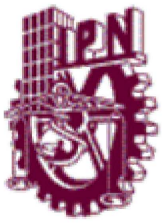
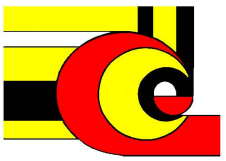


Instituto Politécnico Nacional

Centro de Investigación en Computación



Diseño e Implementación de un Kernel de Tiempo Real para un DSP 56827

T E S I S

Que para obtener el grado de

Maestro en Ciencias de la Computación

Presenta: **Ing. Rafael Ramos Ramos**

Director: **M. en C. Ricardo Barrón Fernández**

México, D.F., Abril de 2004.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

RESUMEN

Los Sistemas Operativos de Tiempo Real pequeños se han convertido en partes fundamentales de los Sistemas Incrustados que alojan Aplicaciones de Tiempo Real, estos núcleos proporcionan el manejo de las tareas, el manejo de interrupciones y la sincronización de las tareas.

El *kernel* de Tiempo Real presentado en la tesis está basado en primitivas que ejecutan el cambio de contexto necesario, el manejo de las secciones críticas, la planificación y el despacho, la creación, la finalización y suspensión de las tareas, así como los servicios de sincronización y comunicación entre los procesos.

Este *Kernel* está diseñado e implementado en un Procesador Digital de Señales (DSP, por sus siglas en inglés). Estos dispositivos se han usado con gran frecuencia en las Aplicaciones de Tiempo Real para llevar acabo funciones de control y procesamiento de señales a través de sistemas y filtros digitales.

El *kernel* está estructurado en forma modular, dividido en cuatro capas: la capa de máquina, la capa de manejo de listas, la capa de manejo del procesador, la capa de servicio. Dentro de ellas, se distribuyen todas las primitivas que componen el *Kernel* de Tiempo Real.

El *Kernel* presentado, se enfoca a los Sistemas de Tiempo Real donde las restricciones de tiempo son fundamentales para su funcionamiento, por ello proporciona la capacidad de crear dos tipos de tareas: las críticas y las de no tiempo real (no tienen plazo).

Entre las ventajas que presenta este *kernel* es proporcionar al desarrollador de la Aplicación de Tiempo Real la posibilidad de dividir su aplicación en pequeñas tareas que podrán ser controladas por el *Kernel*. Logrando con esto que el desarrollador use las llamadas al sistema dentro de las tareas de acuerdo a sus requerimientos.

Como desventaja se podría citar que el tamaño del *Kernel*, excede la capacidad de la memoria interna del DSP empleado para su implementación, sin embargo esto se soluciona empleando la memoria externa proporcionada por la Tarjeta de Evaluación del DSP.

ABSTRACT

Small Real-Time Operating Systems are a fundamental part of the Embedded Systems lodging Real-Time Applications. These *kernels* provide the process management, the handling of interruptions and the process synchronization.

The Real-Time Kernel presented in the thesis is based on primitives that execute the necessary change of context, the handling of the sections critics, the planning and dispatch, creation, conclusion and suspension of the tasks, as well as the services of synchronization and communication between the processes.

The Kernel is designed and implemented on a DSP. These devices have been used very frequently in Real-Time Applications in order to perform control functions and the processing of signals through digital systems and filters.

The Kernel is structured in modular form, divided in four modules: machine layer, management list layer, processor management layer, and service layer. Within there, all the primitives that compose the Real-Time Kernel are distributed.

The presented Kernel is focused on Real-Time systems where the time restrictions are fundamental for good operation. The Kernel however provides the capacity to create two types of tasks: critics ones and non real-time ones.

Among the advantages of the proposed kernel, are to provide to the developer of the Real-time Application the possibility to divide its application into small tasks that could be controlled by a Kernel, obtaining with this that the developer uses the system calls within the tasks according to its requirements.

As a disadvantage it could be mentioned that size of the Kernel, exceed the capacity of the internal memory of the DSP used for his implementation, nevertheless this is solved using the external memory provided by Evaluation Board of the DSP.

INTRODUCCIÓN

En esta tesis se presenta el diseño y la implementación de un *kernel* (núcleo) de tiempo real pequeño y limitado en las primitivas, así como en el conjunto de mecanismos para la sincronización y comunicación de las tareas, dicho núcleo está diseñado para emplearse en la arquitectura de procesadores de la familia 56800 de la marca Motorola, en específico para el procesador digital de señales 56F827.

El *kernel* está estructurado en cuatro capas: la capa de máquina, la capa de manejo de listas, la capa de manejo del procesador, y la capa de servicio. Dentro de ellas, se implementan un conjunto de funciones que ejecutan las actividades de cambio de contexto, el manejo de las secciones críticas, la planificación, el despacho de las tareas, la creación de tareas, la finalización y suspensión de las tareas, así como los servicios de sincronización y comunicación entre los procesos. Ya que el *Kernel* está enfocado a las aplicaciones de control en Tiempo Real, es necesario tener una definición de un Sistema de Tiempo Real (STR).

Un STR es cualquier sistema informático que debe responder dentro de un período finito y específico de tiempo a estímulos generados externamente; es predecible, multitarea y está basado en prioridades[1].

Las aplicaciones de Tiempo Real usan los Procesadores Digitales de Señales (DSP por sus siglas en inglés) para realizar sus funciones de control, debido a su tamaño, gran velocidad de procesamiento y buena capacidad de almacenamiento, esto ayuda a implementar un *Kernel* de Tiempo Real en tales dispositivos. Dado que el *kernel* está en conexión directa con el *hardware*, éste debe proveer la administración de procesos, manejo de interrupciones y sincronización de procesos.

Es bien sabido que un Sistema Operativo de Tiempo Real (SOTR) es aquel que debe satisfacer las restricciones temporales explícitas de modo que si no se cumplieran, se darían en el sistema consecuencias de riesgo severo e incluso el fallo total. No se trata de que los SOTR sean sistemas rápidos sino de que sean fiables.

Esto está claramente influenciado por el hecho de tener que satisfacer restricciones temporales para evitar estados y situaciones catastróficas. Interesa más que un sistema nunca tenga situaciones peligrosas a que el promedio de las tareas sea muy bueno. Esto supone diseñar el SOTR

pensando más en el peor caso, que en optimizar el rendimiento medio, por lo que la rapidez del sistema también resulta necesaria.

Para evitar las situaciones de riesgo es preciso atender con una alta prioridad a las señales externas provenientes del sistema, ya que pueden informarnos de un cambio de estado en él. El SOTR es capaz de ver en qué situación está el sistema en cada momento para poder actuar según requiera la ocasión. Como ya se ha indicado, aunque prevalece la fiabilidad, el SOTR debe ser muy rápido a la hora de atender las señales exteriores (interrupciones, señales, etc.). Cuando surge una señal exterior, el SOTR debe desalojar de la CPU el proceso que la ocupa. Por tanto al diseñar un SOTR se trata de minimizar todo aquello que conlleve un alto precio en el tiempo de la CPU.

Los sistemas operativos de tiempo real se han convertido en indispensables por ejemplo en las siguientes aplicaciones:

- Control de tráfico aéreo.
- Múltiples robots cooperativos.
- Complejas cadenas de fabricación
- Comunicaciones en tiempo real en sistemas de información distribuidos y de gran escala.
- Sistemas de control para defensa.

Es de especial interés en un SOTR la carga de una CPU. Se define un *factor de utilización o tiempo de carga como medida del porcentaje de uso que está haciendo el computador de la CPU*. Aunque en principio podría parecer que el factor óptimo de uso de la CPU es el 100 %, esto no es cierto en los Sistemas de Tiempo Real, ya que los cambios o ampliaciones no podrían realizarse sin que se produjeran condiciones de riesgo por estar la CPU demasiado ocupada para poder atenderlo todo. Un factor bajo tampoco es aconsejable ya que supone que el procesador está subutilizado. Se acepta un factor próximo al 70 % en los sistemas que esperan crecer o ser ampliados en el futuro.

Aunque todo lo anterior nos podría llevar a pensar en usar siempre SOTR frente a los sistemas operativos normales, es preciso considerar lo siguiente:

- Los SOTR son materiales de *software* comerciales, no de uso público como es LINUX.
- Los SOTR son un producto relativamente caro.
- Los SOTR tienen un *kernel* muy pequeño para poder ser alojados en memoria. Eso supone más simplicidad, lo cual provoca que se pierda

en opciones y recursos, como las que poseen otros sistemas operativos.

- Los SOTR carecen o tienen recursos útiles limitados y flexibles para el programador, lo cual introduce grandes retrasos.

A menudo se ha venido haciendo un tratamiento exclusivo de los Sistemas Operativos de Tiempo Compartido existentes para que soporten Tiempo Real.

Por supuesto, los SOTR tienen una evolución marcada claramente por los propios Sistemas de Tiempo Real, ya que cada vez las Aplicaciones de Tiempo Real requieren de más determinismo y más predecibilidad.

Dentro del *kernel* planteado las tareas son clasificadas en duras o críticas y de no tiempo real; para su planificación se emplea el algoritmo EDF (Earliest Deadline First – Primero el Plazo más Corto), en donde se elige como tarea de mayor prioridad aquella que tiene el plazo más corto, cabe mencionar que las tareas críticas se planifican directamente con este algoritmo teniendo en cuenta el plazo de dichas tareas, sin embargo aunque las tareas de no tiempo real son planificadas con el mismo algoritmo que las tareas críticas es necesario realizar un mapeo de sus prioridades fijas (ya que éstas no tienen plazos) hacia plazos absolutos para que de esta manera sean planificadas correctamente. Como mecanismo de comunicación se cuenta con *buffers* asíncronos cíclicos que permiten la comunicación entre una tarea a varias, además de emplear los semáforos binarios para la sincronización entre las tareas que desean compartir un mismo recurso.

ANTECEDENTES

El proyecto de diseñar e implementar un *Kernel* de Tiempo Real surge de la necesidad de proporcionar a las Pequeñas y Medianas Empresas Mexicanas dedicadas al desarrollo de Aplicaciones de Tiempo Real empleando Microcontroladores y Procesadores Digitales de Señales, un núcleo capaz de garantizar que un conjunto de tareas en el sistema cumplan satisfactoriamente las restricciones temporales impuestas por las Aplicaciones de Tiempo Real, y que además proporcione un conjunto de primitivas básicas para la sincronización y la comunicación entre las tareas.

Además de las consideraciones anteriores y después de concertarse una serie de pláticas con desarrolladores de Aplicaciones de Tiempo Real, de la empresa mexicana Servicios Eléctricos de Potencia Computarizados (SEDPC), e intercambiando sus experiencias, donde exponían que uno de los Procesadores Digitales de Señales que prometía un gran desempeño y un buen valor agregado, por ser un dispositivo híbrido (DSP y Microcontrolador) era el DSP56F827 de Motorola, y porque a través de ellos, en un principio se pudo hacer uso de la Tarjeta de Evaluación del DSP, se optó por implementar en dicho dispositivo el Kernel de Tiempo Real.

Otras consideraciones para realizar este proyecto son que actualmente en México existe poco o escaso desarrollo de núcleos de Tiempo Real para Microcontroladores y DSP's. Los *kernel*s utilizados por las empresas mexicanas son desarrollados en el extranjero, lo cual repercute en el costo del producto final. Sin embargo en algunas ocasiones los *kernel*s son diseñados por ellas mismas pero con escasas características de un *Kernel* de Tiempo Real, principalmente estos núcleos sólo tratan la administración de los procesos.

Existen varios trabajos en desarrollo sobre *Kernel*s de Tiempo Real pero están enfocados a las computadoras personales o microprocesadores de uso general, es por ello que con el proyecto del *Kernel* de Tiempo Real implementado en el DSP se desea proporcionar un sistema base que se pueda emplear en Aplicaciones de Tiempo Real relacionadas con los sistemas incrustados.

OBJETIVOS

Objetivos Específicos:

- Diseñar e implementar un núcleo de Tiempo Real para el DSP56F827 de Motorola.
- Proporcionar funciones básicas de administración de procesos utilizando un algoritmo de planificación con asignación dinámica.
- Implementar un mecanismo de sincronización básico y un mecanismo de comunicación no muy difundido como los *Buffers* Asíncronos Cíclicos.

Objetivo General:

- Suministrar *software* base empleando el diseño estructurado para desarrollar Aplicaciones de Tiempo Real implementándolas en alguno de los procesadores de la Familia DSP56800 de Motorola.

ALCANCES

El Kernel de Tiempo Real proporcionará la administración de las tareas a través de un conjunto de estructuras de datos denominadas Bloques de Control de Tareas, aplicando el algoritmo EDF para la planificación de las mismas.

Dentro del núcleo se implementará un mecanismo de sincronización de tareas (semáforos) empleando estructuras de datos que contendrán la información de los semáforos, los cuales funcionaran como un candado para que las tareas tengan acceso a un recurso o continúen su ejecución.

Para que las tareas puedan compartir o intercambiar datos se implementará un mecanismo de comunicación poco usual pero que permite que varias tareas tengan acceso a los datos más actualizados colocados por una tarea dentro de la estructura que manipula el intercambio, dicha estructura es denominada *Buffer* Asíncrono Cíclico. Sin embargo este kernel se plantea para que sólo sea alojado en la memoria interna o externa del DSP desde la cual será ejecutado, no se pretende desarrollar una herramienta de monitoreo, ni un interprete de comandos que se ejecute desde una computadora personal y tenga comunicación directa con el kernel.

El kernel constará de un conjunto de primitivas que se emplearán para desarrollar Aplicaciones de Tiempo Real y las cuales se compilarán y se descargarán a la memoria del DSP usando un cargador y un enlazador compatible con el DSP.

La implementación del kernel se hará empleando herramientas sofisticadas (como el *IDE CodeWarrior*) que cuenten con compiladores de alto nivel y depuradores para la etapa de prueba y mantenimiento.

CAPÍTULO 1 - ESTADO DEL ARTE

1.1 El kernel

El *Kernel* consiste en la parte principal del código del sistema operativo, el cual se encarga de controlar y administrar los servicios proporcionados, así como las peticiones de recursos y de *hardware* con respecto a uno o varios procesos.

Según la teoría clásica de sistemas operativos entendemos un Sistema Operativo como un intermediario entre los programas de usuario y el *hardware*, es decir, el *software* encargado de proporcionar el entorno necesario donde será posible ejecutar otros programas. Cada aplicación de usuario se ejecuta sobre la máquina utilizando lo que llamamos un proceso (que podría entenderse como un programa en ejecución). Pero es necesario ser consciente de que existe mucho bajo el concepto de proceso.

Podemos definir un *kernel* como la porción más pequeña de un sistema operativo que proporciona la planificación de tareas, despacho y la comunicación entre tareas [7]. Sin embargo, el *kernel* es conocido también como el *software* que constituye el núcleo del sistema operativo, dónde se realizan las funcionalidades básicas como la gestión de procesos, la gestión de memoria y de entrada-salida.

En general el *kernel* del sistema operativo debe realizar las siguientes funciones:

- **Ejecución de programas**- capacidad del sistema de cargar un programa en memoria y ejecutarlo.
- **Operaciones de E/S** - dado que los programas no pueden ejecutar operaciones de E/S directamente, el sistema operativo debe proveer medios para realizarlas.
- **Manipulaciones del sistema de archivos** - capacidad de los programas de leer, escribir, crear y borrar archivos. Para el caso de los sistemas incrustados, esto puede ser opcional.
- **Comunicaciones** - intercambio de información entre procesos ejecutándose en la misma computadora o en otra conectada a través de una red. Implementándose mediante memoria compartida o por paso de mensajes.

- **Detección de errores** - asegurar un cómputo correcto mediante la detección de errores en la CPU, memoria, dispositivos de E/S o en los programas de usuario.

Pero el área de interés de nuestra parte es el *Kernel* de Tiempo Real que varía un poco con el clásico enfoque usado en los sistemas operativos comerciales y genéricos, donde las siguientes actividades son básicas:

- **Administración de procesos** – Es el servicio principal del sistema operativo que incluye varias funciones tales como la creación y terminación de procesos, la planificación de los procesos, el despacho, el cambio de contexto, y otras actividades relacionadas.
- **Manejo de interrupciones** – Su objetivo es proveer el servicio a la solicitud de la interrupción que ha sido generada por un dispositivo periférico, como puede ser el teclado, puerto serial, convertidor análogo-digital o cualquier interfaz con un sensor.
- **Sincronización de procesos** - El acceso concurrente a datos compartidos puede llevar a inconsistencias en los datos. El mantenimiento de la consistencia en los datos requiere mecanismos para asegurar la ejecución ordenada de procesos cooperativos. Los mecanismos de consistencia deben permitir la aplicación correcta de los algoritmos de planificación sin perjudicar a la predecibilidad.

1.2 El Sistema Operativo de Tiempo Real

La característica principal de un SOTR[1], al igual que en un STR[2], es que sea determinista, es decir, garantice que el sistema se ejecutará dentro de una restricción temporal especificada.

Los especialistas consideran que la mejor estructura para un SOTR es una jerárquica formada por niveles, en cuyo nivel más bajo se encuentra el *kernel*. Estos niveles pueden verse como una pirámide invertida (véase Figura 1.1), donde cada uno de los niveles se encuentra construido sobre un nivel inferior.

Los requerimientos generales para el *kernel* subyacente en un SOTR son los siguientes [19]:

- Multitarea.

[1] Sistema Operativo de Tiempo Real
[2] Sistema de Tiempo Real

- Planificación (Scheduling) por desalojo (Preemptive).
- Cambio de contexto rápido.
- Tamaño pequeño.
- Rapidez y flexibilidad en la comunicación y sincronización entre tareas.
- Facilidad de comunicación entre tareas y niveles de interrupción.
- Capacidad de responder rápidamente a interrupciones externas.
- Límite de ejecución.
- Dotación de particiones fijas o variables para la gestión de memoria
- Minimizar los intervalos durante los cuales las interrupciones están deshabilitadas.
- Capacidad de bloquear acceso al código y datos de memoria.
- El *kernel* ha de mantener un reloj de tiempo real.

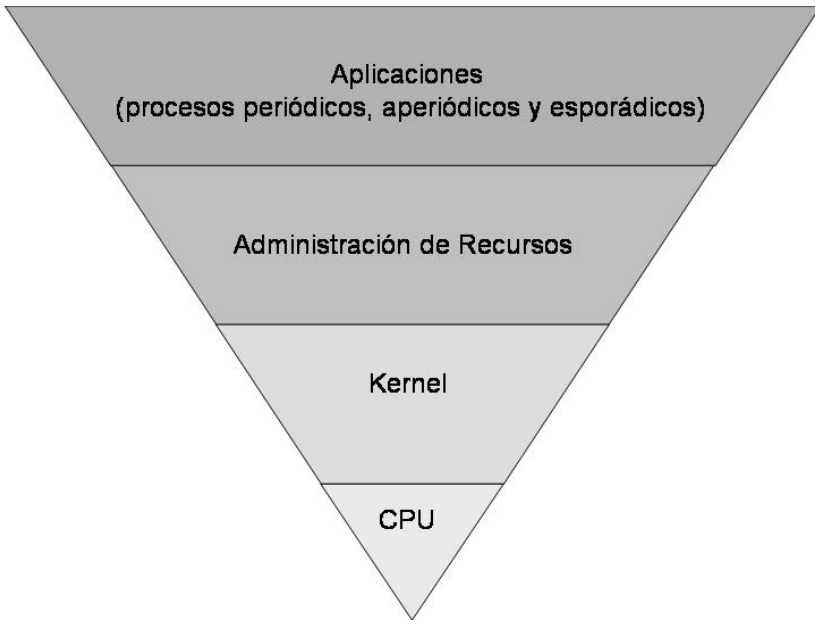


Figura 1.1- Jerarquía Funcional de un SO en un Escenario de Tiempo Real.

El *kernel* debe encargarse de introducir los diferentes procesos en la CPU. La naturaleza síncrona de los eventos en el mundo real impone la necesidad de ejecutar un conjunto de procesos o tareas concurrentes. La multitarea es la mejor forma de tratar los diferentes eventos discretos externos.

La planificación de los procesos por desalojo (Preemptive) basado en prioridades consiste en ejecutar aquellas tareas de mayor prioridad que se encuentren preparadas. Tan pronto como una tarea de alta prioridad se encuentre preparada, se vaciará la CPU de cualquier tarea de menor prioridad que se esté ejecutando en ese instante. Los eventos que se producen en tiempo real tienen una cierta precedencia inherente. Tiene vital importancia que esa precedencia se observe a la hora de introducir los

diferentes procesos en la CPU. La evacuación rápida de un proceso de la CPU para que entre otro, debe ser lo más rápida posible.

El tamaño pequeño del *kernel* permite que los SOTR residan en memoria, lo cual redunda en beneficio de la rapidez.

El *kernel* debe proveer de un método de sincronización efectivo para el acceso simultáneo a recursos, no produciendo un desalojo cuando una tarea se encuentra en la región crítica. En un sistema de tiempo real hay muchas tareas trabajando como componentes de una misma aplicación. Por esto, es importante, que el sistema posea mecanismos rápidos y potentes de comunicación entre las tareas y los niveles de interrupción. Aunque la mayoría de los eventos que se producen en el mundo real se deben a interrupciones, es deseable tratar éstas a nivel de tarea para poder aprovechar el uso de colas, prioridades, etc.

Un *Kernel* de Tiempo Real, debería optimizar la ejecución en el peor de los casos, por ejemplo, un sistema que ejecuta una función siempre en 50 ms es más deseable que un sistema que normalmente la ejecuta en 10 ms pero ocasionalmente tarda 75 ms.

Las funcionalidades generales incluidas en los *kernel*s de sistemas incrustados desarrollados para aplicaciones específicas o arquitecturas concretas son las siguientes:

- Gestión eficiente de recursos.
- Planificación (Scheduling) de tareas y comunicaciones.
- Multitarea con baja sobrecarga por cambios de contexto.
- Equilibrado de carga de trabajo.
- Reconfiguración y recuperación dinámica.
- Operación de dispositivos.
- E/S síncrona y asíncrona.
- Respuesta rápida a interrupciones externas.
- Primitivas IPC (memoria compartida, semáforos, eventos asíncronos).
- Posibilidad de bloqueo y desbloqueo de memoria.
- Uso limitado de memoria virtual.
- Soporte de tiempo real para cumplimiento de plazos de respuesta: planificación basada en prioridades, relojes de tiempo real, alarmas especiales, primitivas para retardar, parar o reanudar tareas, etc.
- Tamaño ajustable a las necesidades de empotramiento.

A los SOTR se les debe exigir una serie de cualidades básicas, entre las que podemos destacar las siguientes:

- Determinismo.
- Capacidad para responder a sucesos rápidamente.
- Control del sistema por parte del usuario.
- Gestión de prioridades.
- Fiabilidad.
- Gestión de memoria.
- Comunicación entre tareas.
- Código reentrante.
- Tamaño reducido de código.
- SOTR distribuidos.
- Configurabilidad.
- Adaptabilidad.

Determinismo se puede definir como la tendencia que tiene el sistema a realizar una determinada acción en un tiempo predefinido. El minimizar el tiempo de respuesta a interrupciones garantiza un mayor nivel de determinismo.

Capacidad para responder a sucesos rápidamente, distinguiendo entre sucesos síncronos y asíncronos. Para esto se necesita una rutina de interrupciones para dar una respuesta "inmediata" a los sucesos asíncronos. Requiere que el desalojo y realojamiento de procesos de la CPU se haga con rapidez.

Control del sistema por parte del usuario de modo que los usuarios puedan conmutar entre distintos modos de ejecución. Por ejemplo, un operario que acciona un botón de emergencia de una máquina.

En los sistemas operativos tradicionales la **prioridad** es dinámica, el propio sistema operativo va incrementando o decrementando la prioridad conforme pasa el tiempo. En los SOTR la prioridad debe ser estática con prioridades determinadas, al menos para varios procesos especialmente críticos. Las interrupciones procedentes del exterior tienen una prioridad fija, que no depende de tiempos de espera o ejecución. Además el SOTR debe hacer un análisis de las restricciones de tiempo de las tareas para conocer si ellas se rechazan según los plazos (seguimiento preventivo y predictivo) y cómo interfieren con las demás.

La **fiabilidad** en los SOTR indica que en caso de fallo del *hardware* ha de haber una solución de tipo *software*. Deben estar contempladas todas las respuestas que se darían a cada uno de los posibles fallos que se pudiesen presentar.

La **gestión de memoria** en los SOTR ha de ser más estricta que en los sistemas operativos tradicionales. Cuanta mayor facilidad se trata de dar al usuario mayor va a ser el tamaño del *kernel*. En los SOTR el *kernel* debe ser lo menor posible.

La **comunicación entre tareas** debe ser muy rápida. Suele ser explícita, la tiene que hacer el usuario.

El **código reentrante** se refiere a la posibilidad de que dos procesos puedan emplear un mismo código de programa, sin tener que contar una copia del mismo para cada uno y sin que haya problemas de interferencias entre ellos al manejar el mismo código.

En tiempo real conviene que el **repertorio de rutinas** empleado sea lo menor posible, a costa de mayor precio de programación por parte del usuario, de modo que se minimice el número de primitivas y el *kernel*.

Los **SOTR distribuidos** son un asunto complicado. Se trata de minimizar los tiempos de respuesta por parte de la red: buses de campo, redes industriales. Han de minimizar el tiempo desde que se recoge algo en un sensor hasta que llega a un actuador a dar la orden que sea, pasando por el gestor de la red. Otros problemas son considerar que puede haber sobrecarga en la red y problemas de sincronización de los relojes de los distintos elementos del sistema distribuido.

La **configurabilidad** es un requisito cada vez más actual. Consiste en que el SOTR sirva para una amplia gama de sistemas: desde pequeños sistemas incrustados hasta sistemas donde cada nodo de la red es un supercomputador.

La **adaptabilidad** indica la necesidad de adaptación a cambios que se producen en el entorno de operación del sistema de tiempo de ejecución. Los tipos básicos de adaptación son la preventiva y la reactiva. La adaptación preventiva trata de garantizar un nivel de prestaciones y funcionalidad del *software* de operación haciendo hipótesis sobre el comportamiento futuro del sistema basándose en su comportamiento presente. La adaptación reactiva se realiza en respuesta a eventos inesperados.

1.3 Clasificación de los SOTR

Estos sistemas operativos de tiempo real se pueden clasificar como [3]:

1. *Kernel* propietario, basado en prioridades para aplicaciones incrustadas, se pueden citar como ejemplo a VxWorks, QNX, VRTX32.
2. Extensiones de Tiempo Real de Sistemas Operativos de Tiempo Compartido, tales como RT-UNIX, RT-LINUX, RT-MACH, entre otros.
3. *Kernel* de investigación, ejemplos: MARS, CHAOS, Spring, TIMIX.

En la clasificación de los *kernel*s de tiempo real, la primera categoría incluye muchos *kernel*s comerciales, que por muchos aspectos son optimizados de versiones de Sistemas Operativos de Tiempo Compartido. En general, el objetivo de tales *kernel*s o núcleos[3] es lograr un alto desempeño en términos del promedio al tiempo de respuesta de los eventos externos. Como una consecuencia, la principal característica que distingue estos núcleos son un rápido cambio de contexto, un tamaño pequeño, manejo de interrupciones eficiente, la habilidad de mantener código y datos en la memoria principal, el uso de primitivas desalojables, y la presencia de rápidos mecanismos de comunicación para mandar señales o eventos.

El manejo del tiempo lo realizan a través de un reloj de tiempo real, el cual es usado para iniciar los cálculos, generar las señales de alarma y verificar los intervalos de los servicios del sistema.

La comunicación entre procesos y la sincronización ocurre usualmente por semáforos binarios, buzones, eventos y señales. Sin embargo los recursos mutuamente exclusivos raramente son controlados por protocolos de acceso para prevenir la inversión de prioridades.

La segunda categoría de núcleos incluyen extensiones de tiempo real de sistemas de tiempo compartido comercial. Como ventaja principal de este tipo de sistemas operativos consiste en el uso de dispositivos periféricos estándares e interfaces que permiten incrementar el desarrollo de Aplicaciones de Tiempo Real y simplificar la portabilidad sobre diferentes plataformas de *hardware*.

Sin embargo, su desventaja principal radica en que los mecanismos básicos del *kernel* no son los apropiados para el manejo de los cálculos con restricciones de tiempo real. Esto se visualiza de mejor manera cuando es necesario crear tareas de forma dinámica. Por otra parte la prioridad puede ser reductiva para representar una tarea con diferentes atributos, tales como importancia, plazo, período.

[3] La palabra kernel o núcleo se usará de manera indistinta en este texto.

Este tipo de núcleos son utilizados principalmente en Aplicaciones de Tiempo Real no críticas, donde la falla de las restricciones temporales no causa serias consecuencias al ambiente controlado.

Tomando en cuenta las deficiencias mostradas en las dos categorías anteriores se ha decidido crear una categoría en donde se colocan los núcleos generados por las investigaciones enfocadas a desarrollar nuevos paradigmas computacionales y nuevas estrategias de planificación logrando y garantizando un alto comportamiento de la predicción del tiempo. Dichos sistemas operativos con las características antes mencionadas son llamados Sistemas Operativos de Tiempo Real Duro o Crítico.

Las principales características que distinguen estos núcleos de los demás son las siguientes:

- La habilidad de tratar tareas o procesos con restricciones temporales explícitas. Tales como períodos o plazos.

- La presencia de mecanismos de garantía que permitan verificar por anticipado sí las restricciones de la aplicación pueden ser cumplidas durante la ejecución.

