diseño modular de las aplicaciones.

Tarea se refiere a un programa simple que resuelve parte de una aplicación de tiempo real y se diseña como un módulo independiente. El código de la tarea se escribe como una función que es de ciclo infinito. A cada tarea le corresponde además de un área de memoria del código de la parte de aplicación que realice, un área de memoria para el bloque de control de tarea (TCB) y la pila de la tarea (figura 2.1); el bloque de control de tarea es una estructura de datos que contiene información a cerca de la tarea que le corresponde y que usa el sistema operativo para llevar el control y estado de todas las tareas, y de la pila que se le haya asignado a la tarea. La pila de cada tarea se usa como lugar de almacenamiento individual de cada tarea.

Compartir la CPU significa poder cambiar la ejecución de una tarea a otra, sin que se altere el objetivo de cada tarea. El cambio de ejecución de una tarea a otra se efectúa realizando un cambio de contexto es decir, se guardan los contenidos de los registros internos (contexto) de la CPU producidos por la ejecución de la tarea actual y se restauran los contenidos de los registros internos de la CPU de la tarea que se quiere iniciar ó reanudar su ejecución (figura 2.1). Uno de los registros internos más importantes que hacen posible el cambio de ejecución de una tarea a otra es el contador de programa, que se encarga de indicar la instrucción que se ejecutará a continuación; al guardar el contenido de este registro cuando se va a realizar un cambio de ejecución, se puede reanudar la ejecución de la tarea que dejó de ejecutarse (suspendida) al restaurar el contenido del contador de programa posteriormente.

El contexto de la CPU se guarda normalmente en la pila de la tarea que está ejecutándose actualmente y posteriormente se restaura de la misma.

Cuando se efectúa un cambio de contexto una tarea deja el control de la CPU y otra lo toma. Por lo tanto, estas dos tareas involucradas puede decirse que realizan un cambio de estado; una cambia de un estado de no-ejecución, a un estado de ejecución y la otra cambia de un estado de ejecución, a un estado de no-ejecución.

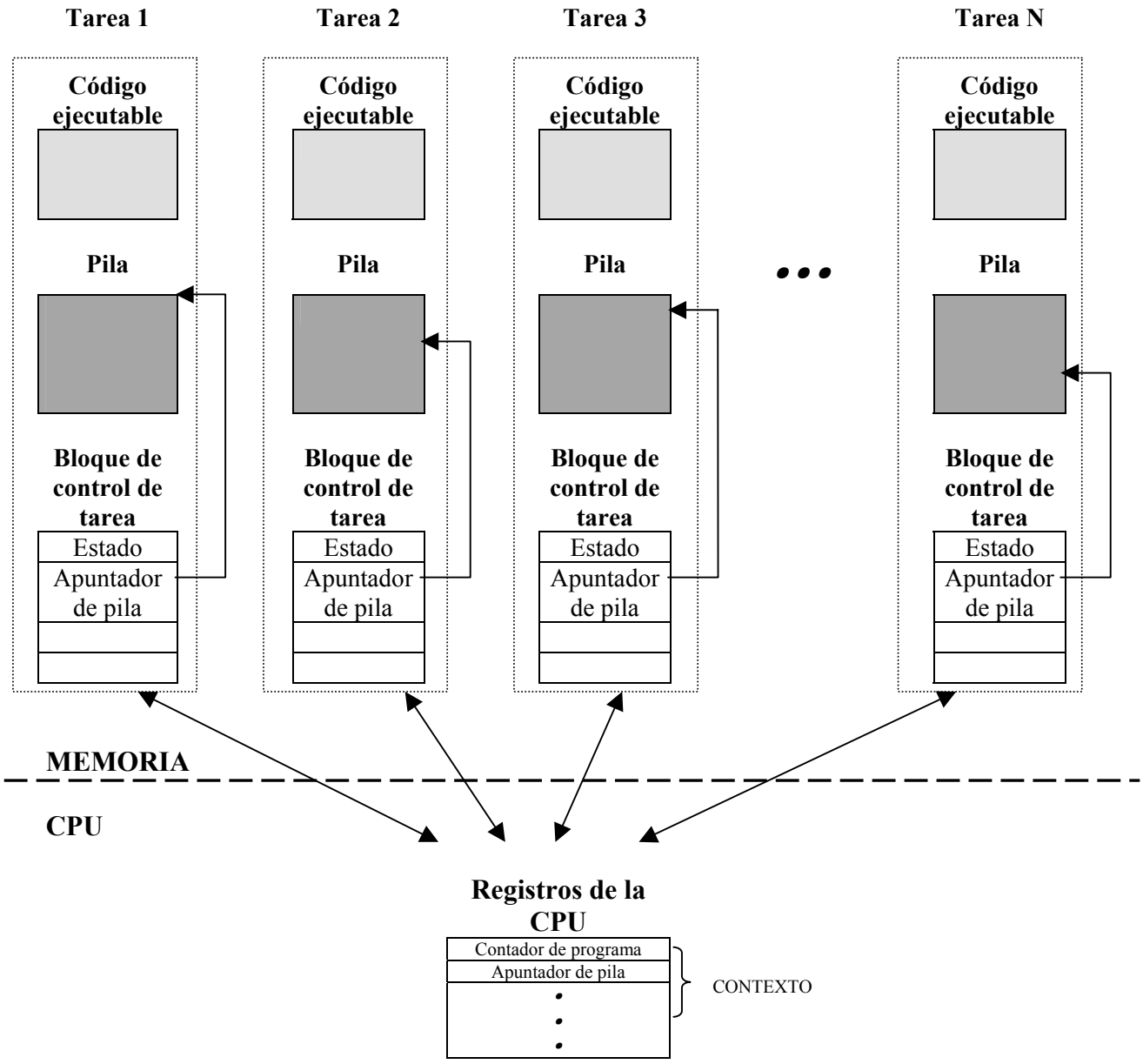


Figura 2.4 Tareas múltiples.

Cuando una tarea se encuentra en un estado de no-ejecución, se refiere a que la tarea puede estar en uno de los estados en el cual no se tiene el control de la CPU, como son BLOQUEADA (cuando una tarea espera un evento) ó PREPARADA (cuando una tarea está lista para obtener el control de la CPU). El estado de EJECUCIÓN se da cuando la tarea tiene el control de la CPU. La decisión la toma una parte del sistema multitarea llamado kernel y específicamente el planificador. En la figura 2.2 se puede observar los

estados en los que puede estar una tarea en un sistema operativo, en esta figura, las flechas indican los estados a los que se puede pasar, por ejemplo una tarea en el estado preparada para ejecución puede pasar al estado de ejecución.

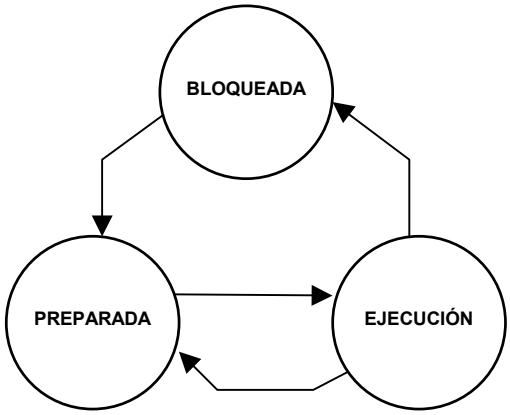


Figura 2.5 Estados en los que puede encontrarse una tarea en un sistema operativo.

2.4 NÚCLEO DE PROGRAMACIÓN

En un sistema multitareas, como lo es el sistema operativo en tiempo real, la parte de código encargado del manejo de las tareas y la comunicación entre tareas se conoce como núcleo (kernel) de programación. Es un programa base sobre el que se programan las tareas para el diseño de las aplicaciones. El núcleo proporciona servicios que ayudan a mejorar el desempeño de la CPU. El servicio fundamental del núcleo es el cambio de contexto.

Al usar un núcleo de tiempo real se facilita el diseño sistemas, pero se necesita usar memoria extra; ROM para incluir el kernel y los servicios que proporcione, y RAM para los bloques de control de tarea y la pila de cada tarea. Por lo que algunos

microcontroladores de pastilla simple no se pueden usar para este tipo de diseño ya que estos cuentan con poca RAM.

2.5 PLANIFICADOR

El que toma la decisión de qué tarea se va a ejecutar en un sistema multitarea es el planificador ó despachador, el cual forma parte del kernel. El planificador básicamente decide qué tarea que se encuentra preparada para ejecutarse va a pasar al estado de ejecución. Muchos de los kernel de tiempo real están basados en la prioridad que se le asigna a cada tarea dependiendo su importancia [4]. Los kernel basados en prioridad dan el control de la CPU a la tarea de prioridad más alta que esté preparada para ejecución. Sin embargo, dependiendo del momento en el que se le dé el control de la CPU a la tarea de prioridad más alta los kernel pueden ser de dos tipos:

- El kernel que siempre está ejecutando la tarea de prioridad más alta. Este tipo de kernel siempre le da el control de la CPU a la tarea de prioridad más alta que está preparada para ejecución. Cuando un evento hace que una tarea de prioridad mayor a la que está en estado de ejecución pase al estado preparada para ejecución, en ese momento el kernel le da el control a la tarea de mayor prioridad, dejando suspendida la que se estaba ejecutando. La tarea suspendida reanudará su ejecución cuando la tarea de mayor prioridad deje el control de la CPU. Este tipo de kernel se muestra en la figura 2.3.

- El kernel que no siempre está ejecutando la tarea de prioridad más alta. En este tipo de kernel en ocasiones no se está ejecutando la tarea que tenga mayor prioridad que está preparada para ejecución. Cuando un evento hace que una tarea prioridad mayor a la que está en ejecución pase al estado preparada para ejecución, al terminar el evento la tarea de menor prioridad continua ejecutándose, hasta que esta decide dejar el control de la CPU y en este momento el kernel decide ejecutar la tarea de mayor prioridad. En la figura 2.4 se ilustra este kernel.

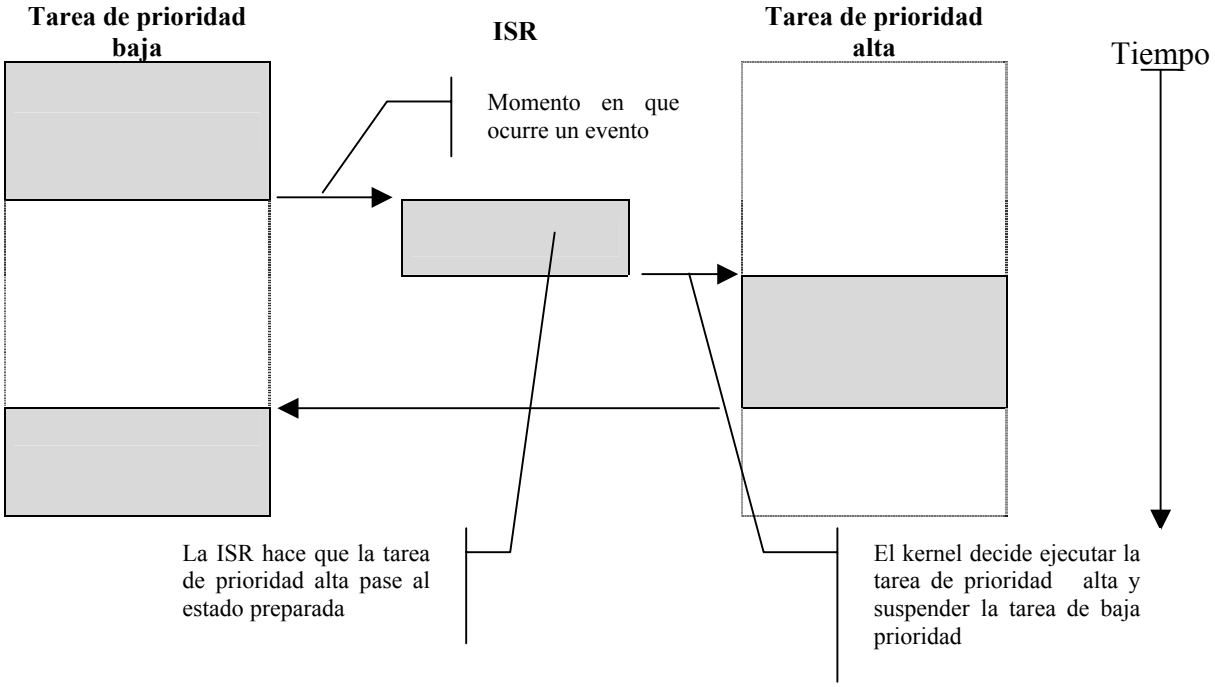


Figura 2.6 Kernel que siempre ejecuta la tarea de prioridad más alta.

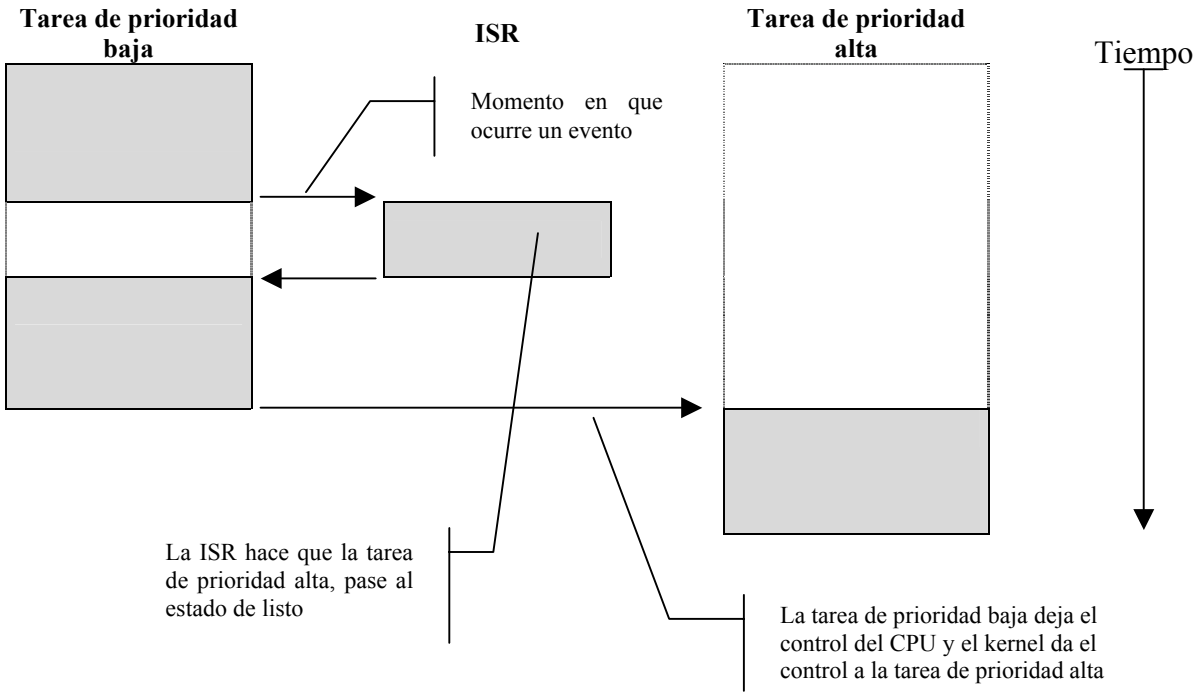


Figura 2.7 Kernel que no siempre ejecuta la tarea de prioridad más alta.

2.6 SERVICIOS DE LOS NÚCLEOS DE PROGRAMACIÓN

Los núcleos de programación proveen un tipo de biblioteca de funciones conocidas como servicios, que el diseñador de aplicaciones en tiempo real usa principalmente para:

- Llevar una base de tiempo común en el sistema de tiempo real.
- Sincronizar la ejecución de las tareas.
- Controlar el acceso a las unidades de entrada y/ó salida.
- Enviar y recibir mensajes (compartir datos) entre tareas.
- Manejo de memoria.

Dentro de los servicios que ofrecen los núcleos de programación se encuentran:

- El semáforo. Es un mecanismo protocolario utilizado para las funciones de controlar el acceso de unidades de entrada y/ó salida que estén usando dos ó más tareas (exclusión mutua), señalar cuando ocurre un evento ó sincronizar la ejecución de dos tareas. El semáforo consiste básicamente en una llave que adquiere el código para poder continuar su ejecución. Si el semáforo está en uso, la tarea solicitante quedará suspendida hasta que el semáforo (la llave) sea liberado por su poseedor actual (otra tarea).

- Relativos al tiempo. Este tipo de servicio permite a alguna tarea poder retardarse asimisma un número entero del sistema de base de tiempo. Un sistema de base de tiempo típicamente ocurre cada 10 a 200 ms, dependiendo de la aplicación. La base de tiempo también se usa para dar límites de tiempo de espera de algún evento.

- Buzón de mensaje. Este servicio se usa para enviar y recibir mensajes de una tarea a otra ó de una ISR a una tarea. El mensaje típicamente es un apuntador de tamaño variable. Los códigos que van a recibir (esperar) y enviar (colocar) deben coincidir con el mensaje, es decir hacia donde apuntan. Cuando un código necesita enviar un

mensaje, este lo deposita al buzón (por medio de un servicio del kernel) y cuando se va a recibir un mensaje, el código receptor del mensaje checa el buzón si hay un mensaje lo adquiere y sino espera por él.

- Cola de mensaje. Se utiliza para enviar más de un mensaje a una tarea. Este servicio puede ser considerado como un arreglo de buzones.

- Manejadores de memoria. Estos se emplean en los sistemas para poder optimizar el uso de ella, ya que en este tipo de sistemas está limitada la cantidad de memoria.

2.7 EL NÚCLEO DE PROGRAMACIÓN DE TIEMPO REAL μCOS-II

μCOS-II es un núcleo de programación de tiempo real creado por Jean J. Labrosse. Es una versión actualizada de μCOS. μCOS-II se entiende como sistema operativo microcontrolador versión 2. μCOS-II fue creado especialmente para desarrollar sistemas de computación que están integradas a un sistema mayor, como son los dispositivos usados para controlar, supervisar o asistir la operación de un equipo, estos sistemas son o incluyen computadoras ó microprocesadores digitales.

El código de μCOS-II está escrito, como la mayor parte de los sistemas operativos de esta naturaleza, en ANSI C para poder ser usado en procesadores que tengan entre sus herramientas de programación un compilador C compatible con ANSI C.

Las aplicaciones que se diseñan usando μCOS-II son divididas en módulos independientes llamados tareas (funciones de C que crea el usuario), que resuelven parte de la aplicación. A cada tarea se le asigna una prioridad única según su importancia en la aplicación. μCOS-II es un kernel que siempre ejecutará la tarea de mayor prioridad de las que se encuentren preparadas para ejecución. μCOS-II cuenta con los servicios más

comunes como son base de tiempo, buzón y cola de mensajes y semáforo, control del uso de memoria, sincronizar la ejecución de las tareas, controlar el acceso a las unidades de entrada y/ó salida y la comunicación entre tareas.

2.5.1 SISTEMA DE TAREAS EN μCOS-II

μCOS-II puede manejar hasta 64 tareas, de las cuales 2 están reservadas para el sistema y las otras 62 son las disponibles para el usuario. Las reservadas para el sistema tienen las prioridades más bajas, es decir *OS_LOWEST_PRIO* y *OS_LOWEST_PRIO-1* (*siendo OS_LOWEST_PRIO* el valor de prioridad más alto definido por el usuario, el valor de la prioridad mientras mayor sea menor es su prioridad). Las tareas pueden tener un valor de prioridad entre 0 y *OS_LOWEST_PRIO-2*.

En la figura 2.5 se muestra los estados en que puede estar una tarea en μCOS-II, los cuales son: **espera**, **dormida**, **preparada**, **interrumpida** (ISR) y en **ejecución**; los estados de espera, dormida, preparada e interrumpida son estados de no ejecución.

Una tarea está en **espera** cuando deja de ejecutarse por cierto tiempo decidido por el usuario (retardo) y/ó por la ocurrencia de un evento.

Se dice que una tarea se encuentra **dormida** cuando no está disponible para μCOS-II, pero ésta reside en memoria.

Una tarea **preparada** es aquella que no está en ejecución por ser de menor prioridad a la que se está ejecutando actualmente.

La tarea en **ejecución** es la tarea de prioridad más alta que está **preparada** para correr.

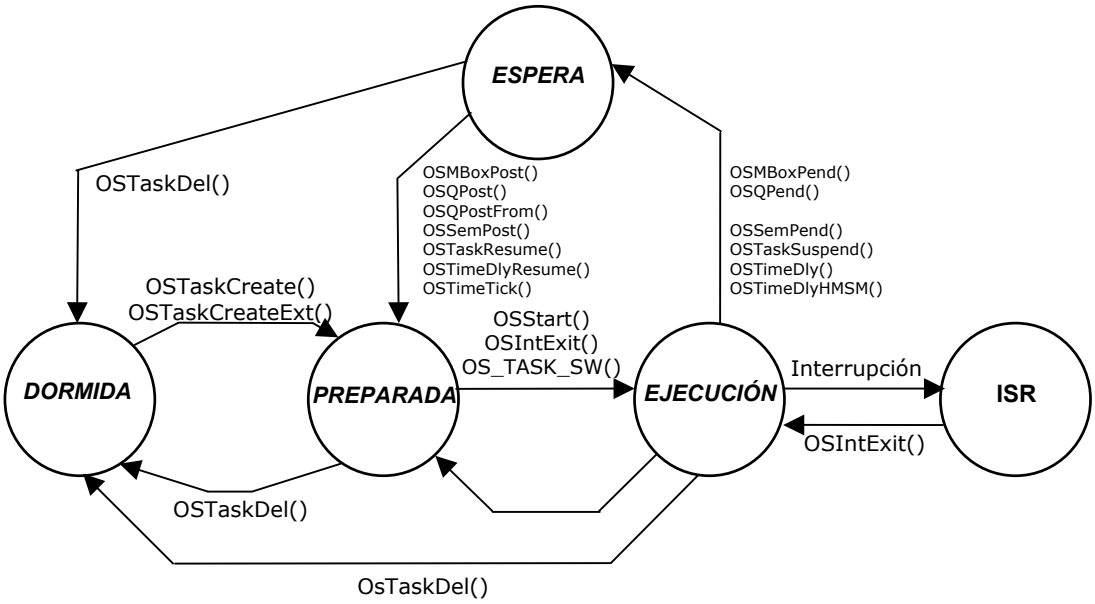


Figura 2.8 Estados de las tareas y funciones con las que se cambian de estado.