- La posibilidad de caracterizar tareas con parámetros adicionales, los cuales son usados para analizar el desempeño dinámico del sistema.

- El uso de protocolos específicos para el acceso a los recursos que eviten la inversión de prioridades y limiten el bloqueo en los recursos mutuamente exclusivos.

En esta categoría existen ya una gran cantidad de sistemas operativos, y la principal diferencia entre ellos consiste en la arquitectura en la cual ellos fueron desarrollados, el enfoque estático o dinámico adoptado para la planificación de los recursos compartidos, los tipos de tareas manejadas por el *kernel*, su algoritmo de planificación, el tipo de análisis ejecutado para verificar la planificabilidad de las tareas y la existencia de técnicas para la tolerancia a fallos.

Existen varios paradigmas de diseño de Sistemas Operativos de Tiempo Real, entre los cuales destacan dos principalmente:

- SOTR activados por eventos – en donde cualquier actividad es iniciada en respuesta a la ocurrencia de un evento particular causado por en el entorno del sistema.

- SOTR activados por tiempo – en los cuales las actividades del sistema se inician en instantes predefinidos de un tiempo globalmente sincronizado.

1.4 SOTR para PC o Estaciones de Trabajo

Actualmente existen un gran número de SOTR en el mercado, pero no todos están bien documentados. Algunos de los más documentados son: QNX[4], Linux de Tiempo Real (RT-Linux)[5]. Sin embargo estos núcleos son enfocados a las computadoras personales y son usados para aplicaciones de tiempo real a alto nivel, es por ello, que existe otra sección de núcleos de tiempo real enfocados a ambientes incrustados en donde se emplean Microcontroladores y Procesadores Digitales de Señales, como ejemplo podemos citar, MicroC OS-II, DSP-OS, ERIKA, MARS, entre otros.

En QNX el *kernel* sólo realiza planificación y paso de mensajes. Cuenta con una serie de servicios implementados como procesos de sistema:

- `Proc`: Gestor de procesos.
- `Fsys`: Gestor de archivos.
- `Dev:` Gestor de dispositivos.
- `Net`: Gestor de red.

El micro*kernel* maneja la creación de procesos, gestión de memoria, y control de cronómetros. El enfoque para el procesamiento distribuido transparente permite lanzar procesos a través de la red QNX, permitiendo la herencia completa del entorno, incluyendo archivos abiertos, directorio actual, descriptores de archivos, e ID de usuarios. El micro*kernel* también incluye servicios POSIX 1003.1 (certificado) y muchos servicios de tiempo-real POSIX 1003.1b, además de diagnóstico de alta-velocidad de muestreo de eventos.

Entre las principales características podemos citar:

(a) relojes y cronómetros POSIX 1003.1b,
(b) interrupciones totalmente anidadas,
(c) manejadores de interrupciones ligables y eliminables dinámicamente,
(d) primitivas de depuración incrustadas para depuración local y remota por la red,
(e) recursos y límites del sistema configurables por el usuario,
(f) capacidades para nombrar procesos a través de la red, y

[4] www.qnx.com
[5] www.realtimelinux.org

(g) planificación de procesos según el borrador de POSIX 1003.1b.

Cuenta con una interfaz gráfica denominada Photon, con las siguientes características:

- Bajo requerimiento de memoria.
- Flexible y extensible.
- Conectividad entre plataformas.
- Construida como un *kernel* y una serie de procesos cooperantes.

Suministra gráficos profesionales de alta calidad incluso en dispositivos incrustados pequeños. Photon da un soporte gráfico escalable, con capacidades multimedia, y una interfaz totalmente personalizable.

Sobre el micro*kernel* se añaden una serie de gestores básicos (entradas, gráficos y gestor de fuentes), y un conjunto de componentes opcionales que cooperan entre sí. Esta arquitectura permite agregar o quitar módulos para escalar desde un pequeño sistema incrustado hasta unas potentes estaciones de trabajo gráficas.

La micro-GUI de Photon permite aplicaciones de Internet, electrónica de consumo, navegación en coche, telefonía, instrumentación médica, etc. Permite ver y probar en vivo sus interfaces, depurarlas, y añadirlas al sistema ya que los sistemas de desarrollo y tiempo de ejecución son la misma cosa.

Linux por su parte está diseñado para optimizar el rendimiento medio y tratar de repartir imparcialmente la CPU, no considera el peor caso. Por ello no es adecuado en tiempo real crítico. En concreto cuenta con:

- Planificación no predecible (depende de la carga).
- Resolución baja de cronómetros (10ms).
- *Kernel* no apropiativo.
- Deshabilita interrupciones como mecanismo de sincronización.
- Utiliza memoria virtual.
- Reordena solicitudes de E/S por eficiencia.

Linux está diseñado como un Sistema Operativo de Tiempo Compartido, y como tal trata de optimizar los casos medios, pero un Sistema de Tiempo Real debe tener en cuenta el peor caso. Esto produce contradicciones entre ambos tipos de sistemas, lo que es bueno para el caso medio tiende a deteriorar el peor caso. Un ejemplo clásico es la memoria virtual y la paginación por demanda. Algunos de los aspectos más relevantes que hacen que Linux no sea un Sistema Operativo de Tiempo Real son:

- El planificador está diseñado para repartir imparcialmente la carga (tiempo compartido) y además no es escalable, es decir, cuanto mayor es la carga del sistema peor realiza sus funciones. Esto hace que la planificación no sea predecible.
- La resolución de los cronómetros de UNIX es baja para un amplio número de aplicaciones de tiempo real.
- Como en los UNIX tradicionales (no System V ni Solaris), el *kernel* de Linux utiliza una política de planificación no apropiativa con lo cual es difícil acotar el tiempo de ejecución en este modo.
- Linux utiliza entre otros mecanismos de sincronización la deshabilitación de interrupciones para proteger ciertas secciones críticas, por lo que la atención a los dispositivos se pierde.
- La memoria virtual muy útil en sistemas de propósito general, genera problemas a la hora de acotar el tiempo de ejecución de los procesos.
- Por razones de eficiencia, las colas de peticiones de E/S se reordenan, por ejemplo, para optimizar el acceso a disco, esto introduce impredecibilidad en la E/S de procesos críticos.

En Real-Time Linux (RT- Linux), se construye un *Kernel* de Tiempo Real bajo Linux, de esta forma, Linux se convierte en una tarea del *Kernel* de Tiempo Real que se ejecuta sólo cuando no hay tareas de tiempo real. Por otra, parte se apropia de Linux, cuando hay tareas de tiempo real que necesitan la CPU.

El objetivo de RT-Linux es alcanzar el rendimiento de un SOTR duro, una planificación personalizable con un ajuste temporal fino, y una baja latencia de interrupción, y todo ello con los mínimos cambios del *kernel* de Linux para poder seguir utilizando todos los servicios de los que dispone el sistema. Inspirado en MERT[6], utiliza el concepto de Máquina Virtual para la emulación de interrupciones, y construye un pequeño SOTR bajo el *kernel* de Linux. Para este SO, el propio Linux no es más que una tarea nula. Se interpone una capa de emulación *software* entre el *kernel* de Linux y el controlador *hardware* de interrupciones. Esto evita que Linux pueda deshabilitar las interrupciones. Interrupciones como CLI (Clear Interruption), STI (Set Interruption) e IRET (Interruption Return) son sustituidas por sus versiones emuladas por *software*.

Para evitar la sobrecarga del cambio de contexto, invalidación de TLB (Translation Lookaside Buffer)[7], llamadas al sistema, todas las tareas de Tiempo Real se ejecutan en el mismo espacio de direcciones, y se ejecutan

[6] Sistema operativo que fue hecho en los laboratorios de Bell en los años 70.

[7] Es un mecanismo que se emplea para la gestión de memoria en particular con la interpretación de direcciones, en donde se reservan partes de la memoria principal para tablas de página de memoria relacionadas [13].

como módulos *kernel*, lo que permite su carga dinámica. La asignación de recursos es estática, es decir, en tareas de Tiempo Real no se deben utilizar funciones tales como *kmalloc*.

El propio planificador es un módulo que se carga en el *kernel*, por defecto, implementa un algoritmo apropiativo basado en prioridades estáticas, pero también implementa los algoritmos Tasa Monótona (RM), y Primero el Plazo más Corto (EDF).

Una aplicación de RT-Linux se divide en dos partes: una de tiempo real que debe ser rápida, simple y pequeña, otra que se ejecuta en espacio de usuario y debe controlar aspectos como E/S, etc. Estas dos tareas se comunican mediante dos FIFOS de tiempo real (son unidireccionales).

Un FIFO de tiempo real es aquel que se comporta de manera no bloqueante en el lado de la aplicación de Tiempo Real y de manera convencional en la tarea de usuario. El *buffer* de los FIFOs se asigna en el espacio de direcciones del *kernel*.

1.5 SOTR para MCU y DSP

Como se mencionó en el apartado anterior existen en el mercado un conjunto de *Kernel*s de Tiempo Real enfocados al área de sistemas incrustados en los cuales están presentes los Microcontroladores y los Procesadores Digitales de Señales. En este apartado se mencionarán algunas características de dos núcleos que han servido de guía e inspiración para el diseño del *Kernel* de Tiempo Real para el DSP56F827, uno es microC OS-II[8] y el otro es DSP-OS[9], que al momento de la redacción de este documento ha sido cambiado de nombre conociéndosele ahora como Fusion RTOS.

MicroC OS-II es un núcleo portable, escalable, apropiativo, de tiempo real, multitarea. Está escrito en ANSI C y contiene una pequeña porción de código en lenguaje ensamblador para adaptarlo a diferentes arquitecturas de procesadores. Entre sus características sobresalen las siguientes:

- **Apropiativo o Expropiativo** – Significa que MicroC OS-II siempre ejecuta la tarea lista con la prioridad más alta.

[8] www.ucos-ii.com
[9] www.dspos.com

- **Multitarea** – MicroC OS-II puede controlar hasta 64 tareas; sin embargo, no se recomienda usar toda su capacidad, dado que es necesario reservar unas tareas para el control del *kernel*.
- **Determinista** – Los tiempos de ejecución para la mayoría de las funciones y servicios son deterministas, lo cual significa que se puede saber siempre cuanto tiempo MicroC OS-II tomará en ejecutar una función o servicio.
- **Servicial** – Provee un número de servicios del sistema, tales como semáforos, semáforos de exclusión mutua, banderas de eventos, buzones de mensajes, cola de mensajes, entre otros.

Otro de los Sistemas Operativos de Tiempo Real para Microcontroladores y en específico para los Procesadores Digitales de Señales (DSP's) es el sistema operativo DSP-OS. Es un núcleo basado en prioridades, apropiativo, multitarea, diseñado para optimizar la arquitectura de los DSP's. DSP-OS provee una serie de servicios para el monitoreo del desempeño de las tareas, estadísticas del uso de la pila de *software* en tiempo de ejecución, aserciones de sobreflujo de la pila, detección de errores, entre otras funcionalidades.

Los componentes de este Sistema Operativo de Tiempo Real son los siguientes:

- **Tareas**: Procesos independientes que son ejecutados, cuentan con opciones que incluyen la prioridad, tiempo y desalojo.

- **Grupos de Pila Compartida**: Grupos de tareas que comparten una memoria de pila.

- **Tareas de Recursos Limitados**: Son procesos, usualmente lanzados por interrupciones.

- **Eventos**: Usados para la sincronización de tareas y la comunicación entre procesos.

- **Colas**: Usadas para la comunicación entre procesos. Una cola puede aceptar un mensaje de múltiples emisores.

- **Buzones**: Usados para la comunicación de procesos, un buzón puede pasar dos palabras desde una tarea a otras, o desde una interrupción a una tarea.

- **Semáforos**: Usados por las tareas para la sincronización y la protección de los datos.

- **Temporizadores**: Usados para lanzar procesos dentro del sistema operativo.

1.6 El Kernel para el DSP56F827 (KTR-DSP)

A pesar de una variedad de Sistemas Operativos de Tiempo Real, en ocasiones el conseguir dicho sistema requiere de una gran inversión económica, es por ello, que puede ser factible desarrollar su propio *Kernel* de Tiempo Real, para emplearlo en el desarrollo de las Aplicaciones de Tiempo Real propias y específicas al *hardware* con que se cuenta a la hora de desarrollar la aplicación.

Dado que actualmente los desarrollos de Aplicaciones de Tiempo Real se usan a menudo con los sistemas incrustados y en ellos como unidad principal de control se utiliza el Procesador Digital de Señales, se decidió desarrollar un núcleo de tiempo real que pueda ser empleado para el control de las tareas que forman parte de dichas aplicaciones, permitiendo con esto contar con un núcleo que controlará específicamente las funcionalidades del Procesador Digital de Señales que esté en uso.

El KTR-DSP evoluciona como una alternativa más para agregar funcionalidad y disminuir los costos en el diseño de aplicaciones de Sistemas de Control en Tiempo Real. El KTR-DSP se propone con la finalidad de administrar tareas periódicas de acuerdo a sus restricciones específicas de tiempo (como su plazo de vencimiento) y a la vez que proporcione las funciones básicas ofrecidas por los *Kernel*s de Tiempo Real comerciales, como son el uso de semáforos, la comunicación entre procesos, entre otras.

El Procesador Digital de Señales seleccionado para implementar el núcleo de Tiempo Real, es un procesador de 16 bits que cuenta con una arquitectura Harvard, que realiza operaciones con una frecuencia de hasta 80 MHz, cuenta con temporizadores y un banco de memoria que puede ser expandible.

CAPÍTULO 2 - ARQUITECTURA Y CONJUNTO DE INSTRUCCIONES BÁSICAS DEL DSP56F827

2.1 El DSP56F827

El DSP56F827 es miembro de la familia Procesadores Digitales de Señales basado en el núcleo DSP56800 de Motorola.

Caracterizado por su tamaño de la memoria y las opciones de periféricos, el chip multifuncional DSP56F827 ofrece soluciones para una amplia variedad de aplicaciones. El núcleo provee más potencia de desarrollo que cualquier otro chip de control disponible mientras ahorra espacio y dinero en el diseño[11].

La potencia de procesamiento del DSP y la funcionalidad de un Microcontrolador con un conjunto flexible de periféricos se combinan en un solo chip (DSP56F827). Este diseño de chip proporciona una solución extremadamente rentable y compacta para un número de aplicaciones.

La familia de chips se diseña para proporcionar el control de aplicaciones tales como:

- Motores de inducción de CA – para industrias y aplicaciones tales como lavadoras, ventiladores.
- Motores de escobilla de CC – para elevadores de ventanas de automóviles, antenas eléctricas, juguetes.
- Motores sin escobilla de CC - utilizados en automotores y aplicaciones incluyendo los ventiladores de la PC, ventiladores del techo, sopladores, lavadoras, sistemas de manejo de la energía eléctrica.
- Y para el control de potencia durante:
    - Aplicaciones generales de conversión/inversión
    - Y en Unidades de Potencia Ininterrumpida – en línea, línea interactiva, y espera.

Cualquier DSP56F827 puede ser empleado en las siguientes aplicaciones:

- Control Automotriz.
- Modem de línea de potencia.
- Fuentes de alimentación ininterrumpida.
- Implementación de sistemas de Telefonía.
- Supresión de Ruido.

- Seguridad casera.
- Codificadores.
- Regulación de temperatura.
- Control y monitoreo remoto.
- Contestadoras digitales de teléfonos.
- Sistemas de administración de combustible.
- Reconocimiento de voz.
- Sistemas de seguridad y rompimiento de cristales.
- Control de semáforos (vehiculares).
- Lectores de etiquetas de identificación.

2.2 Descripción de la Familia DSP56800

El núcleo DSP56800 está construido sobre una arquitectura estilo Harvard constituido por tres unidades operando en paralelo, permitiendo hasta seis operaciones por ciclo de instrucción. El modelo de programación estilo microprocesador y el conjunto optimizado de instrucciones permite la generación directa de código compacto y eficiente para ambas aplicaciones DSP y MCU[10]. El conjunto de instrucciones es altamente eficiente para compiladores C/C++ en beneficio de permitir el desarrollo rápido y eficiente de las aplicaciones de control.

Los chips DSP56800 soportan la ejecución de programas a partir de las memorias interna o externa, proporcionando dos líneas dedicadas de interrupción y 32 líneas multipropósito de entrada-salida. El controlador del DSP incluye memoria Flash de Programa y de Datos, cada una programable a través del puerto común del grupo de acción de pruebas JTAG (Joint Test Action Group), con RAM de Programa y de Datos. El controlador también soporta la ejecución de programas desde la memoria externa. El núcleo del DSP56800 es capaz de tener acceso a dos operandos de datos desde la memoria RAM interna de Datos por ciclo de instrucción.

El controlador del DSP también proporciona un completo conjunto de periféricos programables estándares: Interfaces de Comunicación Serial (Serial Comunications Interfaces - SCIs), Interfaz de Periférico Serial (Serial Peripheral Interface – SPI), y un contador de tiempo cuádruple (Quad Timer). Cualquiera de estas interfaces puede usarse como periféricos de propósito general de entrada-salida (General Purpose In/Out - GPIO) si las funciones anteriores no son requeridas. Un controlador de interrupciones interno y un GPIO dedicado, son incluidos también en algunos DSP's.

[10] Microcontroller Unit- Microcontrolador

2.3 Diagrama de Bloques del Núcleo DSP56800

Un diagrama de bloques completo de la arquitectura del núcleo DSP56800 se muestra en la Figura 2.1.

El núcleo DSP56800 es alimentado por una memoria de programa y de datos, una interfaz de memoria externa, y varios periféricos apropiados para aplicaciones incrustadas. Los bloques incluyen:

- Unidad Aritmética Lógica de Datos (Data ALU)
- Unidad de Generación de Direcciones (AGU)
- Controlador de Programa y Unidad *Hardware* de Lazos
- Unidad de Manipulación de Bits
- Puerto de Emulación en el chip (OnCE)
- Controlador de Interrupciones
- Bus de Puente externo
- Buses de Dirección
- Buses de Datos

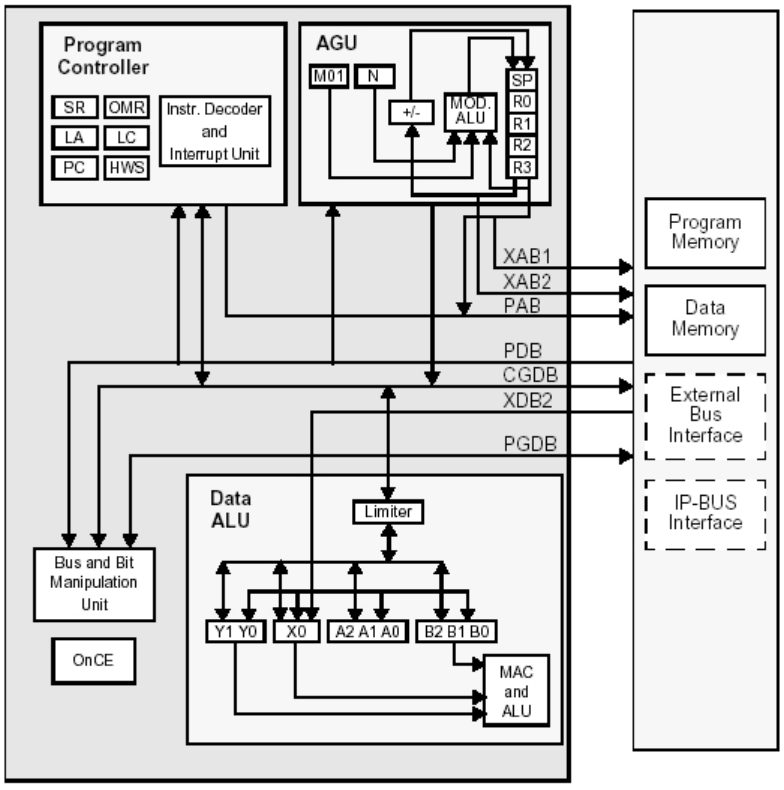


Figura 2.1- Diagrama de Bloques de la Arquitectura del Núcleo DSP56800.

El Controlador de Programa, la AGU y la ALU de Datos cada uno contienen un conjunto de registros y una lógica de control discreta, así que cada uno puede funcionar independientemente y en paralelo a los otros. Asimismo, cada unidad funcional interactúa con las otras unidades, con la memoria, y con los periféricos mapeados en memoria a través de los buses internos de datos y direcciones, según lo demostrado en la Figura 2.2.

Es posible que en un solo ciclo de instrucción el Controlador de Programa traiga una primera instrucción, la AGU genera dos direcciones para una segunda instrucción y la ALU de Datos ejecute una multiplicación en una tercera instrucción. De manera similar, la Unidad de Manipulación de Bits puede realizar una operación de la tercera instrucción descrita previamente en vez de la multiplicación en la ALU de Datos. La arquitectura se canaliza para tomar ventaja de las unidades paralelas y significativamente disminuir el tiempo de ejecución de cada instrucción.

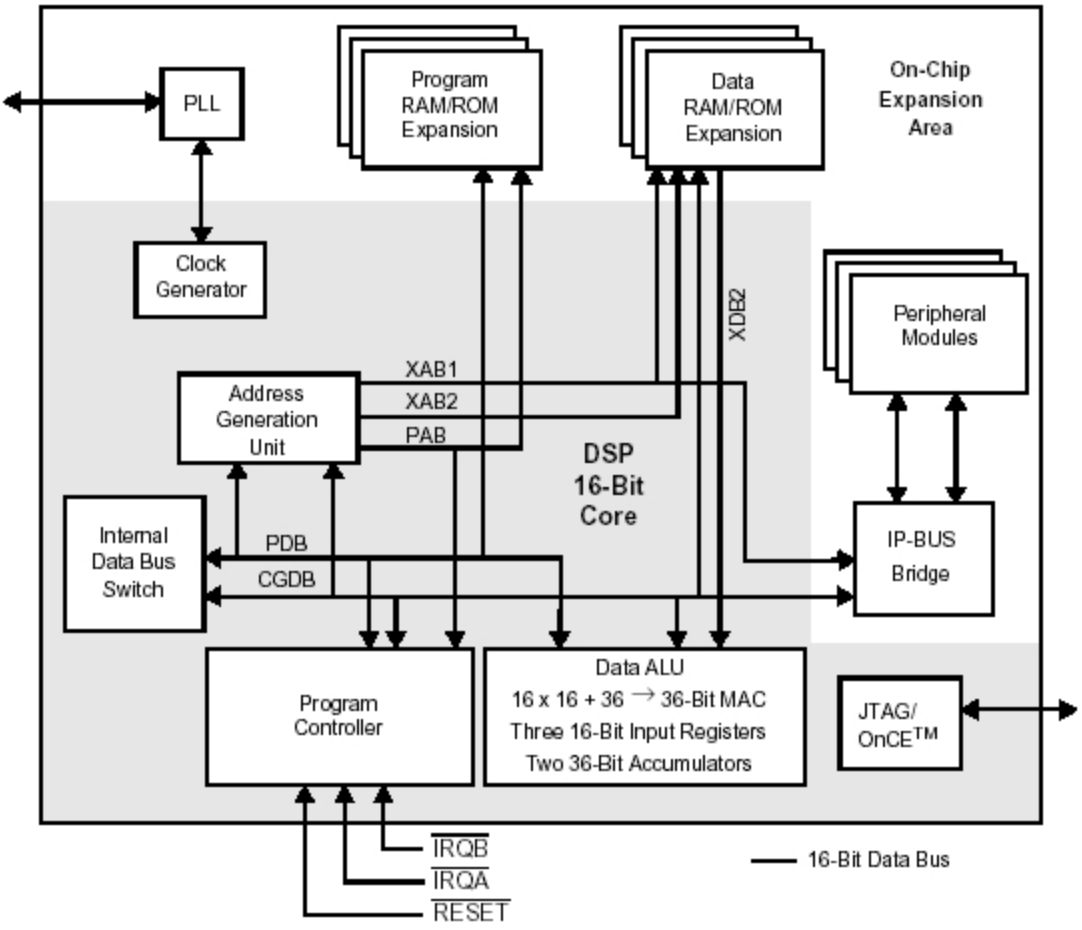


Figura 2.2- Diagrama de Bloques de los Buses del Núcleo DSP56800.

2.4 Características y Beneficios del DSP56F827

Como el DSP56F827 está basado en el núcleo DSP56800 consiste de unidades funcionales operando en paralelo, incrementando el desempeño de la máquina. La arquitectura estilo Harvard consta de tres unidades operando en paralelo. Estas tres unidades de ejecución permiten hasta seis operaciones durante cada ciclo de instrucción. El conjunto de instrucciones es también altamente eficiente para compiladores de C.

Sus características son:

- Hasta 40 MIPS a una frecuencia del núcleo de 80 MHz [10].
- Funcionalidad de DSP y MCU en una arquitectura unificada y eficiente para el lenguaje C.
- Ciclos DO y REP por *Hardware*.
- Memoria Flash de Programa con 63K palabras[11] de 16 bits.
- Memoria RAM de Programa con 1K palabras de 16 bits.
- Memoria Flash de Datos con 4K palabras de 16 bits.
- Memoria RAM de Datos con 4K palabras de 16 bits.
- Hasta 64K palabras de 16 bits para la expansión de la memoria externa de Programa y de Datos.
- Depuración por el puerto JTAG/OnCE.
- Conjunto de instrucciones MCU que soporta las funciones de DSP y Microcontrolador: MAC, Unidad de Manipulación de Bits, 14 modos de direccionamiento.
- Chip programable de 8 canales de selección.
- Convertidor Análogo-Digital (ADC) de 12 bits, con 10 canales.
- Interfaz Serial Síncrona (SSI).
- Interfaz de Puerto Serial (SPI).
- Interfaces de Comunicación Serial (SCIs).
- Contador de Tiempo Cuádruple de propósito general.
- Contador de Tiempo del Día (TOD).
- Embalaje LQFP[12] de 128 pines.

Sus beneficios:

- La memoria Flash proporciona almacenamiento confiable permanente, eliminando así los dispositivos de almacenaje requeridos.
- Fácil para programar con herramientas flexibles de desarrollo de aplicaciones.

[11] Debe entenderse como "*palabras o words*" a la amplitud en bits que tiene el DSP para el manejo de datos.
[12] Low Profile Quad Flat Pack –Paquete horizontal de perfil bajo.

- Optimizado para la eficiencia con compiladores del lenguaje C.
- Actualización sencilla de la memoria Flash con SPI, SCI, u OnCE usando el cargador de arranque del chip.
- Suporta protocolos de comunicación de 9 bits.
- Interfaz sencilla con los otros dispositivos seriales asíncronos de comunicación y las memorias eléctricamente borrables externas.
- Capacidad de Convertidor Digital-Análogo (DAC) utilizando el contador de tiempo cuádruple para proporcionar independencia a cada canal del contador de tiempo.
- Depuración sofisticada usando OnCE para ver el contenido del núcleo, de los periféricos y de la memoria.
- Modo de Paro análogo para ahorro de energía.

Un diagrama de bloques completo del DSP56F827 se muestra en la Figura 2.3.

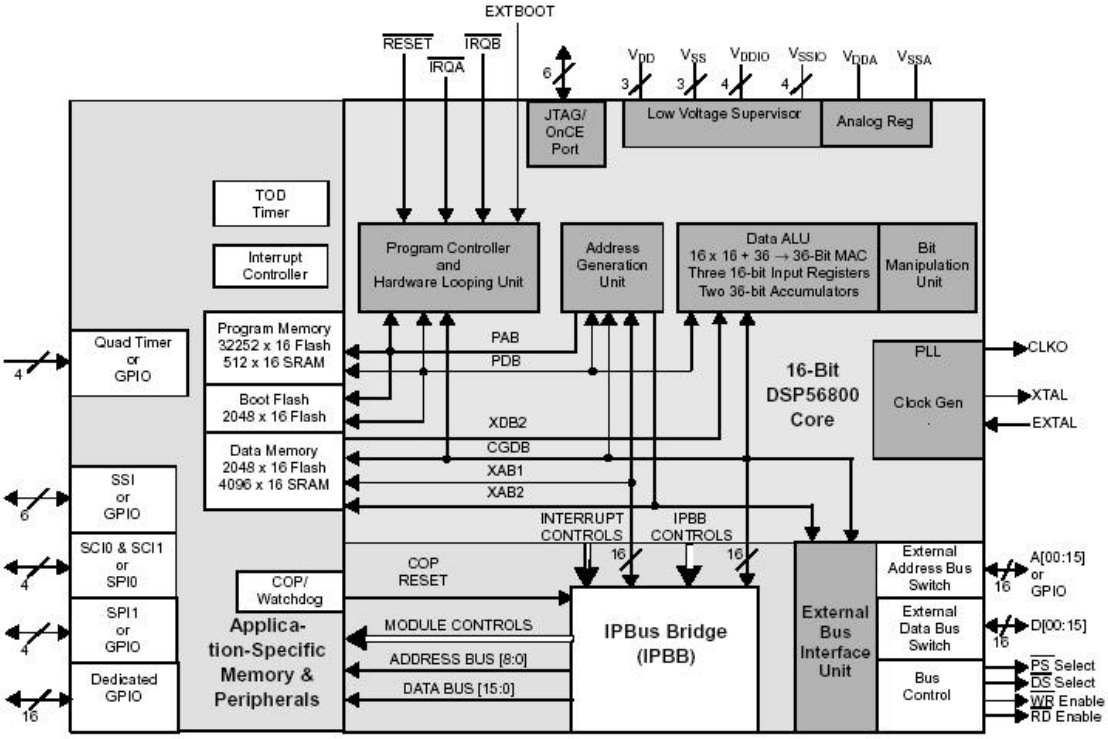


Figura 2.3- Diagrama de Bloques del DSP56F827.