Cuando una tarea está en ejecución y ocurre una interrupción, la tarea se convierte en tarea **interrumpida**.

Una tarea pasa al estado de **espera** a partir de estar en **ejecución** llamando alguna de las siguientes funciones: *OSMBoxPend, OSQPend, OSSemPend, OSTaskSuspend, OSTimeDly y OSTimeDlyHMSM*. Esto se puede observar en la figura 2.5, y a partir de esta forma se pueden determinar con qué funciones se puede pasar a los diversos estados de una tarea.

Para llevar a cabo sus funciones cada tarea se le asigna un TCB y un área de pila. El TCB contiene las variables que se muestran en la tabla 2.1, donde se puede ver la función que desempeña cada variable. El área de pila que se le asigna a cada tarea es de tamaño definible por el usuario y es controlado por el TCB por medio de las variables *OSTCBStkPtr, OSTCBStkBottom* y *OSTCBStkSize*, definidos en la tabla 2.1

Tabla 2.1 Variables del bloque de control de tarea y función de cada una.

Variable	Función
OSTCBStkPtr	Contiene el apuntador que controla la pila de una tarea, este apunta a la parte alta de la pila.
OSTCBExtPtr	Es un apuntador a una extensión del bloque de control de tarea definida por el usuario
OSTCBStkBottom	Es un apuntador de la parte más baja de la pila de la tarea.
OSTCBStkSize	Mantiene el tamaño de la pila en número de elementos.
OSTCBId	Es usado como identificador de la tarea.
OSTCBNext y OSTCBPrev	Son apuntadores que se usan para encadenar todos los TCB's.
OSTCBEventPtr	Apuntador de bloque de control de evento.
OSTCBMsg	Es un apuntador de un mensaje que ha enviado a una tarea.
OSTCBDly	Es usado para que la tarea se retarde un intervalo de tiempo.
OSTCBStat	Contiene el estado en el que se encuentra la tarea.
OSTCBPrio	Contiene la prioridad de la tarea.
OSTCBX, OSTCBY, OSTCBBitX y OSTCBitY	Son valores que se usan para acelerar el proceso de hacer que una tarea lista para ejecución o hacer que una tarea espere por un evento.
OSTCBDelReq	Contiene un valor que indica si la tarea le fue solicitada su borrado ó no.

2.5.2 PLANIFICADOR DE TAREAS

La labor del planificador de μCOS-II es determinar qué tarea tiene la prioridad más alta de las que están preparadas para ejecutarse, y sea la próxima en ejecución. Las funciones que realizan esto son *OSSched* y *OSIntExit*. La primera es llamada a nivel-tarea y *OSIntExit* es a nivel-interrupción. El tiempo de planificación es constante sin importar el número de tareas creadas en la aplicación. El diagrama de flujo de la función *OSShed* se muestra en la figura 2.6, la función *OSIntExit* tiene algunas diferencias con *OSShed* al momento de implementarlo, pero básicamente tienen el mismo diagrama de flujo. Estas diferencias se tratarán más adelante.

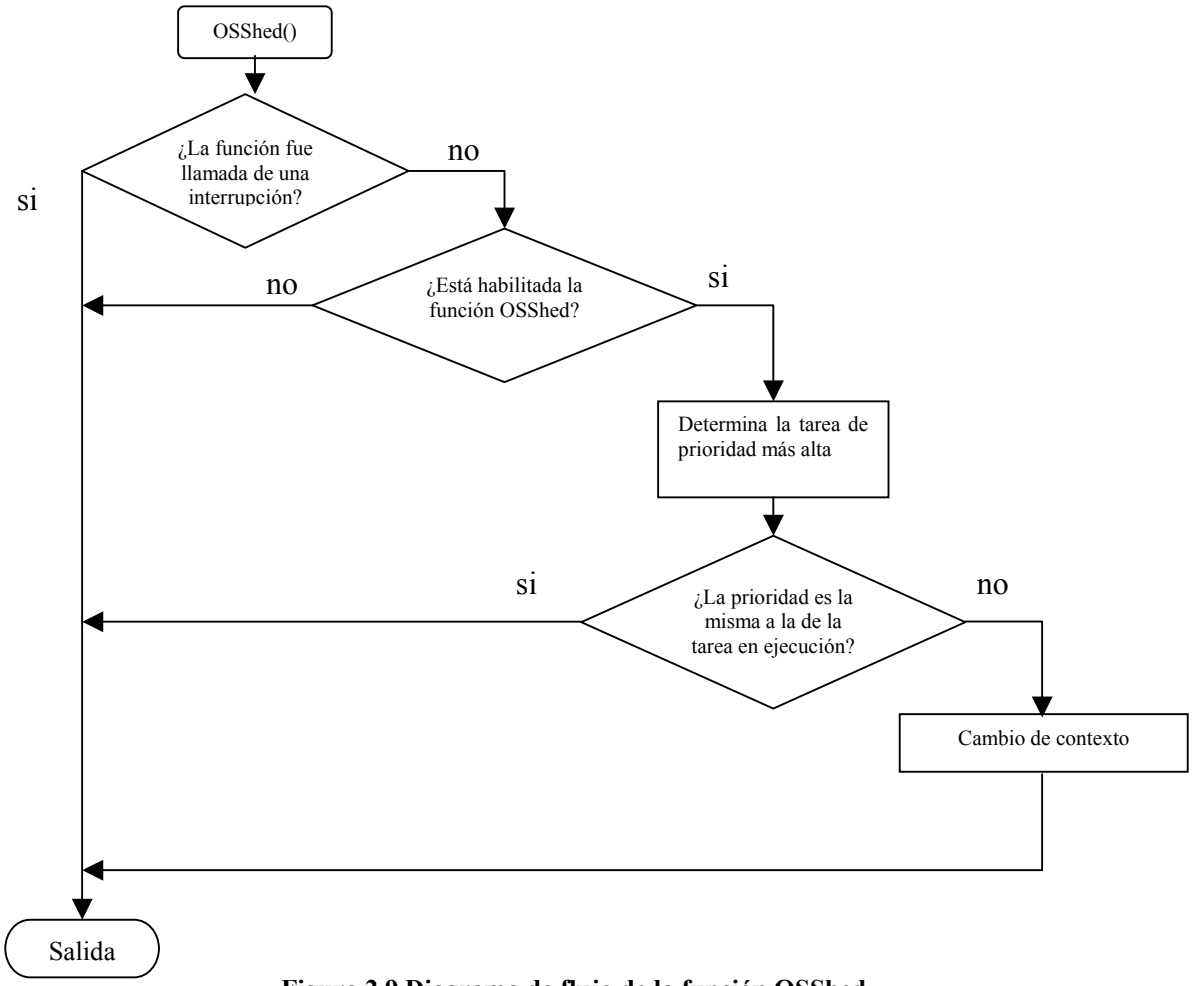


Figura 2.9 Diagrama de flujo de la función OSShed.

2.5.3 INTERRUPCIONES EN μCOS-II

Las rutinas de servicio de interrupción que se usen en μCOS-II, deben de realizar en algunas acciones en un orden determinado. Estas acciones son necesarias y no debe realizarse la rutina en forma clásica, ya que durante la rutina de servicio de interrupción una tarea puede pasar al estado preparada y ser de prioridad más alta a la interrumpida, además de que el sistema operativo debe conocer que el sistema se encuentra interrumpido. El proceso que debe seguir es el siguiente:

1. Guardar todos los registros de la CPU.

2. Llamar la función *OSIntEnter*, que forma parte de μCOS-II.
3. Ejecutar el código del usuario para la rutina de servicio de interrupción.
4. Llamar la función *OSIntExit*.
5. Restaurar los registros de la CPU.
6. Ejecutar una instrucción de retorno de interrupción.

Los registros se guardan en la pila correspondiente de la tarea actualmente en ejecución (la tarea pasa al estado interrumpida), ya que, como anteriormente se menciona, puede haber durante la interrupción una tarea de prioridad más alta que pase al estado preparada y se debe guardar el estado en que se encuentra la CPU en ese momento. *OSIntEnter* sirve para que μCOS-II se entere que ha ocurrido una interrupción. Después de ejecutar la rutina de servicio de interrupción se llama a la función *OSIntExit* para que μCOS-II sepa que ya salió de la interrupción y busque si hay una nueva tarea de prioridad más alta, para que posteriormente se restauren los registros de la nueva tarea ó la tarea interrumpida. Finalmente, se sale de la rutina de servicio de interrupción usando la instrucción adecuada.

2.5.4 SISTEMA DE TIEMPO DE μCOS-II

En μCOS-II una tarea se puede retardar su ejecución por un cierto tiempo y también se le puede asignar un plazo de espera de un evento (si el evento no ocurre en el plazo que se le da, la tarea se ejecuta). Para llevar a cabo lo anterior, μCOS-II necesita de un sistema de tiempo, este sistema se le conoce como *tic*, que es una interrupción que se genera periódicamente. Esta interrupción comúnmente tiene una frecuencia entre 10 y 100 Hertz.

La rutina de servicio de interrupción del sistema de tiempo debe realizar en orden las siguientes operaciones:

1. Guardar los registros del procesador.
2. Llamar a *OSIntEnter*.

3. Llamar la función *OSTimeTick*, que también forma parte de μCOS-II, y se encarga básicamente de decrementar la variable *OSTCBDly* del TCB de las tareas y mantener el número de *tics* desde que inicia en operación el sistema.
4. Llamar a *OSIntExit*.
5. Restaurar los registros del procesador.
6. Ejecutar una instrucción de retorno de interrupción.

Las operaciones que tiene que realizar la rutina de servicio de interrupción del sistema de tiempo son como las que se mencionaron para una interrupción manejada en μCOS-II.

2.5.5 SERVICIOS EN μCOS-II

Todos los servicios que maneja μCOS-II se muestran en la tabla 2.2, donde están agrupados de acuerdo a la actividad de cada función y una breve descripción, para mayor información se puede consultar el manual en [4].

Tabla 2.2 Servicios de μCOS-II.

Grupo	Funciones (servicios)	Descripción
Interrupción	*OSIntEnter()*	Notifica a μCOS-II que una ISR está siendo procesada
	OSIntExit()	Notifica a μCOS-II que una ISR ha sido completada.
Buzón de mensajes	*OSMboxAccept()*	Permite ver si un mensaje está disponible en el buzón de mensaje deseado.
	OSMboxCreate()	Crea e inicializa un buzón de mensaje.
	OSMboxPend()	Se usa cuando una tarea espera recibir un mensaje.
	OSMboxPost()	Envía un mensaje a una tarea por medio de un buzón de mensaje.
	OSMboxQuery()	Obtiene información a cerca de un mensaje en el buzón de mensaje.
Memoria	*OSMemCreate()*	Crea e inicializa una partición de memoria.
	OSMemGet()	Obtiene un bloque de memoria de una partición de memoria.

Tabla 2.2 (Continuación)

Grupo	Funciones (servicios)	Descripción
Memoria	*OSMemPut()*	Retorna un bloque de memoria para una partición de memoria.
	OSMemQuery()	Obtiene información a cerca de una partición de memoria.
Cola de mensajes	*OSQAccept()*	Checa si un mensaje está disponible en la cola de mensajes deseada.
	OSQCreate()	Crea una cola de mensajes.
	OSQFlush()	Vacía los contenidos de la cola de mensajes y elimina todos los mensajes enviados a la cola.
	OSQPend()	Es usado cuando una tara quiere recibir mensajes de la cola.
	OSQPost()	Envía un mensaje para una tarea por medio de una cola.
	OSQpostFront()	Envía un mensaje para una tarea por medio de una cola. Pero, el mensaje que envía es el insertado al frente de la cola.
	OSQQuery()	Obtiene información a cerca de una cola de mensajes.
Planificación	*OSSchedLock()*	Deshabilita la planificación de tarea hasta su contraparte OSSchedUnlock().
	OSSchedUnlock()	Re-habilita la planificación de tarea.
Semáforo	*OSSemAccept()*	Checa si una fuente de datos compartida está disponible o un evento ha ocurrido.
	OSSemCreate()	Crea e inicializa un semáforo.
Semáforo	*OSSemPend()*	Es usado cuando una tarea quiere acceso exclusivo a una fuente de datos compartida, necesita sincronizar su actividad con una ISR ó una tarea, o para esperar a que ocurra un evento.
	OSSemPost()	Un semáforo es señalado al llamar esta función.
	OSSemQuery()	Obtiene información a cerca de un semáforo.
Tareas	*OSTaskChangePrio()*	Cambia la prioridad de una tarea.
	OSTaskCreate()	Crea una tarea para que sea manejada por µCOS-II.
	OSTaskCreateExt()	Crea una tarea para que sea manejada por µCOS-II. Pero con información adicional a cerca de la tarea.

Tabla 2.2 (Continuación)

Grupo	Funciones (servicios)	Descripción
Tareas	*OSTaskDel()*	Borra una tarea especificada por el número de prioridad de la tarea a borrar.
	OSTaskDelReq()	Solicita que una tarea se borre asimisma.
	OSTaskQuery()	Obtiene información a cerca de una tarea.
	OSTaskResume()	Hace que continúe su ejecución una tarea que fue suspendida por medio de *OSTaskSuspend*.
	OSTaskStkChk()	Calcula el monto de espacio de pila libre y usada de una tarea especificada.
Tiempo	*OSTaskSuspend()*	Suspende la ejecución de una tarea incondicionalmente.
	OSTimeDly()	Permite a una tarea retardarse asimisma por un número de *tics*.
	OSTimeDlyHMSM()	Permite a una tarea retardarse asimisma por un tiempo especificado en horas, minutos, segundos y milisegundos.
	OSTimeResume()	Hace continuar con la ejecución de una tarea que haya sido retardada a través de una llamada a *OSTimeDly* ó *OSTimeDlyHMSM*.
	OSTimeGet()	Obtiene el valor actual del sistema de reloj (*tic*).
Otros	*OSTimeSet()*	Establece el valor actual del sistema de reloj (número de *tics*).
	OSTimeTick()	Procesa un *tic*.
	OSInit()	Inicializa a μCOS-II y debe ser llamada antes que *OSStart*, que realmente inicia el sistema multitareas.
	OSStart()	Inicia la multitarea en μCOS-II.
	OSStatInit()	Determina el máximo valor que un contador puede alcanzar cuando ninguna tarea se está ejecutando.
	OSVersion()	Obtiene la versión actual de μCOS-II.
	OS_ENTER_CRITICAL()	Macro usado para deshabilitar las interrupciones del procesador.
	OS_EXIT_CRITICAL()	Macro usado para habilitar las interrupciones del procesador.

2.5.6. MANEJO BÁSICO DE LAS TAREAS EN μCOS-II

Una tarea se ve como cualquier otra función c, contiene un tipo de retorno y un argumento, pero esta nunca debe retornar. El tipo de retorno de una tarea debe ser siempre declarado como *void*. La estructura general de una tarea se muestra en listado 2.1.

Para que una tarea pueda ser manejada por μCOS-II, esta debe ser creada. Una forma de crear una tarea es pasando su dirección, la dirección de su argumento, la dirección de inicio de su pila y su prioridad al servicio *OSTaskCreate*; el formato de la instrucción es como sigue:

*INT8U OSTaskCreate(void (*task) (void *pd), void *pdata, OS_STK *ptos, INT8U prio);*

Donde:

INT8U, es un tipo de dato entero de 8 bits no signado que retorna la función: si no hay ningún error en la creación retorna 0, si la prioridad está en uso 40, si la prioridad es inválida 42 y si no hay TCB disponibles 70.

*void(*task)(void *pd),* es un apuntador al código de la tarea.

*void *pdata,* es un apuntador de dato opcional usado para pasar parámetros a la tarea cuando se crea.

*OS_STK *ptos*, es un apuntador a la parte alta de la pila de la tarea. *OS_STK* está definido como un tipo de dato entero no signado de 16 bits.

INT8U prio, es la prioridad de la tarea.

Una tarea debe involucrar al menos un servicio provisto por μCOS-II, ya sea para esperar a que expire un tiempo, suspenda la tarea ó espere a que ocurra algún evento (esperando un buzón de mensaje, cola ó semáforo). Esto permite que otras tareas puedan ganar el control de la CPU.

```
void TuTarea (void *pdata)
{
        .                       /*      Hacer algo con el pdata                         */
    for (;;)    {               /* Cuerpo de la tarea, siempre debe ser un bucle infinito */
        /*CODIGO DEL USUARIO                                                            */
        /*Llamar al menos uno de los servicios de uC/OS-II:
        /*      OSQPend();                                                              */
        /*      OSTimeDly();                                                            */
        /*      OSTimeDlyHMSM();                                                        */
        /*      OSTaskSuspend();                                                        */
        /*      OSSemPend();                                                            */
        /*      OSTaskDel();                                                            */
        /*      OSMboxPend();                                                           */
        /*CODIGO DEL USUARIO*/
        }
    }
```

Listado 2.1 Estructura de una tarea en µCOS-II.

µCOS-II debe ser inicializado para que se encargue de manejar las tareas. Para inicializar µCOS-II se emplea la función principal. En el listado 2.2 se muestra un ejemplo de aplicación típico, donde se ve la función principal *main* y la tarea *task1*. La función principal hace lo siguiente:

- Llama al servicio *OSInit*. Se encarga de inicializar las variables y las estructuras de datos de µCOS-II, prepara también el sistema de planificación y crear una tarea llamada "ociosa". La tarea ociosa tiene la prioridad más baja de las tareas y se emplea para que el procesador siempre tenga código por ejecutar, cuando no haya tareas que estén preparadas para ejecución.

- Posteriormente, se debe crear cuando menos una tarea antes de iniciar el proceso de multitareas. En este caso, se crea únicamente la tarea que se llama *Task1*, se pasa un argumento nulo, la dirección de inicio de la pila que tiene un tamaño de 1k de datos y tiene una prioridad de 25.

- Al final del programa principal se llama al servicio *OSStart*. Este servicio debe ser llamado solamente una vez y se encarga de iniciar el sistema multitareas de μCOS-II. Busca la tarea de prioridad más alta que haya sido creada y es la primera tarea en ejecutarse.

La tarea tiene la estructura como la del listado 2.1, *pdata* se usa únicamente para no producir una advertencia (*warning*).

```
OS_STK Task1Stk(1024);
void    main(void)
{
        INT8U   err;
        .
        OSInit();
        .
        OSTaskCreate(Task1,(void *)0,&Task1Stk[0],25);
        .
        OSStart();
}

void    Task1(void *pdata)
        {
                pdata=pdata;
                for (;;)  {             /*          CODIGO DE LA TAREA          */
                        .
                        .
                }
        }
```

Listado 2.2 Ejemplo de una tarea en μCOS-II.

CAPÍTULO 3

DISEÑO E IMPLEMENTACIÓN DEL SISTEMA OPERATIVO BASADO EN μCOS-II PARA EL TMS320F240

La arquitectura de una aplicación implementada con un sistema operativo basado en μCOS-II consta de cuatro partes: El programa de aplicación, el código independiente del procesador (kernel), código específico de la aplicación (configuración del kernel) y código específico del procesador (diseño del código para el procesador a usar). Cada parte se compone de diversos archivos que se aprecian en la figura 3.1.

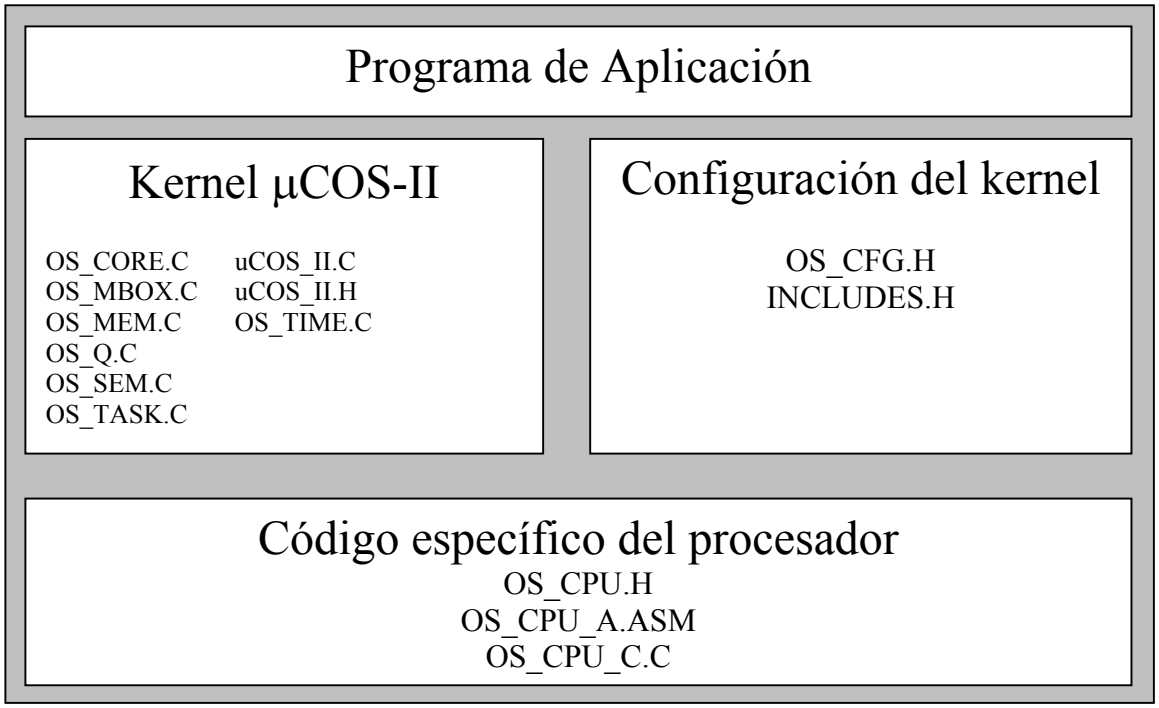


Figura 3.1 Arquitectura de una aplicación usando un sistema operativo basado en μCOS-II.

El código independiente del procesador permanece sin cambio para cualquier tipo de procesador que se tenga (microprocesador, microcontrolador ó DSP), siempre y cuando cuente con las siguientes características [4]:

1. Tenga un compilador C que genere código reentrante.
2. Las interrupciones puedan ser deshabilitadas y habilitadas desde C.
3. El procesador soporte interrupciones que ocurran en intervalos regulares.
4. El procesador soporte una pila que pueda almacenar entre 128 bytes y 8 kbytes por tarea (La pila es usada para almacenar variables locales, parámetros de la función, dirección de retorno y los registros de la CPU cuando ocurre una interrupción).
5. El procesador tenga instrucciones para cargar y almacenar el apuntador de pila y otros registros de la CPU, también en la pila ó en memoria.

El procesador TMS320F240 cuenta con las características mencionadas y algunas otras que pueden apreciarse en el apéndice A.

Para que el sistema operativo se pueda usar en un procesador específico, se requiere de tres archivos: archivo cabecera de definiciones del procesador específico (*OS_CPU.H*), archivo de funciones en leguaje ensamblador del procesador específico (*OS_CPU_A.ASM*) y archivo de funciones en lenguaje C del procesador específico (*OS_CPU_C.C*)[4]. Estos archivos son diferentes para cada procesador que se emplee. El diseño del sistema operativo basado en μCOS-II para el TMS320F240 consiste precisamente en crear los archivos mencionados. Los archivos deben contener lo siguiente:

- Configuración del valor de una constante #define (*OS_CPU.H*),
- Declaración de 10 tipos de datos (*OS_CPU.H*),
- Declaración de 3 macros #define (*OS_CPU.H*),
- 6 funciones escritas en lenguaje C (*OS_CPU_C.C*) y
- 4 funciones escritas en lenguaje ensamblador (*OS_CPU_A.ASM*).

El programa de aplicación y la configuración del kernel depende de la aplicación específica que se desarrolle, esto se puede ver en el siguiente capítulo 4.

3.1 ARCHIVO DE DEFINICIONES DEL PROCESADOR ESPECÍFICO (OS_CPU.H)

De acuerdo a [4], el archivo *OS_CPU.H* debe contener la definición de tipos de datos que correspondan al compilador específico (*typedef*), la definición de tres macros (*OS_EXIT_CRITICAL, OS_ENTER_CRITICAL* y *OS_TASK_SW*) y una constante que especifica el tipo de pila que tiene el procesador (*OS_STK_GROWTH*).

Se definen tipos de datos en el sistema operativo, ya que los diversos procesadores tienen diferente tamaño de palabra y como consecuencia diferente tamaño en los tipos de datos que se maneje en el compilador C. En la primer columna de la tabla 3.1 muestra los nombres de los tipos de datos definidos en μCOS-II y en la segunda columna la asignación que se realizó de acuerdo a los tipos de datos que se tienen en el compilador C del TMS320F240 [5], así como sus tamaños en bits. No todos los tipos de datos corresponden exactamente a los tamaños de bits que son manejados por μCOS-II, esto es debido a que el DSP maneja como mínimo un tamaño de 16 bits y el compilador procesa hasta 2 palabras en conjunto, es decir 32 bits.

Tabla 3.1 Asignación de los tipos de datos en el sistema operativo.

Tipo de dato definido en μCOS-II		Tipo de dato asignado del compilador C del TMS320C2XX	
Nombre	Tamaño (bits)	Nombre	Tamaño (bits)
BOOLEAN	8	unsigned char	16
INT8U	8	unsigned char	16
INT8S	8	signed char	16
INT16U	16	unsigned int	16
INT16S	16	signed int	16
INT32U	32	unsigned long	32
INT32S	32	signed long	32
FP32	32	Flota	32
FP64	64	Double	32
OS_STK	16	unsigned int	16

Se tiene que definir los macros *OS_EXIT_CRITICAL, OS_ENTER_CRITICAL y OS_TASK_SW*. *OS_ENTER_CRITICAL* y *OS_EXIT_CRITICAL* lo emplea μCOS-II para habilitar y deshabilitar las interrupciones respectivamente, con la finalidad de que se tenga acceso exclusivo a alguna parte de código. En la tabla de instrucciones del TMS320C240 hay dos instrucciones que se aprovechan para hacerlo, *SETC INTM* (deshabilita interrupciones mascarables) y *CLRC INTM* (habilita las interrupciones mascarables). El otro macro es *OS_TASK_SW*, el cual es llamado por μCOS-II cuando hay una tarea de prioridad más alta que necesita pasar al estado de ejecución. Para que μCOS-II haga que una tarea pase al estado de ejecución, debe darle el control de la CPU a esta tarea que tiene una prioridad más alta que la que se estaba ejecutándose y debe suspender la tarea que estaba en ejecución. A la tarea suspendida se le debe guardar el estado en que deja la CPU, para poder continuar su ejecución en otro momento. Una forma simple de guardar el estado de la CPU es generar una interrupción por programa. Por lo que *OS_TASK_SW* debe generar una interrupción por software para que se pueda realizar el cambio de la tarea, ya que al generar la interrupción se guarda en la pila física del TMS320F240 la dirección de retorno de la interrupción, y se complementa con en el vector de interrupción que llama a la rutina de servicio de interrupción para que termine con el cambio de tarea. En el TMS320F240 existen varias interrupciones por software que se llaman con la instrucción *INTR*, la última de las interrupciones es la 31, que es una opción para definir el macro *OS_TASK_SW*.

Finalmente, se define una constante que configura la forma en que crece la pila *OS_STK_GROWTH*. En el compilador C, el manejo de la pila se realiza por programación y crece de una memoria baja a una alta por lo que *OS_STK_GROWTH* adquiere el valor de 0, si la pila creciera de memoria alta a baja el valor debería ser 1. Este valor es usado para el manejo de la memoria correspondiente a las pilas de las tareas. El código correspondiente al archivo *OS_CPU.H* con las características que se mencionaron se puede ver en el apéndice B.

3.2 ARCHIVO DE FUNCIONES EN LEGUAJE ENSAMBLADOR DEL PROCESADOR ESPECÍFICO (OS_CPU_A.ASM)

Para el archivo *OS_CPU_A.ASM* son necesarias las siguientes funciones que deben ser escritas en lenguaje ensamblador:

- Función que inicia la ejecución de la tarea de prioridad más alta (*OSStartHighRdy*).
- Función que realiza cambio de contexto a nivel tarea (*OSCtxSw*).
- Función que realiza cambio de contexto a nivel interrupción *OSIntCtxSw*.
- Rutina de servicio de interrupción del tic (*OSTickISR*).

Estas funciones se escriben en lenguaje ensamblador porque no se pueden acceder a los registros del procesador directamente desde el compilador C.

3.2.1 FUNCIÓN QUE INICIA LA EJECUCIÓN DE LA TAREA DE PRIORIDAD MÁS ALTA (*OSStartHighRdy*)

Esta función es llamada por otra llamada *OSStart*. *OSStart* es el servicio de μCOS-II que inicia el proceso de multitareas. *OSStartHighRdy* es usada para inicializar la tarea de prioridad más alta lista para ejecutarse en el inicio del proceso de multitareas. El procedimiento que debe realizar en orden es el siguiente [4]:

1. Llamar la función definida por el usuario *OSTaskSwHook*. Esta función se emplea para personalizar algún proceso al momento de llamar *OSStartHighRdy*.

2. Obtener el apuntador de pila de la primer tarea de prioridad más alta que va a ejecutarse. Cuando llega a ejecutarse esta parte del código, el apuntador *OSTCBHighRdy* apunta al bloque de control de tarea de la tarea de prioridad más alta y de allí toma la dirección de donde quedó su pila para que actualice el apuntador de pila.

Apuntador de pila = *OSTCBHighRdy->OSTCBStkPtr*

3. *OSRuning* = TRUE. Con la variable *OSRuning* se le dice a μCOS-II que ya inició el proceso de multitareas.

4. Restaurar todos los registros que se encuentran en la pila de la tarea que se va a ejecutar. Para que de esta forma la tarea obtenga el control de la CPU.

5. Ejecutar la instrucción de retorno de una interrupción. Con la ejecución de la instrucción de retorno el contador de programa apunta al inicio del código correspondiente a la tarea e inicia el procesador con la ejecución de las instrucciones.

El código de esta función para el TMS320F240 se muestra en el listado 3.1. Para crear esta función se tuvo que considerar lo siguiente:

- En el listado puede observarse que los nombres de las funciones y variables, van antecedidos por un guión bajo, esto es para poder distinguir las funciones y variables cuando están escritas en lenguaje ensamblador [5].

- El TMS320F240 no cuenta con un apuntador de pila (SP) que pueda modificarse, ya que la pila que tiene es física y el manejo de la pila lo tiene el procesador. Sin embargo, el compilador C maneja una pila por programación en lugar de usar los 8 niveles con que cuenta de pila física, ya que no son suficientes y se debe de tener el control de la pila. Para crear una pila por programación el compilador C, utiliza los registros auxiliares cero y uno (AR0 y AR1), ocupando a AR0 como el apuntador de trama y AR1 como el apuntador de pila.

- Para restaurar el contexto de la tarea se tuvo que auxiliar de una función que forma parte del compilador C llamada *I$$REST*, la cual está relacionada con *I$$SAVE*. La función *I$$SAVE* se encarga de salvar el contexto actual de la CPU en la pila en el

orden que se muestra en la figura 3.2 y lo que hace *I$$REST* es restaurar el contexto de la CPU en el orden contrario.

```
_OSStartHighRdy:
        call    _OSTaskSwHook().
        lar     AR1,#_OSTCBHighRdy    ; AR1 apunta a _OSTCBHighRdy
        lar     AR1,*                 ; AR1 apunta al TCB de la tarea de prioridad más alta
        lar     AR1,*                 ; AR1 se convierte en el apuntador de
                                      ; pila de la tarea de prioridad más alta
        lacc    #01h
        LDPK    _OSRunning            ; Actualiza el DP para _OSRunning
        sacl    _OSRunning            ; OSRunning=1
        b       I$$REST               ; restaura el contexto de la tarea lista de prioridad mas alta
```

Listado 3.1 Código de la función *OSStartHighRdy*.

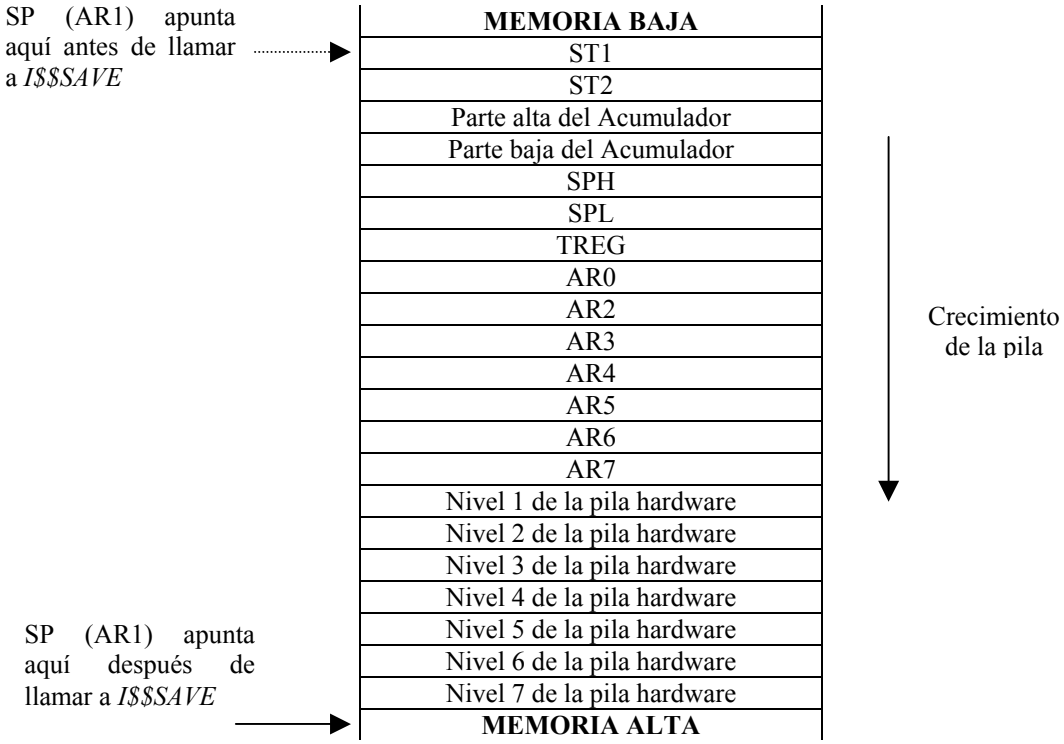


Figura 3.2 Orden en que se almacenan los registros internos de la CPU usando la función *I$$SAVE*.

- Al final del código de la función *I$$REST*, tiene una instrucción de retorno de interrupción, por lo que es innecesario la instrucción de retorno de interrupción en la función *_OSStartHighRdy*.

3.2.2 FUNCIÓN QUE REALIZA CAMBIO DE CONTEXTO A NIVEL TAREA (*OSCtxSw*)

Esta función es la rutina de servicio de interrupción de la interrupción por software usada por la macro *OS_TASK_SW*, para realizar un cambio de contexto a nivel tarea. Este cambio de contexto se efectúa debido a que se ha llamado un servicio de µCOS-II, y este último hizo que una tarea de prioridad más alta a la actual que está ejecutándose, esté lista para ejecutarse. Al final de la llamada del servicio, µCOS-II llama a la función *OSSched*, la cual concluye que la tarea actual no es la más importante para continuar su ejecución. Entonces, *OSSched* carga la dirección de la tarea de prioridad más alta en *OSTCBHighRdy*, para después ejecutar la instrucción de interrupción por software al ejecutar el macro *OS_TASK_SW*. Pero *OSTCBCur* (apuntador que contiene la dirección del bloque de control de tarea de la tarea actual) todavía apunta al bloque de control de tarea de la tarea de prioridad baja. La instrucción de interrupción por software hace que el contenido del contador de programa se guarde en la pila física. Finalmente, se ejecuta *OSCtxSw*. El código de *OSCtxSw* debe efectuar lo siguiente [4]:

- Salvar los registros del procesador, para que la tarea suspendida pueda continuar su ejecución posteriormente sin que se alteren las condiciones del procesador.

- Llamar la función definida por el usuario *OSTaskSwHook* para un cambio de contexto especial.

- Salvar el apuntador de pila en el bloque de control de tarea de la tarea suspendida. *OSTCBCur* todavía apunta al bloque mencionado.

OSTCBCur->OSTCBStkPtr=apuntador de pila

- Actualizar el apuntador de pila con el de la tarea de prioridad más alta. *OSTCBHighRdy* es el que apunta a esta tarea.

Apuntador de pila=*OSTCBHighRdy->OSTCBStkPtr*

- Actualizar *OSTCBCur* con la dirección del bloque de control de tarea de la tarea de más alta prioridad, para convertirse en la tarea que está ejecutándose actualmente.

OSTCBCur=OSTCBHighRdy

- La prioridad de la tarea más alta que está lista para ejecutarse (*OSPrioHighRdy*) debe ser el de la tarea que está ejecutándose (*OSPrioCur*) para después restaurar el contexto del TMS320F240 usando de la pila de la tarea que pasó a ser la tarea actual (antes la tarea de más alta prioridad).

OSPrioCur=OSPrioHighRdy

- Restaurar los registros de la nueva tarea que se encuentran en su pila.

- Ejecutar la instrucción de retorno de la interrupción.

Para escribir el código de la función para el TMS320F240 se tomaron las mismas consideraciones que en la función *OSStartHighRdy*. El código de *OSCtxSw* se muestra en el apéndice B.

3.2.3 FUNCIÓN QUE REALIZA CAMBIO DE CONTEXTO A NIVEL INTERRUPCIÓN (*OSIntCtxSw*)

Esta función es llamada por *OSIntExit* para ejecutar un cambio de contexto a partir de una ISR (rutina de servicio de interrupción), es decir a nivel interrupción. Este cambio de contexto es parecido al que se realiza a nivel tarea, pero se debe tomar en cuenta cómo se modifica el contenido de la pila física y la pila por programación al entrar al ejecutar la rutina del servicio de interrupción. Tal que debe ajustarse los contenidos de la pila física y la pila por programación, para que la tarea interrumpida quede con el contenido de pila apropiado.

Para entender *OSIntCtxSw*, se necesita analizar la secuencia de eventos que llevan a μCOS-II a llamar a la función. Para la siguiente descripción observe la figura 3.3, y considere que la interrupción no está anidada (es decir, una ISR no es interrumpida), las interrupciones están habilitadas, el procesador está ejecutando el código de una tarea y la ISR tiene la estructura que se mencionó en el capítulo 2:

- Cuando se genera una interrupción, la instrucción actual se completa, reconoce la interrupción e inicia el procedimiento de manejo de la interrupción. El procesador va a la rutina de servicio de interrupción apropiada. En la rutina, si el código se escribió en lenguaje C y sino se crea ninguna variable temporal, los registros del TMS320F240 son guardados en la pila por programación usando la función *I$$SAVE* (figura 3.3-1) y se deja un espacio de memoria para almacenamiento temporal de datos (figura 3.3-2) que usa el compilador. Luego, μCOS-II requiere también de llamar a *OSIntEnter* ó incrementar la variable global *OSIntNesting* en uno. Hasta aquí, la pila de la tarea interrumpida tiene el contexto de la tarea interrumpida y un espacio de memoria para almacenamiento temporal. La rutina de servicio de interrupción puede ahora iniciar el servicio del dispositivo que generó la interrupción y posiblemente hacer una tarea de prioridad más alta lista para ejecutarse al llamar uno de los servicios de μCOS-II.

- Cuando la rutina de servicio de interrupción completa el servicio del dispositivo que generó la interrupción, la rutina de servicio llama a *OSIntExit*. *OSIntExit* le dice básicamente que es momento de regresar al código de tarea. Si el código fue escrito en lenguaje C, al llamar a la función *OSIntExit* causa que la dirección de retorno de la función y el apuntador de trama de pila (AR0) sean almacenados en la pila y se deje un espacio de memoria para almacenamiento temporal (figura 3.3-3, 3.3-4 y 3.3-5 respectivamente).

- *OSIntExit* inicia deshabilitando las interrupciones debido a que se va a ejecutar código crítico, llamando el macro *OS_ENTER_CRITICAL*. *OSIntExit* entonces concluye que la tarea interrumpida no tiene la suficiente prioridad para seguir ejecutándose porque otra tarea de prioridad más alta está lista para ejecutarse. En este caso, el apuntador

OSTCBHighRdy se le hace apuntar al *OS_TCB* de la tarea nueva, y *OSIntExit* llama a *OSIntCtxSw* para ejecutar el cambio de contexto. Al llamar *OSIntCtxSw* causa que la dirección de retorno sea metida en la pila física.

Si se llevara acabo el cambio de contexto con estas condiciones en la pila de manera similar a *OSCtxSw*, sería incorrecto el contenido de las pilas. El contenido de pila por programación válido para realizar el cambio de contexto, son las etapas (1) y (2) de la figura 3.3 y se deben ignorar la (3), (4) y (5) de la misma figura. Para ajustar el apuntador de pila (AR1) una constante (en este caso 4) se efectúa con la instrucción *sbrk*. Además se debe ajustar la pila física, ya que en la parte superior de la pila contiene la dirección de retorno al llamar la función *OSIntCtxSw*, el ajuste se hace con la instrucción pop (saca el dato que está en la parte alta de la pila). Después del ajuste de las pilas, se continúa con el cambio de contexto de la misma forma que se realizó con *OSCtxSw*. El código resultante se muestra en el apéndice B.

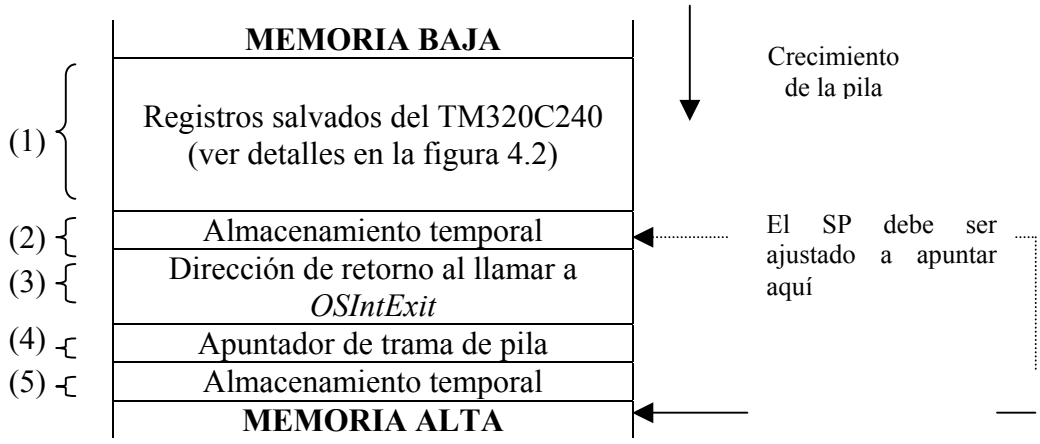


Figura 3.3 Ajuste de apuntador de pila que debe efectuar la función *OSIntCtxSw*.