CAPÍTULO 3 - DISEÑO Y ESTRUCTURA DEL KERNEL

3.1 Consideraciones para el Diseño de un KTR

Generalmente los núcleos de Tiempo Real son diseñados para ser utilizados en los Sistemas de Tiempo Real, estos sistemas pueden describirse como sistemas que constan de los siguientes tres componentes: el subsistema controlado, el subsistema de control, y el subsistema operador.

El subsistema controlado representa la aplicación o entorno (p.ej. una planta industrial), que dicta los requisitos de tiempo real.

El subsistema de control gobierna algunos equipos de computación y comunicación para uso desde el sistema controlado.

El subsistema operador inicia y vigila la actividad del sistema entero. La interfaz entre el sistema controlado y el de control consta de una serie de dispositivos como sensores y actuadores. La interfaz entre el subsistema de control y el operador consiste en una interfaz hombre-máquina. El subsistema controlado está implementado mediante tareas, tareas de aplicación, que se ejecutan utilizando los equipos gobernados por el subsistema de control. Este último subsistema puede construirse mediante un número elevado de procesadores, equipados con recursos locales como memoria y discos, interconectados por una red de área local de tiempo real. Estos procesadores y recursos están gobernados por un SOTR.

De acuerdo con esto y con la clasificación de tareas (periódicas, aperiódicas, esporádicas)[13], podíamos pensar que la principal responsabilidad de un SOTR es garantizar que la ejecución individual de cada tarea satisfaga los requisitos temporales impuestos. Sin embargo, con objeto de cumplir esta responsabilidad, el objetivo del SOTR no puede establecerse sólo con minimizar el tiempo de respuesta medio de cada tarea de aplicación; en su lugar, el aspecto fundamental es que sea predecible, es decir, la conducta temporal y funcional debe ser tan determinista como sea necesario para satisfacer la especificación. Así, un *hardware* rápido y unos

[13] Periódicas: Son tareas activadas entre intervalos regulares de tiempo fijo.
Aperiódicas: Son tareas manejadas por eventos que permiten que los eventos lleguen simultáneamente o dentro de un tiempo arbitrariamente pequeño uno de otro.
Esporádicas: Son tareas manejadas por eventos, ellas son activadas por una señal externa o un cambio de alguna relación [20].

algoritmos eficientes son útiles al construirlo; sin embargo, no son suficientes para garantizar la conducta predecible necesaria para tal sistema.

Para que una colección de tareas cooperantes pueda tener los recursos que necesita, es obvio que un buen SOTR debe ser capaz de realizar una asignación de recursos y una planificación de la CPU integrada. Pero no sólo eso, es también necesario acotar las primitivas que suministra. Estos paradigmas han llevado al desarrollo de dos arquitecturas diferentes, denominadas arquitecturas activadas por eventos (ET – Event Task) y arquitecturas activadas por tiempo (TT- Time Task).

En ambos casos, la predecibilidad del SOTR se alcanza utilizando estrategias (diferentes) para evaluar, antes de la ejecución de cada tarea, los recursos necesarios, y disponibles para satisfacer esas necesidades. Sin embargo, en la arquitectura de tareas activadas por eventos (ET), las necesidades de recursos y la disponibilidad pueden variar en tiempo de ejecución, y deben ser juzgadas dinámicamente. Por el contrario, en la arquitectura de tareas activadas por tiempo (TT), las necesidades pueden calcularse fuera de línea, basándose en análisis anteriores a la ejecución de la aplicación específica; si esas necesidades no pueden ser anticipadas, se utiliza la estimación del peor caso.

Los defensores de las arquitecturas TT critican el enfoque de la arquitectura ET, debido a que por su propia naturaleza, pueden caracterizarse por un número excesivo de posibles conductas que deben ser analizadas cuidadosamente con objeto de establecer su predecibilidad. Por el contrario, los defensores de las arquitecturas ET indican que estas arquitecturas son más flexibles que las TT, y son ideales para una amplia clase de aplicaciones que no permiten predeterminar sus requisitos de recursos. En particular, argumentan que las TT, debido al enfoque de la estimación del peor caso, son más propensas a desperdiciar recursos con objeto de suministrar una conducta predecible.

Tanto en las arquitecturas ET como TT, para determinar los recursos necesarios y su disponibilidad se deben tener en cuenta los requisitos temporales de las aplicaciones. De aquí que las cuestiones de gestión del tiempo, que caracterizan la conducta temporal del sistema, sean cruciales en el diseño de un Sistema de Tiempo Real.

Existen una serie de estándares que se deben considerar para el diseño e implementación de los núcleos en Tiempo Real, debido a ello paralelamente al desarrollo y construcción de *kernel*s de sistemas operativos para objetivos específicos de aplicaciones y plataformas arquitectónicas, la industria ha desarrollado estándares para caracterizar la funcionalidad típica ofrecida

por el sistema operativo. Uno de estos esfuerzos es el estándar POSIX 1003.1b (anteriormente 1003.4). Este estándar derivado de UNIX, incluye entre otras las siguientes funciones:

- E/S tanto síncronas como asíncronas.
- Primitivas IPC (memoria compartida, semáforos, y eventos asíncronos).
- La capacidad para bloquear memoria.
- Planificación por prioridades.
- Archivos de tiempo real
- Cronómetros.

Generalmente estos Sistemas Operativos de Tiempo Real se justifican para su uso cuando las aplicaciones para las cuales se desean aplicar se pueden dividir en fragmentos de código, en donde cada fragmento se puede caracterizar como una tarea o proceso y en donde dicha tarea llevará acabo una función específica de la aplicación, además es necesario saber de antemano las restricciones temporales para cada función.

3.2 Estructura del KTR para el DSP

El *Kernel* de Tiempo Real del DSP provee las siguientes actividades básicas: Administración de Procesos, el Manejo de Interrupciones y la Sincronización de Procesos. Estas actividades sirven como base para plantear su estructura.

El KTR-DSP se encaja en los *kernel*s denominados monolíticos por contar con todas las funcionalidades posibles integradas en el núcleo. Se trata de un programa de tamaño considerable que se debe recompilar por completo cada vez que se desea añadir una nueva posibilidad [13].

La estructura del *kernel* tiene una forma jerárquica, dicha estructura puede ser dividida en cuatro capas: la capa de máquina, la capa de administración de listas, la capa de administración del procesador y la capa de servicio[16].

La Figura 3.1 muestra la estructura planteada para el desarrollo del KTR-DSP.

3.2.1 Capa de Máquina

La Capa de Máquina es aquella que interactúa directamente con el procesador, por lo tanto está escrita en lenguaje ensamblador. Las funciones principales de este nivel tratan con actividades como el cambio de contexto,

el manejo de interrupciones, y el manejo del temporizador. Estas funciones no son visibles al nivel de usuario.

El cambio de contexto forma parte de las actividades primordiales del *kernel*, y se presenta cuando una tarea es desalojada (del uso de la unidad central de proceso) por otra tarea que en ese instante tiene mayor prioridad que la tarea que se encuentra en ejecución. En el cambio de contexto se guarda el estado de la tarea actual y se carga el estado de la tarea a ejecutarse.

Figura 3.1- Estructura Jerárquica del Kernel de Tiempo Real.

El manejo de interrupciones permite atender un servicio cuando se hace una solicitud de interrupción, en el *Kernel*, este manejo se lleva a cabo considerando uno de los enfoques suministrados (Despachador de Interrupciones Super-rápido, Rápido o Normal) por la librería de funciones proporcionadas por el SDK de Motorola para ser usadas con la familia de DSP 56800.

El manejo del temporizador, es fundamental para el buen funcionamiento del *Kernel*, ya que permite activar funciones en intervalos de tiempo definidos o solicitar el servicio a una interrupción[14]. Cabe aclarar que este manejo del tiempo, se lleva a cabo a través de la implantación de una interrupción que controle o invoque un servicio para atender las diversas funciones del temporizador.

En particular, cada vez que se hace una solicitud de servicio por la interrupción del temporizador, la rutina de servicio correspondiente debe

[14] Se debe entender el término interrupción como "una señal del tiempo que causa que el procesador suspenda la ejecución del proceso o tarea actual e inicie otro proceso (el que da el servicio)".

salvar el contexto de la tarea en ejecución, incrementar el contador del tiempo del sistema; si el tiempo actual es mayor que el tiempo de vida del sistema entonces debe generar un error de tiempo; si el tiempo actual es mayor que algún plazo crítico de alguna tarea, debe generar un error de sobreflujo de tiempo para dicha tarea; debe también reanudar las tareas que se encuentran ociosas si cualquiera de ellas tiene que iniciar un nuevo período, en el supuesto caso que al menos una tarea ha sido reanudada se invoca el planificador. Además la rutina de servicio del temporizador debe remover todas las tareas zombis para las cuales su plazo ha expirado; carga también el contexto de la tarea actual y regresa de la interrupción.

3.2.2 Capa de Administración de Listas

La Capa de Administración de Listas[15] mantiene la pista del estado de las diferentes tareas. En esta capa se provee de funciones básicas para insertar o remover tareas de una lista.

En esta etapa se deben definir las estructuras de datos que controlaran las tareas o procesos, así como los demás componentes del *kernel*, como lo son los semáforos y los *buffers* de comunicación. Al mismo tiempo se deben definir las estructuras para manejar las listas que controlaran los componentes antes citados.

Para llevar un control y saber el estado de los procesos se define una estructura de datos denominada Bloque de Control de Tareas (véase Figura 3.2), en el cual se almacenan los datos más importantes de la tarea, como son su período, plazo de vencimiento, su prioridad, el apuntador a la pila de la tarea, el factor de utilización, y los apuntadores para ligarlo hacia las otras tareas cuando se encuentre dentro de la lista.

[15] En el texto se usará de forma indistinta el término “lista” o “cola” refiriéndose con estos términos “A un conjunto de tareas esperando por un dado tipo de recurso y ordenadas de acuerdo a un criterio”.

Apuntador al tope de la pila
Índice del siguiente BCT
Índice del anterior BCT
Plazo
Estado de la tarea
Prioridad
Tiempo de Retardo
Período
Tipo de la tarea
Tiempo de ejecución
Factor de utilización

Figura 3.2- Estructura del Bloque de Control de Tarea para el KTR-DSP.

La lista para el control de tareas, se implementa a través de un arreglo en el cual tiene un apuntador al inicio y es el tope de la lista que sirve para controlar y tener seguimiento de las demás tareas ligadas a dicha lista (véase Figura 3.3).

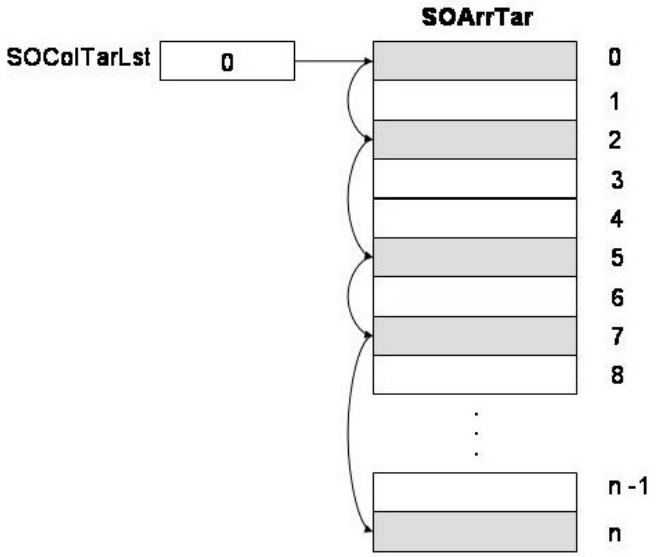


Figura 3.3- Implementación de la Lista de Control de Tareas a través de un Arreglo de BCT's.

Para el control de los semáforos se crea una estructura de datos denominada Bloque de Control de semáforos en la cual se almacena un contador, una lista del semáforo para llevar el control de los procesos bloqueados y un apuntador al siguiente bloque de control dentro de la lista (véase Figura 3.4). Al igual que los procesos, la lista de los semáforos se implementa a través de un arreglo.

Figura 3.4- Estructura del Bloque de Control del Semáforo para el KTR-DSP.

En los *buffers* de comunicación empleados para el intercambio de datos entre los procesos se crea un estructura de datos denominada *Buffer* Asíncrono Cíclico (BAC), en la cual a su vez contiene un apuntador a una lista de *buffers*, que son empleados para almacenar el dato a compartir entre las tareas; contiene un apuntador al *buffer* más reciente, que es el que contiene el dato actual colocado en dicha estructura; a su vez puede contener el número de *buffers* dentro de la estructura, así como su dimensión. Se define también otra estructura de datos en la cual se almacena el dato a compartir entre las tareas, esta estructura contiene un apuntador al dato, otro apuntador que indica el siguiente *buffer* dentro de la lista y cuenta también con un contador para indicar el número de tareas que están usando el *buffer* (véase Figura 3.5).

3.2.3 Capa de Administración del Procesador

La Capa de Administración del Procesador se encarga de las operaciones de planificación y despacho de tareas[18].

El mecanismo de planificación es el encargado de seleccionar a una tarea para ejecutarse de acuerdo a la política de planificación correspondiente. El algoritmo de planificación seleccionado nos indica la política de planificación que se emplea para la selección de las tareas. En el *Kernel* de Tiempo Real para el DSP se implementa el algoritmo de planificación denominado EDF por sus siglas en inglés que literalmente se puede traducir como “Primero el Plazo más Corto”, este algoritmo cuenta con una regla de planificación dinámica[16] que selecciona las tareas o procesos de acuerdo a sus plazos de vencimiento absolutos (plazo de la tarea más el tiempo de

[16] Con este método de planificación todas las tareas activas son reordenadas cada vez que una nueva tarea entra al sistema u ocurre un evento.

activación de la j-ésima instancia de la tarea), específicamente las tareas con plazos más cortos serán ejecutadas en prioridades más altas. El plazo absoluto queda expresado en Ec-1.

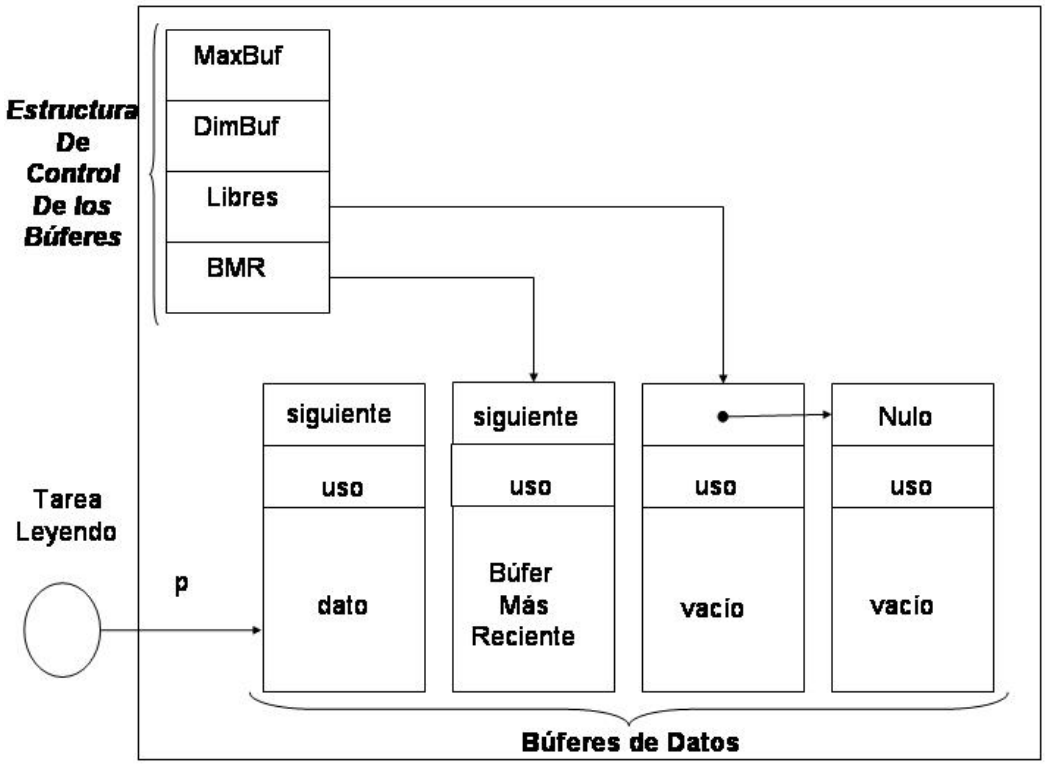


Figura 3.5- Estructura General del Buffer Asíncrono Cíclico para el KTR-DSP.

$$d_{i,j} = \Phi_i + (j-1)T_i + D_i \qquad \textbf{Ec. 3–1}$$

Donde:

$d_{i,j}$ Plazo absoluto de la j-ésima instancia de la tarea t_i.

Φ_i Fase de la tarea t_i; que es, el tiempo de arribo de su primer instancia ($\Phi_i = r_{i,1}$).

j Número de instancia de la tarea t_i.

T_i Período de la tarea t_i.

D_i Plazo relativo de la tarea t_i, que es el tiempo dentro del cual una tarea de tiempo real debería completar su ejecución.

Por otra parte, este algoritmo es intrínsecamente preferente (expropiativo), es decir, la tarea actualmente en ejecución es expulsada por cualquier otra instancia periódica con el plazo más corto (mayor prioridad) cuando ésta se vuelve activa.

En el diseño del Kernel se decidió emplear el algoritmo de planificación EDF porque con su empleo se logra alcanzar el máximo factor de utilización de la CPU, es decir de alrededor del cien por ciento[8]. Además de éste valor en el desempeño de la CPU, el algoritmo EDF permite planificar tareas periódicas y aperiódicas, a diferencia de otros algoritmos de prioridad fija como el algoritmo de Tasa Monótona (Rate Monotonic) que generalmente se emplea para la planificación de tareas periódicas y en donde se logra un factor de utilización de la CPU de alrededor del setenta por ciento para una cantidad larga de tareas. Según Giorgio Buttazzo en su libro Hard Real-Time Systems en donde analiza el desempeño del algoritmo EDF y otros algoritmos más en dos escenarios de tareas de tiempo real, las periódicas y las aperiódicas, en ambos, Buttazzo caracteriza al algoritmo EDF como un algoritmo óptimo, además de caracterizarlo como un algoritmo apropiativo (preemptive) donde el desalojo de una tarea de menor prioridad por otra de mayor prioridad es permitido en cualquier instante, permitiendo a la vez que las tareas tengan un tiempo de arribo arbitrario.

3.2.4 Capa de Servicio

La Capa de Servicio proporciona todos los servicios visibles al usuario haciéndole parecer un conjunto de llamadas al sistema. Los servicios típicos son creación, abortar, suspensión de tareas y consulta de las operaciones del sistema.

En esta capa es donde se implementan las primitivas para que el usuario pueda hacer uso de los semáforos para la sincronización de sus procesos. Así como también el usuario puede invocar las primitivas que manipulan los *buffers* asíncronos cíclicos para permitir que una tarea intercambie datos con otra u otras.

Dado que el kernel emplea para la planificación el algoritmo EDF y aunque el algoritmo puede planificar cualquier conjunto de tareas periódicas hasta alcanzar un factor de utilización de la CPU del cien por ciento no es un algoritmo adecuado para implementar los protocolos de herencia de prioridades para los semáforos, por las siguientes consideraciones según Rajkumar [15]:

- Si se usa el algoritmo EDF, no es posible predecir con anterioridad cual tarea fallará su plazo bajo sobrecargas transitorias.
- No existen esquemas dentro de los algoritmos de planificación de plazos para resolver problemas prácticos tales como la insuficiencia de niveles de prioridad y buferización.
- El análisis de algoritmos de planificación con prioridades dinámicas surge para ser tratado sólo bajo condiciones idealizadas y no puede ser así cuando ciertas restricciones prácticas son añadidas al problema. Por otro lado, los algoritmos de planificación de prioridades estáticas son analizables para un amplio ámbito de problemas, es por ello que es más fácil implementar estos protocolos empleando el algoritmo de Tasa Monótona, en lugar de los algoritmos con prioridades dinámicas como el EDF.

Sin embargo existen otros protocolos orientados a los algoritmos de planificación dinámica (tales como la Política de Pila de Recursos –Stack Resource Policy) que tratan también el problema de inversión de prioridades, pero que no se implementaron en este kernel por no formar parte de los alcances de la tesis y porque se prevé que se implementen en trabajos futuros.

Los semáforos implementados en el kernel son definidos como semáforos binarios, estos semáforos sencillos se usan principalmente para la sincronización de tareas y la exclusión mutua en las regiones críticas de las tareas. En el kernel dichos semáforos no tienen expiración de plazo de espera.

3.3 Estados del Proceso o Tarea

En cualquier *kernel* que soporta la ejecución de actividades concurrentes sobre un simple procesador, donde los semáforos son usados para la sincronización y la exclusión mutua, existen al menos tres estados en los cuales una tarea puede entrar:

- Ejecución. Una tarea entra en este estado cuando inicia su ejecución en el procesador.
- Listo. Éste es el estado de todas las tareas que están listas para ser ejecutadas, pero que no pueden ser ejecutadas porque el procesador está asignado a otra tarea.
- Espera. Una tarea entra a este estado cuando ejecuta una primitiva de sincronización para esperar por un evento. Cuando se usan los semáforos esta operación es una primitiva "espera" sobre un semáforo bloqueado. En este caso, la tarea es insertada en una lista asociada

con el semáforo. La tarea en el tope de esta lista es reanudada cuando el semáforo es desbloqueado por otra tarea que ejecuta una primitiva "señal" sobre ese semáforo.

Existen otros estados en los cuales una tarea puede entrar cuando ésta existe en el *Kernel*, tales como OCIO, DETENIDO, ZOMBI.

El estado de OCIO se emplea en el *Kernel* de Tiempo Real que soporta la ejecución de tareas periódicas, y una tarea periódica entra en este estado cuando completa su ejecución y tiene que esperar por el inicio del siguiente período. Para que una tarea sea reanudada por el temporizador debe notificar el fin de su ciclo ejecutando un llamado específico al sistema (en el caso de nuestro *kernel* la primitiva es *SOTarTermCcl*), el cual coloca la tarea en el estado de OCIO y asigna el procesador a otra tarea lista para ejecutarse. En el tiempo adecuado, cada tarea periódica que se encuentra en el estado de OCIO será reanudada por el *kernel* e insertada en la lista de tareas listas a ejecutarse. Esta operación es realizada por una rutina activada por el temporizador, el cual verifica, en cada tic o ciclo de reloj, si algunas tareas tienen que ser reanudadas.

Cuando una tarea ejecuta una primitiva "detener o retardo", la cual suspende un proceso por un intervalo de tiempo dado, coloca la tarea en un estado de sueño (DETENIDO), hasta que sea reanudada por el temporizador después de alcanzar el intervalo.

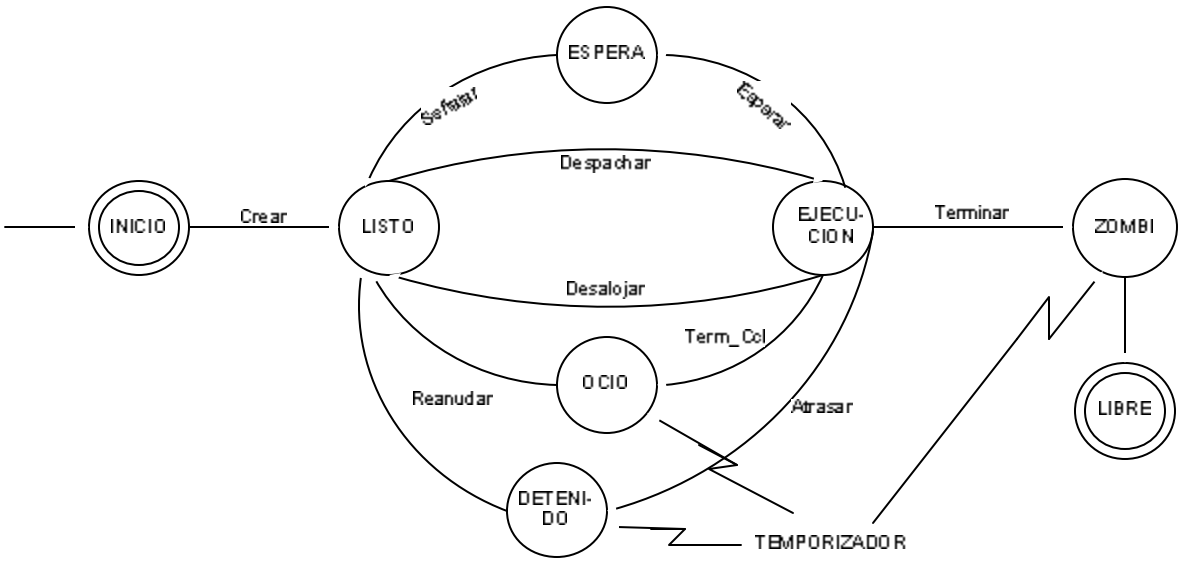
En el KTR-DSP existen siete estados en los cuales una tarea puede entrar en un determinado instante, los cuales se enumeran a continuación:

- LISTO.- En este estado se encuentra una tarea cuando es creada al invocar la primitiva SOTarCrear().
- EJECUCION.- Este estado se le asigna a una tarea cuando ocupa la CPU y va a iniciar su ejecución, mientras la tarea se esté ejecutando tendrá este estado, de lo contrario ocupará alguno de los otros estados.
- ESPERA.- Una tarea llega a este estado cuando desea ocupar una región crítica que está siendo salvaguardada por un semáforo, por lo tanto ya que existe otra tarea teniendo acceso a dicha región, dicha tarea se coloca en una lista de espera del semáforo y se le asigna este estado hasta que el semáforo sea liberado por la tarea que lo está ocupando y sea colocada nuevamente en la lista de tareas próximas a ejecutarse, si dicha tarea tiene la mayor prioridad entra en ejecución y podrá ocupar el semáforo.
- OCIO.- Cuando una tarea es periódica y ya terminó su ciclo de ejecución es colocada en una lista de tareas ociosas con este estado

para indicar que dicha tarea está en espera de volverse activar en su siguiente período.

- ZOMBI.- En este estado se colocan todas aquellas tareas críticas que han sido invocadas para ser eliminadas del sistema pero que todavía tienen ocupando ancho de banda del procesador, debido al factor de utilización. La rutina de servicio del temporizador verifica que las tareas en estado ZOMBI se liberen y sean colocadas en el estado LIBRE dentro de una lista de BCT's libres.
- DETENIDO.- Las tareas llegan a este estado cuando se invoca la primitiva SOTmpEsp(), indicando que la tarea debe esperarse por una cierta cantidad de tics o ciclos de reloj. Una vez alcanzada dicha cantidad la tarea se coloca en el estado de LISTO.
- LIBRE.- Este estado es el que en un principio tienen todas las tareas cuando se crean los BCT's. A este estado también llegan las tareas zombis cuando son liberadas.

La Figura 3.6 muestra un diagrama de transición de estados que experimenta una tarea cuando existe dentro del KTR-DSP.



- **Crear**: Esta transición se lleva acabo al invocar la rutina "SOTarCrear()" que permite que una tarea forme parte del sistema y la lleva a su primer estado (LISTO).
- **Despachar**: Una vez que se realizó la planificación se realiza esta transición para que la tarea que ha sido seleccionada entre en ejecución y llegue al estado de EJECUCION.
- **Desalojar**: Tiene lugar esta transición cuando una tarea que está en ejecución es retirada del uso de la CPU y es reemplazada por otra de mayor prioridad, una vez reemplazada se regresa la tarea a la lista de tareas próximas a ejecutarse.
- **Term_Ccl**: Esta transición es una consecuencia de invocar la rutina "SOTarTermCcl()" para indicar que una tarea periódica ha finalizado su ciclo de ejecución y estará en espera de volver a iniciar su ciclo en su próximo período.
- **Atrasar**: Cuando una tarea requiere suspenderse por cierta cantidad de tics se llega a esta transición. Dicha transición es consecuencia de invocar la rutina "SOTmpEsp()".
- **Reanudar**: Esta transición es consecuencia de salir de los estados de OCIO y DETENIDO por haber concluido la espera necesaria y con ello regresar a la lista de tareas próximas a ejecutarse. Esta transición es activada por el temporizador.
- **Esperar**: Cuando una tarea que hace uso de un semáforo ocupado por otra tarea tiene lugar esta transición logrando con esto que la tarea que quiere ocupar el semáforo se coloque en una lista de espera del mismo semáforo y se llegue al estado de ESPERA.
- **Señalar**: Esta transición sucede cuando el semáforo ocupado por otra tarea es liberado y la tarea se tiene que regresar a la lista de tareas próximas a ejecutarse, para que dicha transición suceda la tarea que ocupa el semáforo debe invocar la rutina "SOSemLiberar()" para así dar oportunidad a otra tarea de ocupar el semáforo.

3.4 Tipos de Tareas Ejecutadas en el Kernel

En el *kernel* se pueden configurar dos tipos de tareas: las críticas y las de no tiempo real, estas tareas para que puedan ser ejecutadas necesitan ser planificadas con anterioridad, garantizando que dichas tareas no van a fallar con sus períodos o plazos (deadlines). Las tareas dentro del kernel son independientes una de otra. Dentro del mismo no se implementa ningún esquema de restricciones de precedencia configurable desde el momento en que se crea una tarea en él.