3.2.4 RUTINA DE SERVICIO DE INTERRUPCIÓN DEL TIC (*OSTickISR*)

Como ya se mencionó en el capítulo 2, μCOS-II necesita tener una fuente de tiempo periódico (*tic*) para mantener la secuencia de los retardos de tiempo y los tiempo-fuera. Un tic debe ocurrir entre 10 y 100 veces por segundo o hertz. Este puede ser generado por un

temporizador que contenga el procesador que se vaya a emplear ó una fuente externa. En los dos casos se requiere manejar una interrupción periódica que sirva como tic. *OSTickISR* es la rutina de servicio de interrupción del tic. *OSTickISR* se trata de una rutina de servicio de interrupción común, que debe llamar a la función *OSTimeTick* [4].

El código de la función *OSTickISR* para el TMS320F240, se puede escribir en lenguaje ensamblador, siguiendo los pasos para escribir una rutina de servicio de interrupción normal que se utilice en el sistema operativo. Pero debe de percatarse que se va a llamar a la función *OSIntExit*, y este a su vez, puede llamar a la función *OSIntCtxSw* para un cambio de contexto. La cual, fue escrita suponiendo que la rutina de servicio de interrupción sería escrita en lenguaje C. En este caso, la diferencia que se produce si se escribe la ISR en lenguaje ensamblador es en el manejo de la pila por programación. En lenguaje C se reserva un espacio de memoria para almacenamiento temporal, después de salvar el contexto al iniciar la ISR (figura 3.3-2) [5]. Este espacio reservado debe incluirse al escribir la ISR en lenguaje ensamblador incrementando en 1 al apuntador de pila. Esta modificación al apuntador de pila debe ser ajustada al final de la ISR de la misma forma como lo hace el compilador C, decrementando en 1 al apuntador de pila. El listado de esta rutina se puede ver en el apéndice B.

Para que la rutina *OSTickISR* se ejecute periódicamente, el TMS320F240 debe generar una interrupción periódica que tenga como ISR a la función *OSTickISR*. El DSP cuenta con una interrupción en tiempo real (RTI), la cual genera interrupciones periódicas de frecuencias bajas. Para configurar la RTI se usan los registros CKCR1 y RTICR (apéndice C y D, *ini.asm*). El registro CKCR1 se configura dependiendo la frecuencia de la señal de reloj de entrada, que en este caso se usó 10 MHz, si la información introducida correctamente en el registro CKCR1, el sistema de conteo de la RTI (WDCLK) trabajará a una frecuencia de 15625 Hz. Con el registro RTICR, se configura el preescalamiento del sistema de conteo WDCLK y la interrupción local de este periférico, se hizo un preescalamiento de factor 256 por lo que debe una interrupción cada 16.384 ms (61.04 Hz). Finalmente, para que se pueda ejecutar la rutina de servicio de interrupción se debe configurar el vector de interrupción correspondiente a la RTI para que pueda ejecutar la

función OSTickISR (apéndice C y D, *vector.asm*). De esta forma, se tiene el tic a una frecuencia de 61.04 Hz.

3.3 ARCHIVO DE FUNCIONES EN LENGUAJE C DEL PROCESADOR ESPECÍFICO (OS_CPU_C.C)

Para que μCOS-II se ejecute en un procesador se deben escribir 6 funciones en lenguaje C, estas son: *OSTaskStkInit, OSTaskCreateHook, OSTaskDelHook, OSTaskSwHook, OSTaskStatHook* y *OSTimeTickHook.* De las 6 funciones, 5 deben ser declaradas pero no necesitan contener código, ya que estas funciones son para personalizar algunos servicios que ofrece μCOS-II, éstas son *OSTaskCreateHook, OSTaskDelHook, OSTaskSwHook, OSTaskStatHook* y *OSTimeTickHook*. Y la otra función, *OSTaskStkInit*, no solamente necesita ser declarada sino contener código.

OSTaskCreate y *OSTaskCreateExt* llaman a la función *OSTaskStkInit* para llenar con el contenido inicial a la trama de pila de una tarea que está siendo creada. *OSTaskStkInit* requiere de tres argumentos que son transferidos al crear una tarea, estos son, dirección de inicio del código de la tarea (*task*), la dirección de inicio de la pila asignada a la tarea (*ptos*) y un dato que se quiera transferir al momento de ser creada la función (*pdata*). La trama de pila inicial debe tener un contenido adecuado, para que la tarea se ejecute correctamente la primera vez. Para que una tarea se ejecute en μCOS-II, se realiza un cambio de contexto, no se efectúa de manera convencional como si se tratara de una función, por lo que el contenido inicial de la pila es el contexto inicial de la tarea. Otros puntos importantes son: que la tarea no tiene ninguna dirección a donde retornar y tiene un argumento al cual posiblemente se acceda.

Para escribir el código de *OSTaskStkIni* para el TMS320F240, se debe analizar el procedimiento de la ejecución de una tarea la primera vez.

- El cambio de contexto se realiza con la ejecución de *OSIntCtx* ó *OSCtxSw* si μCOS-II determina que la tarea debe ejecutarse. Las funciones *OSIntCtx* y *OSCtxSw*

toman el contexto de la tarea que se va a ejecutar de la pila por programación, ejecutando la función *I$$REST*. El contenido del nivel 1 de la pila debe ser el contexto de la tarea.

- La última instrucción de la función *I$$REST* es un retorno de interrupción, al ejecutar esta instrucción, la CPU ejecuta la instrucción indicada por el último nivel de la pila física. La instrucción indicada, debe ser la primera instrucción del código de la tarea que se vaya a ejecutar por primera vez.

- Durante la ejecución de la tarea se debe poder acceder al argumento *pdata*. Para que se pueda acceder al argumento se debe seguir las convenciones que maneja el compilador C. Para acceder a un argumento, el compilador C lo hace referenciando al apuntador de trama y considerando que el contenido de la pila sea como el que se muestra en la figura 3.4 [5]. En la figura, la parte sombreada es donde se encontraba el contexto del procesado.

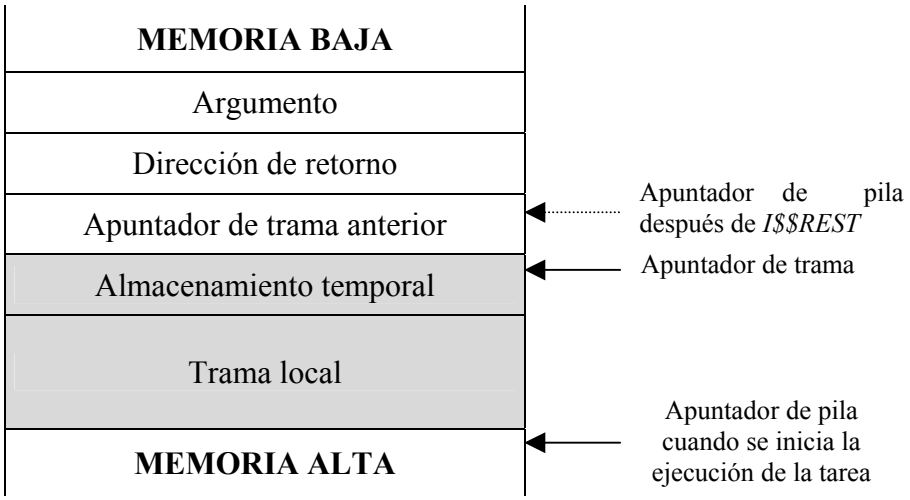


Figura 3.4 Contenido de la pila al inicio de ejecución de una tarea que haya sido escrita en lenguaje C.

Con este análisis se concluye que el contenido de la pila debe ser como el de la figura 3.5. Esta figura muestra lo siguiente: la posición del apuntador de pila antes de que la función de cambio de contexto restaure la tarea (4), la posición del contexto en memoria (3),

ubicación del apuntador depila después del cambio de contexto (5), la supuesta dirección de retorno de la tarea (2), el argumento (1) de la tarea y la ubicación del inicio de la pila.

El contenido inicial de la pila por programación que se muestra en la figura 3. 5 debe tener valores que no deban afectar al ejecutarse por primera vez. La tabla 3.2 propone los valores mencionados y da la justificación del mismo.

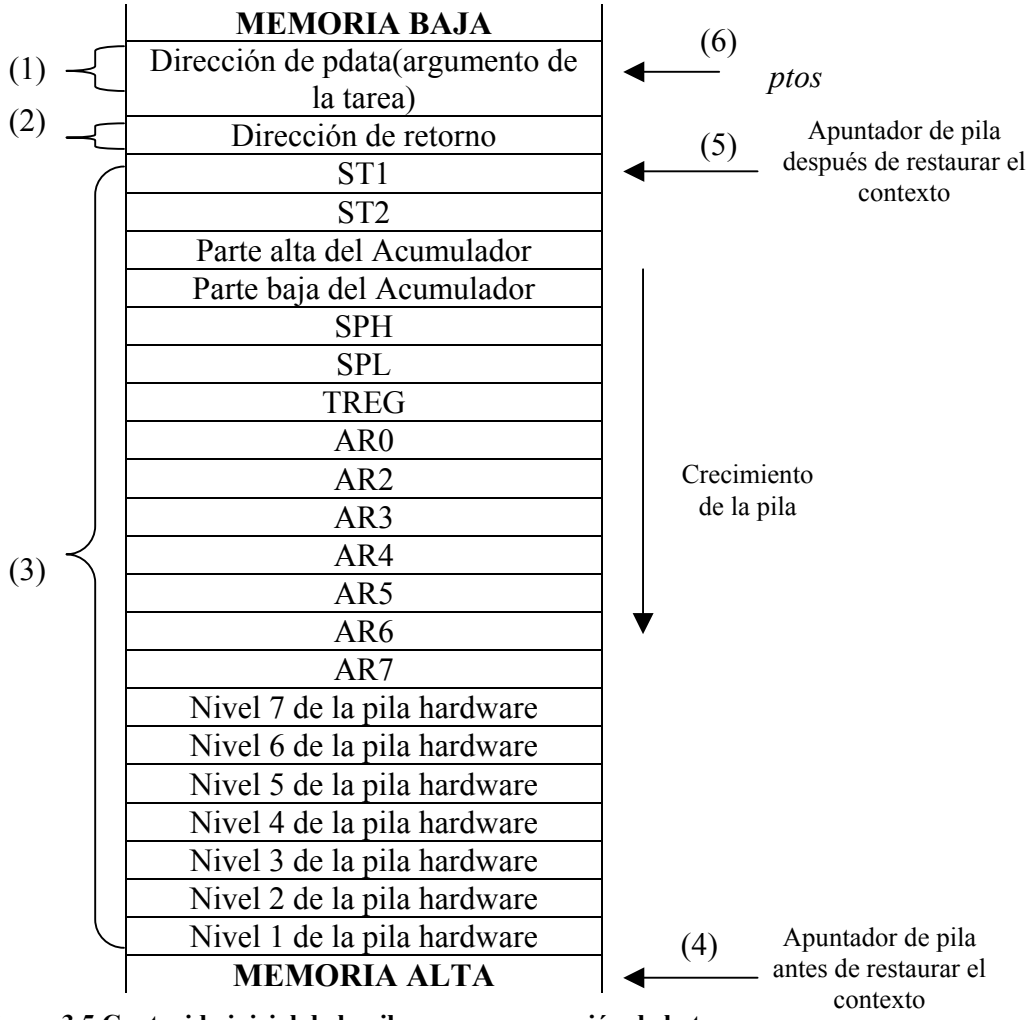


Figura 3.5 Contenido inicial de la pila por programación de la tarea

La función *OSTaskStkInit* debe llenar la pila por programación con el contenido mostrado en la figura 3.5 y con los valores señalados en la tabla 2. La dirección de inicio de la pila por programación lo da el argumento *ptos*, el cual indica dónde se debe guardar el primer dato de la tabla 2, e incrementando la dirección el siguiente, etc. El código de esta función se muestra en el apéndice B.

Tabla 3.2 Valores del contenido inicial de la pila por programación de una tarea.

Contenido de pila	Valor	Justificación
Argumento	Pdata	Argumento de la tarea
Dirección de retorno	0x0000	No importa, nunca retorna la tarea
ST1	0x2000	ARB=AR1
ST2	0x2200	ARP=AR1, interrupciones habilitadas y DP=0, el registro auxiliar actual tiene que ser el apuntador de pila (AR1), al inicio de ejecución de la tarea por convención del compilador C
Parte alta del Acumulador	0x0000	Cualquier valor
Parte baja del Acumulador	0x0000	Cualquier valor
SPH	0x0000	Cualquier valor
SPL	0x0000	Cualquier valor
TREG	0x0000	Cualquier valor
AR0	0x0000	Cualquier valor
AR2	0x2222	Cualquier valor
AR3	0x3333	Cualquier valor
AR4	0x4444	Cualquier valor
AR5	0x5555	Cualquier valor
AR6	0x6666	Cualquier valor
AR7	0x7777	Cualquier valor
Nivel 1 de la pila hardware	task	Dirección de inicio del código de la tarea, cuando la instrucción de retorno de interrupción se ejecuta, continua con la ejecución de la instrucción que se encuentra en la dirección de este valor
Nivel 2 de la pila hardware	0x0000	Cualquier valor
Nivel 3 de la pila hardware	0x0000	Cualquier valor
Nivel 4 de la pila hardware	0x0000	Cualquier valor
Nivel 5 de la pila hardware	0x0000	Cualquier valor
Nivel 6 de la pila hardware	0x0000	Cualquier valor
Nivel 7 de la pila hardware	0x0000	Cualquier valor

CAPÍTULO 4

PRUEBA DE FUNCIONAMIENTO DEL SISTEMA OPERATIVO

Para comprobar el funcionamiento del sistema operativo se debe desarrollar una aplicación donde se ocupen recursos de μCOS-II e implementarlo en el TMS320F240. Para esto, se efectuaron dos aplicaciones: una que consiste en hacer que enciendan un banco de LED's que tiene el módulo de evaluación del TMS320F240 y la otra, que es una aplicación a ingeniería eléctrica, consiste en simular la sincronización de un generador al bus infinito

4.1 ENCENDIDO DE LED'S

La aplicación consiste en incrementar el valor que contiene el puerto digital en 1 con un retardo de 0.5 segundos. Para diseñar una aplicación usando el sistema operativo se requiere:

- Configurar μCOS-II.
- Diseñar la función principal.
- Diseñar las tareas.

μCOS-II posee elementos que son configurables, esto se hace en el archivo *OS_CFG.H*. Normalmente, para cada aplicación ó proyecto que se realice debería de tener el suyo. Fue creado para un mejor manejo de la memoria RAM. La tabla 4.1 muestra la configuración de μCOS-II que se utilizó para esta aplicación.

Tabla 4.1 Configuración de constantes de μCOS-II para la aplicación de encendido de LED's.

CONSTANTES	VALOR	DESCRIPCIÓN
OS_MAX_EVENTS	0	No se ocupa ningún tipo de evento.
OS_MAX_MEM_PART	0	No se hace ningún tipo de manejo de partición de memoria.
OS_MAX_QS	0	No se ocupa ninguna cola de mensajes.
OS_MAX_TASKS	1	Máximo número de tareas de aplicación, en este caso se necesita solamente una tarea.
OS_LOWEST_PRIO	63	Especifica la prioridad más baja de las tareas.
OS_TASK_IDLE_STK_SIZE	512	Especifica el tamaño de la pila que tendrá la tarea "ociosa".
OS_TASK_STAT_EN	0	Está deshabilitada la tarea de estadísticas del sistema.
OS_TASK_STAT_STK_SIZE	512	No se usa la tarea de estadísticas del sistema.
OS_CPU_HOOKS_EN	0	Especifica la habilitación de funciones especificadas por el usuario. El 0 indica que no están habilitadas.
OS_MBOX_EN	0	No se ocupa ningún buzón de mensajes.
OS_MEM_EN	0	Está deshabilitado el manejo de partición de memoria
OS_Q_EN	0	No se ocupa ninguna cola de mensajes.
OS_SEM_EN	0	No se ocupa ningún semáforo
OS_TASK_CHANGE_PRIO_EN	0	No se hará ningún cambio de prioridad
OS_TASK_CREATE_EN	1	Se habilita la creación de tareas por medio de la función *OSTaskCreate*.
OS_TASK_CREATE_EXT_EN	0	Está deshabilitada la creación de tareas por medio de la función *OSTaskCreateExt*.
OS_TASK_DEL_EN	0	No se va a usar la función *OSTaskDel* para borrar una tarea.
OS_TASK_SUSPEND_EN	0	No se va a emplear la función *OSTaskSuspend* para suspender una tarea.
OS_TICKS_PER_SEC	61	La función *OSTimeTick* va a ser llamada a una frecuencia de 61 Hertz.

La aplicación solamente requiere de una tarea, la cual va a prender los LED's y va a retardarse asimisma 0.5 segundos. Por lo que la función *main* debe hacer en orden lo siguiente:

- Inicializar el sistema operativo.
- Configurar los periféricos necesarios para la aplicación.
- Crear la tarea.
- Empezar el funcionamiento del sistema operativo.

El código para la función *main* de esta aplicación se muestra en el listado 4.1, donde los comentarios indican que hace cada parte del programa. Las funciones *OSInit* y *OSStart,* no requieren de ningún argumento y son parte de los recursos del sistema operativo. La tarea que se crea usando la función *OSTaskCreate* se le asigna una prioridad de 61. Y el

único periférico que se configura es el sistema guardián, para asegurar que está deshabilitado.

```
void main(void)
        {
        OSInit();                                   /*Inicializa el sistema operativo*/
          asm("    .global  _ConfigureWatchDog");     /*configura el sistema guardian*/
          asm("      call      _ConfigureWatchDog");
 OSTaskCreate(ParpadeoTask, (void *)0, (void *)&ParpadeoStk[0], 61); /*crea la tarea*/

        OSStart();                              /* inicia multitarea */
        }
```

Listado 4.1 Función principal de la aplicación encendido de LED's.

La finalidad de la tarea es obtener una salida por el puerto que contiene los LED's con un retardo de tiempo. El retardo se obtiene a partir del sistema de tiempo que cuenta el sistema operativo. La función para obtener retardos es *OSTimeDly*, que tiene por argumento un valor entero que corresponde al tiempo en *tics* que se retardará. Los *tics* se configuraron, como se muestra en el capítulo 3, a una frecuencia de 61.03515625 Hz (0.016384 s), por lo que un retardo de 0.5 s sería 31 *tics* aproximadamente (0.507904 s). El código de la tarea sería como en el listado 4.2. En el listado se puede ver que se llama a la función *Habilitar*, esta función habilita el sistema *tic*.

```
void ParpadeoTask(void *data)
        {
    volatile INT8U LEDTick=0;                   /*  inicializa la variable a cero    */
    extern void Habilitar(void);
    Habilitar();
            while (1)
               {
               outport(0x000C,LEDTick++);      /* saca el valor de la variable al puerto*/
               OSTimeDly(31);                  /* se retarda 31 tics para continuar      */
               }
        }
```

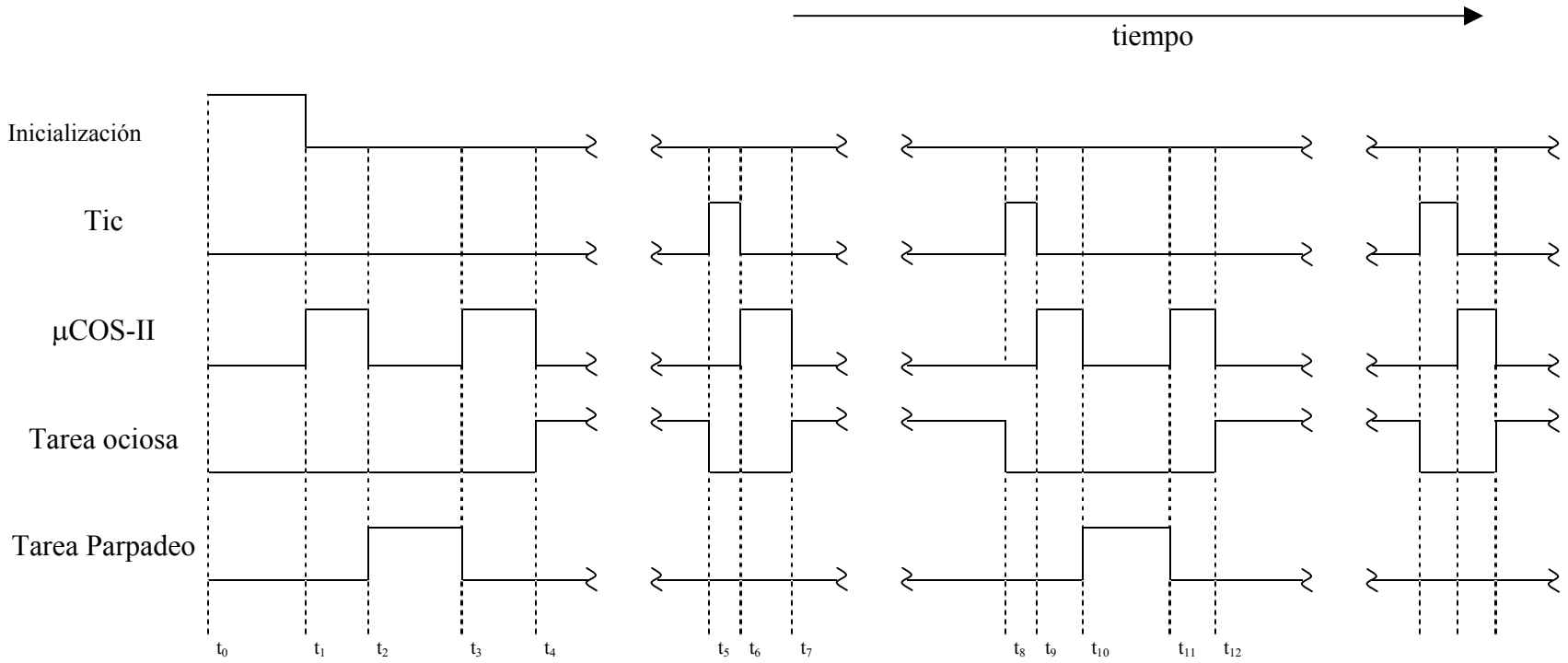
Listado 4.2 Código de la tarea *ParpadeoTask*.

El código fuente de toda la aplicación se muestra en el apéndice C. Con la función *main* y la tarea creada, se compilan todos los archivos que corresponden a la aplicación (función *main*, tarea, archivos de μCOS-II y configuración) para generar el código máquina que se va a ejecutar en el TMS320F240. Al ejecutar el código obtenido, la secuencia de ejecución es como se muestra en la figura 4.1 en el cual sucede lo siguiente:

t_o - t_1, inicializa el TMS320F240, llama a la función *main* y ejecuta *OSInit*, que inicializa las estructuras y variables empleadas por μCOS-II. Además de crear la tarea "ociosa", es decir, le crea su TCB y su pila con valores iniciales para que μCOS-II pueda manejar a la tarea, también la deja en estado preparada para ejecución (esta tarea siempre está preparada para ejecución). También ejecuta la función *ConfigureWatchDog* para deshabilitar el sistema de guardián. Y crea la única tarea de la aplicación, *ParpadeoTask*, es decir se le asigna su TCB y su pila con valores iniciales, y la deja preparada para ejecución.

t_1-t_2, en la última parte de la función *main* (se ejecuta solamente una vez), llama a *OSStart*, la cual ejecuta el sistema de planificación para determinar cuál de las tareas que están preparadas para ejecución es la que tiene prioridad más alta. De las dos tareas creadas, la tarea *ParpadeoTask* es la que tiene mayor prioridad, por lo que el planificador decide ejecutarla. En este momento, se dice que inicia el sistema multitareas.

t_2-t_3, ejecuta la tarea *ParpadeoTask*, la primer función llamada es *Habilitar* (se ejecuta una vez), la cual activa la interrupción en tiempo real que es la generadora del tic. Se habilita la interrupción de tiempo real hasta este momento, porque ya está iniciado el sistema de multitareas y hay una tarea del usuario en ejecución. Si se ejecutara antes, podría causar problemas al ejecutar el código correspondiente a *OSTickISR*. Después, se ejecuta la parte del código de la tarea que va a ser manejada por μCOS-II. Se envía el contenido de la variable *LEDTick* al puerto correspondiente a los LED's y se incrementa en 1 a la variable. Esta primera vez que se ejecuta la tarea el valor es 0, por lo que todos los LED's estarán apagados. Ejecuta el retardo, dejando a *Parpadeotask* en espera y llama al sistema de planificación.



NOTA: Un nivel alto significa que se está ejecutando el código y un nivel bajo no se está ejecutando el código. El procesador no ejecuta dos códigos a la vez y tampoco puede dejar de ejecutar código en ningún momento.

Figura 4.1 Secuencia de ejecución de la aplicación encendido de LED's.

t_3-t_4, el planificador de μCOS-II, busca la tarea de prioridad más alta preparada para ejecución y concluye que debe continuar en ejecución la tarea "ociosa" (no hay otra tarea prepara para ejecución).

t_4-t_5, ejecuta la tarea ociosa hasta que ocurre la interrupción periódica parte del tic.

t_5-t_6, ejecuta la ISR del tic y llama al planificador.

t_6-t_7, el planificador busca la tarea de prioridad más alta lista para ejecución y concluye que debe continuar en ejecución la tarea "ociosa" (la tarea *ParpadeoTask* continua retardada).

t_7-t_8, ejecuta la tarea "ociosa" hasta que ocurre nuevamente la interrupción periódica parte del tic. Y se repite la ejecución de t_5 a t_8 hasta completar 31 tics debido a que se ha retardado la tarea *ParpadeoTask*. En t8, ocurre la interrupción periódica parte del tic.

t_8-t_9, ejecuta la ISR del tic y llama al planificador.

t_9-t_{10}, el planificador busca la tarea de prioridad más alta preparada para ejecución. Y decide que la tarea que se va a ejecutar es *ParpadeoTask*.

t_{10}-t_{11}, continua su ejecución la tarea *ParpadeoTask* después de completar su retardo. Se envía el contenido de la variable *LEDTick* al puerto correspondiente a los LED's y se incrementa en 1 a la variable. Ahora, el valor es 1, por lo que se prende uno de los LED's. Ejecuta el retardo, dejando a *Parpadeotask* en espera y llama al sistema de planificación.

t_{11}-t_{12}, el planificador busca la tarea de prioridad más alta lista para ejecución y concluye que debe continuar en ejecución la tarea "ociosa" (la tarea *ParpadeoTask* continua retardada). Después, se repite de t_5 a t_{12} periódicamente.

El resultado del programa al ejecutarse, observando el módulo de evaluación, es lo siguiente: los LED's que corresponden al puerto prenden con un retardo de 0.5 segundos iniciando con todos los LED's apagados y posteriormente se van prendiendo en forma de conteo binario (0 apagado y 1 encendido), como se muestra en la figura 4.2.

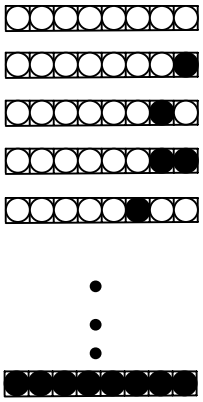


Figura 4.2 Secuencia de encendido del banco de LED's del módulo de evaluación del TMS320F240.

4.2 SINCRONIZACIÓN DE UN GENERADOR

Para simular la sincronización de un generador al bus infinito se emplearon dos generadores de funciones conectados al módulo de evaluación del TMS320F240, como se muestra en la figura 4.3. El generador de funciones 1 está conectado al convertidor analógico digital 1 del TMS320F240 (ADC1) y el generador 2 al convertidor analógico digital 2 del TMS320F240 (ADC2). El generador de funciones 1 simula al bus infinito y el generador de funciones 2 simula un generador. De esta forma, el generador de funciones 1 debe permanecer su valor de tensión y frecuencia sin cambio para simular al bus (para este caso se empleó una señal como la mostrada en la figura 4.4) y el generador de funciones 2 debe variar su frecuencia y tensión.

El módulo de evaluación se encarga de indicar el momento en que el generador 2 se encuentra sincronizado al generador1, es decir el generador 2 tenga un valor de tensión y frecuencia semejante al generador 1, además de estar en fase.

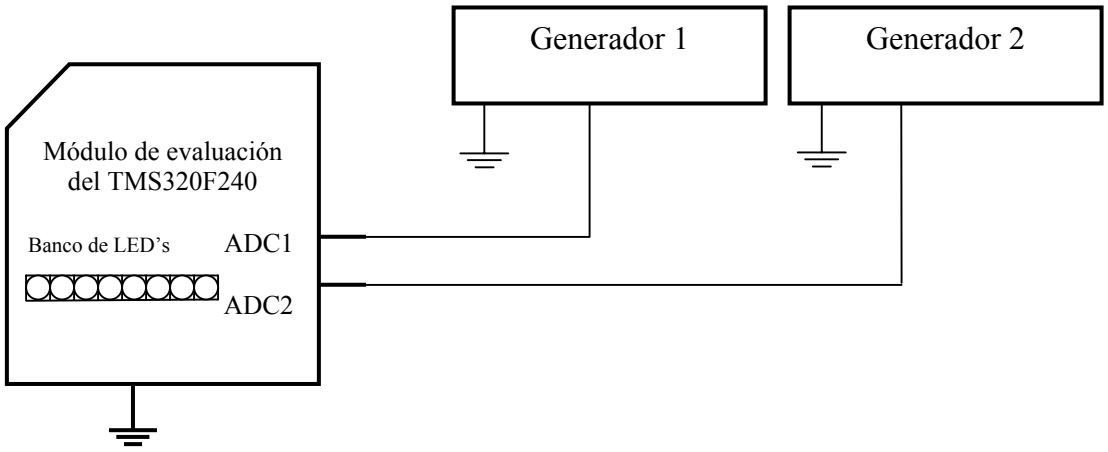


Figura 4.3 Conexión de dos generadores al módulo de evaluación del TMS320F240.

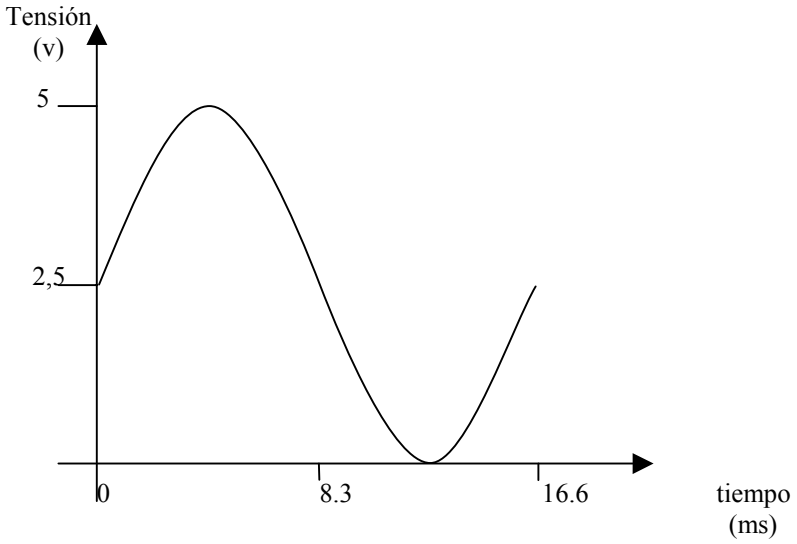


Figura 4.4 Señal del generador que simula el bus infinito.

Para desarrollar la aplicación se basó en la de encendido de LED's, dejando la tarea *ParpadeoTask* como parte de esta aplicación. Aunque no tiene que ver con la esta, nos indica visualmente si la aplicación está funcionando. La aplicación se diseñó con una tarea y una interrupción. La tarea se le llamó *Sincroniza* y la interrupción se genera por el sistema de conversión analógico-digital (ADC).

La tarea *Sincroniza* y la ISR del ADC están sincronizadas, como se muestra en la figura 4.5. Para que la tarea *Sincroniza* se ejecute, la ISR del ADC debe mandarle una señal por medio de uno de los servicios del sistema operativo, en este caso se usó un semáforo. La ISR se emplea para hacer el muestreo de las señales de los generadores y la tarea *Sincroniza* se encarga de realizar los cálculos pertinentes para determinar si los dos generadores están en fase y con valores de tensión y frecuencia semejantes, para que al momento de lograr estas condiciones la tarea *Sincroniza* lo indique prendiendo todos los LED's.

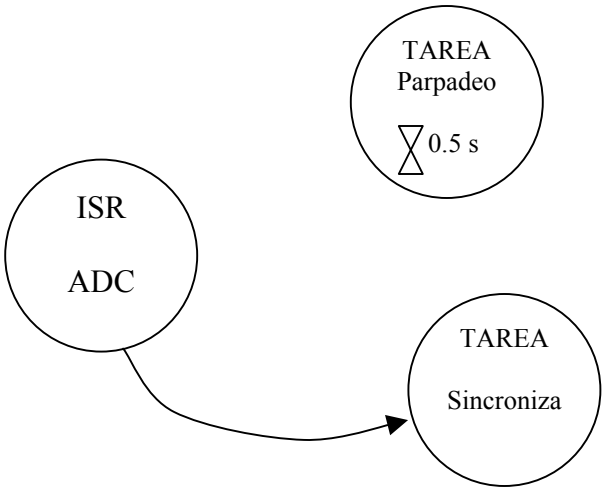


Figura 4.5 Diagrama de la sincronización de un generador.

Para desarrollar esta aplicación usando μCOS-II se requiere:

- Configurar μCOS-II.
- Diseñar la función *main*.
- Diseñar las tareas.
- Diseñar la ISR del ADC.

La tabla 4.2 muestra la configuración de μCOS-II que se utilizó para esta aplicación, la cual es ligeramente diferente a la tabla 4.1, ya que se ocupan dos tareas, un semáforo y una interrupción.

Tabla 4.2 Configuración de constantes de µCOS-II para la sincronización de un generador.

CONSTANTES	VALOR	DESCRIPCIÓN
OS_MAX_EVENTS	2	Se va a ocupar un semáforo. El valor mínimo que se debe tener es 2 cuando se va a ocupar algún evento.
OS_MAX_MEM_PART	0	No se hace ningún tipo de manejo de partición de memoria.
OS_MAX_QS	0	No se ocupa ninguna cola de mensajes.
OS_MAX_TASKS	2	Máximo número de tareas de aplicación, en este caso se necesita solamente una tarea.
OS_LOWEST_PRIO	63	Especifica la prioridad más baja de las tareas.
OS_TASK_IDLE_STK_SIZE	512	Especifica el tamaño de la pila que tendrá la tarea "ociosa".
OS_TASK_STAT_EN	0	Está deshabilitada la tarea de estadísticas del sistema.
OS_TASK_STAT_STK_SIZE	512	No se usa la tarea de estadísticas del sistema.
OS_CPU_HOOKS_EN	0	Especifica la habilitación de funciones especificadas por el usuario. El 0 indica que no están habilitadas.
OS_MBOX_EN	0	No se ocupa ningún buzón de mensajes.
OS_MEM_EN	0	Está deshabilitado el manejo de partición de memoria
OS_Q_EN	0	No se ocupa ninguna cola de mensajes.
OS_SEM_EN	1	Se habilita la generación de código para el manejo de semáforos.
OS_TASK_CHANGE_PRIO_EN	0	No se hará ningún cambio de prioridad
OS_TASK_CREATE_EN	1	Se habilita la creación de tareas por medio de la función *OSTaskCreate*.
OS_TASK_CREATE_EXT_EN	0	Está deshabilitada la creación de tareas por medio de la función *OSTaskCreateExt*.
OS_TASK_DEL_EN	0	No se va a usar la función *OSTaskDel* para borrar una tarea.
OS_TASK_SUSPEND_EN	0	No se va a emplear la función *OSTaskSuspend* para suspender una tarea.
OS_TICKS_PER_SEC	61	La función *OSTimeTick* va a ser llamada a una frecuencia de 61 Hertz.

La función *main* tiene el mismo código que la aplicación anterior, pero se adiciona la creación de la tarea *Sincroniza* con prioridad 62 y la creación de un semáforo llamado *SEMMEDICION*. La función *OSSemCreate* se utiliza para crear el semáforo, la función tiene un argumento que es un valor entero, en este caso se le dio el valor inicial de 0, que indica que no está liberado el semáforo *SEMMEDICION*. El código de la función *main* se muestra en el apéndice D.

La tarea Parpadeo es la misma que *ParpadeoTask*, pero en esta se le agregó la configuración del sistema ADC del TMS320F240 llamando la función *ConfiguraADC*. El sistema ADC fue configurado para que el manejador de eventos inicie la conversión de ADC1 y ADC2, con una frecuencia de 960 Hz y finalizada las conversiones se genere una

interrupción. El código de la función *ConfiguraADC* se muestra en el apéndice D en *ini.asm.*

La tarea *Sincroniza* indica el momento en el cual se considera que las señales de los dos generadores están sincronizados, de acuerdo al diagrama de flujo mostrado en la figura 4.6. En el inicio de la tarea *Sincroniza*, inicializa variables. Después, inicia la parte de código que es controlado por μCOS-II. La espera de la señal de la ISR del ADC se hace con la función *OSSemPed*, si el semáforo está liberado la tarea continúa su ejecución, sino se mantiene en el estado de espera hasta que este sea liberado por la ISR del ADC con la función *OSSemPost*.

El cálculo del valor RMS de la diferencia de tensión entre los generadores se efectuó con el algoritmo recursivo empleado en [6]:

$$X'_{RMSk} = X'_{RMSk} + X_K^2 - X_{K-N}^2 \qquad (2)$$

$$X_{RMSk}^2 = \frac{X'_{RMSk}}{N} \qquad (3)$$

Donde X_{RMSk}, representa el valor RMS, X_K es la muestra actual, X_{K-N} es la muesta N y, $X'_{RMSk}=0$ y $X_{-1}, X_{-2,}, ..X_{-N}=0$.

El método empleado para estimar la frecuencia, es el de cruce por cero modificado [7]. El método consiste en aplicar interpolación lineal entre las dos muestras más cercanas laterales del nivel de cero como se muestra en la figura 4.7. El periodo de la señal (T), que va desde t_2 hasta t_5, es la suma de las fracciones de tiempo de muestreo t_2-t_3 y t_4-t_5 más el número de muestras tomadas entre t_3 y t_4. Las fracciones de tiempo de t_2-t_3 y t_4-t_5 se calcula de la siguiente forma:

$$t_{23} = \frac{N_3}{N_3 - N_2}$$

$$t_{45} = \frac{N_4}{N_4 - N_5}$$

Donde N_2, N_3, N_4 y N_5 son los niveles de la señal en los tiempos t_2, t_3, t_4 y t_5 respectivamente. El código de la tarea *Sincroniza* se muestra en el apéndice D.

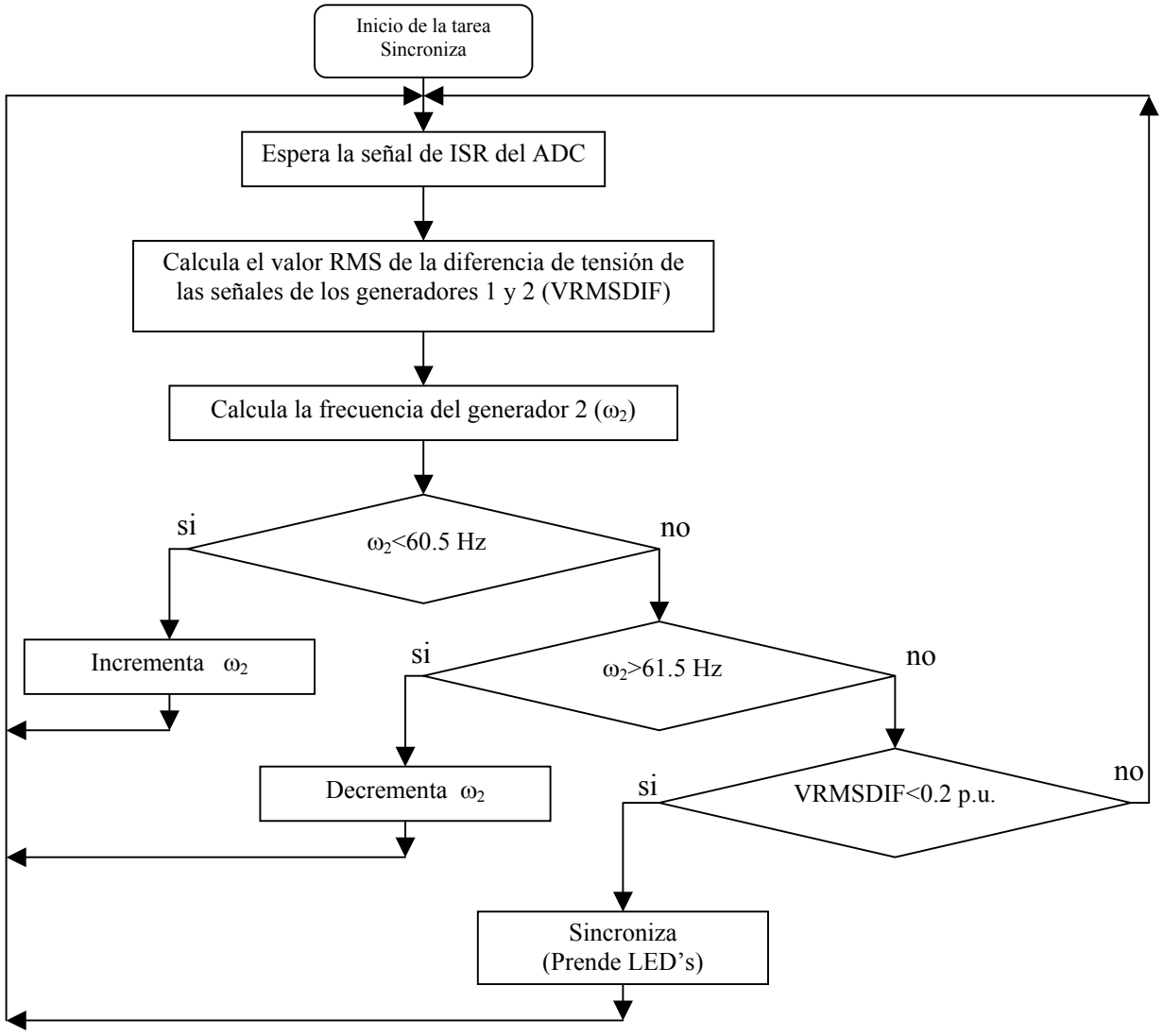


Figura 4.6 Diagrama de flujo de la tarea Sincroniza.