Las tareas críticas tienen un plazo crítico, y las tareas de no tiempo real tienen una prioridad fija.

Las tareas críticas son activadas periódicamente. Si la instancia es terminada con la primitiva "SOTarTermCcl", la tarea es colocada en el estado de OCIO y automáticamente activada por el temporizador al inicio de su próximo período.

Las tareas críticas son planificadas empleando el algoritmo de planificación denominado Primero el Plazo –deadline- más Corto (EDF), para las tareas de no tiempo real se hace un mapeo de su prioridad hacia su valor de plazo de vencimiento, logrando con esto que los dos tipos de tarea se planifiquen con el mismo algoritmo.

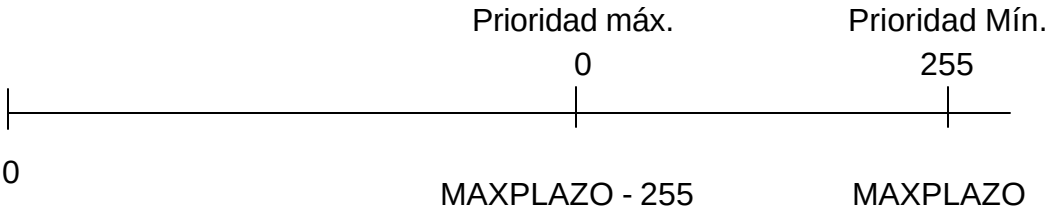


Figura 3.7 . Mapeo de las Prioridades de las Tareas de No Tiempo Real dentro de los Plazos.

Con el propósito de integrar la planificación de estos tipos de tareas y evitar el uso de dos listas de planificación, las prioridades de las Tareas de No Tiempo Real (NTR) son transformadas por plazos así que ellas son siempre mayores que los plazos de las tareas críticas. La regla para asignar las prioridades de las tareas NTR por plazos se muestra en la figura anterior y está dada por la siguiente expresión:

$$d_i^{NTR} = MAXPLAZO - NIVPRIO + P_i \qquad \textbf{Ec. 3–2}$$

Donde *MAXPLAZO* es el valor máximo de la variable SOTmpSist ($2^{31}-1$), *NIVPRIO* es el número de niveles de prioridad manejados por el *kernel* (manejada en el *kernel* como SO_PrioMasBaja), y P_i es la prioridad de la tarea, en el rango $[0, NIVPRIO-1]$ (siendo 0 la más alta prioridad). Este mapeo de prioridades reduce sutilmente el tiempo de vida del sistema pero simplifica grandemente el manejo de las tareas y las operaciones de las listas.

CAPÍTULO 4 - IMPLEMENTACIÓN DEL KERNEL

En esta etapa del desarrollo de la tesis se presentan las rutinas implementadas para el funcionamiento del *Kernel* de Tiempo Real y los diagramas para ilustrar las operaciones realizadas. Así como una descripción de las herramientas empleadas para su realización.

4.1 Herramientas de Software y Hardware

El dispositivo en el cual se ha programado el *kernel* es un Procesador Digital de Señales de la marca Motorola, el modelo empleado es el 56F827. Este procesador pertenece a la familia de Procesadores Digitales de Señales (PDS o DSP por sus siglas en inglés) con núcleo (*core*) 56800. Combina en un solo chip, la potencia del procesamiento de un PDS y la funcionalidad de un microcontrolador con un conjunto flexible de periféricos para crear soluciones efectivas en costo a las aplicaciones de propósito general. El DSP56F827 incluye varios periféricos que son especialmente útiles para aplicaciones, tales como: supresión de ruido, lectores de etiquetas de identificación, seguridad en dispositivos de acceso, medición remota, alarmas y telefonía.

El DSP56F827 soporta la ejecución de programas desde las memorias internas o externas. Dos operandos de datos pueden ser accesados desde la RAM de datos interna en cada ciclo de instrucción. También provee dos líneas de interrupción dedicadas y hasta 64 líneas para entradas y salidas de propósito general, dependiendo de la configuración de los periféricos.

En la Figura 4.1 se ilustra el diagrama de bloques de la arquitectura del DSP56F827 de Motorola que se emplea para la ejecución del *Kernel* y de las Aplicaciones en Tiempo Real.

Para la implementación y su ejecución se emplea la Tarjeta Módulo de Evaluación DSP56F827EVM de Motorola que contiene las herramientas requeridas para iniciar rápidamente la evaluación del procesador.

La tarjeta de desarrollo DSP56F827EVM se usa para demostrar las capacidades del DSP56F827 y proveer una herramienta de *hardware* que permite el desarrollo de aplicaciones para dicho DSP.

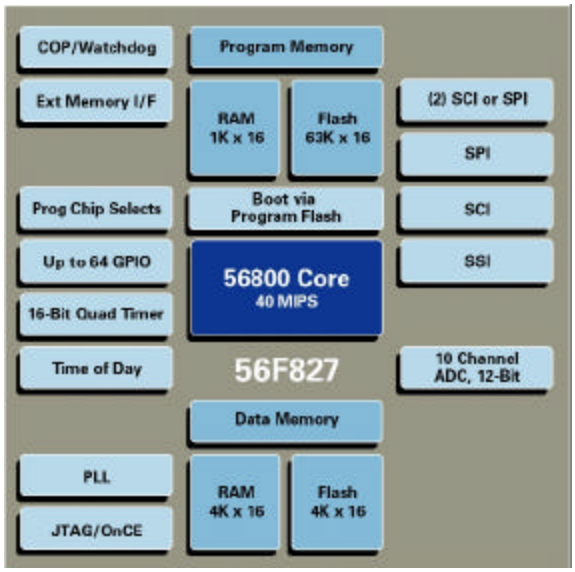


Figura 4.1- Diagrama de Bloques de la Estructura del DSP 56F827 de Motorola.

La DSP56F827EVM es una Tarjeta Módulo de Evaluación que incluye un DSP56F827, un CODEC estereofónico de 16 bits, memoria externa y una interfaz para expansión de memoria. Está diseñada para los siguientes propósitos:

- *Permitir a los nuevos usuarios familiarizarse con las características de la arquitectura de procesadores 56800.* Las herramientas y ejemplos suministrados con el DSP56F827EVM facilitan la evaluación del conjunto de características y los beneficios de esta familia de procesadores.
- *Servir como plataforma de desarrollo para software de Tiempo Real*. La gama de herramientas permiten al usuario desarrollar y simular rutinas, descargar el *software* al chip o a la RAM de la tarjeta, ejecutarlo, y depurarlo usando el puerto JTAG/OnCE (Join Test Action Group / On-chip Emulation). Las características de punto de ruptura (break-point) del puerto OnCE permiten al usuario especificar condiciones de pausa y ejecutar el *software* desarrollado por el usuario en gran velocidad, hasta que la condición de pausa se cumpla. La capacidad para examinar y modificar todos los registros accesibles al usuario, memoria y periféricos a través del puerto OnCE facilitan ampliamente la tarea del desarrollador.
- *Servir como una plataforma de desarrollo de hardware*. La plataforma de *hardware* permiten al usuario conectar periféricos externos. Los periféricos de la tarjeta pueden ser desactivados, proporcionando al usuario la capacidad de reasignar cualquiera de los periféricos del DSP.

En la Figura 4.2 se ilustra el diagrama de bloques de la Tarjeta Módulo de Evaluación DSP56F827EVM de Motorola que se emplea para la ejecución del *Kernel* y de las Aplicaciones en Tiempo Real [12].

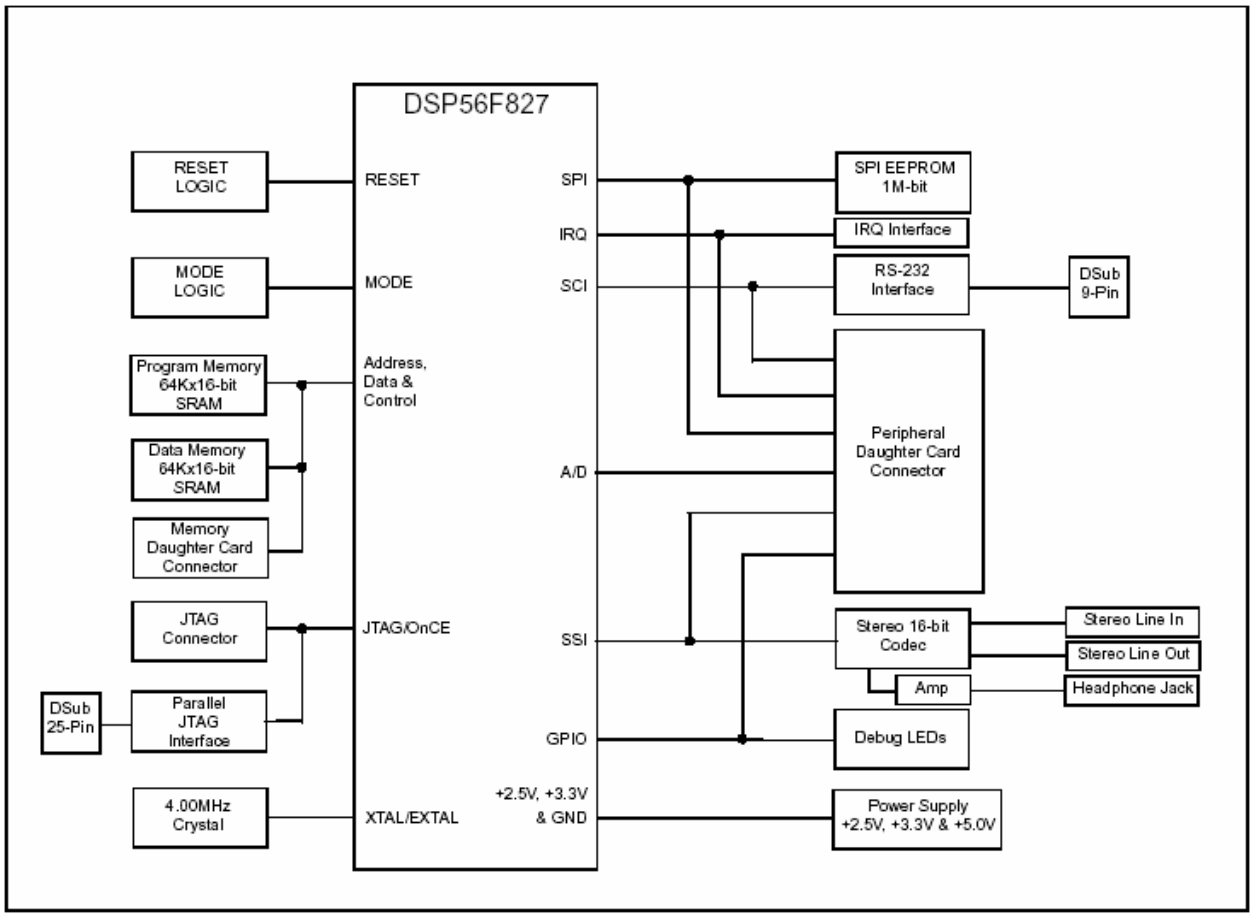


Figura 4.2. Diagrama de Bloques de la Tarjeta de Desarrollo DSP56F827EVM de Motorola, Empleada en la Implementación del KTR para el DSP56827.

Las herramientas de *software* empleadas para el desarrollo del *Kernel* son el conjunto de librerías denominadas SDK (*Software* Development Kit) de Motorola para los procesadores 56800 y el IDE (Integrated Development Environment) CodeWarrior que incluye un compilador del lenguaje de programación "C".

El SDK es desarrollado por Motorola para complementar el ambiente de desarrollo para los procesadores incrustados Motorola 568xx, ya que está diseñado para permitir el rápido desarrollo de eficientes aplicaciones de alto nivel que son portables y reusables, así los usuarios pueden acelerar los productos al mercado.

Las librerías estándares incluyen controladores de periféricos, librerías de procesamiento de señales, algoritmos de aplicaciones específicas, servicios de *software*, aplicaciones de ejemplo, y soporte para SOTR.

El conjunto de desarrollo de *software* de Motorola define un estándar para interfaces de *software* y un marco de trabajo a través de la línea de procesadores 56800 garantizando que la inversión en *software* compatible con el SDK sea mantenida independientemente del procesador objetivo. Adicionalmente, las soluciones específicas son fácilmente integradas dentro de módulos de *software* compatibles de terceros.

El SDK está completamente integrado con el entorno de desarrollo de CodeWarrior y el Módulo de Evaluación y el *hardware* de desarrollo del sistema. Está estructurado para soportar el ciclo de vida del producto de desarrollo desde la evaluación hasta la personalización del desarrollo del *hardware* y *software* para el soporte de los equipos, maximizando el uso de los recursos limitados.

Por otra parte CodeWarrior para Sistemas Incrustados 56800 de Motorola es un *software* desarrollado por Metrowerks que provee un amplio ambiente de desarrollo de *software*.

CodeWarrior es a la vez un Ambiente de Desarrollo Integrado (IDE) basado en ventanas con compiladores del lenguaje de programación "C" altamente eficientes. Este IDE es una herramienta sofisticada para la navegación, edición, compilación y depuración del código. A su vez incluye una intuitiva gestión gráfica del proyecto y el desarrollo del sistema; un ensamblador y un enlazador; un depurador gráfico a nivel de código, y un simulador del conjunto de instrucciones.

El IDE CodeWarrior soporta ambos lenguajes de programación, el lenguaje de Programación "C" y el lenguaje ensamblador de los procesadores 56800 sobre las diversas plataformas de Windows.

Con la ayuda de estas características proporcionadas por el IDE y las librerías del SDK se ha podido implementar el código del *Kernel* de Tiempo Real de manera más eficiente y versátil al momento de depurar el código.

El KTR-DSP se ha implementado en su mayor parte en lenguaje C y en las partes críticas del código como lo son el cambio de contexto, el manejo de las interrupciones y del temporizador se ha implementado con lenguaje ensamblador.

Se opto por emplear el lenguaje de programación C, en vez de emplear completamente el conjunto de instrucciones del ensamblador propio del DSP, porque "C" es un lenguaje de alto nivel y a la vez es un lenguaje conciso y poderoso muy adecuado para la implementación del *Kernel*, ya que hace más fácil la escritura, comprensión, depuración del mismo, además que proporciona la capacidad de poder migrar el núcleo a otros procesadores. Permite al programador manipular direcciones y especificar formatos de almacenamiento, es también un lenguaje de alto nivel que soporta procedimientos parametrizados, expresiones lógicas de control, y compilación independiente [4].

4.2 Rutinas Implementadas para el Kernel

Ahora bien, dado que el *Kernel* está estructurado a través de capas, se hace un apartado a cada una de ellas para un mejor entendimiento de la implementación.

Dentro de cada apartado se enumeran en una tabla las rutinas de la capa correpondiente, especificando dentro de ella el nombre de la rutina, los parámetros que la rutina emplea, una breve descripción, el tiempo de ejecución expresado en notación "gran O, O()" y por último el número de ciclos que resultaron de simular dicha rutina en el CodeWarrior. Los tiempos de ejecución son medidos por ciclos de reloj del oscilador. Es decir, los tiempos expresados en unidades variaran dependiendo de la configuración que tenga el oscilador del reloj del DSP en uso. Para mayor información de la notación O() consultar [1].

4.2.1 Rutinas para la Capa de Máquina

La primera capa que se presenta en este apartado es la Capa de Máquina que como ya se ha mencionado es aquella en donde se interactúa directamente con el procesador por lo cual se ha escrito código en lenguaje ensamblador para acceder a los componentes que deben interactuar llevando acabo algunas funciones del *kernel*, como son el cambio de contexto, el manejo de interrupciones, y el manejo del temporizador o cronómetro. Cabe aclarar que estas funciones no son visibles al nivel de usuario y no deben manipularlas.

Para una mejor visualización de las rutinas implementadas en el *Kernel* éstas son nombradas con el prefijo SO seguido de tres o más letras del componente al cual dan servicio o la operación que realizan.

Las funciones implementadas en el *Kernel* para esta capa se encuentran en diversos archivos dentro del conjunto de archivos que conforman el código del *Kernel* de Tiempo Real, los cuales se enuncian a continuación: **KTR.asm**, **KTR.h**, **KTR.c**. La Tabla 4.1 muestra las rutinas implementadas en el KTR-DSP para dicha capa.

El proceso de desalojo de una tarea que se encuentra en ejecución por otra tarea que tiene mayor prioridad se presenta en la Figura 4.3.

Tabla 4.1- Rutinas de la Capa de Máquina Implementadas en el KTR para el DSP56827.

Rutina	Parámetros	Descripción	Tiempo de ejec.: T(n)	Ciclos de reloj sim. con el Code Warrior.
SOTarStkIni()		Inicializa la pila (*stack*) de cada tarea que es creada en el sistema, de manera que cuando ésta sea invocada contenga los registros necesarios para poder realizar el cambio de contexto de la tarea. Los registros se almacenan simulando la existencia de una interrupción.	O(1)[17]	268
SOCmbCtx()		Permite salvar el contexto (estado de la tarea) en el espacio asignado para almacenar los registros del DSP de la tarea a ser suspendida y que pudiesen estar en uso en el momento en que la tarea requiere ser suspendida para dar oportunidad a otra tarea para que ocupe el CPU.	O(1)	174

[17] Esta notación indica el tiempo de ejecución, y se emplea para representar "alguna cantidad constante".

		Al mismo tiempo esta rutina restaura los valores de los registros de la tarea que va ocupar la CPU.		
SOIntCmbCtx()		Hace lo mismo que la rutina anterior, sólo que ésta es invocada en las rutinas de servicio a las interrupciones, creadas por el usuario.	O(1)	174
SOIniTarLst()		Es invocada cuando se hace la inicialización del *Kernel*, y se emplea para poner en ejecución la tarea que tiene la mayor prioridad en el tope de la lista de las tareas listas para ejecutarse, el objetivo de esta rutina es colocar los valores de los registros almacenados en la pila (*stack*) de la tarea en los registros del DSP, para que estos estén disponibles cuando se lleve a cabo la ejecución de la tarea.	O(1)	258
SOTicISR()		Rutina de servicio de la interrupción del temporizador. Da servicio cada vez que el temporizador hace una interrupción, e invoca a la subrutina denominada **SOTicSrv()** que tiene el objetivo de actualizar el contador del tiempo del sistema, y checar las tareas que están en espera del tic del reloj, para que puedan ser colocadas en la lista de	$O(n^2)$	4266

		tareas a ejecutarse y poder competir por la CPU. También checa las tareas que están en estado ZOMBI y aquellas que están en estado de OCIO, para liberarlas y reanudarlas, respectivamente.		
SOIntDshbl()		Es una macro que permite deshabilitar las interrupciones en el DSP.	O(1)	22
SOIntHbl()		Es una macro que tiene la función inversa de la anterior, es decir, habilita las interrupciones en el DSP.	O(1)	18
SOIntEnt()		Indica al *Kernel* la entrada a una interrupción, su objetivo es incrementar el contador del nivel de interrupciones. Se emplea cuando el usuario requiere definir una interrupción.	O(1)	66
SOIntSal()		Indica al *kernel* la salida de una interrupción, su función es disminuir el contador del nivel de interrupciones y verificar si la interrupción ha colocado una tarea en la lista de tareas a ejecutarse; si es así, realiza el cambio de contexto invocando la rutina **SOIntCmbCtx()**.	O(n) n= Número Máximo de Tareas + Tareas Aux. Activas.	1032

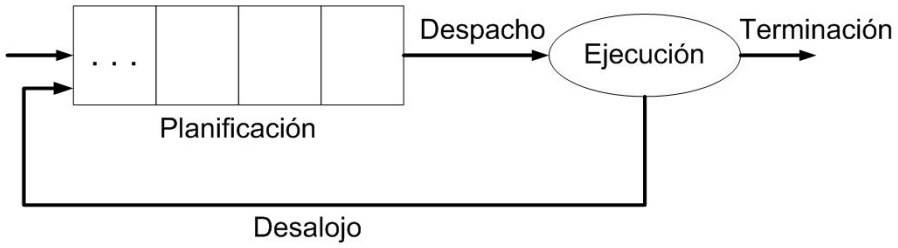


Figura 4.3- Desalojo de una Tarea en el KTR.

4.2.2 Rutinas de la Capa de Administración de Listas

Otra de las capas implementadas en el *Kernel* es la Capa de Administración de Listas, la cual mantiene la pista del estado de las diferentes tareas. En esta capa se provee de funciones básicas para insertar a una lista y remover de una lista.

Las rutinas implementadas en esta capa del *Kernel* se encuentran en el archivo **so_lista.c**, y se enumeran en la Tabla 4.2.

Cabe aclarar que ya que no existe un intérprete de comandos para el kernel, la creación de tareas se tiene que realizar desde el código de la aplicación, si se desea crear una nueva tarea para la aplicación, dicha tarea deberá crearse explícitamente antes de compilar nuevamente y poner en ejecución a la aplicación.

Tabla 4.2- Rutinas Implementadas para la Capa de Manejo de Listas del KTR para el DSP56827.

Rutina	Parámetros	Descripción	Tiempo de ejec.: T(n)	Ciclos de reloj sim. con el Code Warrior.
SOLstInsertar()	Int16 indice cola *cola_ins	Permite encontrar el lugar adecuado para insertar un nodo a la lista y actualizar los apuntadores del nodo. El criterio de inserción es en forma ascendente de acuerdo al plazo (deadline) de la tarea, en este caso	O(n)	250

		el nodo (véase Figura 4.4).		
SOLstExtraer()	Int16 indice cola *cola_ext	Extrae un nodo de la lista, y actualiza los apuntadores del nodo extraído (véase Figura 4.5).	O(1)	174
SOLstObtPrmr()	Int16 *cola_ext	Esta rutina es útil para poder extraer el nodo que está en el tope de la lista. Es decir el primer elemento de la lista (véase Figura 4.6).	O(1)	112
SOLstPrmrPlz()	cola *cola_chq	Devuelve el valor del plazo del nodo que está en el tope de la lista.	O(1)	70
SOLstVacia()	cola *cola_chq	Devuelve un valor (FALSO o VERDAD) para indicar si la lista está vacía o no.	O(1)	58
SOLstInsEsp()	Int16 indice cola *cola_ins	Similar a la rutina **SOLstInsertar()**, la única diferencia es el criterio por el cual son ordenados los nodos en la lista. El criterio usado es el valor de tiempo de espera. Se ordena de forma ascendente.	O(n)	184
SOLstPrmrEsp()	cola *cola_chq	Devuelve el valor del tiempo de espera del nodo que está en el tope de la lista.	O(1)	70

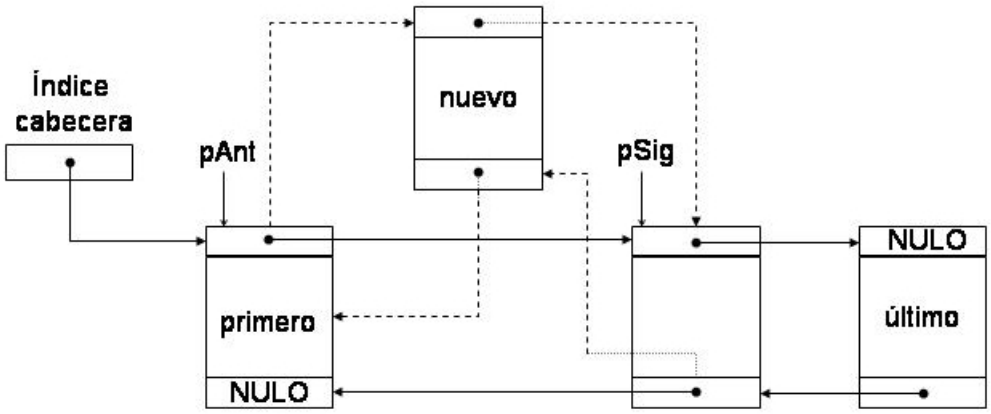


Figura 4.4- Inserción de un BCT en una Lista.

4.2.3 Rutinas para la Capa de Administración del Procesador

En la Capa de Administración del Procesador se han implementado las rutinas que se encargan de las operaciones de planificación y despacho de tareas, dichas rutinas se enumeran en la Tabla 4.3 y el código de estas primitivas se encuentra en los archivos **KTR.h** y **KTR.C**. Los detalles del cambio de contexto se mencionan en los siguientes párrafos.

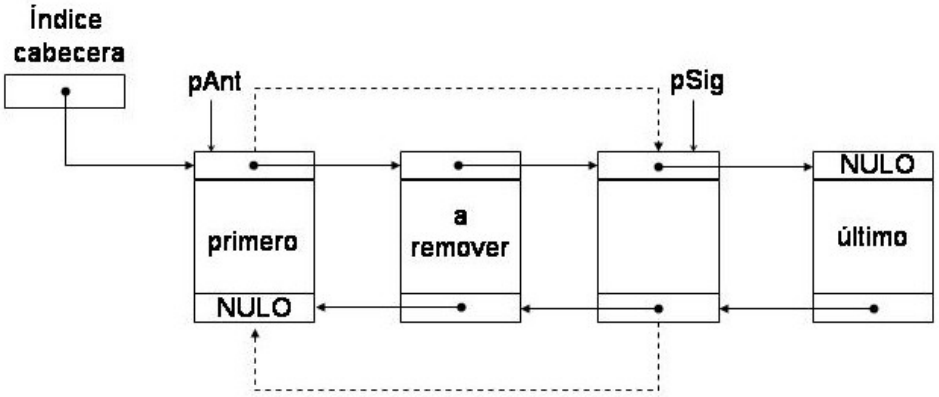


Figura 4.5- Extracción de un BCT desde una Lista.

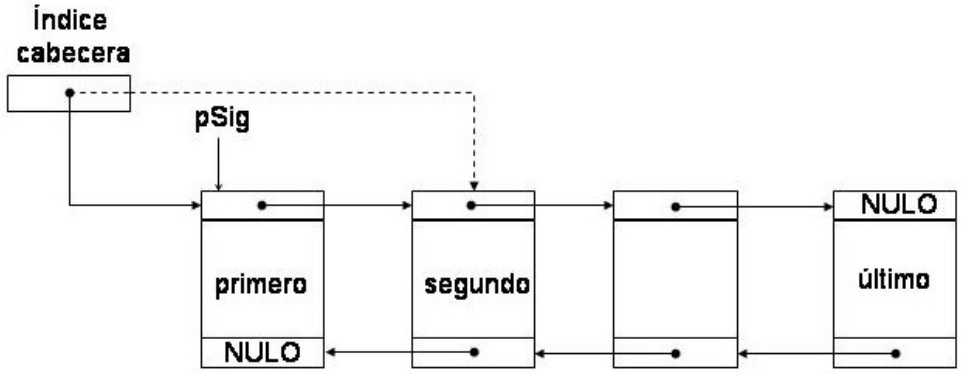


Figura 4.6- Extracción del BCT que está en el Tope de una Lista.

Tabla 4.3- Rutinas de la Capa de Administración del Procesador Implementadas en el KTR para el DSP56827.

Rutina	Parámetros	Descripción	Tiempo de ejec.: T(n)	Ciclos de reloj sim. con el Code Warrior.
SOPlanificar()		Selecciona la tarea que está en el tope de la lista de tareas a ejecutarse tomando como criterio de selección el plazo más corto.	O(n)	1124
SODespacha()		Macro que actualiza la tarea actual e invoca la macro **SOTARCAMBIO().**	O(1)	392
SOTARCAMBIO()		Macro que emplea una instrucción en ensamblador para lanzar una interrupción de *software* (SWI) para invocar la rutina de cambio de contexto.	O(1)	182

El contexto de una tarea es generalmente el contenido de todos los registros de la CPU. El código del cambio de contexto sólo necesita guardar los valores de los registros de la tarea que será desalojada y cargar dentro de la CPU los valores de los registros para la tarea a reanudarse [6].

La Figura 4.7 muestra el estado de algunas variables del *Kernel* y estructuras de datos sólo antes de invocar la macro SOTARCAMBIO(), por consideración, se supone la CPU con unos cuantos registros de propósito general, sin embargo el núcleo del DSP56800 tiene 22 registros, es por ello que se emplean unos cuantos con la finalidad de ilustrar su empleo. Los pasos siguientes describen los números de la figura:

1. SOBCTAct apunta a el BCT de la tarea que será suspendida (la tarea de baja prioridad).
2. El apuntador de la pila de la CPU (registro SP) apunta al tope de la pila de la tarea que será desalojada.
3. SOBCTColLst apunta al BCT de la tarea que se ejecutará después de completar el cambio de contexto.
4. El campo .SOBCTStkTar en el BCT apunta al tope de la pila de la tarea a reanudar.
5. La pila de la tarea a reanudar contiene los valores deseados de los registros a cargar dentro de la CPU; estos valores pudiesen haber sido grabados por un cambio de contexto anterior.

La Figura 4.8 muestra el estado de las variables y las estructuras de datos después de invocar SOTARCAMBIO() y después de salvar el contexto de la tarea a suspender. A continuación se explica lo que sucede en la figura:

1. El llamado a SOTARCAMBIO invoca la instrucción de interrupción por *software*, la cual obliga al procesador a salvar el valor actual de los registros SR y PC dentro de la pila de la tarea actual; el procesador entonces cambia de dirección hacia el manejador de la interrupción por *software*, el cual es responsable de completar los pasos restantes del cambio de contexto.
2. El manejador de la interrupción por *software* inicia salvando los registros de propósito general, R0, R1, R2, y R3, en este orden.
3. El registro apuntador de la pila es entonces salvado dentro del BCT de la tarea actual. En este punto, el registro SP de la CPU y SOBCTAct ? SOBCTStkTar están apuntando a la misma localidad dentro de la pila de la tarea actual.