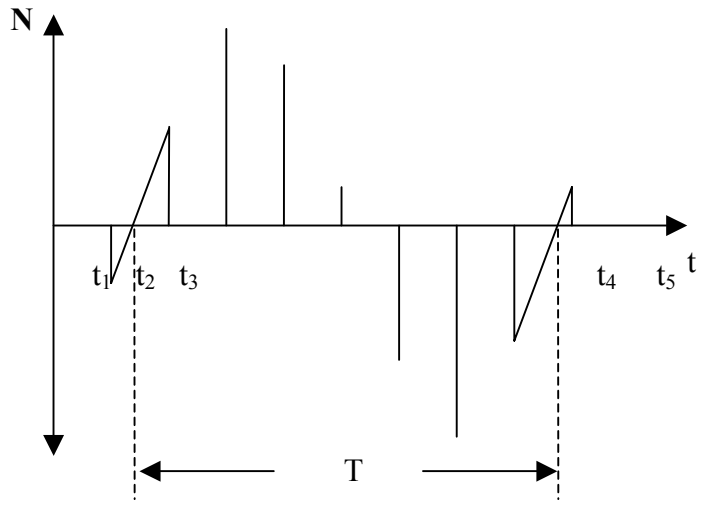


Figura 4.7 Cálculo del periodo de una señal por cruce por cero modificado.

La secuencia de ejecución de esta aplicación no se puede analizar tan fácilmente, debido a que hay dos eventos que ocurren periódicamente, pero en diferentes tiempos. La interrupción del tic que ocurre a una frecuencia de 61 Hz y la interrupción del sistema ADC que se presenta a 960 Hz. Sin embargo, se puede analizar la secuencia de ejecución del inicio de la aplicación, la que se muestra en la figura 4.8 y hace lo siguiente.

t_0-t_1, Inicializa el TMS320F240, llama a la función *main* y ejecuta *OSInit*, que inicializa las estructuras y variables empleadas por μCOS-II. Además de crear la tarea "ociosa", es decir, le crea su TCB y su pila con valores iniciales para que μCOS-II pueda manejar a la tarea, también la deja en estado preparada para ejecución (esta tarea siempre está preparada para ejecución). Además, ejecuta la función *ConfigureWatchDog* para deshabilitar el sistema de guardián. Y crea la tarea *Parpadeo*, la tarea Sincroniza y el semáforo *SEMMEDICION*.

t_1-t_2, en la última parte de la función *main,* la cual se debe ejecutar solamente una vez, llama a *OSStart*. Este servicio llama al sistema de planificación para determinar cuál de las tareas que están preparadas para ejecución es la que tiene prioridad más alta. De las tareas creadas, la tarea *Sincroniza* es la que tiene mayor prioridad, por lo que el planificador decide ejecutarla, iniciando el sistema multitareas.

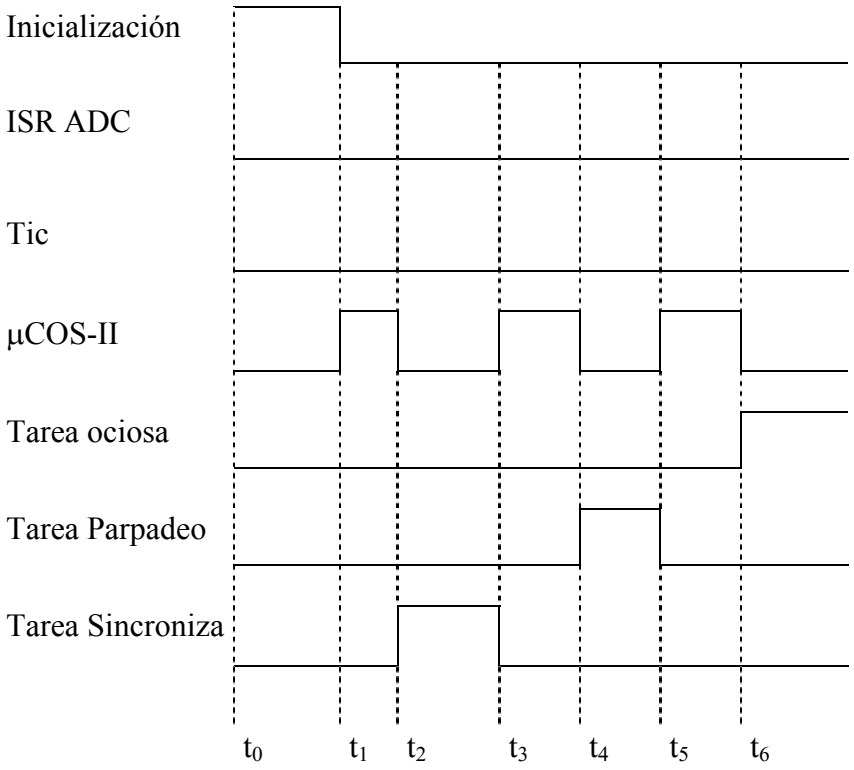


Figura 4.8 Secuencia de ejecución inicial de la aplicación sincronización de un generador.

t_2-t_3, ejecuta la tarea *Sincroniza*, inicializando variables y llama a la función OSSemPend, para preguntar si el semáforo está liberado y en este momento no lo está, por lo que la tarea *Sincroniza* pasa al estado de espera. Después, llama al planificador.

t_3-t_4, el planificador de μCOS-II, busca la tarea de prioridad más alta preparada para ejecución y concluye que es la tarea *Parpadeo,* dándole el control de la CPU.

t_4-t_5, ejecuta la tarea *Parpadeo*, la primera función llamada es *ConfiguraADC*, configurando el sistema ADC y después, ejecuta la función *Habilitar*, que habilita las interrupciones del sistema ADC y tic. Posteriormente, se ejecuta la parte del código de la tarea que va a ser manejada por μCOS-II. Se envía el contenido de la variable *LEDTick* al puerto correspondiente a los LED's y se incrementa en 1 a la variable. Esta primera vez que se ejecuta la tarea el valor es 0, por lo que todos los LED's estarán apagados. Ejecuta el retardo, dejando a la tarea *Parpadeo* en espera y llama al sistema de planificación.

t_5-t_6, el planificador de μCOS-II, busca la tarea de prioridad más alta preparada para ejecución y concluye que debe continuar en ejecución la tarea "ociosa" (no hay otra tarea prepara para ejecución).

Después de t_6, hay varias posibilidades de ejecución, dependiendo el evento: cuando ocurre un tic y cuando hay un fin de conversión en el sistema ADC. Sobresaliendo, la que ocurre cuando hay una interrupción por el fin de conversión del sistema ADC. La secuencia de ejecución se muestra en la figura 4.9 y es como sigue:

t_7-t_8, en t_7 ocurre una interrupción generada por el sistema ADC, en la ISR se llama a la función *OSIntEnter* para que μCOS-II se entere de que ha ocurrido una interrupción, después se realiza se procesa las señales muestreadas. Se llama a la función *OSSemPost*, para liberar el semáforo (dando la señal de continuación a la tarea *Sincroniza*, pasándola al estado preparada para ejecución). Finalmente, se ejecuta la función *OSIntExit* para indicar la salida de la ISR y llama al planificador.

t_8-t_9, el planificador determina cuál de las tareas preparadas para ejecución es la que tiene mayor prioridad. La tarea que puede asegurarse que está prepara para ejecución en este momento es la tarea *Sincroniza* y no importa que otra tarea se encuentra en el mismo estado porque la tarea *Sincroniza* es la que tiene la mayor prioridad de todas las que fueron creadas. Por lo tanto, el planificador le da el control de la CPU a esta tarea.

t_9-t_{10}, en la tarea *Sincroniza*, se determina si ya se considera que están sincronizados los generadores, y después regresa a llamar la función *OSSemPost* para esperar el muestreo, dejando a la tareas *Sincroniza* nuevamente en el estado de espera y llama al planificador.

t_{10}-t_{11}, el planificador busca nuevamente, la tarea de prioridad más alta que esté en el estado preparada para ejecución. Y puede ser la tarea "ociosa" ó la tarea *Parpadeo*.

El resultado del programa al ejecutarse, observando el módulo de evaluación, es lo siguiente: los LED's que corresponden al puerto prenden con un retardo de 0.5 segundos iniciando con todos los LED's apagados y posteriormente se van prendiendo en forma de conteo binario (0 apagado y 1 encendido), como se muestra en la figura 4.2, pero si los generadores están sincronizados este hará que prendan todos los LED's sin importar cual sea la secuencia. Y si el generador 2 se sale de sincronía los LED's continúan su conteo.

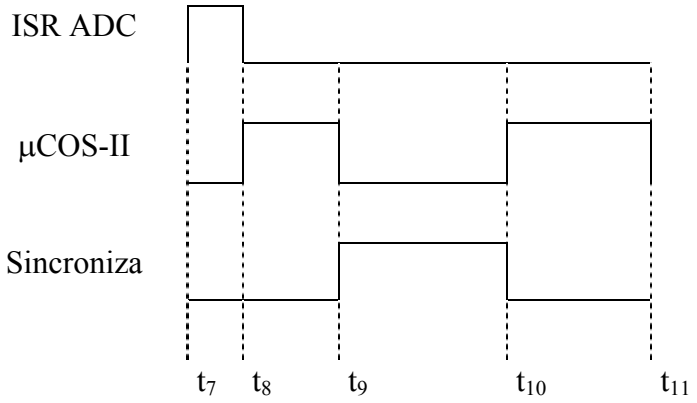


Figura 4.9 Secuencia de ejecución de la aplicación sincronización de un generador cuando ocurre una interrupción generada por el sistema ADC.

CAPÍTULO 5

CONCLUSIONES, APORTACIONES Y TRABAJOS FUTUROS

5.1 CONCLUSIONES

Para diseñar e implementar el sistema operativo basado en μCOS-II en el TMS320F240, se necesita conocer la arquitectura del procesador, las herramientas de programación (ensamblador y compilador C) y su relación entre las mismas, además de entender la filosofía de programación de μCOS-II.

Para desarrollar aplicaciones con el sistema operativo como las desarrolladas en la tesis y aun más complejas, se deben conocer todas las capacidades que tiene el sistema operativo, es decir todos los servicios que ofrece. Esto es, porque cada tarea necesita ganar el control del procesador y la única forma de que pueda ganarlo es llamando uno de los servicios. Los servicios son el contacto que tiene la tarea con el núcleo de programación.

Al incluir la aplicación *encendido de LED´s* en la aplicación de *sincronización de un generador*, refleja la independencia que tiene las tareas entre sí y la flexibilidad para el mantenimiento, modificación ó actualización de las aplicaciones desarrolladas en el sistema operativo.

5.2 APORTACIONES

- Se tiene un sistema operativo de tiempo real para el TMS320F240 basado en de μCOS-II, del cual no se tiene referencia.

- Ejemplos de aplicación basados en un sistema operativo de tiempo real que pueden servir de guía para aplicaciones más complejas.

- Análisis de la secuencia de ejecución de las dos aplicaciones que ayudan a entender la filosofía de el uso del sistema operativo.

5.3 TRABAJOS FUTUROS

- Implementar el sistema operativo en otros procesadores. Cada vez hay mejores herramientas de programación que facilitan la implementación de las aplicaciones y procesadores con mejor desempeño en el cual se puede implementar.

- Desarrollar aplicaciones más complejas usando un sistema operativo de tiempo real.

- Analizar otros núcleos de programación.

REFERENCIAS

[1] Guevara López, Pedro y José de Jesús Medel Juárez. "*Introducción a los Sistemas en Tiempo Real*". IPN, México, D. F. 2003.

[2] Embedded System. Sitio Web "http://www.embedded.com".

[3] *MicroC/OS-II.* Sitio Web "http://www.micrium.com".

[4] Labrosse, Jean J. "*MicroC/OS-II, The Real-Time Kernel*". R&D Books. 1999.

[5] Texas Instruments. "*TMS320C2x/C2xx/C5x Optimizing C Compiler, User's Guide*". 1995.

[6] Cortés Mateos, Raúl; Domitilo Libreros y Rafael San Vicente Cisneros. "*Medición del valor RMS del Voltaje y Corriente con el DSP TMS320C240*". 4º Congreso Nacional de Ingeniería Electromecánica y de sistemas, ESIME-SEPI, México D. F., Nov 1998, pags 281-284.

[7] Backmutsky, V y V. Zmudikov. "*Accurate Frequency Estimation in Power Systems by DSP*", Proc. 18th IEEE Conf. of Israel, Tel-Aviv, March 1995, 5.2.4, 5 pp.

[8] Ganssle, Jack G. "*The art of designing embedded systems*". Boston, Mass. Newnes, 1999.

[9] Kernighan, W. Brian y Dennis M. Ritchie. "*El Lenguaje de Programación* C". Segunda Edición. Pearson Educación.1991.

[10] Liu, Jane W. S . "*Real-Time systems*". Prentice Hall, 2000.

[11] Saucedo Martínez, David Jaime y Domitilo Libreros. *"Sistemas Operativos de Tiempo Real y su Aplicacion al Monitoreo de Subestaciones Eléctricas*". 4º Congreso Nacional de Ingeniería Electromecánica y de sistemas, ESIME-SEPI, México D. F., Nov 1998, pags 336-342.

[12] Saucedo Martínez, David Jaime. *"Sistemas Operativos de Tiempo Real y su Aplicacion a un Control Supervisorio*". Tesis para obtener el Grado de Maestro en Ciencias, México, D. F. 1996.

[13] Simon, David E. "*An Embedded Software Primer*". Addison-Wesley, 2001.

[14] Stallings, William. "*Sistemas Operativos*". Madrid, PRENTICE HALL, 1997.

[15] Tanenbaum, A.S, "*Sistemas Operativos Modernos*". Segunda Edición, Prentice-Hall, 2003.

[16] Texas Instruments. "*DSP Controllers Reference Set Volumen 1. CUP System and Instruction",* Diciembre 1997.

[17] Texas Instruments. "*DSP Controllers Reference Set Volumen 2. Peripherial Library and Specific Devices",* Diciembre 1997.

[18] Texas Instruments. "*TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide*". 1995.

[19] Wellings, Andy y Alan Burns,. "*Real-time systems and programming languages*". Addison-Wesley, Harlow, England, 1996.

[20] Wu Astro. "*Designing an Embedded Operating System with the TMS320 Family of DSPs*". Application Brief: SPRA296. DSP Applications, TI Asia, 1998.

APÉNDICE A

CARACTERÍSTICAS GENERALES DEL DSP TMS320F240 Y SU COMPILADOR C

El DSP TMS320F240 es un procesador de señales digitales que tiene dentro del mismo encapsulado periféricos, por lo que se dice que es un DSP-microcontrolador. Es fabricado por Texas Instruments y pertenece a la familia de los DSP de la serie TMS320, diseñado especialmente para aplicaciones de control de motores. Este cuenta con varias herramientas de desarrollo para su programación, entre ellas se encuentra un compilador C que cumple con el estándar ANSI.

CARACTERÍSTICAS GENERALES DEL DSP TMS320F240

UNIDAD DE PROCESAMIENTO CENTRAL DEL TMS320F240

- Unidad aritmética lógica central de 32 bits.
- Acumulador de 32 bits.
- Multiplicador paralelo de 16 bits x 16 bits con un resultado de 32 bits.
- Tres registros de corrimiento para escalamiento.
- 8 registros auxiliares de 16 bits con una unidad aritmética dedicada para direccionamiento indirecto de memoria de datos.

MEMORIA:

- 544 palabras de 16 bits en la pastilla de memoria RAM de acceso dual para datos ó programa.
- 16K palabras de 16 bits en la pastilla de memoria flash EEPROM para programa.
- 224K palabras de 16 bits de máximo direccionamiento de espacio de memoria (64K palabras de espacio para programa, 64K palabras para espacio de datos, 64K palabras para espacio de entrada/salida, y 32K palabras de espacio global).
- Módulo de interfase de memoria externa con un generador de estados de espera por software, un bus de direcciones de 16 bits, y un bus de datos de 16 bits.
- Soporte de estados de espera físico.

CONTROL DE PROGRAMA:

- Operación pipeline de cuatro niveles.
- Pila por hardware de ocho niveles.
- Seis interrupciones externas: interrupción de protección para manejo de alimentación, reinicialización, interrupción no mascarable y tres interrupciones mascarables.

CONJUNTO DE INSTRUCCIONES:

- Código fuente compatible con generaciones de DSP's de la familia TMS320 de punto fijo (C2x, C2xx y C5x).
- Operación de repetir una sola instrucción.
- Instrucciones multiplicar/acumular de un ciclo.
- Instrucciones de movimiento de bloque de memoria para manejo de programa/datos.
- 3 Instrucciones de interrupción por software.
- Capacidad de direccionamiento indexado.
- Capacidad de direccionamiento indexado especial para aplicar transformada rápida de Fourier índice-2.

POTENCIA:

- Tecnología CMOS estática.
- Cuatro modos de ahorro de energía.
- Emulación: estándar IEEE 1149.1

MANEJADOR DE EVENTOS:

- 12 canales de comparación/modulación de ancho de pulso (PWM), de los cuales 9 son independientes.
- Tres temporizadores de propósito general de 16 bits con seis modos, incluyendo conteo ascendente continuo y conteo ascendente/descendente continuo.
- Tres unidades de comparación completa de 16 bits con capacidad de banda muerta.
- Tres unidades de comparación simple de 16 bits.
- Cuatro unidades de captura.

OTRAS:

- Velocidad: Tiempo de ciclo de instrucción 50 ns (20 MIPS), con la mayor parte de instrucciones de un ciclo.
- Arquitectura tipo Harvard modificado.
- Dos convertidores analógico digital de 10 bits.
- 28 terminales de entrada/salida multiplexadas, individualmente programables.
- Módulo de reloj basado en un bucle de captura de fase (PLL).
- Módulo temporizador guardián (watchdog) con interrupción en tiempo real.
- Interface de comunicación serial (SCI).
- Interface periférica serial (SPI).

REGISTROS INTERNOS

ACUMULADOR: Registro de 32 bits que se encuentra en la sección de la unidad aritmética lógica central de la CPU. Recibe la salida de la unidad aritmética lógica central y es capaz de la ejecución de corrimientos de bits sobre su contenido con la ayuda del bit de acarreo. Algunas instrucciones dividen el acumulador en dos partes iguales, la parte alta (ACCH) y la parte baja (ACCL).

CONTADOR DE PROGRAMA (PC): La lógica de generación de dirección de programa usa el contador de programa de 16 bits para direccionar memoria de programa interna y externa. El PC mantiene la dirección de la próxima instrucción a ser ejecutada.

PILA: El DSP tiene una pila física de 8 niveles de profundidad con un ancho de 16 bits. La lógica de generación de dirección-programa la usa para almacenamiento de dirección de retorno en la parte alta de la pila, cuando ocurre una llamada a subrutina ó una interrupción. Cuando no está usándose para estos propósitos se puede usar como almacenamiento temporal de datos.

REGISTRO TEMPORAL (TREG): Registro temporal de 16 bits que pertenece a la sección de multiplicación, mantiene uno de los multiplicandos. El otro viene de memoria de datos o programa.

REGISTRO PRODUCTO (PREG): Registro de 32 bits que recibe el resultado de la multiplicación. Se ubica en la sección de multiplicación.

REGISTROS AUXILIARES (AR7-AR0): Son registros de 16 bits que proveen un flexible y poderosos direccionamiento indirecto. Se puede acceder a cualquier localidad en el espacio de memoria de datos de 64K usando el contenido de un registro auxiliar.

REGISTROS DE ESTADO (ST0 y ST1): Este DSP tiene dos registros de estado, ST0 y ST1, estos contienen el estado y bits de control. Ambos tienen un tamaño de 16 bits. La figura A.1 y A.2 muestran la organización estos registros, y la descripción de cada bit se muestra en la tabla A.1.

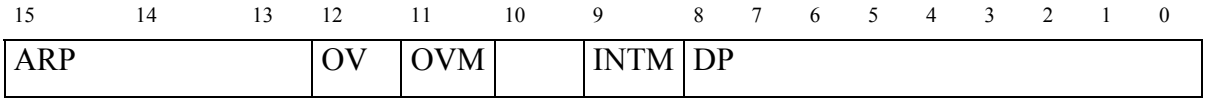


Figura A.1 Registro de estado ST0.

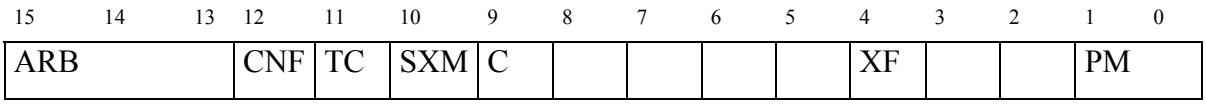


Figura A.2 Registro de estado ST1.

Tabla A.1 Descripción de los bits de los registros de estado ST0 y ST1.

Nombre	Descripción
ARB	Buffer del apuntador del registro auxiliar. Contiene el valor previo del apuntador del registro auxiliar (ARP).
ARP	Apuntador del registro auxiliar. Este campo de 3 bits selecciona que registro auxiliar (AR) se usará en modo de direccionamiento indirecto.
C	Bit de acarreo. Este bit se pone en 1 si el resultado de una suma genera un acarreo, ó se pone en 0 cuando el resultado de una resta genera un préstamo.
CNF	Bit de configuración de la memoria tipo DARAM que está en la pastilla. Si se le asigna un 0 esta RAM de acceso dual se mapea en espacio de datos; y si se le asigna un 1 se mapea en espacio de programa.
DP	Apuntador de página de datos. Si una instrucción usa modo de direccionamiento directo, el campo de 9 bits de DP se concatena con los 7 bits menos significativos de la palabra de instrucción para formar una dirección de memoria de datos completa de 16 bits.
INTM	Bit de modo de interrupción. Habilita o deshabilita las interrupciones mascarables.