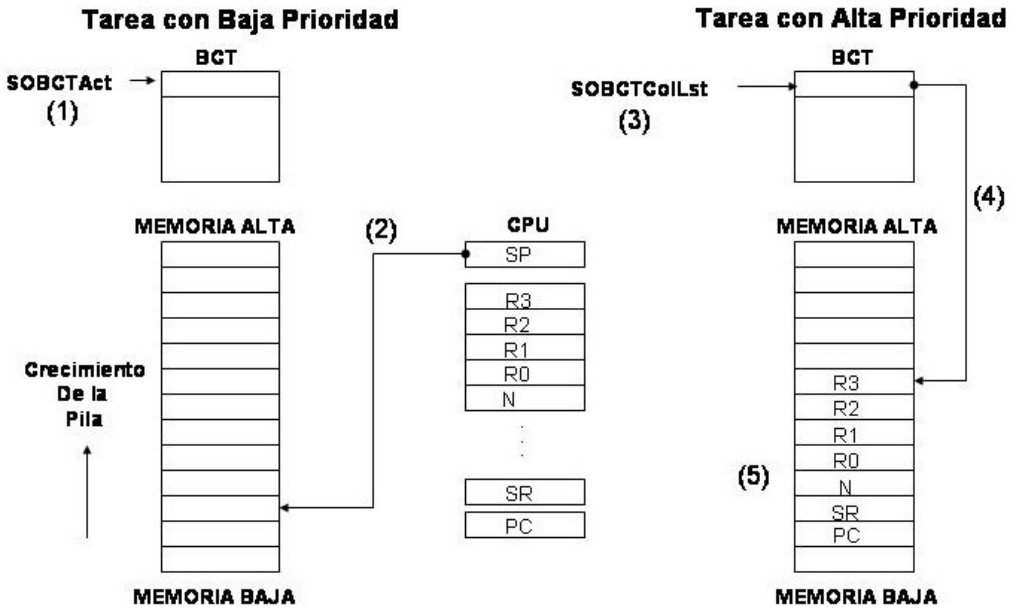


Figura 4.7- Estructuras del Kernel cuando SOTARCAMBIO es Invocada.

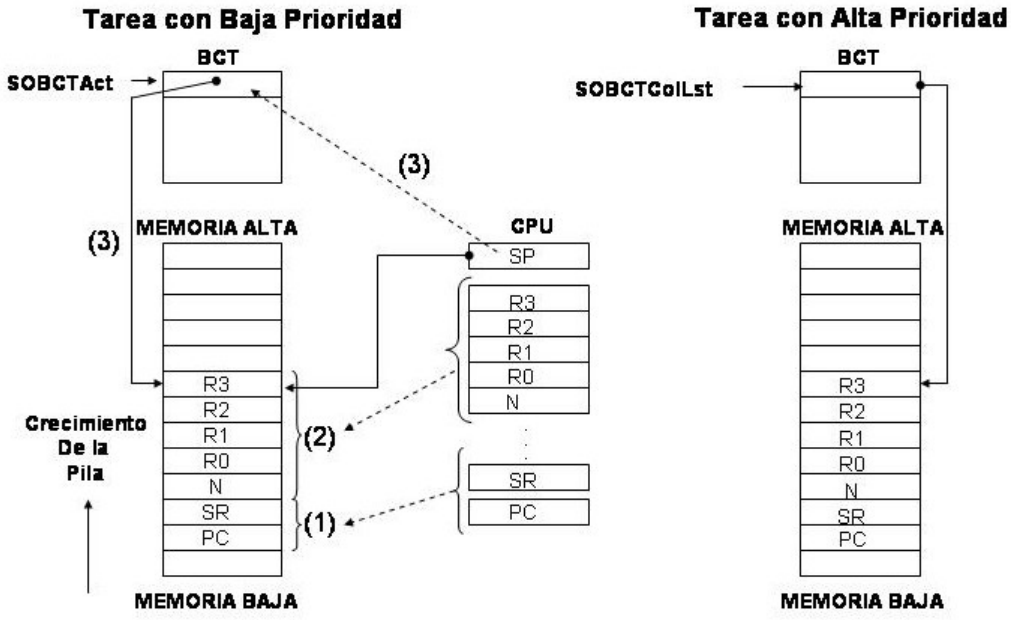


Figura 4.8- Salvando el Contexto de la Tarea Actual.

La Figura 4.9 muestra el estado de las variables y las estructuras de datos después de ejecutar la última parte del código de cambio de contexto. Los números que se señalan en la figura se detallan en la siguiente lista:

1. Ya que la nueva tarea actual es ahora la tarea que será reanudada, el código del cambio de contexto copia SOBCTColLst a SOBCTAct.

2. El apuntador de la pila de la tarea a reanudar se extrae del BCT (de SOBCTColLst? SOBCTStkTar) y se carga dentro del registro SP de la CPU. En este punto, el registro SP apunta a la localidad de la pila que contiene el valor del registro R3.
3. Los registros generales son extraídos de la pila en orden inverso (R3, R2, R1, y R0).
4. Los registros PC y SR son cargados de regreso dentro de la CPU al ejecutarse la instrucción de regreso desde la interrupción (RTI). Ya que el PC cambió, la ejecución del código reanuda en el punto al cual el PC está apuntando, lo cual pasa a ser el código de la nueva tarea.

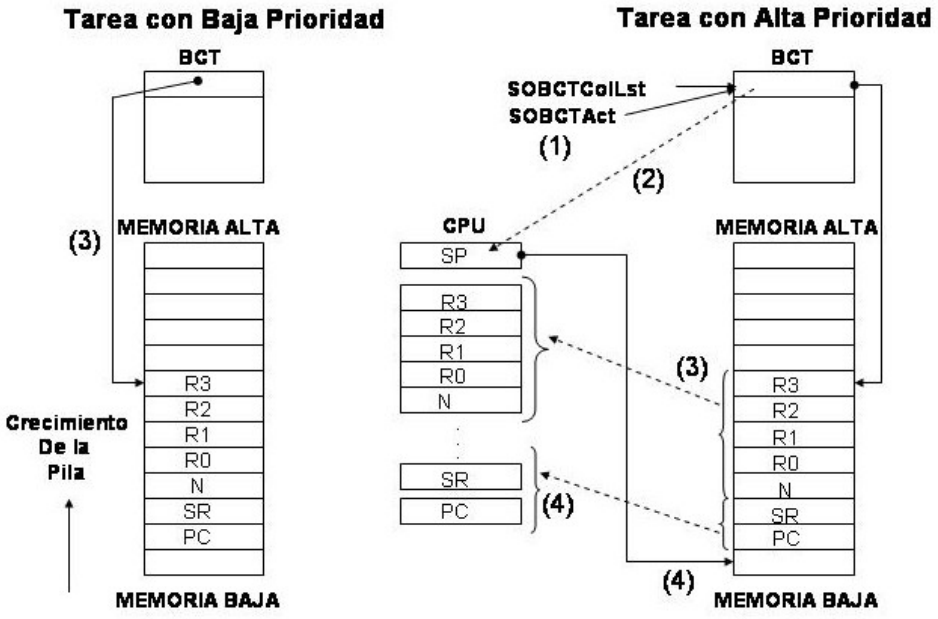


Figura 4.9- Reanudando la Tarea Actual.

La Capa de Administración del Procesador es una de las más importantes dentro del código del *kernel* ya que en ella se elige a la tarea de más alta prioridad y se invoca el cambio de contexto para que ella pueda hacer uso de la CPU.

4.2.4 Rutinas para la Capa de Servicio

La última capa de la estructura del *Kernel* es la Capa de Servicio, la cual proporciona todos los servicios visibles al usuario como un conjunto de llamadas al sistema. Los servicios típicos son creación, cancelación, suspensión de tareas, consulta de las operaciones del sistema, sincronización de tareas, y comunicación entre las tareas.

Las rutinas implementadas en esta capa se localizan en los archivos **KTR.c**, **so_tar.c**, **so_tiempo.c**, **so.sem.c**, **so_bac.c**, **so_info.c**. La Tabla 4.4 enumera dichas rutinas.

Tabla 4.4- Rutinas de la Capa de Servicio Implementadas en el KTR para el DSP56827.

Rutina	Parámetros	Descripción	Tiempo de ejec.: T(n)	Ciclos de reloj sim. con el Code Warrior.
SOTarCrear()	void (*SOTarDireccion)() SO_STK *ptp Int16 SOTarTipo Int32 SOTarPer Int32 SOTarTmpEje	Crea una tarea y la pone en el estado LISTO. Se pueden definir con esta rutina el tipo de tarea y los parámetros que la componen como son el plazo de vencimiento, que a la vez se toma como el período con el cual se repite una instancia de la tarea, y por último el tiempo de ejecución[18] del peor caso, para que se pueda garantizar la planificabilidad.	O(n)	3766
SOTarTermCcl()		Inserta una tarea en la lista de tareas ociosas. Esta rutina se emplea cuando se requiere	O(n)	1240

[18] Este tiempo de ejecución tiene que ser calculado tomando en cuenta el número de instrucciones de ensamblador empleadas por la rutina al momento de producirse el código objeto de la aplicación. Teniendo en cuenta esto, se debe localizar el tiempo de ejecución de cada instrucción de acuerdo con lo especificado en el manual del DSP y hacer la sumatoria de los tiempos, de esta manera obtenemos el tiempo de ejecución. Sin embargo este tiempo puede ser también estimado de acuerdo a la experiencia del programador.

		especificar el fin del ciclo de una tarea periódica. Es necesario que esta rutina se coloque siempre al final de la tarea periódica dentro del ciclo repetitivo que forma a la tarea para garantizar que la siguiente instancia se ejecutará correctamente.		
SOTarTermProc()		Termina la tarea en ejecución. Esta rutina se emplea con las tareas de no tiempo real.	O(n)	1736
SOTarFinalizar()	Int16 proceso	Termina una tarea específica, devolviendo las estructuras de datos a la lista de tareas libres.	O(n)	702
SOTarGarantia()	Int16 proceso	Garantiza la planificabilidad de una tarea crítica, checando que se respete el factor de utilización del conjunto de tareas críticas creadas.	O(n)	1348
SOIniSist()		Inicializa las estructuras y variables que son empleadas en el *Kernel*. Esta rutina es importante que el usuario la ejecute antes de crear cualquier tarea en	O(n)	3476

		el *kernel*.		
SOAbortar()	Int16 Error	Se invoca cuando sucede un error en alguna de las rutinas del *Kernel*. Los posibles valores a usar son: TMP_SOBREPASADO TMP_EXPIRADO NO_GARANTIA NO_BCT NO_SEM NO_TMP_ESP	O(1)	97
SOIniciar()		Inicia la multitarea en el *kernel*, es decir, toma la primera tarea lista para ejecutarse y la coloca en estado de EJECUCION para que inicie.	O(1)	466
SOIniEstd()		Inicializa los datos para usar la tarea de estadística.	O(n)	
SOTarOcio()		Se emplea para crear una tarea que siempre estará ejecutándose, cuando no haya ninguna otra tarea lista para ejecutarse.	O(n)	148
SOTmpEsp()	UInt16 tics	Coloca la tarea que la invoca, en una lista de espera, y se reanuda después del número de Tics indicado, esto permite colocar a la tarea en el estado DETENIDO.	O(n)	894
SOTmpEspRndr()	Int16 proceso	Reanuda a la tarea	O(n)	2184

		especificada que se encuentra en espera de tics del reloj, si la tarea no está en espera, no hace nada.		
SOSemCrear ()	Int16 valor	Adquiere un BCS de la lista de BCS libres y lo inicializa para poder ser empleado por las tareas.	O(1)	188
SOSemFinalizar ()	sem semaforo	Permite desalojar un BCS si éste no está en uso y lo inserta a la lista de BCS libres.	O(1)	142
SOSemAdquirir()	sem semaforo	Es invocada por una tarea para indicar la espera de un evento o indicar que la tarea está dentro de una sección crítica. Si otra tarea está ocupando el semáforo, éste inserta a la tarea en una lista de espera y coloca a la tarea en estado ESPERA. De lo contrario disminuye el contador del semáforo.	O(n)	824
SOSemLiberar()	sem semaforo	Indica la ocurrencia de un evento o la salida de una sección crítica, lo cual permite verificar si una tarea se encuentra en la	O(n)	1934

		lista de espera del semáforo, si existe la tarea, ésta se extrae y se coloca en la lista de tareas listas a ejecutarse. De lo contrario incrementa el contador del semáforo.		
SOBacCrear()	BAC *nom_bac, BUFBAC *lst_bacs, Int16 tam_msj, Int16 num_buf	Permite inicializar las estructuras de datos para alojar los *buffers* asíncronos cíclicos, sus parámetros incluyen el apuntador a la estructura de control de los *buffers*, el apuntador a la lista de *buffers*, el tamaño del mensaje y el número de *buffers* dentro de la lista.	O(n)	612
SOBacReservar ()	BAC *b	Reserva el *buffer* en el BAC, para poder escribir un mensaje, con esto se impide que otra tarea escriba datos sobre el mismo *buffer*.	O(1)	100
SOBacPonerMsj ()	BAC *b, BUFBAC *bbac	Permite colocar un *buffer* con el dato actualizado en la estructura BAC, para que pueda ser utilizado por otra tarea.	O(1)	144
SOBacObtMsj ()	BAC *b	Se invoca para	O(1)	104

		poder adquirir el dato más actual del *buffer* a través de un apuntador desde la estructura BAC indicada como parámetro.		
SOBacLiberar ()	BAC *b, BUFBAC *bbac	Una vez terminado de utilizar el *buffer* de la estructura BAC es necesario desalojarlo y regresarlo a la lista de *buffers* libres para que puedan ser empleados por otras tareas.	O(1)	126
SOInfoTmp()		Retorna el tiempo transcurrido del sistema en milisegundos, desde que se invoco la rutina de inicialización del *kernel*.	O(1)	50

Para la sincronización se han implementado semáforos binarios (no se implementa ningún Protocolo de Herencia de Prioridades por las razones enunciadas en el capítulo DISEÑO Y ESTRUCTURA DEL KERNEL, veáse página 29) que permiten bloquear y desbloquear una sección crítica y para la comunicación entre tareas se ha implementado un mecanismo de comunicación denominado *Buffers* Asíncronos Cíclicos [17], que pueden ser usados para tareas que necesiten intercambiar datos entre ellas como sería la adquisición de datos de un sensor.

CAPÍTULO 5 - PRUEBAS Y RESULTADOS

5.1 Consideraciones para las Pruebas

Las pruebas del *Kernel* de Tiempo Real se realizaron empleando una PC de Escritorio y en ocasiones una PC Portátil, ambas se usaron como máquina anfitriona con las siguientes características:

- PC de Escritorio
  - Procesador Pentium 4 a 2 GHz.
  - Memoria RAM 768 Mb.
  - Disco Duro de 80 Gb.
  - Sistema Operativo Windows XP.
- PC Portátil
  - Procesador Celeron a 766 Mhz.
  - Memoria RAM 192 Mb.
  - Disco Duro de 20 Gb.
  - Sistema Operativo Windows XP.

En la PC está instalado el IDE CodeWarrior y el SDK de Motorola. Dado que el CodeWarrior es un *software* bajo licencia se solicitó una licencia de prueba a la compañía Metrowerks, para poder realizar la implementación y las pruebas del *Kernel*.

Para llevar acabo las pruebas se empleó también la Tarjeta de Evaluación DSP56F827EVM, en la cual se descargaron y ejecutaron los ejemplos de prueba haciendo uso de la memoria externa de la tarjeta, con capacidad de hasta 64K palabras en la memoria de datos y 64K palabras en la memoria de programa, se empleó siempre la memoria externa de la tarjeta dado que la memoria interna Flash no era suficiente para descargar el KTR y las aplicaciones.

Para poder emplear la Tarjeta junto con la PC es necesario hacer ciertas conexiones. La Figura 5.1 muestra las interconexiones necesarias para llevar a cabo las pruebas con la Tarjeta DSP56F827EVM.

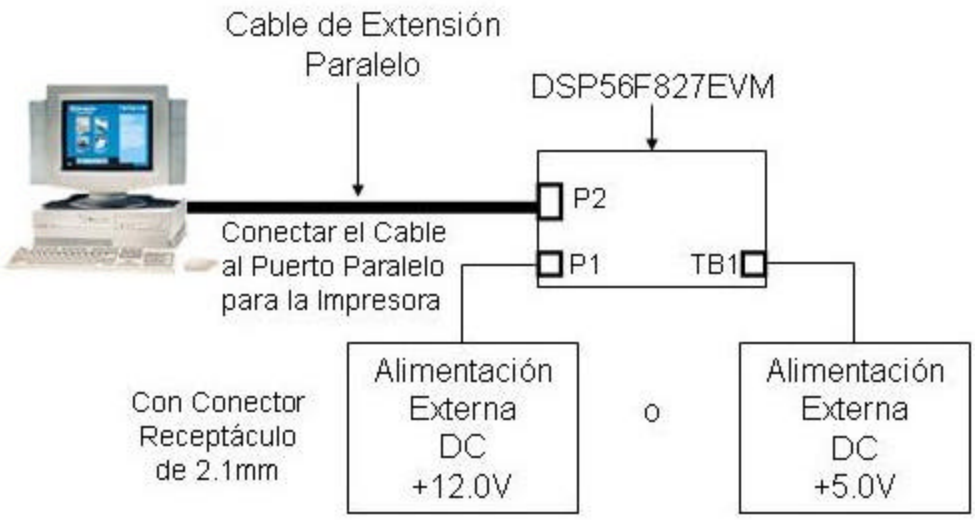


Figura 5.1- Conexión de los Cables de la Tarjeta DSP56F827EVM.

Los pasos a seguir para conectar los cables de la tarjeta se numeran a continuación:

1. Conectar el cable paralelo de extensión al puerto paralelo de la computadora anfitriona (donde está instalado el CodeWarrior y el SDK).
2. Conectar el otro extremo del cable paralelo de extensión a P2 (puerto paralelo, interfaz RS232), sobre la tarjeta DSP56F827EVM. Esto proporciona la conexión con la cual la computadora anfitriona controlará la tarjeta.
3. Cerciorarse que la fuente de alimentación externa de +12.0 V CC 1.2 A o la fuente de alimentación externa de laboratorio de +5.0 V CC 1.0 A no está conectada a una fuente de alimentación de 120 V CA.
4. Aplicar la alimentación a la fuente de alimentación externa. El LED verde de encendido, LED7, se iluminará cuando la alimentación esté correctamente aplicada.

5.2 Resultados

Los resultados alcanzados hasta ahora son haber implementado este *Kernel* de Tiempo Real usando un compilador del lenguaje C denominado CodeWarrior compatible con las instrucciones en ensamblador del DSP 56827 de Motorola. La versión del CodeWarrior es la 4.1. Las características implementadas son la creación de tareas, suspensión de tareas, finalización de tareas, la creación de semáforos para permitir la exclusión mutua entre las tareas, el control de las secciones críticas a través de la habilitación y deshabilitación de las interrupciones, la comunicación de las tareas a través de los *buffers*.

La Tabla 5.1 muestra las características del KTR-DSP y otros núcleos de Tiempo Real similares.

Tabla 5.1- Comparación de núcleos de Tiempo Real.

Características/ Kernel	Porta-ble	Escala-ble	Apropia-tivo	Multi-tarea	Determi-nista	Utilida-des	Manejo De Interrup-ciones
MicroC/OS-II	X	X	X	X	X	X	X
DSP-OS	X		X	X	X	X	X
KTR-DSP	X	X	X	X	X	X	X

Como características distintivas del KTR-DSP se encuentran el empleo de un algoritmo de planificación con asignación dinámica, el algoritmo EDF y el mecanismo de comunicación implementado denominado Buffers Asíncronos Cíclicos, que no es un mecanismo muy difundido y sólo pocos núcleos de tiempo real lo proporcionan.

Para realizar las pruebas se crearon cuatro aplicaciones para demostrar el uso general del *kernel*, el empleo de semáforos, la utilización de los *buffers* para la comunicación entre tareas. Las aplicaciones se enumeran a continuación:

- Aplicación 1- Uso del CPU en el KTR.
- Aplicación 2- Empleo de las Tares Críticas y de No Tiempo Real Manipulando LED's.
- Aplicación 3- Empleo de los *Buffers* Asíncronos Cíclicos para Intercambio de Datos.
- Aplicación 4- Empleo de los semáforos para la solución del problema de los Filósofos Comensales.

Para poder ejecutar las aplicaciones se tuvieron que realizar ciertos cambios en los archivos de configuración del Proyecto. Un proyecto es el área de trabajo donde CodeWarrior almacena todos los archivos y dependencias entre ellos para poder realizar la aplicación.

Para realizar una aplicación con el CodeWarrior se crea un proyecto nuevo usando la plantilla "Utilidades del SDK Incrustado" (véase Figura 5.2), después de colocar el nombre, se selecciona la plantilla de acuerdo al procesador para el cual se desea realizar la aplicación, seleccionando de entre ellas el soporte para SOTR o no y a la vez indicando si es una aplicación para la memoria Externa RAM, para la memoria Flash o para ambas (véase Figura 5.3).

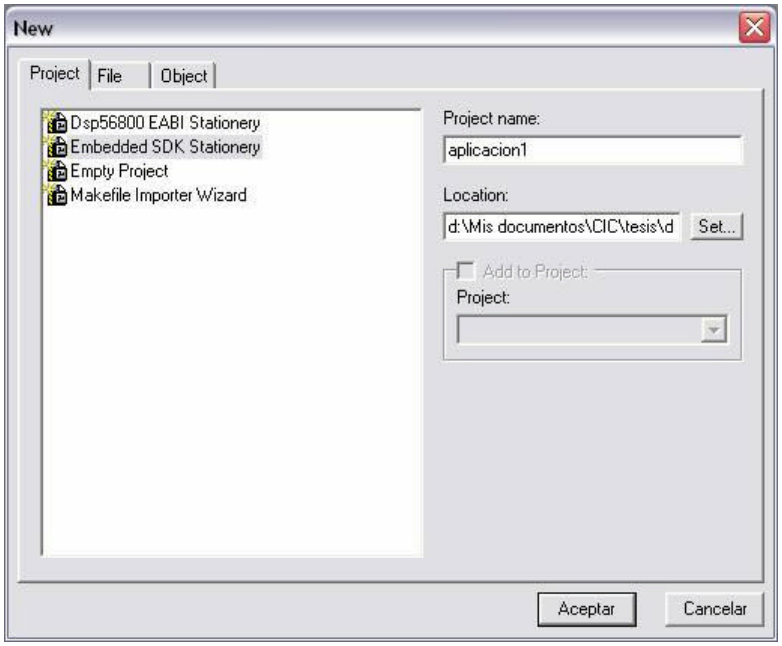


Figura 5.2- Creación de un Nuevo Proyecto en el IDE CodeWarrior.

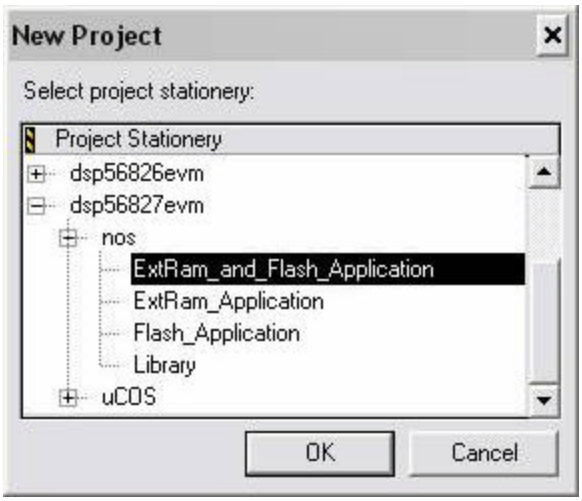


Figura 5.3- Selección del Tipo de Aplicación.

Una vez realizado los pasos anteriores el IDE CodeWarrior genera el proyecto junto con un conjunto de archivos organizados en grupos en donde se pueden visualizar de manera semejante a un explorador del Sistema Operativo Windows.

Dentro de los cambios al proyecto, hay que sustituir el archivo **main.c** que se genera por defecto, por el archivo de la aplicación. Además es necesario cambiar un archivo de configuración del SDK, denominado **appconfig.h**, en este archivo se activan o desactivan las funcionalidades del SDK, así como también los periféricos de la Tarjeta de Evaluación, los cambios que generalmente se hacen son los siguientes:

- Sustituir esta línea *#undef INCLUDE_TIMER /* Timer support */*
- Por *#define INCLUDE_TIMER /* Timer support */*
- Incluir *#define INCLUDE_UCOS /* Soporte del SDK para SOTR */*

Sin embargo estas modificaciones también dependen de la aplicación en particular, ya que se pueden activar otros periféricos de la Tarjeta de Evaluación como podrían ser los LED's que se emplean de vez en cuando para probar la multitarea del *Kernel*. Esto se haría incluyendo la siguiente línea *#define INCLUDE_LED*, en el archivo **appconfig.h**.

Otro cambio realizado al proyecto principal es la creación de un grupo que contenga los archivos del código del *Kernel* de Tiempo Real. Esto se crea dando clic en el menú *Project*, *Creat Group*..., e indicando el nombre del grupo, que para nuestros fines se le denomina *Codigo KTR*. En este grupo se incluyen los archivos del *kernel*. Para agregar los archivos una vez colocados

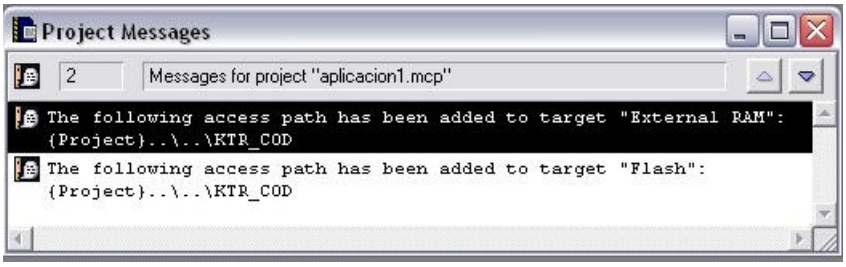
en el grupo, se da clic en el menú *Project*, *Add files*..., y se localizan los archivos y se indican cuales son los archivos que se van agregar. Automáticamente el IDE CodeWarrior detecta una nueva ruta de acceso a los directorios y la agrega a la configuración de rutas de acceso para el destino (*Target*) de la aplicación (RAM Externa o Flash) (véase Figura 5.4).

Los archivos incluidos en el grupo denominado *Codigo KTR* son los siguientes:

- **KTR.asm** – Contiene todas las rutinas programadas en ensamblador que son fundamentales para el *Kernel*.
- **KTR.c –** Contiene las primitivas internas de uso general en el *Kernel*.
- **KTR.h –** Contiene las definiciones de las constantes, variables y estructuras de datos empleadas en el *Kernel*.
- **So_bac.c -** Incluye las primitivas para el manejo de los *Buffers* Asíncronos Cíclicos.
- **So_bac.h -** Define las variables y estructuras de datos empleadas en los BAC's.
- **So_info.c -** Contiene las rutinas para consultar información del *Kernel*.
- **So_lista.c –** Incluye las rutinas para la administración de las listas empleadas en el *Kernel*.
- **So_sem.c –** Aloja las rutinas para el empleo de los semáforos.
- **So_tar.c –** Incluye las rutinas encargadas de administrar las tareas en el núcleo.
- **So_tiempo.c –** Contiene las rutinas encargadas de llevar el control del tiempo en el núcleo.

Sin embargo existen dos archivos que también se deben incluir en este grupo, pero que son dependientes de la aplicación. Los cuales deben localizarse en el mismo directorio del archivo proyecto de la aplicación y se mencionan a continuación:

- **Inc_maestro.h –** Incluye los archivos de cabecera que se emplean con el núcleo y la aplicación.
- **So_cnfg.h –** Este archivo permite activar o desactivar algunas rutinas del núcleo, para que se ahorre un poco de espacio.



(a)

(b)

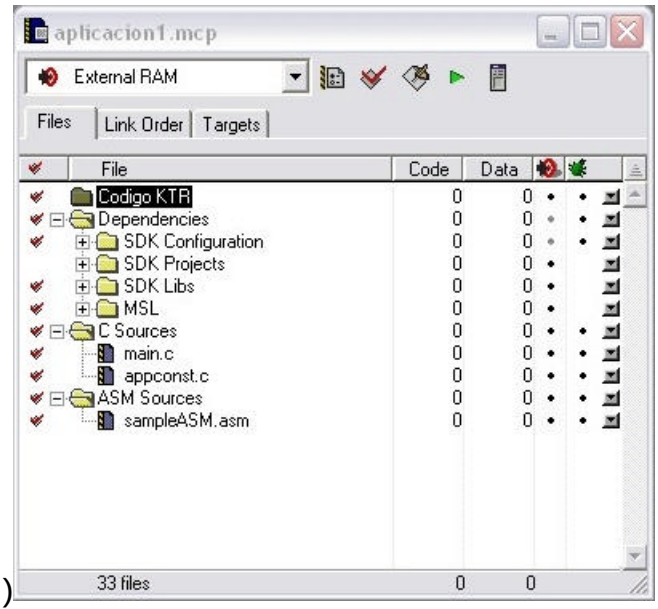


Figura 5.4- Anexo de Rutas de Acceso a los Directorios en el IDE CodeWarrior. (a) Muestra las nuevas rutas. (b) Muestra como quedan los archivos en el proyecto.

Llevando a cabo todas estas consideraciones es posible ya empezar a desarrollar las aplicaciones usando el núcleo de Tiempo Real.

5.2.1 Aplicación 1

Para probar el empleo del *Kernel* se creo una aplicación de ejemplo que mide el uso de la CPU, mientras están en ejecución un conjunto de tareas. Este cálculo del uso de la CPU se realiza empleando las tareas de no tiempo real considerando una serie de iteraciones e incrementando un contador que se usa para determinar el uso de la CPU, dicho contador se incrementa cada vez que en el sistema no haya una tarea de mayor prioridad en ejecución que la tarea ociosa.

Cabe aclarar que no se usan tareas periódicas en este ejemplo, porque cada vez que se crea una tarea periódica se hace el cálculo general del factor de utilización de la CPU de acuerdo con los parámetros de la tarea y el conjunto de tareas periódicas creadas.

El ejemplo consiste en la creación 13 tareas en el *Kernel* de Tiempo Real, todas estas tareas no tienen plazos o restricciones temporales explícitas, las cuales se enumeran enseguida:

1 tarea ociosa, que se ejecuta cada vez que no hay ninguna tarea de mayor prioridad lista para entrar en ejecución.

1 tarea de estadística, que toma como referencia un contador de ocio, para saber cuanto ha sido el uso del CPU.
1 tarea inicial, que crea un conjunto de tareas para la interacción del ejemplo y lleva el control del ejemplo verificando el número de iteraciones realizadas por el conjunto de tareas.
10 tareas semejantes, que cada una incrementa un contador interno, adquiere y libera un semáforo para sincronizar la ejecución.

Una vez ejecutado este ejemplo se obtiene una impresión en la consola del compilador, que nos indica el uso del CPU, aproximadamente el tiempo de ejecución del ejemplo es de unos 12 segundos, desde que se invoca la función main() del ejemplo, pero la ejecución de las 10 tareas desde su creación hasta la finalización de las mismas, toma 10 segundos. El uso del CPU de este conjunto de 10 tareas que se sincronizan con el semáforo fue de 37%, esto se puede observar en la Figura 5.5 que muestra la consola del compilador.

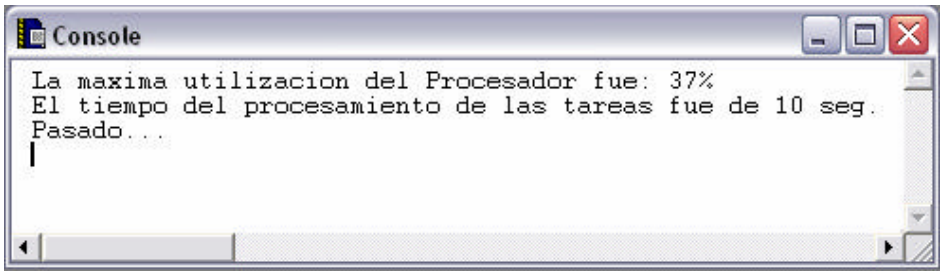


Figura 5.5- Salida del Ejemplo del Uso del CPU.

5.2.2 Aplicación 2

Esta aplicación se crea para probar la interacción de las tareas críticas y las tareas de no tiempo real manipulando los LED's que contiene la Tarjeta de Evaluación, dado que son seis los diodos emisores de luz proporcionados, se crea una tarea para cada LED, tres simulando tareas críticas (periódicas) y tres simulando tareas de no tiempo real.

El ejercicio es trivial, sin embargo enfatiza la interacción de los dos tipos de tareas que se pueden crear en el *Kernel*.

El ejemplo consta de siete tareas, tres periódicas, tres de no tiempo real y una ociosa, ambos tipos de tareas se invocan empleando código reentrante.

Las tareas tienen la finalidad de conmutar el estado del LED entre prendido y apagado, dependiendo del tiempo especificado a cada una de ellas. A las tareas críticas cuando se crean se le especifica como parámetro el tiempo en el cual ellas deben reanudar su ejecución. En el caso de las tareas de no tiempo real se especifica el tiempo en el cual deben de reanudar empleando la primitiva de retardo SOTmpEsp(). Con esto se logra que ambos tipos de tareas prendan o apaguen un LED en determinados intervalos de tiempo.

En esta aplicación no existe ninguna salida en la consola del IDE CodeWarrior, ya que el funcionamiento del ejemplo se ve directamente en la Tarjeta de Evaluación.

5.2.3 Aplicación 3

Para probar el empleo de los *Buffers* Asíncronos Cíclicos en el *Kernel*, se creo una aplicación trivial de ejemplo con dos tareas principales, donde una tarea envía un número (0-99) a través de un BAC y la otra recupera dicho número, el cual suma con los recibidos anteriormente. Una vez terminado el ciclo de 99 repeticiones, la última tarea despliega la suma.

El ejemplo consiste en la creación 3 tareas en el *Kernel* de Tiempo Real:

1 tarea ociosa, que se ejecuta cada vez que no haya ninguna tarea de mayor prioridad lista para entrar en ejecución.

1 tarea emisora, la cual reserva un *buffer* del BAC y coloca el dato que se envía en el *buffer*.

1 tarea receptora, la cual recibe el dato del *buffer* y lo suma a los anteriores datos. Al finalizar el ciclo de repeticiones (de 0 a 99) la tarea despliega datos en consola que se muestran en la Figura 5.6.

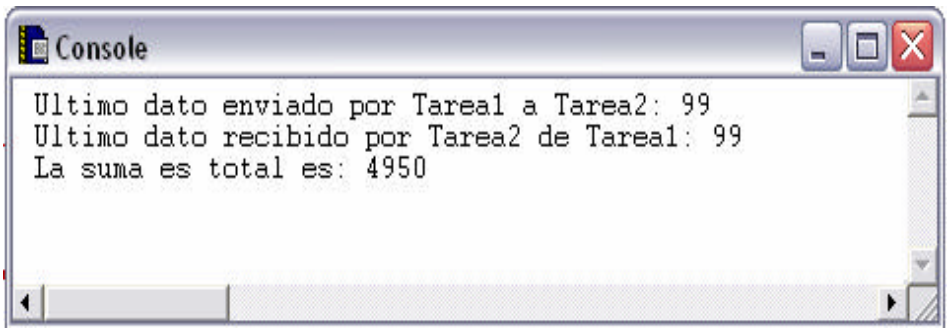


Figura 5.6- Salida de la Consola del IDE con el Ejemplo de BAC.

5.2.4 Aplicación 4

El uso de los semáforos es muy común cuando se desean sincronizar dos o más tareas, en base a ello los semáforos se han empleado para resolver el problema de sincronización denominado el problema de los filósofos comensales o la cena de los filósofos. El problema se puede enunciar de la siguiente manera. Cinco filósofos se sientan a la mesa. Cada uno tiene un plato de espaghetti. El espaghetti es tan escurridizo, que un filósofo necesita de dos tenedores para comerlo. Entre cada dos platos hay un tenedor [22].

La vida del filósofo consta de períodos alternados de comer y pensar. (Esto es una abstracción, incluso para los filósofos, pero las demás actividades son irrelevantes en este caso). Cuando un filósofo tiene hambre, intenta obtener un tenedor para su mano izquierda y otro para su mano derecha, alzando uno a la vez y en cualquier orden. Si logra obtener los dos tenedores, come un rato y después deja los tenedores y continúa pensando.

Para probar esta solución se implementó una aplicación de ejemplo haciendo uso del código del Kernel y en específico de los semáforos, demostrando con ello la funcionalidad de los mismos.

El ejemplo consiste en crear 5 tareas que representaran a los filósofos, cuando uno de ellos desee tomar su tenedor para iniciar a comer , éste verificará que los tenedores de la derecha y de la izquierda estén disponibles para que él los pueda utilizar y así comenzar a comer, al término de la comida, el filósofo devuelve los tenedores para que los otros filósofos los puedan ocupar. En el ejemplo los tenedores se representan con semáforos, como regla se sigue que exista un tenedor por filósofo.

Como no existen rutinas adecuadas para el despliegue de información (ni en el CodeWarrior, ni en el Kernel) que permitan visualizar claramente la funcionalidad del ejemplo, se tuvo que implementar dicho ejemplo usando los LED's que trae la tarjeta de evaluación y así poder visualizar alguna salida. Debido a esta situación en este documento no se presentan imágenes de la salida del ejemplo. Sin embargo se puede expresar que el uso de los semáforos es bueno para el problema de sincronización de las tareas, ya que le da el acceso sólo a aquella que tiene la capacidad de ingresar a una sección crítica en un momento dado.

CONCLUSIONES

Llegado a este punto de la Tesis se puede afirmar que el *Kernel* de Tiempo Real planteado y presentado en este documento, es una alternativa más para los diseñadores y desarrolladores de Aplicaciones de Tiempo Real, ya que provee mecanismos para planificar tareas críticas y de no tiempo real, para sincronizar tareas cuando comparten recursos y también para intercambiar datos entre las tareas del sistema. Además, ya que está diseñado e implementado para funcionar en un Procesador Digital de Señales contribuye a la implementación de algoritmos de Filtrado Digital de Señales en Tiempo Real.

El KTR-DSP posee varias características básicas con las cuales todo Kernel de Tiempo Real debe contar.

El KTR-DSP es *determinista*, ya que a través de la prueba de planificabilidad que se hace al crear una tarea crítica se puede decir si la tarea es planificable o no.

El kernel también cuenta con la capacidad de responder a sucesos rápidamente a través de las interrupciones, cuenta además con la gestión de prioridades empleando un algoritmo de asignación dinámica conocido como Primero el Plazo más Corto.

Otra característica más proporcionada por el kernel es la comunicación entre tareas llevada acabo por los buffers. Además para ahorrar espacio y esfuerzo en la programación de aplicaciones el kernel soporta la ejecución de código reentrante es decir se puede usar una misma función para crear varias tareas. Permite además Configurar las primitivas necesarias para usarse en cada aplicación en particular.

El *kernel* propuesto es una opción viable para algunas aplicaciones de tiempo real que requieran plantear su conjunto de tareas en base a los plazos de vencimiento que requiere cada una, a la vez permite definir las tareas como periódicas y tareas de no tiempo real.

Con esto el diseñador de la aplicación en tiempo real, se preocupará de dividir su aplicación en pequeñas tareas que indicarán el tiempo de ejecución y el plazo máximo que la tarea se puede tardar en completarse.

RECOMENDACIONES Y TRABAJOS FUTUROS

- Modificar el algoritmo de planificación para que se adapte de manera más flexible para la implementación de mecanismos de distribución de recursos que tratan de eliminar la inversión de prioridades como son los protocolos de herencia de prioridades.

- Como otra extensión al trabajo presentado se encuentra la migración del código del *Kernel* de Tiempo Real en base a las especificaciones del estándar POSIX de Tiempo Real (1003.1B).

- Ya que en el presente trabajo no se incluye un intérprete de comandos, es necesario que éste se implemente dentro del kernel y sirva de comunicación directa entre el usuario y el sistema.

- Implementar otros mecanismos de sincronización entre las tareas, como podrían ser las banderas de eventos o las señales.

- Se puede considerar también la ampliación de los mecanismos de comunicación entre tareas, y poder implementar aquellos como los buzones de mensajes y las colas de mensajes muy usadas en otros kernels de tiempo real.

Todas estas consideraciones deben ser tomadas en cuenta para complementar el presente trabajo y así poder ofrecer más primitivas posibles de usarse en una gran variedad de Aplicaciones de Tiempo Real, sin embargo siempre será necesario verificar que el núcleo no sea muy extenso en tamaño para así poderlo alojar en la memoria principal de nuestro DSP o microcontrolador objetivo.

APÉNDICE A - CONJUNTO DE INSTRUCCIONES DEL NÚCLEO DSP56800

La arquitectura del DSP se puede ver como varias unidades funcionales operando en paralelo[9]:

- ALU de Datos
- AGU
- Controlador de Programa
- Unidad de Manipulación de Bits

La meta del conjunto de instrucciones es mantener ocupadas a cada una de estas unidades en cada ciclo de instrucción. Esto logra la máxima velocidad, el consumo mínimo de energía, y el uso mínimo de la memoria de programa.

La capacidad del rango completo de instrucciones combinada con los modos flexibles de direccionamiento proporciona un poderoso lenguaje ensamblador para los algoritmos de procesamiento digital de señales y cómputo de propósito general.

El conjunto de instrucciones se ha diseñado también para permitir la codificación eficiente de algoritmos de DSP, de control de código y compiladores de lenguaje de alto nivel. El tiempo de ejecución es mejorado por las capacidades de bucle por *hardware*. En este apartado se introducen las instrucciones MOVE disponibles en el núcleo del DSP, el concepto de movimientos paralelos, los formatos de instrucciones del DSP, el modelo de programación del núcleo del DSP, grupos del conjunto de instrucciones, un resumen del conjunto de instrucciones en forma tabular, y una introducción al pipeline de instrucciones.

Introducción a los Movimientos y Movimientos en Paralelos

Para simplificar la programación, un conjunto de instrucciones de movimiento (MOVE) se encuentra en el núcleo del DSP. Esto facilita no sólo la tarea de programar el DSP, sino también disminuye el tamaño de código del programa y mejora la eficacia, que alternadamente disminuye el consumo de energía y las MIPS requeridas para realizar una tarea dada.

Algunos ejemplos de instrucciones MOVE se listan en el Ejemplo 5.1.

Ejemplo 5.1- Tipos de la Instrucción MOVE.

```
MOVE <cualquier_registro_del_DSP>,   <cualquier_registro_del_DSP>
MOVE <cualquier_registro_del_DSP>,   <Memoria_X_Datos>
MOVE <cualquier_registro_del_DSP>,   <registro_periferico_Int>
MOVE <Memoria_X_Datos >,             <cualquier_registro_del_DSP>
MOVE <registro_periferico_Int>,      <cualquier_registro_del_DSP>
MOVE <valor_inmediato>,              <cualquier_registro_del_DSP>
MOVE <valor_inmediato>,              <Memoria_X_Datos>
MOVE <valor_inmediato>,              <registro_periferico_Int>
```

Para cualquier instrucción MOVE que tiene acceso a la memoria de datos X o a un registro de un periférico interno mapeado en memoria, se soportan siete modos de direccionamiento diferentes. Modos de direccionamiento adicionales están disponibles en el subconjunto de registros del DSP que tienen acceso más frecuentemente, incluyendo los registros de la ALU de Datos, y todos los apuntadores de la Unidad de Generación de Direcciones.

Para todas las instrucciones MOVE en el DSP56800, la sintaxis dispone de la fuente y el destino como sigue: **SRC, DST**. La fuente de los datos que se moverán y el destino son separados por una coma, sin espacios antes o después de la coma. La sintaxis del ensamblador indica también cual memoria está siendo accesada (memoria de programa o datos) en cualquier movimiento de la memoria. La Tabla 5.2 presenta la sintaxis para especificar el correcto espacio de memoria para cualquier acceso de memoria; se presenta un ejemplo de acceso a la memoria de programa donde la dirección está contenida en el registro R2 y la dirección del registro se post-incrementa después del acceso. Los dos ejemplos para los accesos a la memoria de datos X, presentan un modo de direccionamiento indirecto por registro de dirección en el primer ejemplo y un direccionamiento absoluto en el segundo.

Tabla 5.2- Símbolos de Espacio de Memoria.

Símbolo	Ejemplos	Descripción
P:	P:(R2)+	Acceso a la Memoria de Programa
X:	X:(R0) X:$C000	Acceso a la Memoria de Datos X

El conjunto de instrucciones del DSP56800 soporta dos tipos de movimientos –el movimiento paralelo simple y la lectura dual en paralelo. Ambos son

considerados “movimientos paralelos” y son extremadamente poderosos para algoritmos de Procesamiento Digital de Señales y cálculos numéricos.

El movimiento paralelo simple permite que una operación aritmética y un movimiento de la memoria sean completados con una instrucción en un ciclo de instrucción. Por ejemplo, es posible sumar dos números mientras se lee o escribe un valor de la memoria en la misma instrucción.

La Figura 5.7 ilustra un movimiento paralelo simple, que utiliza una palabra de programa y se ejecuta en un ciclo de instrucción.

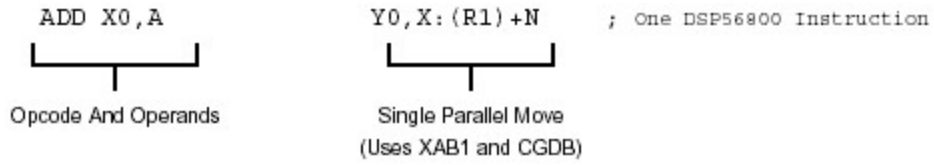


Figura 5.7 - Movimiento Paralelo Simple.

En el movimiento paralelo simple, ocurre lo siguiente:

- El registro X0 es sumado al registro A y el resultado se almacena en el acumulador A.
- El contenido del registro Y0 se mueve a la memoria de datos X en la localidad contenida en el registro R1.
- Después de completar el movimiento de memoria, el registro R1 se post-actualiza con el contenido del registro N.

La lectura dual en paralelo permite que una operación aritmética tenga lugar y dos valores se lean de la memoria de datos X con una instrucción en un ciclo de instrucción. Por ejemplo, es posible ejecutar en la misma instrucción una multiplicación de dos números, con o sin redondear el resultado, mientras se leen dos valores desde la memoria de datos X hacia dos registros de datos de la ALU.

La Figura 5.8 ilustra un movimiento paralelo doble, que utiliza una palabra de programa y se ejecuta en un ciclo de instrucción.

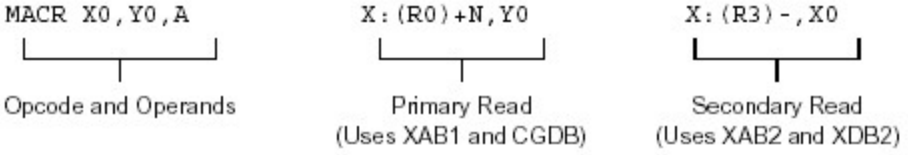


Figura 5.8- Movimiento Paralelo Dual.

En el movimiento paralelo dual, ocurre lo siguiente:

- El contenido de los registros X0 e Y0 son multiplicados, el resultado se suma al acumulador A, el resultado final se almacena en el acumulador A.
- El contenido de la localidad de la memoria de datos X apuntada con el registro R0 se mueve al registro Y0.
- El contenido de la localidad de memoria de datos X apuntada con el registro R3 se mueve al registro X0.
- Después de completar los movimientos de memoria, el registro R0 es post-actualizado con el contenido del registro N y el registro R3 se disminuye en uno.

Ambos tipos de movimientos paralelos utilizan un subconjunto de modos de direccionamiento disponibles en el DSP56800, y los registros disponibles para la parte de movimiento de instrucciones son también un subconjunto del conjunto total de registros del núcleo del DSP.

Estos subconjuntos incluyen los registros y modos de direccionamiento encontrados más frecuentemente en los algoritmos de Procesamiento Digital de Señales y cálculos numéricos de alto desempeño. También, los movimientos paralelos permiten que un movimiento ocurra sólo con una operación aritmética en la ALU de Datos. Un movimiento paralelo no está permitido, por ejemplo, con una instrucción JMP, LEA o BFSET.

Formatos de Instrucciones

Las instrucciones tienen una, dos o tres palabras de longitud. La instrucción es concretada por la primera palabra de la instrucción. Las palabras adicionales pueden contener información sobre la instrucción misma o un operando para la instrucción. Ejemplos de código fuente en lenguaje ensamblador para varias instrucciones se muestran en la Tabla 5.3.

Desde los formatos de instrucciones enumerados en la Tabla 5.3, puede ser visto que el DSP ofrece procesamiento paralelo usando la ALU de Datos, la AGU, el Controlador de Programa y la Unidad de Manipulación de Bits. En el ejemplo del movimiento paralelo, el DSP puede realizar una operación señalada por la ALU (datos ALU) y hasta dos transferencias de datos especificadas con las actualizaciones del registro de dirección (AGU), y también descifrará la instrucción siguiente y traerá una instrucción de la memoria de programa (Controlador de Programa), todo en un ciclo de la instrucción. Cuando una instrucción tiene más de una palabra de longitud, es requerido un adicional ciclo de instrucción-ejecución. La mayoría de las

instrucciones involucradas con la ALU de Datos son basadas en registros (es decir, los operandos están en los registros de datos de la ALU) y permiten que el programador mantenga cada unidad de procesamiento paralelo ocupada. Instrucciones que están orientadas a la memoria (por ejemplo, una manipulación de bits), todas las instrucciones lógicas, o instrucciones que causan un cambio en el control de flujo (tal como un salto) impiden el uso de todos los recursos del procesamiento paralelo durante su ejecución.

Tabla 5.3- Formatos de Instrucciones.

Op-code [1]	Operandos[2]	Transferir por CGDB[3]	Transferir por XDB2[4]	Transferir por PDB[5]	Comentarios
ADD	#$1234,Y1				No movimiento paralelo
ANDC	#$7C,X:$E27				No movimiento paralelo
ENDDO					No movimiento paralelo
TSTW	X:(SP-9)				No movimiento paralelo
MAC	A1,Y0,B				No movimiento paralelo
LEA	(R2)-				No movimiento paralelo
MOVE		R0,Y0			No movimiento paralelo
CMP	X0,B	Y0, X:(R2)+			Movimiento paralelo simple
NEG	A	X:(R1)+N, X0			Movimiento paralelo simple
SUB	Y1,A	X:(R0)+,Y0	X:(R3)+,X0		Lectura dual en paralelo
MPY	X1,Y0,B	X:(R1)+N, Y1	X:(R3)+,X0		Lectura dual en paralelo
MACR	X0,Y0,A	X:(R1)+N, Y0	X:(R3)-,X0		Lectura dual en paralelo

MOVE				X0 ,P:(R1)+	Movimiento de memoria de programa
JMP	$3C10				Salto a una dirección de 16 bits

1. Indica los datos de la ALU, AGU, Controlador de Programa, o Unidad de Manipulación de Bits que se ejecutarán.
2. Especifica los operando usados por el opcode.
3. Indica transferencia de datos opcionales por el bus CGDB (Core Global Data Bus).
4. Indica transferencia de datos opcionales por el bus XDB2 (External Data Bus).
5. Indica transferencia de datos opcionales por el bus PDB (Program Data Bus).

Modelo de Programación del DSP56800

Los registros en el modelo de programación del núcleo del DSP56800 se muestran en la Figura 5.9.

Grupo de Instrucciones

El conjunto de instrucciones se divide en los siguientes grupos:

- Aritmético
- Lógico
- Manipulación de Bits
- Bucle
- Movimiento
- Control de Programa

Instrucciones Aritméticas

Las instrucciones aritméticas ejecutan todas las operaciones aritméticas dentro de la ALU de Datos. Pueden afectar un subconjunto o todos los bits del registro de condición del código. Las instrucciones aritméticas son típicamente basadas en registros (modos de direccionamiento directos por registro son usados por los operandos) de modo que la operación de la ALU de Datos indicada por la instrucción no utiliza el CGDB o el XDB2, aunque algunas instrucciones pueden operar también sobre datos inmediatos u operandos en memoria. Las transferencias de datos opcionales (movimientos paralelos) pueden especificarse con muchas instrucciones aritméticas. Esto permite movimiento de datos paralelo sobre el CGDB y sobre el XDB2 durante una operación de la ALU de Datos. Esto permite que los nuevos datos sean pre-traídos para usarlos en las siguientes

instrucciones y los resultados calculados por las instrucciones anteriores sean almacenados. Las instrucciones aritméticas se ejecutan típicamente en un ciclo de instrucción, aunque algunas de las operaciones pueden tomar ciclos adicionales con diversos modos de direccionamiento del operando. *Las instrucciones aritméticas son la única clase de instrucciones que permiten movimientos paralelos.*

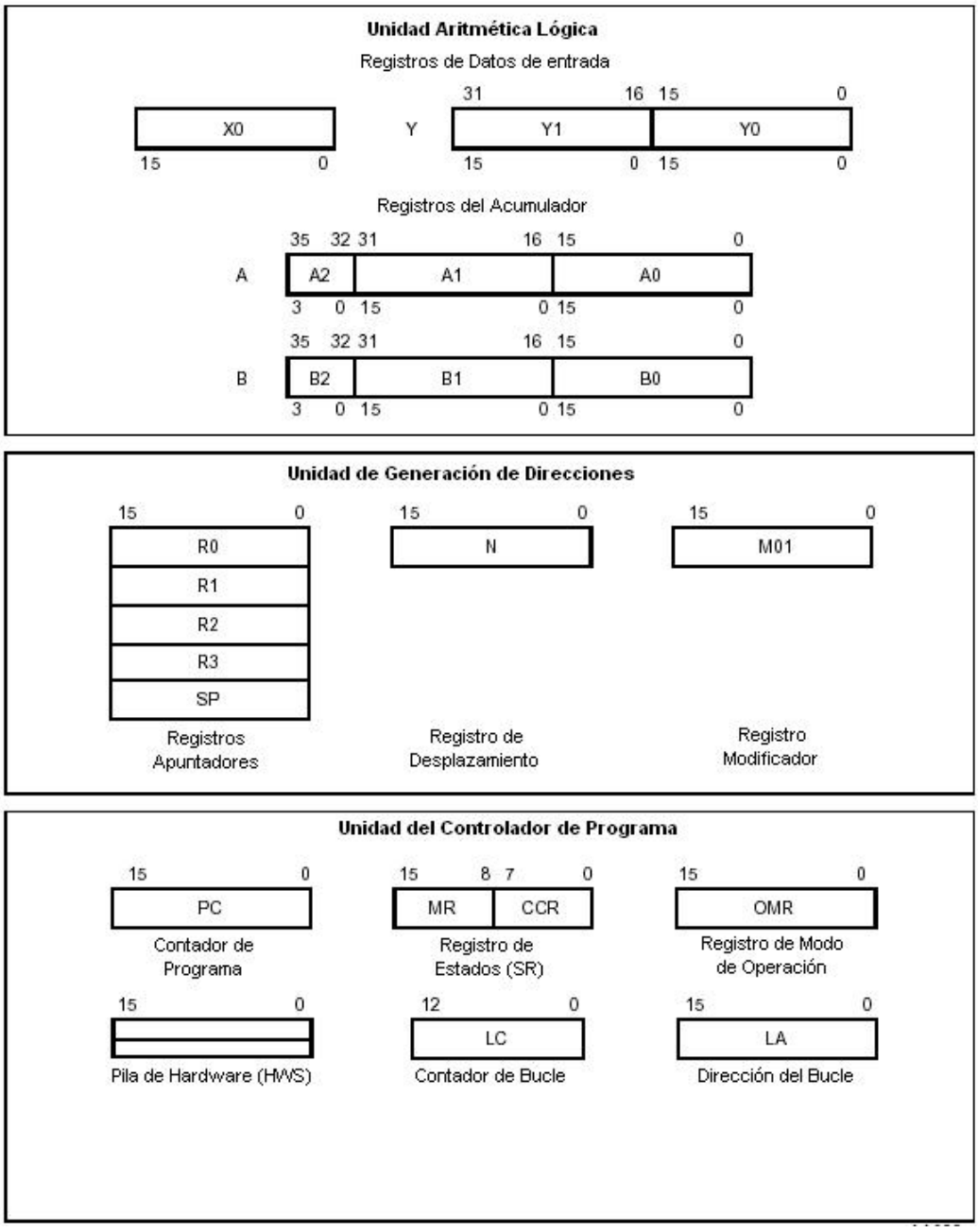


Figura 5.9- Modelo de Programación del Núcleo DSP56800.

Además de los cambios aritméticos presentes aquí, otros tipos de cambios están disponibles también en el grupo de instrucciones lógicas. La Tabla 5.4 enumera las instrucciones aritméticas.

Tabla 5.4- Lista de Instrucciones Aritméticas.

Instrucción	Descripción
ABS	Valor absoluto
ADD	Suma
ASL	Corrimiento aritmético a la izquierda (36 bits)
ASLL	Corrimiento aritmético multi-bits a la izquierda[1]
ASR	Corrimiento aritmético a la derecha (36 bits)
ASRAC	Corrimiento aritmético multi-bits a la derecha con acumulación[1]
ASRR	Corrimiento aritmético multi-bits a la derecha[1]
CLR	Borrado
CMP	Comparación
DEC(W)	Decremento de la palabra superior del acumulador
DIV	División[1]
IMPY(16)	Multiplicación de enteros[1]
INC(W)	Incremento de la palabra superior del acumulador
MAC	Multiplicación-Acumulación con signo
MACR	Multiplicación-Acumulación con signo y redondear
MACSU	Multiplicación-Acumulación con signo y sin signo[1]
MPY	Multiplicación con signo
MPYR	Multiplicación con signo y redondear
MPYSU	Multiplicación con signo y sin signo[1]
NEG	Negación
NORM	Normalización[1]
RND	Redondeo
SBC	Resta larga con acarreo[1]
SUB	Resta
Tcc	Transferencia condicional[1]
TFR	Transferencia de datos del registro de la ALU a un acumulador
TST	Prueba un acumulador de 36 bits
TSTW	Prueba un registro de 16 bits o una localidad de memoria[1]

1. Estas instrucciones no permiten movimiento de datos paralelo.

Instrucciones Lógicas

Las instrucciones lógicas ejecutan todas las operaciones lógicas dentro de la ALU de Datos. También afectan los bits del registro de condición del código. Las instrucciones lógicas están basadas en registros. Así como en las instrucciones aritméticas, algunas instrucciones pueden también operar sobre los operandos en memoria. Estas instrucciones se ejecutan en un ciclo de instrucción.