Tabla A. 1 (Continuación).

Nombre	Descripción
OV	Bit de bandera de sobreflujo.
OVM	Bit de modo de sobreflujo. Si es 0 el resultado del sobreflujo queda en forma normal en el acumulador. Si es 1 el resultado del sobreflujo se indica llevando el contenido del acumulador al valor más positivo ó negativo dependiendo como haya ocurrido el sobreflujo.
PM	Modo del corrimiento del producto. PM determina el monto que el valor de PREG será corrido para que sea llevado a la Unidad aritmética lógica central ó a memoria de datos.
SXM	Bit de modo de extensión de signo. Si es cero suprime la extensión de signo, sino produce la extensión de signo.
TC	Bit de bandera de prueba/control.
XF	Bit de estado del pin XF.

COMPILADOR "C" PARA EL TMS320F240

Programar un DSP con un lenguaje de alto nivel como el C, proporciona transportabilidad (capacidad de llevar un mismo código de programa a diferentes procesadores) y facilidad de mantenimiento. Un programa puede ser rápidamente diseñado como un prototipo en lenguaje C y posteriormente, optimizado a un procesador específico.

Texas Instruments proporciona el compilador C TMS320C2X/C2XX/C5X, que convierte programas en estándar ANSI C a programas en lenguaje ensamblador ó directamente a lenguaje máquina, para los DSP's de la familia TMS320C2X, TMS320C2XX y TMS320C5X.

Aunque el uso de esta herramienta da muchas ventajas para el diseño de aplicaciones, esta fue creada para toda una familia de procesadores, por lo que se necesita

ciertas consideraciones para que definan al procesador seleccionado y la circuitería externa a la pastilla (periféricos y memoria externa) que contiene al procesador.

Para definir al procesador que fue seleccionado, los periféricos y la memoria externa se necesita realizar lo siguiente:

- Crear un programa para los vectores de interrupción,
- crear un programa para configuración de periféricos,
- y especificar el mapa de memoria.

Cada procesador tiene diferente ubicación de los vectores de interrupción en el mapa de memoria, en el caso del TMS320F240 los vectores inician en la dirección 0x0 en memoria de programa, como se puede ver en el mapa de memoria del módulo de evaluación del procesador (figura A.3). Un método para ubicar apropiadamente los vectores de interrupción consiste en crear un archivo separado del programa principal, tal archivo tendría una tabla de vectores escritos en lenguaje ensamblador. En el apéndice C y D se muestra la de tabla de vectores de dos aplicación en el TMS320F240 (vector.asm).

La mayoría de las aplicaciones usan las unidades de entrada/salida, por lo que se necesita configurarlos. Para facilidad esto se logra creando un programa en lenguaje ensamblador. Aunque la aplicación no requiera de periféricos, se necesita configurar cuando menos el sistema guardián de la integridad del sistema (watchdog). Además, se recomienda crear este programa como un archivo separado, para después enlazarse con los demás archivos de la aplicación que se esté desarrollando. En el apéndice C y D, se muestra un programa de este tipo (*init.asm*).

Otra consideración muy importante es la especificación del mapa de memoria, ya que cada sistema normalmente tiene un mapa de memoria diferente. Para el caso del módulo de evaluación del TMS320F240 el mapa de memoria se muestra en la figura A.3, como ya se había mencionado. Esto es importante ya que en este caso, el sistema de memoria se compone de varios tipos, además de ser externas e internas. De esta manera se

decide que código ó datos se va a escribir en cada memoria del sistema de memoria, con el fin de manejar al sistema de memoria en forma más eficiente. La especificación del mapa de memoria se hace en el archivo de comando del enlazador (para identificarlo el archivo tiene la extensión *.cmd*). Este archivo además de definir el mapa de memoria, como su nombre lo indica, se usa para dar comandos al enlazador para crear el archivo objeto (lenguaje máquina y datos que se guardan en el procesador). El compilador usa 7 secciones para guardar el código generado (.text, .cinit, .switch, .const, .bss, .stack y .sysmem). Ejemplos de este archivo puede verse en el apéndice C y D (emv.cmd).

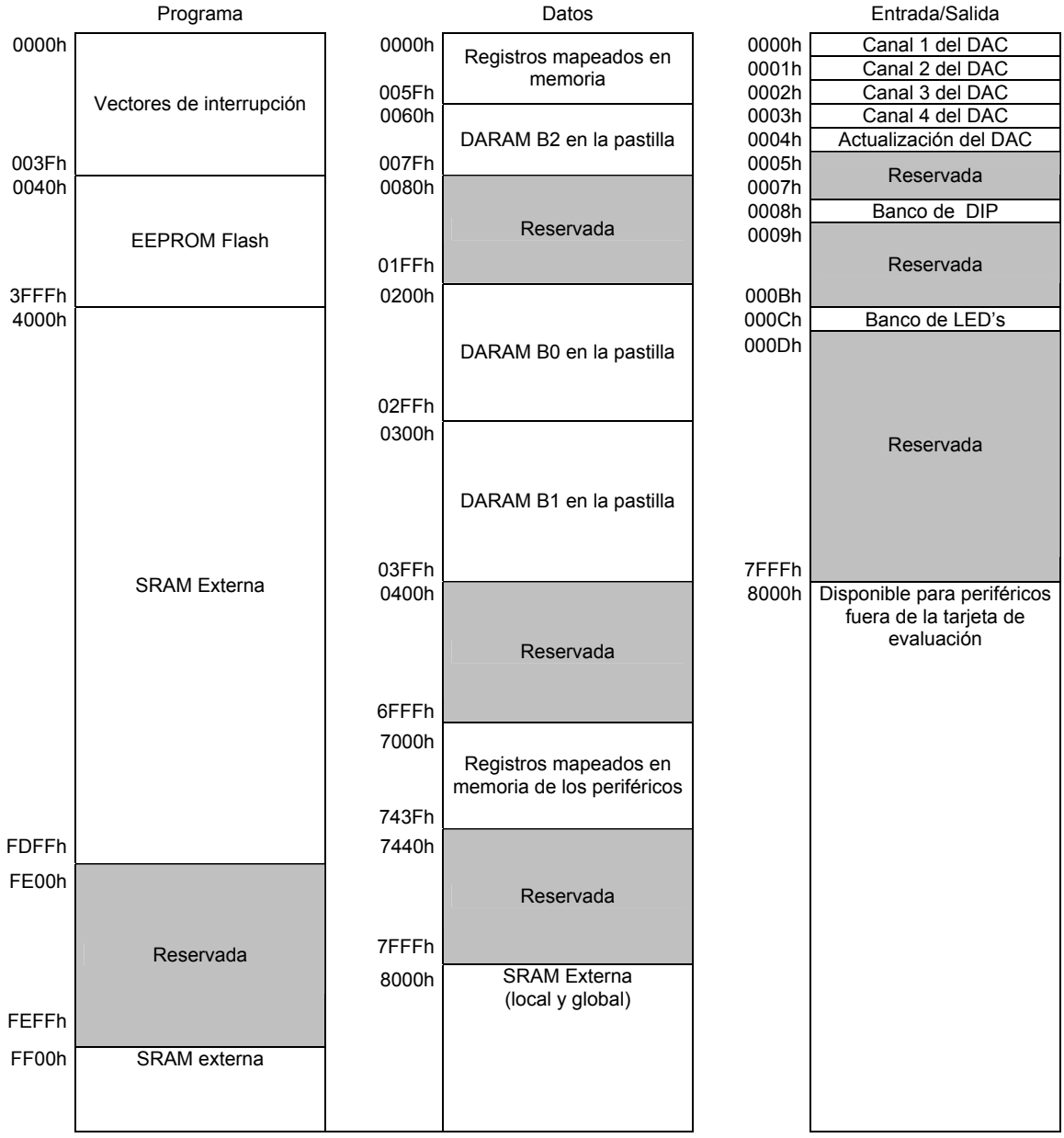


Figura A.3 Mapa de memoria del módulo de evaluación del microcontrolador DSP TMS320F240.

APÉNDICE B

Código elaborado para el procesador TMS320F240 para que el sistema operativo en tiempo real basado en μCOS-II pueda usarse en el desarrollo aplicaciones.

OS_CPU.H

```
typedef unsigned char     BOOLEAN;
typedef unsigned char     INT8U;          /*cantidad de 8 bits no signada    */
typedef signed   char     INT8S;          /*cantidad de 8 bits  signada    */
typedef unsigned int      INT16U;         /*cantidad de 16 bits no signada    */
typedef signed  int       INT16S;         /*cantidad de 16 bits signada    */
typedef unsigned long     INT32U;         /*cantidad de 32 bits no signada    */
typedef signed   long     INT32S;         /*cantidad de 32 bits signada    */
typedef float             FP32;           /*punto flotante de precision simple*/
typedef double            FP64;           /*punto flotante de doble precision    */

typedef unsigned int   OS_STK;

#define   OS_ENTER_CRITICAL()       asm("     SETC INTM");
#define   OS_EXIT_CRITICAL()        asm("     CLRC INTM");
#define   OS_STK_GROWTH             0
#define   OS_TASK_SW()              asm("     INTR 31");
```

OS_CPU_A.ASM

```
_OSStartHighRdy:
        call   _OSTaskSwHook          ;llama a la funcion, si se necesita realizar un cambio de
                                      ;contexto personalizado
        lar       AR1,#_OSTCBHighRdy  ; AR1 apunta a _OSTBHighRdy
        lar    AR1,*                  ; AR1 se actualiza como apuntador de pila con el contenido
                                      ;de _OSTBHighRdy
        lar    AR1,*
```

```
        lacc   #01h                     ; OSRunning=1
        LDPK   _OSRunning
        sacl   _OSRunning
        b      I$$REST                  ; retorno de la funcion
_OSCtxSw:
        call   I$$SAVE                  ; salva el contexto de la tarea

_OSCtxSw_comun:
        call      _OSTaskSwHook         ;llama a la funcion, si se necesita realizar un cambio de
                                        ;contexto personalizado

        lar       AR2,#_OSTCBCur        ; AR2 apunta  al TCB de la tarea vieja
        mar       *,AR2                 ; ARP=2
        lar    AR0,*,AR0                ; AR0 apunta al TCB de la tarea vieja
        sar    AR1,*                    ; guarda el apuntador de pila en el TCB
        lar    AR0,#_OSTCBHighRdy       ; AR0 apunta a la variable que contiene la
                                ; dirección del TCB de la tarea nueva
        lar    AR0,*                    ; AR0 apunta al TCB de la tarea nueva
        lar    AR1,*,AR2                ; AR1 se actualiza como el apuntador de pila
                                        ; de la nueva tarea
        sar    AR0,*,AR1                ; La variable OSTCBCur se actualiza
        ldpk   _OSPrioHighRdy           ;
        lacc   _OSPrioHighRdy           ; Se actualiza el valor de la variable
        ldpk   _OSPrioCur               ; OSPrioCur (prioridad de la tarea actual)
        sacl   _OSPrioCur               ;
        b      I$$REST                  ; Restaura el contexto de la tarea


_OSIntCtxSw:

        pop                             ;ajusta la pila física, contiene la direccion de retorno de OSIntCtxSw
        sbrk   #4                       ;ajusta la pila por programacion
        b      _OSCtxSw_comun           ;continua la ejecucion del cambio de contexto comun a la funcion
                                        ;OSCtxSw


_OSTickISR:

        call   I$$SAVE          ; llama a I$$SAVE para salvar el contexto de la tarea actual
        call   _OSIntEnter      ; llama a OSIntEnter para notificar al ucos-ii de una
                                ;interrupcion
        call   _OSTimeTick      ; llama a OSTimeTick para incrementar el contador Tick
        adrk       1            ;ajuste de la pila por programacion al escribirle codigo del ISR en
                                ;esamblador
        call   _OSIntExit       ; llama a OSIntExit para salir de la interrupcion
        sbrk   1                ;ajuste de la pila por programacion al escribirle codigo del ISR en
                                ;esamblador
        b      I$$REST          ; restaura el contexto de la tarea
```

OS_CPU_C.C

```
void *OSTaskStkInit (void (*task)(void *pd), void *pdata, void *ptos, INT16U opt)
{
    INT16U *stk;
    opt    = opt;                          /*'opt' no esta usandose, para prevenir un warning */
    stk    = (INT16U*)ptos;                /*  carga el apuntador de pila con el inicio de la misma  */
    *stk++ = (INT16U) pdata;               /*  simula el argumento por la llamada a funcion de la tarea*/
    *stk++;                                /*  direccion de retorno, la tarea nunca retorna */
    *stk++ = (INT16U)0x2000;               /* st1                                          */
    *stk++ = (INT16U)0x2200;               /* st0                                          */
    *stk++ = (INT16U)0x0000;               /* parte alta del acumulador=0                  */
    *stk++ = (INT16U)0x0000;               /* parte baja del acumulador=0                  */
    *stk++ = (INT16U)0x0000;               /* PREGH=0                                      */
    *stk++ = (INT16U)0x0000;               /* PREGL=0                                      */
    *stk++ = (INT16U)0x0000;               /* TREG=0                                       */
    *stk++ = (INT16U)0x0000;               /* AR0 = 0x0000, apuntador de trama de la tarea anterior */
    *stk++ = (INT16U)0x2222;               /* AR2 = 0x2222                                 */
    *stk++ = (INT16U)0x3333;               /* AR3 = 0x3333                                 */
    *stk++ = (INT16U)0x4444;               /* AR4 = 0x4444                                 */
    *stk++ = (INT16U)0x5555;               /* AR5 = 0x5555                                 */
    *stk++ = (INT16U)0x6666;               /* AR6 = 0x6666                                 */
    *stk++ = (INT16U)0x7777;               /* AR7 = 0x7777                                 */
    *stk++ = (INT16U)task;                 /* Pila fisica nivel uno, direccion de inicio del codigo de la tarea */
   * stk++ = (INT16U)0x0000;               /*nivel 2 de la pila fisica */
    *stk++ = (INT16U)0x0000;               /*nivel 3 de la pila fisica */
    *stk++ = (INT16U)0x0000;               /*nivel 4 de la pila fisica */
    *stk++ = (INT16U)0x0000;               /*nivel 5 de la pila fisica */
    *stk++ = (INT16U)0x0000;               /*nivel 6 de la pila fisica */
    *stk++ = (INT16U)0x0000;               /*nivel 7 de la pila fisica */
     return ((void *)stk);                                                          (5)
}
```

```
void OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;                    /* previene al compilador de un warning    */
}
```

```
void OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;                    /* previene al compilador de un warning    */
}
```

```
void OSTaskSwHook (void)
{
}


void OSTaskStatHook (void)
{
}

void OSTimeTickHook (void)
{
}
```

APÉNDICE C

Archivos de la aplicación encendido de LED's.

main.c

```
#include "includes.h"

extern void KickDog(void);

OS_STK TimeOfDayStk[OS_TASK_IDLE_STK_SIZE];
static void Parpadeo(void *data);

void main(void)
        {
        OSInit();                                       /*Inicializa el sistema operativo*/
          asm("     .global  _ConfigureWatchDog");      /*configura el sistema guardian*/
          asm("      call      _ConfigureWatchDog");
 OSTaskCreate(ParpadeoTask, (void *)0, (void *)&ParpadeoStk[0], 61)    ; /*crea la tarea*/

        OSStart();                              /* inicia multitarea */
        }


void ParpadeoTask(void *data)
        {
      volatile INT8U LEDTick=0;                 /*  inicializa la variable a cero    */
      extern void Habilitar(void);
      Habilitar();
      data=data;


       while (1)
                {
                outport(0x000C,LEDTick++);      /* saca el valor de la variable al puerto*/
                OSTimeDly(31);                  /* se retarda 31 tics para continuar      */
                }
        }
```

ini.asm

```
        .include f240regs.h

_KickDog:
   .global          _KickDog
   KICK_DOG
   ret

_ConfigureWatchDog
   .global          _ConfigureWatchDog
   KICK_DOG
   LDP   #00E0h
   SPLK             #00BBh,CKCR1          ;CLKIN(OSC)=10MHz,CPUCLK=20MHz
   SPLK             #00C3h,CKCR0          ;CLKMD=PLL Enable,SYSCLK=CPUCLK/2
   SPLK             #40C0h,SYSCR          ;CLKOUT=CPUCLK
   Ret


_Habilitar
    .global _Habilitar
    LDP   #00E0h
    SPLK            #002Fh, WDCR          ;Deshabilita WD
    splk   #0044h,RTICR                   ; configura la interrupcion en tiempo real a 61.04 HZ.
    LDP             #0000h
    SETC            INTM                  ;Deshabilita interrupciones
    SPLK            #0000h,IMR            ;Mascara todas las interrupciones
    LACC            IFR                   ;Lee banderas de interrupción
    SACL            IFR                   ;Limpia todas las banderas de interrupción
    RET
  .end
```

vectors.asm

```
       .include f240regs.h

      .asect  "vectors",0

RESET   B  _c_int0              ; reinicia cuando ocurre un reset
INT1    B  _OSTickISR           ; vector del tic
INT2    EINT                    ; B  _c_int2
        RET
INT3    EINT                    ; B  _c_int3
        RET
INT4    EINT                    ; B  _c_int4
        RET
INT5    EINT                    ; B  _c_int5
        RET
INT6    EINT                    ; B    _c_int6
```

```
        RET
        NOP             ; interrupcion reservada
        NOP
;
; interrupciones por programa
;
INT8    EINT            ; B _c_int8
        RET
INT9    EINT            ; B _c_int9
        RET
INT10   EINT            ; B _c_int10
        RET
INT11   EINT            ; B _c_int11
        RET
INT12   EINT            ; B _c_int12
        RET
INT13   EINT            ; B _c_int13
        RET
INT14   EINT            ; B _c_int14
        RET
INT15   EINT            ; B _c_int15
        RET
INT16   EINT            ; B _c_int16
        RET
TRAP    NOP
        RET             ; B _Trap
;
; vector de interrupcion
;

NMI_VECT EINT           ; B     _Nmi
        RET
        NOP             ; interrupcion reservada
        NOP

INT20   EINT            ; B _c_int20
        RET
INT21   EINT            ; B _c_int21
        RET
INT22   EINT            ; B _c_int22
        RET
INT23   EINT            ; B _c_int23
        RET
INT24   EINT            ; B _c_int24
        RET
INT25   EINT            ; B _c_int25
        RET
INT26   EINT            ; B _c_int26
        RET
INT27   EINT            ; B _c_int27
        RET
INT28   EINT            ; B _c_int28
        RET
INT29   EINT            ; B _c_int29
        RET
INT30   EINT            ; B _c_int30
```

```
        RET
INT31   B  _OSCtxSw          ; vector que sirve como cambio de tarea
```

Evm.cmd

```
/* comandos para generar el archivo objeto de la aplicacion*/
-c
main.obj
vectors.obj
init.obj
ucos_ii.obj
os_cpu_a.obj
os_cpu_c.obj
-o ejemplo1.out
-l rts2xx.lib
-stack 2048
-m ejemplo1.map

/*-----------------------------------------------------------------*/
/* Especificacion de memoria para el TMS320F240          */
/*-----------------------------------------------------------------*/

MEMORY
{
   PAGE 0 :   VECS   : origin =    0h , length =   040h   /* VECTORS */
              PROG   : origin =   40h , length = 0FDFEh   /* PROGRAM */

   PAGE 1 :   MMRS   : origin =    0h , length =   060h   /* MMRS    */
          B2     : origin = 0060h , length =   020h   /* DARAM   */
          B0     : origin = 0200h , length =  0100h   /* DARAM CNF=0*/
          B1     : origin = 0300h , length =  0100h   /* DARAM  */
          DATA1  : origin = 8000h , length =  1000h   /* XDM     */
          DATA2  : origin = 9000h , length =  1000h
          DATA3  : origin = 0A000h , length =  3000h
}

/*-----------------------------------------------------------------*/
/* loclizacion de secciones                                        */
/*-----------------------------------------------------------------*/

SECTIONS
{
   vectors   : { } > VECS     PAGE 0   /* tabla de vectores de interrpcion   */
   .text     : { } > PROG     PAGE 0   /* codigo                             */
   asmtext   : { } > PROG     PAGE 0   /* codigo asm                         */
   .cinit    : { } > PROG     PAGE 0   /* tabla de inicilizacion de datos */
   .switch   : { } > PROG     PAGE 0   /* registros mapeados de memoria    */
   .bss      : { } > DATA1    PAGE 1   /* Block B2                           */
   .const    : { } > DATA2    PAGE 1   /* Block B0                           */
   .sysmem   : { } > B1       PAGE 1   /* Block B1                           */
   .stack    : { } > DATA3    PAGE 1   /* Block B2                           */

}
```

OS_CFG.H

```
#define OS_MAX_EVENTS              0
#define OS_MAX_MEM_PART            0
#define OS_MAX_QS                  0
#define OS_MAX_TASKS               1
#define OS_LOWEST_PRIO             63
#define OS_TASK_IDLE_STK_SIZE      512
#define OS_TASK_STAT_EN            0
#define OS_TASK_STAT_STK_SIZE      512
#define OS_CPU_HOOKS_EN            0
#define OS_MBOX_EN                 0
#define OS_MEM_EN                  0
#define OS_Q_EN                    0
#define OS_SEM_EN                  0
#define OS_TASK_CHANGE_PRIO_EN     0
#define OS_TASK_CREATE_EN          1
#define OS_TASK_DEL_EN             0
#define OS_TASK_SUSPEND_EN         0
#define OS_TICKS_PER_SEC           61
```