La Tabla 5.5 enumera las instrucciones lógicas.

Tabla 5.5- Lista de Instrucciones Lógicas.

Instrucción	Descripción
AND	Y lógico
EOR	O Exclusivo lógico
LSL	Corrimiento lógico a la izquierda
LSLL	Corrimiento lógico multi-bits a la izquierda
LSRAC	Corrimiento lógico a la derecha con acumulación
LSR	Corrimiento lógico a la derecha
LSRR	Corrimiento lógico multi-bits a la derecha
NOT	Complemento lógico
OR	O Inclusivo lógico
ROL	Rotación a la izquierda
ROR	Rotación a la derecha

Instrucciones de Manipulación de Bits

Las instrucciones de manipulación de bits realizan una de las siguientes tres tareas:

- Probar un campo de bits dentro de una palabra.
- Probar y modificar un campo de bits dentro de una palabra.
- Salto condicional basado en la prueba de bits dentro del byte superior o inferior de una palabra.

Las instrucciones de campos de bits pueden operar sobre cualquier localidad de la memoria X, periféricos, o en los registros del DSP. BFTSTH y BFTSTL pueden probar cualquier campo de bits dentro de una palabra de 16 bits. BFSET, BFCLR, y BFCHG pueden probar cualquier campo de bits dentro de una palabra de 16 bits y entonces establecer, borrar, o invertir bits en esta palabra, respectivamente. BRSET y BRCLR pueden probar solamente un campo de 8 bits en el byte superior o inferior de una palabra,

y entonces saltar condicionalmente basándose en el resultado de la prueba. El bit de acarreo del registro de condición del código contiene el resultado de la prueba del bit por cada instrucción. Estas instrucciones son operaciones del tipo leer-modificar-escribir. Las instrucciones BFTSTH, BFTSTL, BFSET, BFCLR, y BFCHG se ejecutan en dos o tres ciclos de instrucción. Las instrucciones BRCLR y BRSET se ejecutan en cuatro o seis ciclos de instrucción.

La Tabla 5.6 enumera las instrucciones de manipulación de bits.

Tabla 5.6- Lista de Instrucciones de Manipulación de Bits.

Instrucción	Descripción
ANDC	Y lógico con datos inmediatos
BFCLR	Prueba y borrado del campo de bits
BFSET	Prueba y fijación del campo de bits
BFCHG	Prueba y cambio del campo de bits
BFTSTL	Prueba del campo de bits bajo
BFTSTH	Prueba del campo de bits alto
BRSET	Salto sí los bits seleccionados están fijos (en uno)
BRCLR	Salto sí los bits seleccionados están limpios (en cero)
EORC	O Exclusivo lógico con datos inmediatos
NOTC	Complemento lógico sobre una localidad de memoria o los registros
ORC	O Inclusivo lógico con datos inmediatos

Nota:

Debido a la canalización (pipelining) de instrucciones, sí un registro de la AGU (Rn, N, SP, o MO1) es directamente cambiado por una instrucción de campo de bits, el nuevo contenido no puede estar disponible para usarse hasta la segunda instrucción siguiente.

Instrucciones de Bucle

Las instrucciones de bucle establecen parámetros de bucle y dan comienzo al bucle del programa con cero gastos. Permiten el bucle en una simple instrucción (REP) o en un bloque de instrucciones (DO). Para el bucle con DO, la dirección de la primera instrucción en el bucle del programa es salvada en la pila de *hardware* para permitir el ciclo sin gastos indirectos. La última dirección del bucle DO se especifica con una dirección absoluta de 16 bits. Ninguna localidad en la pila de *hardware* se requiere para la instrucción REP. La instrucción ENDDO se utiliza solamente cuando se rompe el bucle; de otra manera, es mejor usar MOVE #1, LC.

La Tabla 5.7 enumera las instrucciones de bucle.

Tabla 5.7- Lista de Instrucciones de Bucle.

Instrucción	Descripción
DO	Inicia el bucle por *hardware*
ENDDO	Desactiva el bucle actual y saca de la pilas los parámetros
REP	Repite la siguiente instrucción

Instrucciones de Movimiento

Las instrucciones de movimiento mueven datos por los diferentes buses de datos: CGDB, PGDB, XDB2, y PDB. Las instrucciones de movimiento no afectan el registro de condición del código, excepto al bit de límite sí la restricción se realiza cuando se lee un registro acumulador de la ALU de datos. Estas instrucciones no permiten transferencias de datos opcionales. Además de las siguientes instrucciones de movimiento, hay movimientos paralelos que pueden utilizarse simultáneamente con muchas de las instrucciones aritméticas.

La Tabla 5.8 enumera las instrucciones de movimiento.

Tabla 5.8- Lista de Instrucciones de Movimiento.

Instrucción	Descripción
LEA	Carga efectiva de la dirección
POP	Saca un registro de la pila de *software*
MOVE	Mueve datos
MOVE(C)	Mueve registros de control
MOVE(I)	Mueve datos inmediatos
MOVE(M)	Mueve memoria de programa
MOVE(P)	Mueve datos del periférico
MOVE(S)	Movimiento corto absoluto

NOTA:

Debido a la canalización de instrucciones, sí un registro (Rn, SP, o MO1) de la AGU es cambiado directamente por una instrucción de movimiento, el nuevo contenido no puede estar disponible para usarse hasta la segunda instrucción siguiente.

Instrucciones de Control de Programa

Las instrucciones de control de programa incluyen ramificaciones, saltos, ramificaciones condicionales, saltos condicionales y otras instrucciones que afectan el Contador de Programa y Pila de *Software*. Las instrucciones de control de programa pueden afectar los bits del registro de Estado como

se especifica en las instrucciones. También incluidas en este grupo de instrucciones están las instrucciones STOP y WAIT las cuales pueden colocar al chip del DSP en un estado de baja-energía.

La Figura 5.9 enumera las instrucciones de control.

Tabla 5.9- Lista de Instrucciones de Control de Programa.

Instrucción	Descripción
Bcc	Ramificación condicional
BRA	Ramificación
DEBUG	Entra en modo de depuración
Jcc	Salto condicional
JMP	Salto
JSR	Salto a subrituna
NOP	No operación
RTI	Regreso de una interrupción
RTS	Regreso de una subrutina
STOP	Parada del procesamiento (Modo de espera con mínimo consumo de energía)
SWI	Interrupción por *software*
WAIT	Espera por una interrupción (Modo de espera con bajo consumo de energía)

Canalización (Pipeline) de las Instrucciones

La ejecución de la instrucción es canalizada para permitir ejecutar más instrucciones a razón de una instrucción por cada dos ciclos de reloj. No obstante, ciertas instrucciones requieren tiempo adicional para ejecutarse, incluyendo las instrucciones con las siguientes propiedades:

- Exceden la longitud de una palabra.
- Usan un modo de direccionamiento que requiere más de un ciclo.
- Tienen acceso a la memoria de programa.
- Causan un cambio en el control de flujo.

En el caso de un cambio del control de flujo, un ciclo se necesita para borrar la canalización.

Procesamiento de la Instrucción

La canalización permite las operaciones de búsqueda-decodificación-ejecución de una instrucción a suscitarse durante las operaciones de

búsqueda-decodificación-ejecución de otras instrucciones. Mientras una instrucción se ejecuta, la siguiente instrucción a ejecutarse se decodifica, y la instrucción que sigue a la instrucción que se está decodificando es traída desde la memoria de programa. Si una instrucción es de dos palabras de longitud, la palabra adicional será buscada antes que la instrucción siguiente sea buscada.

La Figura 5.10 muestra la canalización; F1, D1, y E1 se refieren a las operaciones de búsqueda, decodificación y ejecución, respectivamente, de la primera instrucción. Note que la tercera instrucción contiene una palabra de extensión de la instrucción y toma dos ciclos para ejecutarse.

Búsqueda	F1	F2	F3	F3e	F4	F5	F6	...
Decodificación		D1	D2	D3	D3e	D4	D5	...
Ejecución			E1	E2	E3	E3e	E4	...
Ciclo de Instrucción	1	2	3	4	5	6	7	...

Figura 5.10- Canalización –Pipelining.

Procesamiento del Acceso a Memoria

Una o más fuentes de memoria del DSP (memoria de datos X y memoria de programa) pueden tener acceso durante la ejecución de una instrucción. Tres buses de direcciones (XAB1, XAB2, y PAB) y tres buses de datos (CGDB, XDB2, y PDB) están disponibles para acceder a la memoria interna durante un ciclo de instrucción, pero sólo uno de los buses de direcciones y uno de los buses de datos está disponible para acceder a la memoria externa (cuando el bus externo está disponible). Si todas las fuentes de memoria son internas al DSP, una o más de las fuentes de memoria pueden tener acceso en un ciclo de instrucción (eso es, acceso a la memoria de programa, o acceso a la memoria de programa más una referencia a la memoria de datos X, o acceder a la memoria de programa con dos referencias a la memoria de datos X).

APÉNDICE B - INTERFAZ DE COMUNICACIÓN DE LA PC CON EL DSP

Se presenta el siguiente capítulo para describir las características del IDE CodeWarrior como la interfaz de comunicación con el DSP que está inmerso en la Tarjeta de Evaluación DSP56F827EVM, ya que a través de él se puede descargar, ejecutar y depurar el código en el DSP.

Cuando se desarrollan aplicaciones incrustadas basadas en MCU o DSP los requerimientos típicos inician con la instalación de herramientas, tales como ensambladores, cargadores de conexión, y depuradores a nivel de ensamble. Estas son requeridas para todo el desarrollo del *software*. Más allá de eso, otras herramientas son requeridas a menudo, como los compiladores de alto nivel para C, simuladores del conjunto de instrucciones, sistemas operativos de tiempo real, bibliotecas de funciones y bibliotecas de aplicaciones [5].

Idealmente, todas de las herramientas deberían ser accesibles desde un solo ambiente de desarrollo e integradas en una forma integral. La solución debe dar a los usuarios la habilidad de integrar cualquier herramienta disponible, *hardware* o *software*, ahora y en el futuro. Por otra parte, dado que los diseñadores tienen necesidades específicas, las herramientas deben permitir a los desarrolladores integrar sus personalizaciones, utilidades de depuración, y/o sus bibliotecas dentro del ambiente integrado. Por último pero no menos, los materiales de aprendizaje en las herramientas deben ser fáciles de conseguir y fáciles de entender.

El Ambiente de Desarrollo Integrado CodeWarrior se dirige a estas necesidades ofreciendo características únicas y acceso integral a una variedad de herramientas de desarrollo, permitiendo la personalización del ambiente por el usuario. Combina un poderoso editor y navegador de código fuente, compiladores y enlazadores verificados, un sofisticado administrador de proyectos, un amigable simulador-depurador, y un extenso conjunto de bibliotecas, toda bajo un solo techo. El IDE hace posible para los desarrolladores editar, administrar, compilar, enlazar y depurar aplicaciones con menos esfuerzo y mejores resultados.

El administrador de proyectos de CodeWarrior (véase Figura 5.11) provee una interfaz automática, rápida, conveniente para orquestar la operación de todas las herramientas a lo largo del ciclo de desarrollo, como es la creación de proyectos, la apertura de proyectos, adición de archivos y el salvado de los proyectos. Además, un desarrollador puede mover archivos en la ventana del Proyecto, marcar archivos para depuración, crear proyectos anidados y construir destinos, y dividir la ventana del proyecto dentro de grupos de archivos.

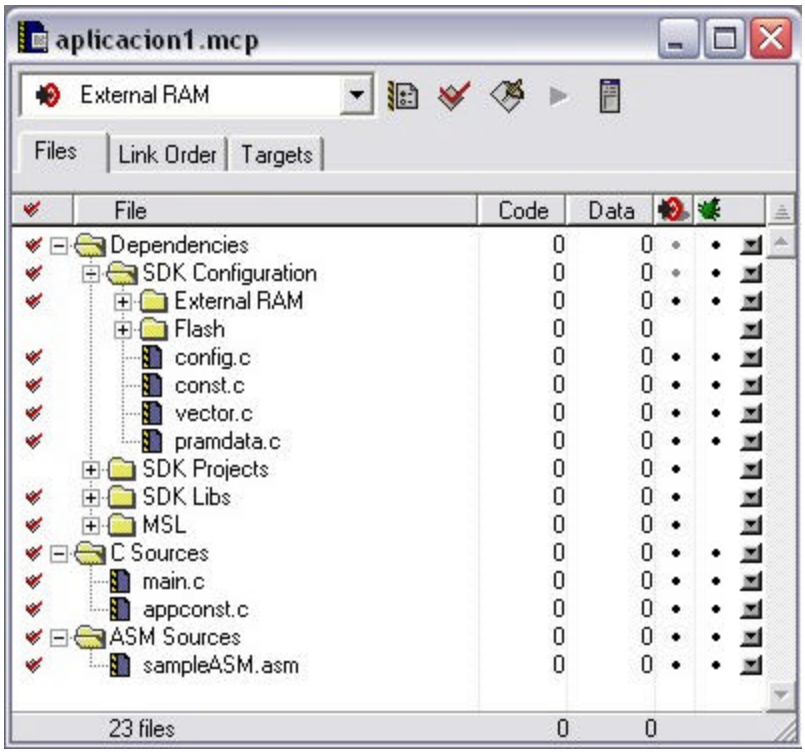


Figura 5.11- Administrador Visual de Proyectos.

El administrador de proyectos mantiene la pista de todos los archivos usados a lo largo del proyecto.

Un navegador permite al usuario clasificar el código a través de una base de datos generada por el compilador. Históricamente los programadores han usado navegadores sobre todo con código orientado a objetos, pero el navegador del IDE CodeWarrior trabaja con el código de procedimientos y el código orientado a objetos. El navegador soporta la mayoría de los compiladores, incluyendo C/C++, Pascal y Java.

El nuevo mercado del Procesamiento Digital de Señales presenta un característico reto para los desarrolladores. Las restricciones de tiempo para el mercado les exigen actuar como desarrolladores de sistemas de alto nivel, programando componentes de sistemas en un nivel muy superior al lenguaje ensamblador enfoque preferido por los diseñadores tradicionales de DSP.

La programación en un lenguaje de alto nivel, como C, ayuda a reducir el tiempo de desarrollo y asegura la portabilidad. Esto es especialmente verdadero cuando los desarrolladores están trabajando con una arquitectura nueva o desconocida. A pesar de eso, la aplicación final debe estar optimizada para cumplir los requerimientos de tiempo real. Si el

compilador C no está optimizado para la arquitectura fundamental del DSP, puede significar largas horas de reajustar manualmente el código en lenguaje ensamblador.

El compilador de CodeWarrior obedece el estándar ANSI (American Nacional Standards Institute) C. Además contiene un completo editor integrado de código fuente que mejora la productividad del desarrollador. El editor puede automáticamente verificar el balance de los paréntesis, corchetes, y las llaves, consecuentemente minimizando los errores de compilación. Los menús emergentes en la ventana del editor proveen navegación instantánea para iniciar de cualquier función o archivo de encabezado dentro del proyecto. La coloración de sintaxis en el editor formatea el código fuente para una fácil identificación de los comentarios y las palabras claves.

El editor de CodeWarrior (véase Figura 5.12) es muy diferente de otros editores en que la depuración del código fuente se hace directamente en la ventana del editor. Esto permite al desarrollador fijar puntos de ruptura y corregir errores en esta ventana y evitar el consumo de tiempo intercambiándose entre el editor y el depurador.

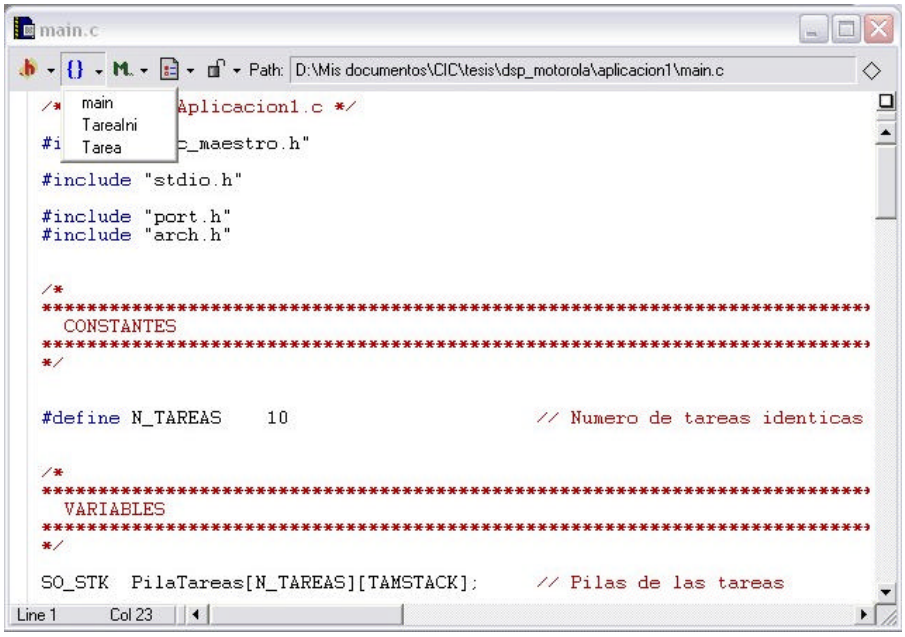


Figura 5.12- Editor del IDE CodeWarrior.

El enlazador es una parte integral del administrador de proyectos. Los desarrolladores pueden especificar el orden en el cual los archivos son compilados en la vista de Enlaces de la ventana del proyecto. Reorganizando el orden de los archivos se impiden los errores de vinculación causados por las dependencias de los archivos.

El depurador de CodeWarrior es una herramienta ideal para que el usuario identifique errores, los remueva, y valide el código. Proporciona comando para ejecutar una o varias líneas de código a la vez, suspende la ejecución cuando el control alcanza un punto especificado, o interrumpe un programa que cambia el valor de una localidad de memoria designada. Despliega información detallada para examinar y cambiar los valores de las variables, examinar el contenido de los registros del procesador, y abrir ventanas de la memoria para desplegar y editar localidades de bytes o word. Cuando el depurar detiene un programa, el usuario puede ver la cadena de las llamadas a funciones. El depurador interactúa con el simulador o con la tarjeta del DSP.

La capacidad multi-objetivo del IDE CodeWarrior da a los usuarios la habilidad de desarrollar código sobre el simulador y el *hardware* al mismo tiempo y también la flexibilidad para migrar proyectos de un procesador a otro dentro de una familia para soportar nuevas generaciones de productos.

Las presiones de los tiempos para el mercadeo no permiten a los desarrolladores de *software* la opción de esperar el trabajo de *hardware* final antes de evaluar el código de la aplicación. El IDE CodeWarrior ofrece un completo ambiente funcional simulado para carga, ejecución y depuración de aplicaciones. Esto incluye acceso a la memoria, configuración de puntos de ruptura, y adelantarse un paso en la ejecución del código.

La ventana de detalle de registros, una característica del depurador de CodeWarrior, ayuda al desarrollador a plantear el modelo de programación de cualquier registro disponible con una explicación detallada de cada bit o grupo de bits, su estado actual, y el significado de este estado. Para los periféricos personalizados, es posible crear un archivo de registro describiendo el contenido del registro.

El IDE CodeWarrior proporciona plantillas de escritorio, las plantillas contienen código que manipulan rutinas de inicialización genéricas para los diferentes módulos de periféricos y memorias de los procesadores.

El código ofrecido en el SDK ha sido desarrollado con, y es completamente compatible con, el conjunto de herramientas del IDE CodeWarrior.

Es por ello que con estas características el IDE CodeWarrior es la mejor herramienta para el desarrollo de aplicaciones con el DSP de Motorola y la interfaz de comunicación entre la PC y el DSP.

APÉNDICE C - CÓDIGO DEL KTR-DSP

Código del Archivo KTR.ASM

```
       SECTION      rtlib


       GLOBAL FSOTarStkIni
       GLOBAL       FOSCtxSw
       GLOBAL FSOCmbCtx
       GLOBAL FSOIntCmbCtx
       GLOBAL FSOIniTarLst
       GLOBAL       FSOTicISR
       GLOBAL  FtimerTick

       XREF   FSOTicSrv
       XREF   FSOIntSal

       XREF   FSOIntAnid
       XREF   FSOIntTmpSrv
       XREF   FSOCorriendo
       XREF   FSOBCTAct
       XREF   FSOBCTColLst
       XREF   FPE_state
       XREF   DispatchRestore

;
; Definir SUPPORT_NESTED_INTERRUPTS a 1 o 0,
; para soportar interrupciones anidadas o no
;

       define SUPPORT_NESTED_INTERRUPTS '1'

       define IPR 'x:$FFFB'

;*****************************************************************
;FSOTarStkIni, rutina que inicializa el stack de una tarea
;*****************************************************************

; SO_STK *SOTarStkIni (void (*Tarea)(void *dato), void *pdato, void *ptp)
;{
       ; Esta rutina esta basada en la implementada en el SDK para ser usada con
el KTR uC-OS II
       ; En el dsp los registros R2 y R3 son usadas para pasar punteros a
funciones, y los
       ; valores de direcciones de 16 bits se regresan en el registro R2

       ; Parametros de entrada:
       ;             r2                  =>     Tarea
       ;             r3                  =>     pdato
       ;             x:(SP-2)     =>  ptp

       ; Registros usados durante la ejecucion:

       ;             y0     => temp
       ;             y1     => SR simulado (constante 0x0100)
       ;             r2     => Tarea
       ;             r3  => pdato
       ;             r1     => ptp
       ;             x0     => constante temp 0
```

```
FSOTarStkIni:

       move   x:(SP-2),R1               ; Carga ptp
       nop

       lea           (R1)+
       move   R2,x:(R1)+                ; Direccion de retorno

       move   #$0100,Y1
       move   Y1,x:(R1)+                ; Crea un nuevo SR con interrupciones
activadas


       clr    X0
       rep           #3
       move   X0,x:(R1)+                ; Limpia los registros N, X0, Y0

       move   R3,x:(R1)+                ; Fija R2 para apuntar a pdato (paramentro
de la tarea)


       if SUPPORT_NESTED_INTERRUPTS==1
              move   IPR,Y0                     ; Salva IPR
              move   Y0,x:(R1)+

       endif


       rep           #9
       move   X0,x:(R1)+                ; Limpia los registros A0, A1, A2, B0, B1,
B2, Y1, R0, R1

       move   r3,x:(R1)+                ; Fija R3

       move   omr,x:(R1)+               ; Fija el registro omr
       move   la,x:(R1)+                ; Fija el registro la
       move   m01,x:(R1)+               ; Fija el registro m01
       move   lc,x:(R1)+                ; Fija el registro lc

       ;
       ;      Salva un hardware stack simulado
       ;
       move   X0,x:(R1)+
       move   X0,x:(R1)+

       ;
       ;      Salva registros temporales de archivo del 0x30 - 0x37 usados por el
compilador
       ;
       rep    #8
       move   X0,x:(R1)+                ; Salva los registros de archivo

       move   #DispatchRestore,R2 ; Crea un jsr para regresar a DispatchRestore
       move   R2,x:(R1)+                ; PC de DispatchRestore
       move   Y1,x:(R1)+                ; SR de DispatchRestore

       ;
       ;      Salva registros permanentes de archivo del 0x38 - 0x3F usados por el
compilador
       ;
       rep    #8
```

```
       move   X0,x:(R1)+                ; Salva los registros de archivo

       move   X0,x:(R1)                 ; Salva FPE_state

       move   R1,R2                     ; regresa ((SO_STK *)stk)
       rts

;}




       PAGE
;***************************************************************************
;FSOIniTarLst: Rutina que carga la tarea que esta en el tope de la lista
;                      de listos, pone la variable SOCorriendo = VERDAD, e inicia
;                      la ejecucion de la tarea.
;***************************************************************************
FSOIniTarLst:

       move   x:FSOBCTColLst,R2               ; R2 = SOBCTColLst

       incw   x:FSOCorriendo                          ; Indica el inicio de la
multitarea

       move   x:(R2),SP                               ; SP = SOBCTColLst
       ;move  R2,SP                                   ; Se pasa R2 porque apunta al
tope
                                                              ; de la pila de
la tarea a iniciar

       bra           ReiniciaTar                           ; Ejecuta la tarea







       PAGE
;***************************************************************************
;FSOCmbCtx: Rutina que salva los registros de la tarea en ejecucion,
;                     asigna la tarea de mayor prioridad, a la tarea en ejecucion
;                     luego hace un rts para llamar la rutina de restauracion de
;                     registros de dicha tarea.
;                     Esta rutina es invocada como si fuese una RSI
;***************************************************************************
FOSCtxSw:
FSOIntCmbCtx:
FSOCmbCtx:

                     lea    (sp)+

                     ;
                     ; Salva registros de archivo permanentes en 0x38 - 0x3F usados
por
                     ; el compilador
                     ;

                     move   #$0038,R2
                     do             #8,FinSlvRA
                     move   x:(R2)+,Y1
                     move   Y1,x:(SP)+
```

```
FinSlvRA:

                    move   x:FSOBCTAct,R2            ; Pre carga SOBCTAct para el
kernel

                    ;
                    ; Salva el estado de emulacion del punto flotante
                    ;

                    move   x:FPE_state,Y1
                    move   Y1,x:(SP)

                    ;

                    ; Logica del kernel
                    ;

                    move   SP,x:(R2)                 ; Salva el tope del stack de
la tarea actual


                    move   x:FSOBCTColLst,R2   ; SOBCTAct = SOBCTColLst
                    move   R2,x:FSOBCTAct

                    move   x:(R2),SP                        ; SP = SOBCTColLst
(Direccion del SP de la tarea)
                                                                ; R2 contiene el
apuntador al tope de la pila
                                                                ; de la tarea a
ejecutarse.

ReiniciaTar:                                        ; Regresa a la nueva tarea

                    pop          y1                             ; Restaura la
emulacion del punto flotante

                    move   y1,x:FPE_state


                    move   #$003F,R2                 ; Restaura registros de
archivo
                    do           #8,FinRstRA               ; (0x38 - 0x3F)
                    pop          y1
                    move   y1,x:(R2)-

FinRstRA:

                    rts                                          ; Resgresa al
invocador (Disparcher o kernel)


      PAGE
;*********************************************************************
;FSOTicISR: rutina que se invoca en cada tic del temporizador
;
;*********************************************************************
FtimerTick:
FSOTicISR:

                    tstw   x:FSOCorriendo            ; Pregunta si ¿SO esta
inicializado?
                    beq    FinSOTicISR
```

```
                    incw   x:FSOIntAnid        ; Incrementa las interrupciones
anidadas

       jsr          FSOTicSrv                 ; Procesa el tic del sistema

       jsr          FSOIntSal                 ; Notifica al KTR el final de la
ISR



FinSOTicISR:

                    rts                                          ; Regresa a la
tarea interrumpida




       ENDSEC

       END
```