APÉNDICE D

Archivos de la aplicación sincronización de un generador.

main.c

```
define EJ2_GLOBALS
#include "C:\respaldo\dsp240\ucos\ejemplo2\includes.h"
#include "c:\respaldo\dsp240\c_comp~1\stdlib.h"
OS_EVENT  *SEMMEDICION;
OS_EVENT  *SEMRMS;
extern void KickDog(void);

OS_STK TimeOfDayStk[OS_TASK_IDLE_STK_SIZE];
OS_STK RMSStk[OS_TASK_IDLE_STK_SIZE];
OS_STK PILAPERIODO[OS_TASK_IDLE_STK_SIZE];

void  RMS(void *pdata);
void SINCRONIZA(void *pdata);
void Parpadeo(void *data);
void incrementaw(void);
void decrmentaw(void);

void main(void)
      {
          extern void ConfigureWatchDog(void);
          SUMA_D_MUESTRAS1=0;
          PERIODO_ACT=0;
          PERIODO_ANT=0;
          CONTADOR=0;
          CONTADOR2=0;
          ConfigureWatchDog();                /* configura sistema guardian  */
          OSInit();
          OSTaskCreate(Parpadeo, (void *)0, (void *)&TimeOfDayStk[0], 62);
          OSTaskCreate(SINCRONIZA, (void *)0, (void *)&PILAPERIODO[0], 61);
          SEMMEDICION=OSSemCreate(0);          /*crea el semaforo y no esta liberado*/
          OSStart();                                /* inicia la multitarea */
          }


void interrupt DefaultISR(void)
          {
           }

void Parpadeo(void *data)
          {
          volatile UBYTE LEDTick=0;
          extern void Habilitar(void);
          extern void ConfiguraADC(void);
          ConfiguraADC();                              /*  configura ADC          */
```

```
        Habilitar();                            /* habilita interrupciones  */
        while (1)
        {
        LEDTick=LEDTick+1;
        LEDS=LEDTick;
        KickDog();
        OSTimeDly(30);
        }
   }

void interrupt c_int6(void)
{
OSIntEnter();
SUMA_D_MUESTRAS=SUMA_D_MUESTRAS1;
CONVERTIDOR1=(*((INT16U *)0X07036))>>6;
CONVERTIDOR2=(*((INT16U *)0X07038))>>6;
SUMA_D_MUESTRAS1=(CONVERTIDOR1)-0X1FF;      /*acondicionamiento de la señal*/
DIFERENCIA=abs(CONVERTIDOR1-CONVERTIDOR2);
OSSemPost(SEMMEDICION);                     /*libera el semaforo*/
OSIntExit();
}


void SINCRONIZA(void *pdata)
{
 INT16U *ACTUAL;
 INT16U BUFFER[16];
 INT8U  err;
 extern void CAL_PER(void);
 VALOR_RMS=0;                               /*inicalizacion de variables*/
 pdata=&BUFFER[0];
 BUFFER[4]=0;
 BUFFER[11]=0;
 ACTUAL=&BUFFER[0];

 while(1)
        {
   OSSemPend(SEMMEDICION,0,&err);
   VALOR_RMS=(DIFERENCIA*DIFERENCIA)-((*ACTUAL)*(*ACTUAL))+VALOR_RMS;
   *ACTUAL=DIFERENCIA;
   *ACTUAL++;
   if(ACTUAL==&BUFFER[16]) ACTUAL=BUFFER;
   CONTADOR2=CONTADOR2+1;
   CAL_PER();
   if(PERIODO_ANT<333233) {
                if(PERIODO_ANT>32230){

                        if(VALOR_RMS<180) LEDS=127;
                                }
                else
                incrementarw();
               }
        else
        decrementarw();
        }
}
```

```
void incrementaw()
{
}

void decrementaw()
{
}
```

ini.asm

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; rutina  :   void KickDog(void)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

_KickDog:
   KICK_DOG
   ret

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; rutina  :   void ConfigureWatchDog(void)
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

_ConfigureWatchDog
   KICK_DOG

   ldp              #00e0h
   splk  #006fh,WDCR              ; habilita watchdog para 1.049 S.
   splk  #0044h,RTICR             ; configura RTI para 61.04 HZ.
                                  ;PLL clockin=10Mhz,CPUCLK=15Mhz,SYSCLK=7.5Mhz
   SPLK             #00bah,CKCR1
   SPLK             #00c1h,CKCR0
                                  ; espera hasta que se estabiliza el PLL
PLLCHK:             BIT      CKCR0,BIT5
   BCND             PLLCHK,NTC
   ret                            ;


;HABILITA INTERRUPCIONES

_Habilitar
   .global     _Habilitar

  SPLK#0FFFFH,EVIFRA;
  SPLK#0FFFFH,EVIFRB;
  SPLK#0FFFFH,EVIFRC;
   ldp              #0000h
   splk  #021h,IMR         ; enable the real time interrup and ADC interrupt
   lacl        IFR
   sacl        IFR
   ret
```

```
;CONFIGURACION DE LOS PERIFERICOS

_ConfiguraADC:
   LDP          #0E1H
   SPLK         #0FFFFH,OCRA
   LDP          #0E0H


;configuracion del convertidor analogico/digital(pagina E0h)

PLLCHK2: BIT  ADCTRL1,BIT7
         BCND  PLLCHK2,NTC
         LACC  ADCFIFO1
         SUB   #0C000h
         BCND  PLLCHK2,LEQ

         SPLK #0001101110000000b,ADCTRL1
         ;        |||| ||||||||||||
         ;        5432 109876543210

;ADCTRL1        ADC Control Register 1
;Bit 15  (1) Suspend-SOFT – completa la conversion antes de detener el emulador
;Bit 14  (0) Suspend-FREE – la operacion no estadeterminada por  Suspend-SOFTbit 15
;Bit 13  (0) ADCIMSTART   - el inicio de conversion inmediata esta deshabilitada
;Bit 12  (1) ADC2EN       - ADC2 esta habilitado
;Bit 11  (1) ADC1EN       - ADC1 esta habilitado
;Bit 10  (0) ADCCONRUN    - el modo de convesion continua esta deshabilitado
;Bit 9           (1) ADCINTEN     - Esta habilitado la interrupcion por ADC
;Bit 8           (1) ADCINTFLAG   - limpia la bandera de interrupcion por ADC
;Bit 7           (0) ADCEOC       - solamente lectura fin de conversion
;Bits 6-4    (000) ADC2CHSEL    - ADC2 Canal 9
;Bits 3-1    (000) ADC1CHSEL    - ADC1 Canal 1
;Bit 0           (0) ADCSOC       - bit de inicio de conversion del ADC


         SPLK #0000010000000100b,ADCTRL2
         ;        ||||| ||||||||||||
         ;        54321 09876543210
;                 ADCTRL2 - ADC Control Register 2
;Bits 15-11 (00000) Reservado
;Bit 10  (1) ADCEVSOC – desmascara el bit del SOC del manejador de eventos
;Bit 9           (0) ADCEXTSOC – el bit SOC de conversion externa esta mascarado
;Bit 8           (0) reservado
;Bits 7-6     (00) ADCFIFO1 – solo lectura
;Bit 5           (0) Reservado
;Bits 4-3     (00) ADCFIFO2 – solo lectura
;Bits 2-0    (010) ADCPSCALE – preescalador del reloj del sistema ADC
; valor de preescalmiento 8
; Periodo SYSCLK = 0.1u sec
; 0.13333u sec x 8 x 5.5=5.86u sec >= 5.5u sec
```

```
;CONFIGURACION DEL MANEJADOR DE EVENTOS
;Se usa el temporizador de proposito general para dar el tiempo de muestreo
;se elige el temporizador de proposito general 1 (5161H ciclos de reloj para una frec 960Hz)
;e=-0.03071901699 (5162) e= 0.01536024576(5161)
;se configura de la siguiente forma:

        ldp   #00e8h            ;dp=>registros del EV

; Configura GPTCON
;*************************************************************************

  SPLK #000000001010101 0b, GPTCON ; configura GP Timer control
        ; |||||||||| |||||||
        ; f ed cba9876543210
;
* bits 0-1        10: GP Timer 1 comparacion de salida activado en alto
* bits 2-3        10: GP Timer 2 comparacion de salida activado en alto
* bits 4-5        10: GP Timer 3 comparacion de salida activado en alto
* bit 6           0: deshabilita salida de GP Timer Compare
* bits 7-8        01: GP Timer 1 inicia la conversion del ADC
* bits 9-10       00: GP Timer 2no inicia conversion
* bits 11-12      00: GP Timer 3 no inicia conversion
*
;*************************************************************************
; Configure T1PER T1CMP and T1CNT *
;*************************************************************************

        SPLK #015625, T1PR ; configura GP Timer period(960hz para un CPUCLK=15MHz)
        SPLK #015625, T1CMPR ; configura Timer compare
        SPLK #0FFFEh, T1CNT ; configura GP Timer counter

;inicia el temporizador

   SPLK         #0001000001000010b,T1CON
;                 fedcba9876543210
;
* bit 0       0: usa su propio PR
* bit 1       1:comparacion  GP Timer habilitado
* bits 2-3    00: GP Timer comp register on under flow
* bits 4-5    00: Selecciona  CLK interno
* bit 6       1: temporizador habilitado
* bit 7       0: usa su propio temporizador
* bits 8-10   000: Preescalador = /1
* bits 11-13  010: conteo continuo ascendente
* bit 14      1: la operacion es afectada cuando se suspende
* bit 15      1: libre

        ret

;rutina que calcula el periodo de la señal y lo deja en _PERIODO_ACT

_CAL_PER:
        setc    sxm
        ldpk    _SUMA_D_MUESTRAS1
        lacc    _SUMA_D_MUESTRAS1
        bgez    ACUMULAR    ;brinca si ACT>=0
```

```
        ldpk    _SUMA_D_MUESTRAS
        lacc    _SUMA_D_MUESTRAS
        blez    CEROANT             ;brinca si ANT<=0
        ldpk    _SUMA_D_MUESTRAS1
        lacc    _SUMA_D_MUESTRAS1;
        ldpk    _SUMA_D_MUESTRAS
        sub     _SUMA_D_MUESTRAS;       ACT-ANT
        ABS

        sacl    _SUMA_D_MUESTRAS;       ANT=ACT-ANT
        ldpk    _SUMA_D_MUESTRAS1
        lacc    _SUMA_D_MUESTRAS1,11
        ldpk    _SUMA_D_MUESTRAS
        ABS
        rpt     #15
        subc    _SUMA_D_MUESTRAS;       acc=ACT/(ACT-ANT)
        sacl    _SUMA_D_MUESTRAS;


        lacc    #0800h;

        sub     _SUMA_D_MUESTRAS;       1-INC
        ldpk    _PERIODO_ACT
        add     _PERIODO_ACT        ;
        ldpk    _PERIODO_ANT
        sacl    _PERIODO_ANT        ;       PERIODO_ANT=(1-INC)+PERIODO_ACT
        ldpk    _SUMA_D_MUESTRAS
        lacc    _SUMA_D_MUESTRAS
        ldpk    _PERIODO_ACT
        sacl    _PERIODO_ACT
        ZAC
        ldpk    _CONTADOR
        sacl    _CONTADOR
        b       FIN

CEROANT
        _SUMA_D_MUESTRAS
        bnz     ACUMULAR            ;brinca si ANT<>0
        ldpk    _PERIODO_ACT
        lacc    _PERIODO_ACT
        ldpk    _PERIODO_ANT
        sacl    _PERIODO_ANT
        ldpk    _PERIODO_ACT
        splk    #0800h,_PERIODO_ACT
        b       FIN

ACUMULAR
        ldpk    _PERIODO_ACT
        lacc    _PERIODO_ACT
        add     #0800h
        sacl    _PERIODO_ACT
FIN     CLRC    SXM
        ret
```

vectors.asm

```
        .include f240regs.h

       .asect  "vectors",0
RESET   B  _c_int0              ; reinicia cuando ocurre un reset
INT1    B  _OSTickISR           ; vector del tic
INT2    EINT                    ; B  _c_int2
        RET
INT3    EINT                    ; B  _c_int3
        RET
INT4    EINT                    ; B  _c_int4
        RET
INT5    EINT                    ; B  _c_int5
        RET
INT6    EINT                    ; B    _c_int6
        RET
        NOP                     ; interrupcion reservada
        NOP
;
; interrupciones por programa
;
INT8    EINT                    ; B  _c_int8
        RET
INT9    EINT                    ; B  _c_int9
        RET
INT10   EINT                    ; B  _c_int10
        RET
INT11   EINT                    ; B  _c_int11
        RET
INT12   EINT                    ; B  _c_int12
        RET
INT13   EINT                    ; B  _c_int13
        RET
INT14   EINT                    ; B  _c_int14
        RET
INT15   EINT                    ; B  _c_int15
        RET
INT16   EINT                    ; B  _c_int16
        RET
TRAP    NOP
        RET                     ; B  _Trap
;
; vector de interrupcion
;

NMI_VECT EINT                   ; B       _Nmi
        RET
        NOP                     ; interrupcion reservada
        NOP

INT20   EINT                    ; B  _c_int20
```

```
        RET
INT21   EINT            ; B  _c_int21
        RET
INT22   EINT            ; B  _c_int22
        RET
INT23   EINT            ; B  _c_int23
        RET
INT24   EINT            ; B  _c_int24
        RET
INT25   EINT            ; B  _c_int25
        RET
INT26   EINT            ; B  _c_int26
        RET
INT27   EINT            ; B  _c_int27
        RET
INT28   EINT            ; B  _c_int28
        RET
INT29   EINT            ; B  _c_int29
        RET
INT30   EINT            ; B  _c_int30
        RET
INT31   B  _OSCtxSw     ; vector que sirve como cambio de tarea
```

Evm.cmd

DEFINICIÓN DEL MAPA DE MEMORIA

```
/* commandos de la aplicacion sincronizacion de un generador  */
-c
main2.obj
vectors.obj
init.obj
ucos_ii.obj
os_cpu_a.obj
os_cpu_c.obj
-o ejemplo2.out
-l rts2xx.lib
-stack 0x1000
-m ejemplo2.map

/*-----------------------------------------------------------------*/
/*ARCHIVO DE COMANDO DEL ENLAZADOR– ESPECIFICACION DE LA MEMORIA para el
F240*/
/*-----------------------------------------------------------------*/

MEMORY
{
   PAGE 0 :  VECS    : origin =    0h , length =   040h   /* VECTORS */
             PROG    : origin =   40h , length = 0FDFEh   /* PROGRAM */
```

```
   PAGE 1 :   MMRS   : origin =    0h , length =   060h   /* MMRS    */
              B2     : origin = 0060h , length =   020h   /* DARAM   */
              B0     : origin = 0200h , length =  0100h   /* DARAM CNF=0*/
              B1     : origin = 0300h , length =  0100h   /* DARAM  */
              DATA1  : origin = 8000h , length =  1000h   /* XDM     */
             DATA2  : origin = 9000h , length =  1000h
             DATA3  : origin = 0A000h , length =  3000h
}

/*--------------------------------------------------------------------*/
/* SECTIONS ALLOCATION                                                */
/*--------------------------------------------------------------------*/

SECTIONS
{
   vectors    : { } > VECS        PAGE 0   /* Tabla de vectores de interrupción  */
   .text      : { } > PROG        PAGE 0   /* Código                             */
   asmtext    : { } > PROG        PAGE 0   /* Código ASM                         */
   .cinit     : { } > PROG        PAGE 0   /* Inicialización de tabla de datos */
   .switch    : { } > PROG        PAGE 0   /* Registros mapeados en memoria */
   .bss       : { } > DATA1       PAGE 1   /* Bloque B2                          */
   .const     : { } > DATA2       PAGE 1   /* Bloque B0                          */
   .sysmem    : { } > B1          PAGE 1   /* Bloque B1                          */
   .stack     : { } > DATA3       PAGE 1   /* Bloque B2                          */

}
```

OS_CFG.H

```
#define OS_MAX_EVENTS              2
#define OS_MAX_MEM_PART            0
#define OS_MAX_QS                  0
#define OS_MAX_TASKS               2
#define OS_LOWEST_PRIO             63
#define OS_TASK_IDLE_STK_SIZE      512
#define OS_TASK_STAT_EN            0
#define OS_TASK_STAT_STK_SIZE      512
#define OS_CPU_HOOKS_EN            0
#define OS_MBOX_EN                 0
#define OS_MEM_EN                  0
#define OS_Q_EN                    0
#define OS_SEM_EN                  1
#define OS_TASK_CHANGE_PRIO_EN     0
#define OS_TASK_CREATE_EN          1
#define OS_TASK_DEL_EN             0
#define OS_TASK_SUSPEND_EN         0
#define OS_TICKS_PER_SEC           61
```

APÉNDICE E

Sistemas operativos de tiempo real comerciales más comunes.

FABRICANTE	SISTEMA OPERATIVO EN TIEMPO REAL	DIRECCION ELECTRÓNICA (Sitio WEB)
3L	Diamond	www.shen.myby.co.uk/threel
Accelerated Technology (Embedded Division of Mentor Graphics)	Nucleus uiPLUS, Nucleus OSEK, Nucleus PLUS	http://www.acceleratedtechnology.com/
Aonix	RAVEN	http://www.aonix.com/
Applied Dynamics International	RTexec	http://www.adi.com/
ARC International	Precise/MQX	http://www.arc.com/
Ardro Engineering	Spartos	http://www.ardro.com/
Auriga	Starlight Linux	http://www.auriga.com/
Avocet Systems	AvSYS Real-Time	http://www.avocetsystems.com/
Blackhawk	Blackhawk OS	http://www.blackhawk-dsp.com/
Blunk Microsystems	TargetOS	http://www.blunkmicro.com/

CMX Systems,	CMX-RTX, CMX-Tiny+, CMX-RTXS	http://www.cmx.com/
Datalight	ROM-DOS	http://www.datalight.com/
DSP OS	Fusion RTOS	http://www.dspos.com/
EBSnet	RTKernel-RISC	http://www.ebsnetinc.com/
esmertec	Jbed	http://www.esmertec.com/
ETAS	ERCOSEK	http://www.etasinc.com/
EUROS Embedded Systems	EUROS	http://www.kaneff.de/
Express Logic	ThreadX	http://www.expresslogic.com/
Eyring	Eyrx, PDOS	http://www.eyring.com/
FORTH	pF/x	http://www.forth.com/
Fujitsu Microelectronics	REALOS	http://www.fujitsumicro.com/
Green Hills Software	INTEGRITY	http://www.ghs.com/
HighTec EDV-Systeme	PXROS	http://www.hightec-rt.com/
Incantation Systems	TxOS -- Titanic	http://www.incantationsystems.com/

Integrated Chipware	icWORKSHOP	http://www.chipware.com/
JK microsystems	eRTOS	http://www.jkmicro.com/
JMI Software Systems	PSX, C Executive	http://www.jmi.com/
KADAK	AMX	http://www.kadak.com/
Keil Software	RTX51, RTX51 Tiny, RTX166, RTX166 Tiny	http://www.keil.com/
Lantronix	SuperTask, TronTask3.0	http://www.lantronix.com/
Lineo	Embedix RT	http://www.lineo.com/
LiveDevices	Realogy Real-Time Architect	http://www.livedevices.com/
LynuxWorks	LynxOS, BlueCat Linux	http://www.lynuxworks.com/
Mentor Graphics	VRTX	http://www.mentor.com/
Metrowerks/Motorola	OSEKturbo	http://www.metrowerks.com/
Micrium	mC/OS-II	http://www.micrium.com/
Micro Digital	pmDOS, smx/smx++, PSMX Portable smx	http://www.smxinfo.com/
Microsoft	Windows CE .NET, Windows XP Embedded	http://www.microsoft.com/

MODCOMP	REAL/IX PX	http://www.modcomp.com/
MontaVista Software	MontaVista Linux	http://www.mvista.com/
OAR	RTEMS	http://www.rtems.com/
On Time Software	On Time RTOS-32, RTKernel	http://www.on-time.com/
OnCore Systems	Linux for Real-Time, OnCore OS	http://www.oncoresystems.com/
OSE Systems	OSE RTOS	http://www.ose.com/
Pumpkin	Salvo	http://www.pumpkininc.com/
QNX Software Systems	QNX, Neutrino	http://www.qnx.com/
Quadros	RTXC	http://www.quadros.com/
RadiSys	Microwave OS-9	http://www.radisys.com/
Red Hat	eCos, Red Hat Embedded	http://www.redhat.com/
REDSonic	REDICE-Linux	http://www.redsonic.com/
SEGGER Microcontroller System	embOS	http://www.segger.com/
SKY Computers	SKYmpx	http://www.skycomputers.com/
Softools	TurboTask, QuickTask	http://www.softools.com/

TenAsys	iRMX III, iRMX for Windows, INtime for Windows	http://www.tenasys.com/
Texas Instruments	Real-Time OS: DSP/BIOS	http://www.ti.com/
The SCO Group	DR-DOS 7.03	http://www.sco.com/
TICS Realtime	Tics	www.cris.com/~Tics/
TimeSys	TimeSys Linux/RT, Real-Time Mach	http://www.timesys.com/
TTTech Computertechnik	TTPos	http://www.tttech.com/
VenturCom	PharLap Real-time ETS Kernel, RTX for Windows	http://www.vci.com/
Wasabi Systems	NetBSD Embedded	http://www.wasabisystems.com/
Wind River	BSD/OS, OSEKWorks, pSOSystem 2.5, pSOSystem 3, VSPWorks, VxWorks, VxWorks AE	http://www.windriver.com/