Código del Archivo KTR.C

```
#define SO_GLOBALES

#include "inc_maestro.h"


/************************** PRIMITIVAS DEL KERNEL
*************************************/

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOIntEnt (). Rutina que indica al Kernel la entrada de una interrupcion */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
void SOIntEnt (void)
{
       SOIntDshbl();
       SOIntAnid++;
       SOIntHbl();
}

/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOIntSal (). Indica al kernel la salida de una interrupcion
       */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
void SOIntSal (void)
{

       SOIntDshbl();

       if (--SOIntAnid == 0) {

              /*Checa si ha sido puesta una tarea lista para ejecutarse por la
interrupcion y
                si es asi realiza el cambio de contexto.
```

```
              */
             if (SOLstPrmrPlz(&SOColTarLst) < SOArrTar[SOTarEje].SOBCTPlz) {


                    SOArrTar[SOTarEje].SOBCTEdo = LISTO;
                    SOLstInsertar (SOTarEje, &SOColTarLst);

                    SOTarEje = SOLstObtPrmr (&SOColTarLst);
                    SOArrTar[SOTarEje].SOBCTEdo = EJECUCION;
                    SOBCTColLst = &SOArrTar[SOTarEje];
                    SOIntCmbCtx();
             }



       }

       SOIntHbl();
}



/* Mecanismo de Planificacion */


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SODespacha (). Asigna el cpu a la primera tarea de la lista de tareas listas
        */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

//  void SODespacha ()    {}



/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOPlanificar (). Selecciona la tarea con el plazo mas corto                 */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

void SOPlanificar (void)
{

       SOIntDshbl();
             if (SOLstPrmrPlz(&SOColTarLst) < SOArrTar[SOTarEje].SOBCTPlz) {
                    SOArrTar[SOTarEje].SOBCTEdo = LISTO;
                    SOLstInsertar (SOTarEje, &SOColTarLst);
                    //SODespacha();
                    /*SODespacha(); */
                    /* Esta lineas se colocan en lugar de la funcion SODespacha */
                    /* Parametros: */
                    /*     SOTarEje -> Indice de la tarea a ejecutarse */
                    /*           SOBCTColLst -> Apuntador al tope de la pila de
dicha tarea */
                    /* Se invoca el macro que invoca una interrupcion que hace el
*/
                    /* cambio de contexto, el numero de la interrupcion es la 4
que es */
                    /* el manejador de la instruccion SWI. */
                    SOTarEje = SOLstObtPrmr (&SOColTarLst);
                    SOArrTar[SOTarEje].SOBCTEdo = EJECUCION;
```

```
                    SOBCTColLst = &SOArrTar[SOTarEje];
                    SOTARCAMBIO();
             }

      SOIntHbl();
}



/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOAbortar (int Error). */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
void SOAbortar (Int16 Error)
{}


/*-------------------------------------------------------------------------------
---------*/
/* SOIniciar - Inicializa la multitarea del Kernel
                                  */
/*-------------------------------------------------------------------------------
---------*/
void         SOIniciar (void)
{


      if (SOCorriendo == FALSO)
      {
            SOTarEje = SOLstObtPrmr(&SOColTarLst);

            SOBCTColLst = &SOArrTar[SOTarEje];
            SOBCTAct = SOBCTColLst;

            SOIniTarLst();
      }

}



/*-------------------------------------------------------------------------------
---------*/
/* SOTicSrv - Rutina de manejo de la interrupcion del Timer
        */
/*-------------------------------------------------------------------------------
---------*/
void SOTicSrv (void)
{
Int16  proceso;
BCT          *proc;
//Int16      cuenta = 0;


      SOIntDshbl();
      SOTmpSist++;
      SOIntHbl();

      if (SOTmpSist >= TMPVIDA)
            SOAbortar(TMP_EXPIRADO);

      if (SOBCTAct->SOBCTTipo == TARCRT)
```

```
            if (SOTmpSist > SOBCTAct->SOBCTPlz)
                  SOAbortar(TMP_SOBREPASADO);

      while ( !SOLstVacia(&SOColTarZmb) &&
                  (SOLstPrmrPlz(&SOColTarZmb) <= SOTmpSist) ) {

            SOIntDshbl();
            proceso = SOLstObtPrmr(&SOColTarZmb);
            proc = &SOArrTar[proceso];
            SOFctUtil = SOFctUtil - proc->SOBCTFctUtil;
            proc->SOBCTEdo = LIBRE;
            SOLstInsertar(proceso, &SOColBCTLbr);
            SOIntHbl();

      }

      /* Checa la lista de tareas en espera de q pasen ciertos tics */
      if (!SOLstVacia(&SOColTarEsp))
      {

            SOIntDshbl();

            proceso = SOColTarEsp;
            proc = &SOArrTar[proceso];

            while (proceso != NULO)
            {

                  if (--proc->SOBCTEsp == 0)
                  {
                        SOLstExtraer(proceso,&SOColTarEsp);
                        proc->SOBCTEdo = LISTO;
                        SOLstInsertar(proceso,&SOColTarLst);
                  }
                  proceso = proc->SOBCTSig;
                  proc = &SOArrTar[proceso];
            }
            SOIntHbl();
      }


      /* Checa la lista de tareas ociosas para ver si ya una inicia su periodo */

      while ( !SOLstVacia(&SOColTarOcs) &&
                  (SOLstPrmrPlz(&SOColTarOcs) <= SOTmpSist) ) {

            SOIntDshbl();
            proceso = SOLstObtPrmr (&SOColTarOcs);
            SOArrTar[proceso].SOBCTPlz += (UInt32) SOArrTar[proceso].SOBCTPer;
            SOArrTar[proceso].SOBCTEdo = LISTO;
            SOLstInsertar(proceso, &SOColTarLst);
            //cuenta++;
            SOIntHbl();

      }

// Estas lineas ya no son necesarias porq despues de esta rutina se invoca la
// rutina SOIntSal() que checa si hay tareas listas para ejecutarse despues de
// la interrupcion
//    if (cuenta > 0)
//          SOPlanificar();
```

```
}



/*---------------------------------------------------------------------------
---------*/
/* SOIniSist - Inicializa las estructuras y variables para el Kernel
                    */
/*---------------------------------------------------------------------------
---------*/

//void SOIniSist (UInt32 tic)
void SOIniSist ()
{


Int16 indice;


       /* Inicializacion de variables para el control del Kernel */

       SOTmpSist =   0L;                                       // Limpia el tic del
sistema
       SOIntAnid =   0;                                        // Limpia el indicador
de int. anidadas
       SOCorriendo = FALSO;                             // Indica q la multitarea no
ha iniciado
       SOContTar =    0;                                // Limpia el contador de
tareas
       SOContOcio =  0L;                                // Limpia el contador de ocio
       SOContCmbCtx= 0L;                                // Limpia el contador de
cambios de contexto

#if SO_TAR_ESTD_AC
       SOContOcioEje = 0L;
       SOContOcioMax = 0L;
       SOEstdLista =       FALSO;                       // Tarea estadistica no esta
lista
#endif


       /* Inicializar la tabla del vector de interrupciones */
       /* Esto se hace automaticamente al usar la biblioteca SDK */

       /* Inicializar las listas de BCT's libres y semaforos */
       /* El proceso que inicia el kernel sera asignado al bct = MAXPROC */
       /* por esta razon el numero de bct's sera igual a MAXPROC + 1 */
       /* no obstante esto puede no ser valido para los semaforos si asi */
       /* se desea. */
       for (indice = 0; indice < (SO_MAX_TARS + SOTARAUX); indice++)
       {
              SOArrTar[indice].SOBCTSig = indice + 1;
       }
       SOArrTar[SO_MAX_TARS].SOBCTSig = NULO;

#if SO_SEM_AC > 0
       for (indice = 0; indice < (SO_MAX_TARS + SOTARAUX); indice++)
       {
              SOArrSem[indice].SOBCSSig = indice + 1;
       }
       SOArrSem[SO_MAX_TARS-1].SOBCSSig = NULO;
#endif
```

```
       SOColTarLst = NULO;
       SOColTarOcs  = NULO;
       SOColTarZmb  = NULO;

       SOColTarEsp = NULO;

       SOColBCTLbr  = 0;
       SOColSemLbr  = 0;
       SOFctUtil    = 0;

       SOTarEje = SOTarCrear(SOTarOcio, (void *)0, &TarOcStk[0], TARNTR,
SO_PRIO_MASBAJA, 0L );

#if SO_TAR_ESTD_AC > 0
       SOTarEje = SOTarCrear(SOTarEstd, (void *)0, &TarEstdStk[0], TARNTR,
SO_PRIO_MASBAJA-1, 0L );
#endif

       /* Puntero al stack del BCT Actual que sera supendedido */
       SOBCTAct = &SOArrTar[SOTarEje];

       /* Puntero al stack del BCT que sera puesto en ejecucion */
       SOBCTColLst = SOBCTAct;

}


/*-------------------------------------------------------------------------
---------*/
/* SOIniEstd - Inicializa los datos para usar la tarea de estadistica
                    */
/*-------------------------------------------------------------------------
---------*/
#if SO_TAR_ESTD_AC > 0
void SOIniEstd (void)
 {
       SOTmpEsp(2);                      /* Sincroniza con el reloj */

       SOIntDshbl();
       SOContOcio   = 0L;                /* Limpia el contador de ocio */
       SOIntHbl();

       SOTmpEsp(SOTics_X_Seg);           /* Determina el Max. valor de cont. Ocioso
en 1 seg. */

       SOIntDshbl();
       SOContOcioMax = SOContOcio;       /* Almacena la cuenta maxima del contador
ocioso en 1 seg */
       SOEstdLista = VERDAD;
       SOIntHbl();
 }
#endif
```

Código del Archivo KTR.H

```
#ifdef   SO_GLOBALES

#define  SO_EXT
```

```
#else
#define  SO_EXT extern
#endif



/* Constantes Globales */


#define MAXPLAZO    0x7FFFFFFF   /* Maximo plazo (deadline) */

#define NULO        -1           /* puntero nulo */
#define VERDAD            1
#define FALSO       0

#define TMPVIDA           MAXPLAZO - SO_PRIO_MASBAJA /*Tiempo de vida del
sistema*/

#if SO_TAR_ESTD_AC              /* Tareas auxiliares para el manejo del kernel */
#define SOTARAUX    2
#else
#define SOTARAUX    1
#endif

#define TAMSTACK    128          /* Tamaño del Stack para las tareas */

/* Tipos de tareas */

#define TARCRT            1           /* tarea critica */
#define TARNTR            2           /* tarea que no es de tiempo real */

/* Estados de las tareas */
#define LIBRE             0           /* BCT no alojado */
#define LISTO       1           /* Estado de listo */
#define EJECUCION   2           /* Estado de ejecucion */
#define DORMIDO           3           /* Estado de dormido */
#define OCIO        4           /* Estado de ocio */
#define ESPERA            5           /* Estado de espera */
#define ESPERA_SEM  6           /* Estado de espera en un semaforo */
#define ZOMBIE            7           /* Estado zombie */


/* Mensajes de error */
#define OK                      0           /* No hay error */
#define      TMP_SOBREPASADO    1           /* Plazo perdido */
#define TMP_EXPIRADO      2           /* Tiempo de vida alcanzado */
#define NO_GARANTIA       3           /* tarea no se puede planificar */
#define NO_BCT                  4           /* Demasiadas tareas */
#define NO_SEM                  5           /* Demasiados semaforos */
#define NO_TMP_ESP        6           /* No hay tiempo de espera */


/* Datos definidos */

typedef int               Int16;
typedef unsigned int      UInt16;
typedef long              Int32;
typedef unsigned long     UInt32;
typedef float             Frac;


typedef Int16cola;              /* Indice de la cola */
typedef Int16       sem;                /* Indice del semaforo */
```

```
typedef Int16proc;                /* Indice del proceso */
typedef Int16       bac;                  /* Indice del buffer BAC */
typedef char*puntero;             /* Puntero a memoria */
typedef Int16SO_STK;                      /* Tipo de dato para la pila */


/* Macros */
#define SOTARCAMBIO()     asm(swi)

/* Declaracion de la estructura de los BCT's
   especificos para el DSP 56F827 con el uso
   del CodeWarrior */

typedef struct bct {
       SO_STK *SOBCTStkTar;       /* Puntero al Tope de la Pila */
       Int16  SOBCTSig;           /* Indice del sig. bct */
       Int16  SOBCTAnt;           /* Indice del ant. bct */

       UInt32 SOBCTPlz;           /* Plazo (deadline) de la tarea */

       Int16  SOBCTEdo;           /* Estado actual de la tarea */
       Int16  SOBCTPrio;          /* Prioridad de la tarea */
       UInt32 SOBCTEsp;           /* Tiempo de espera de la tarea */
       Int16  SOBCTPer;           /* Periodo de la tarea */
       Int16  SOBCTTipo;          /* Tipo de la tarea */
       Int16  SOBCTTmpEje;        /* Tiempo de ejecucion de la tarea */
       Frac   SOBCTFctUtil;       /* Factor de utilizacion del cpu por la tarea */
}BCT;


/* Estructura para el control de los semaforos */
#if SO_SEM_AC > 0
typedef struct bcs {
       Int16  SOBCSCont;          /* Contador del semaforo */
       cola   SOBCSColSem;        /* Cola del semaforo */
       sem           SOBCSSig;           /* Indice al sig. sem. */
}BCS;
#endif

/* Estructuras para el control de los buffers asincronos ciclicos */
typedef struct bufbac {
       Int16  SOBACUso;
       struct bufbac*SOBACSig;
       void   *SOBACDato;
}BUFBAC;

typedef struct bac {
       Int16  SOBACMaxBuf;
       Int16  SOBACDimBuf;
       BUFBAC        *SOBACLibres;
       BUFBAC *SOBACBmr;
}BAC;




/* Declaracion de los arreglos de bloques de control */


SO_EXT BCT SOArrTar[SO_MAX_TARS + SOTARAUX];  /* Arreglo de bct's */

#if SO_SEM_AC > 0
SO_EXT BCS SOArrSem[SO_MAX_TARS + SOTARAUX];        /* Arreglo de bcs's */
```

```
#endif

/* Variables para el uso del Kernel */
SO_EXT  Int16       SOCorriendo;
SO_EXT  Int16SOIntAnid;
SO_EXT  Int16SOContTar;
SO_EXT  UInt32      SOContOcio;
SO_EXT  UInt32      SOContCmbCtx;

#if SO_TAR_ESTD_AC
SO_EXT  Int16SOCPUUso;                          /* Porcentaje usado del CPU */
SO_EXT  UInt32      SOContOcioMax;                  /* Maximo valor que el cont.
ocioso puede tomar en 1 seg */
SO_EXT  UInt32      SOContOcioEje;                  /* Valor alcanzado por el
cont. ocioso en eje. durante 1 seg */
SO_EXT  Int16SOEstdLista;               /* Bandera para indicar q la tarea
estadistica esta lista */
#endif

/* Declaracion de variables a usar */

SO_EXT  proc SOTarEje;                  /* Tarea en ejecucion */
SO_EXT       cola   SOColTarLst;            /* cola de tareas listas */
SO_EXT       cola   SOColTarOcs;            /* cola de tareas ociosas */
SO_EXT       cola   SOColTarZmb;            /* cola de tareas zombie */

SO_EXT       cola   SOColTarEsp;            /* cola de tareas en espera de tics
*/

SO_EXT       cola   SOColBCTLbr;            /* cola de bct's libres */
SO_EXT       cola   SOColSemLbr;            /* cola de bcs's libreas */
SO_EXT       Frac   SOFctUtil;              /* Factor de utilizacion global */



/* Variables Globales para el manejo de las tareas */
SO_EXT  BCT  *SOBCTAct;
SO_EXT       BCT    *SOBCTColLst;
SO_EXT  SO_STK      TarOcStk[SO_TAR_OCIO_TAMSTK];

#if SO_TAR_ESTD_AC
SO_EXT  SO_STK      TarEstdStk[SO_TAR_ESTD_TAMSTK];
#endif

/* Variables para el manejo del tiempo */

SO_EXT       UInt32 SOTmpSist;              /* Tiempo del sistema (numero de
                                    interrupciones generadas desde el
                                    inicio del sistema)*/
SO_EXT       UInt32 SOTmpTic;               /* unidad de tiempo (ms) "Tic" */


#define SOTics_X_Seg            1000


/********** Prototipos de Funciones ****************************/

/*************************************************************************/
/*                        Funciones miscelaneas del KTR          */
/*************************************************************************/
SO_EXT void SOIntHbl (void);
SO_EXT void SOIntDshbl (void);
```

```
 void SOIntEnt (void);
 void SOIntSal (void);

// void      SOIniSist (UInt32 tic);
 void  SOIniSist (void);
 void  SOIniciar (void);

 void  SOIniEstd (void);


/*$PAGE*/
/************************************************************************/
/*                          Funciones para el manejo de listas del KTR          */
/************************************************************************/

 Int16       SOLstExtraer (Int16 indice, cola *cola_ext);
 void  SOLstInsertar (Int16 indice, cola *cola_ins);
 Int16       SOLstObtPrmr (cola *cola_ext);
 UInt32 SOLstPrmrPlz (cola *cola_chq);
 Int16       SOLstVacia (cola *cola_chq);

 void  SOLstInsEsp (Int16 indice, cola *cola_ins);
 Int16       SOLstPrmrEsp (cola *cola_chq);



/************************************************************************/
/*                          Funciones internas para uso del KTR          */
/************************************************************************/
 void  SOPlanificar (void);
 void  SODespacha(void);
 void  SOAbortar (Int16 Error);



/************************************************************************/
/*                          Funciones para el manejo del tiempo          */
/************************************************************************/

 void  SOTicSrv (void);

 void  SOTmpEsp (UInt16 tics);
 Int16       SOTmpEspRndr (Int16 proceso);


/************************************************************************/
/*                          Funciones para el manejo de las tareas        */
/************************************************************************/

 Int16       SOTarGarantia (Int16 proceso);


 Int16       SOTarCrear( void (*SOTarDireccion)(void *pd), void *pdato, SO_STK
*ptp,
                                Int16 SOTarTipo, Int32 SOTarPer, Int32
SOTarTmpEje);

 void  SOTarDormir (void);
 void  SOTarTermCcl (void);
 void  SOTarTermProc (void);
 void  SOTarFinalizar (Int16 proceso);

 void  SOTarOcio (void);
```

```
#if SO_TAR_ESTD_AC
 void  SOTarEstd (void *dato);
#endif


/*****************************************************************************/
/*                          Funciones para el manejo de los semaforos           */
/*****************************************************************************/

 sem   SOSemCrear (Int16 valor);
 void  SOSemFinalizar (sem semaforo);
 void  SOSemAdquirir(sem semaforo);
 void  SOSemLiberar(sem semaforo);


/*****************************************************************************/
/*           Funciones para el manejo de los buffers asincronos ciclicos
      */
/*****************************************************************************/

void         SOBacCrear ( BAC *nom_bac,BUFBAC *lst_bacs, Int16 tam_msj, Int16
num_buf);
struct BUFBAC       *SOBacReservar ( BAC *b);
void         SOBacPonerMsj ( BAC *b,  BUFBAC *bbac);
struct BUFBAC       *SOBacObtMsj ( BAC *b);
void         SOBacLiberar ( BAC *b,  BUFBAC *bbac);




/***************** FUNCIONES EXTERNAS ***********************************/
/*Funciones complementarias programadas en ensamblador, especificas del DSP */
/*****************************************************************************/
  void       SOIniTarLst(void);
  SO_STK     *SOTarStkIni (void (*Tarea)(void *dato), void *pdato, void *ptp);
  void              SOIntCmbCtx (void);
  void              SOCmbCtx (void);
  void              SOTicISR (void);


/*****************************************************************************/
/*Funciones para retornar info sobre algunas variables del Kernel */
/*****************************************************************************/
  UInt32 SOInfoTmp(void);


/*************** MACROS DE FUNCIONES ***********************************/

/* void SOIntHbl (void); */
#define SOIntHbl() asm(bfset #0x0100,SR); asm(bfclr #0x0200,SR)

/* void SOIntDshbl (void); */
#define SOIntDshbl() asm(bfset #0x0300,SR)

#define SODespacha() {    SOTarEje = SOLstObtPrmr (&SOColTarLst); \
                                    SOArrTar[SOTarEje].SOBCTEdo = EJECUCION; \
                                    SOBCTColLst = &SOArrTar[SOTarEje]; \
                                    SOTARCAMBIO();      }
```

Código del Archivo SO_BAC.C

```
#include "inc_maestro.h"



/****************** MANEJO DE LOS BUFFERS ASINCRONOS CICLICOS
*********************************/

/*-------------------------------------------------------------------------------
---------*/
/* SOBacCrear - aloja e inicializa el buffer asincrono ciclico
                     */
/*-------------------------------------------------------------------------------
---------*/
#if SO_BAC_AC > 0
void SOBacCrear (BAC *nom_bac,BUFBAC *lst_bacs, Int16 tam_msj, Int16 num_buf)
{
       Int16  i;

       for (i=0; i < num_buf-1; i++)
       {
              lst_bacs[i].SOBACSig = &lst_bacs[i+1];
              lst_bacs[i].SOBACUso = 0;
       }

       lst_bacs[num_buf-1].SOBACSig = (BUFBAC *) 0;
       lst_bacs[num_buf-1].SOBACUso = 0;

       nom_bac->SOBACMaxBuf = num_buf;
       nom_bac->SOBACDimBuf = tam_msj;
       nom_bac->SOBACLibres = lst_bacs;
       nom_bac->SOBACBmr = (void *) 0;


}
#endif


/*-------------------------------------------------------------------------------
---------*/
/* SOBacReservar - reserva el buffer en el BAC, para poder escribir un mens.
                                    */
/*-------------------------------------------------------------------------------
---------*/
#if SO_BAC_AC > 0
struct BUFBAC       *SOBacReservar ( BAC *b)
{
       BUFBAC *bufbac;

       SOIntDshbl();

       bufbac = b->SOBACLibres;                              // obtiene un buffer
libre
       b->SOBACLibres = bufbac->SOBACSig;                                    //
actualiza la lista de buffers libres

       SOIntHbl();

       return (bufbac);
}
#endif
```

```
/*-------------------------------------------------------------------------
---------*/
/* SOBacPonerMsj - pone el mensaje en el BAC
*/
/*-------------------------------------------------------------------------
---------*/
#if SO_BAC_AC > 0
void          SOBacPonerMsj ( BAC *b,  BUFBAC *bbac)
{
       SOIntDshbl();

       if (b->SOBACBmr->SOBACUso == 0)               // si no es accesado
       {
              b->SOBACBmr->SOBACSig = b->SOBACLibres; // desaloja el buffer mas
reciente
              b->SOBACLibres = b->SOBACBmr;
       }

       b->SOBACBmr = bbac;

       SOIntHbl();
}
#endif


/*-------------------------------------------------------------------------
---------*/
/* SOBacObtMsj - obtiene el apuntador para el buffer mas reciente
                      */
/*-------------------------------------------------------------------------
---------*/
#if SO_BAC_AC > 0
struct BUFBAC       *SOBacObtMsj ( BAC *b)
{
       BUFBAC       *bufbac;

       SOIntDshbl();

       bufbac = b->SOBACBmr;                                  // Obtiene el
apuntador del buffer mas reciente
       bufbac->SOBACUso =bufbac->SOBACUso + 1; // incrementa el contador

       SOIntHbl();

       return(bufbac);
}
#endif


/*-------------------------------------------------------------------------
---------*/
/* SOBacLiberar - libera el buffer solo si no es accesado y no es el buffer
               */
/*                            mas reciente.
                                                              */
/*-------------------------------------------------------------------------
---------*/
#if SO_BAC_AC > 0
void          SOBacLiberar ( BAC *b,  BUFBAC *bbac)
{
       SOIntDshbl();
```

```
	bbac->SOBACUso = bbac->SOBACUso - 1;

	if ( (bbac->SOBACUso == 0) && (bbac != b->SOBACBmr) )
	{
		bbac->SOBACSig = b->SOBACLibres;
		b->SOBACLibres = bbac;
	}

	SOIntHbl();
}
#endif
```

Código del Archivo SO_INFO.C

```
#include "inc_maestro.h"


/************************** Informacion de Estados
**************************************/

/*-------------------------------------------------------------------------------
---------*/
/* SOInfoTmp(); esta rutina devuelve el tiempo del sistema en milisegundos. */
/*-------------------------------------------------------------------------------
---------*/
UInt32 SOInfoTmp(void)
{
	return(SOTmpSist);
}



/*
SOInfoEdo(); esta rutina devuelve el estado de la tarea que es pasada como
parámetro.

SOInfoPlazo(); esta rutina devuelve el plazo de la tarea que se pasa como
parámetro.

SOInfoPeriodo(); de forma semejante a las anteriores rutinas, esta rutina
devuelve el periodo.
*/
```

Código del Archivo SO_LISTA.C

```
#include "inc_maestro.h"


/************************ Manejo de las listas de los BCT's
*****************************/

/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstInsertar (). Inserta una tarea en la cola, basada en su plazo
	*/
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

void SOLstInsertar (Int16 indice, cola *cola_ins)
{
UInt32 plazo;                    /* plazo de la tarea a ser insertada*/
Int16   pAnt;                    /* puntero al bct anterior*/
Int16   pSig;                    /* puntero al bct siguiente */

       pAnt = NULO;
       pSig = *cola_ins;
       plazo = SOArrTar[indice].SOBCTPlz;

       /* Encontrar el elemento antes del punto de insercion */
       while (( pSig != NULO) && (plazo >= SOArrTar[pSig].SOBCTPlz)) {
              pAnt = pSig;
              pSig = SOArrTar[pSig].SOBCTSig;
       }

       if (pAnt != NULO)
              SOArrTar[pAnt].SOBCTSig = indice;
       else
              *cola_ins = indice;

       if (pSig != NULO)
              SOArrTar[pSig].SOBCTAnt = indice;

       SOArrTar[indice].SOBCTSig = pSig;
       SOArrTar[indice].SOBCTAnt = pAnt;
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstExtraer (). Extrae una tarea de la cola
       */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

Int16 SOLstExtraer (Int16 indice, cola *cola_ext)
{
Int16 pAnt, pSig;                /* Punteros auxiliares */

       pAnt = SOArrTar[indice].SOBCTAnt;
       pSig = SOArrTar[indice].SOBCTSig;

       if (pAnt == NULO)         /* Primer elemento */
              *cola_ext = pSig;
       else
              SOArrTar[pAnt].SOBCTSig = SOArrTar[indice].SOBCTSig;
       if (pSig != NULO)
              SOArrTar[pSig].SOBCTAnt = SOArrTar[indice].SOBCTAnt;

       return (indice);

}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstObtPrmr (). Extrae la tarea de la cabeza de la cola
       */
```

```
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

Int16 SOLstObtPrmr (Int16 *cola_ext)
{
Int16   pSig;                     /* puntero al primer bct  */

       pSig = *cola_ext;

       if (pSig == NULO) return (NULO);

       *cola_ext = SOArrTar[pSig].SOBCTSig;
       SOArrTar[*cola_ext].SOBCTAnt = NULO;

       return (pSig);
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstPrmrPlz (). Regresa el plazo de la primera tarea de la cola
        */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

UInt32 SOLstPrmrPlz (cola *cola_chq)

{
       return (SOArrTar[*cola_chq].SOBCTPlz);
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstVacia (). regresa verdadero si la cola esta vacia                    */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

Int16  SOLstVacia (cola *cola_chq)
{
       if (*cola_chq == NULO)
              return (VERDAD);
       else
              return (FALSO);
}




/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstInsEsp (). Inserta a la lista de tareas en Espera, en forma ascendete */
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

void SOLstInsEsp (Int16 indice, cola *cola_ins)
{
UInt32  ticesp;                   /* tics a esperar por la tarea */
Int16   pAnt;                     /* puntero al bct anterior*/
Int16   pSig;                     /* puntero al bct siguiente */

       pAnt = NULO;
```

```
	pSig = *cola_ins;
	ticesp = SOArrTar[indice].SOBCTEsp;

	/* Encontrar el elemento antes del punto de insercion */
	while (( pSig != NULO) && (ticesp >= SOArrTar[pSig].SOBCTEsp)) {
		pAnt = pSig;
		pSig = SOArrTar[pSig].SOBCTSig;
	}

	if (pAnt != NULO)
		SOArrTar[pAnt].SOBCTSig = indice;
	else
		*cola_ins = indice;

	if (pSig != NULO)
		SOArrTar[pSig].SOBCTAnt = indice;

	SOArrTar[indice].SOBCTSig = pSig;
	SOArrTar[indice].SOBCTAnt = pAnt;
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/
/* SOLstPrmrEsp (). Retorna el valor de espera del elemento del tope de la lista
*/
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++*/

Int16 SOLstPrmrEsp (cola *cola_chq)
{
	return (SOArrTar[*cola_chq].SOBCTEsp);
}
```

Código del Archivo SO_SEM.C

```
#include "inc_maestro.h"



/************************** MANEJO DE LOS SEMAFOROS
*************************************/

/*-----------------------------------------------------------------------------
---------*/
/* SOSemCrear - aloja e inicializa un semaforo
	*/
/*-----------------------------------------------------------------------------
---------*/

#if SO_SEM_AC > 0
sem SOSemCrear (Int16 valor)
{
sem semaforo;

	SOIntDshbl();

	semaforo = SOColSemLbr;			/* primer indice de semaforo
libre */
```

```
	if (semaforo == NULO)
		SOAbortar(NO_SEM);


	SOColSemLbr = SOArrSem[semaforo].SOBCSSig;	/* actualiza la lista de sem
libres */
	SOArrSem[semaforo].SOBCSCont = valor;		/* inicializa el contador*/
	SOArrSem[semaforo].SOBCSColSem = NULO;		/* inicializa cola del sem */

	SOIntHbl();

	return (semaforo);
}
#endif


/*-------------------------------------------------------------------------------
---------*/
/* SOSemFinalizar - desaloja un semaforo
*/
/*-------------------------------------------------------------------------------
---------*/

#if SO_SEM_AC > 0
void SOSemFinalizar (sem semaforo)
{
	SOIntDshbl();

	SOArrSem[semaforo].SOBCSSig = SOColSemLbr;	/* inserta el sem a la cabeza
*/
	SOColSemLbr = semaforo;				/* de la lista de sem libres
*/

	SOIntHbl();
}
#endif


/*-------------------------------------------------------------------------------
---------*/
/* SOSemAdquirir - Espera por un evento
*/
/*-------------------------------------------------------------------------------
---------*/
#if SO_SEM_AC > 0
void SOSemAdquirir(sem semaforo)
{
	SOIntDshbl();

	if (SOArrSem[semaforo].SOBCSCont > 0)
		SOArrSem[semaforo].SOBCSCont--;
	else {

		SOArrTar[SOTarEje].SOBCTEdo = ESPERA_SEM;
		SOLstInsertar(SOTarEje, &SOArrSem[semaforo].SOBCSColSem);

		SODespacha();

	}

	SOIntHbl();
}
```

```
#endif



/*---------------------------------------------------------------------------
---------*/
/* SOSemLiberar - Avisa un evento
*/
/*---------------------------------------------------------------------------
---------*/
#if SO_SEM_AC > 0
void SOSemLiberar(sem semaforo)
{
Int16 proceso;

       SOIntDshbl();

       if (!SOLstVacia(&SOArrSem[semaforo].SOBCSColSem)) {
              proceso = SOLstObtPrmr(&SOArrSem[semaforo].SOBCSColSem);
              SOArrTar[proceso].SOBCTEdo = LISTO;
              SOLstInsertar(proceso, &SOColTarLst);

              SOPlanificar();

       }
       else
              SOArrSem[semaforo].SOBCSCont++;

       SOIntHbl();
}
#endif
```

Código del Archivo SO_TAR.C

```
#include "inc_maestro.h"



/***************************** MANEJO DE TAREAS
*************************************/

/*---------------------------------------------------------------------------
---------*/
/* SOTarGarantia - garantiza la planificabilidad de una tarea critica
        */
/*---------------------------------------------------------------------------
---------*/

Int16  SOTarGarantia (Int16 proceso)
{
       SOFctUtil = SOFctUtil + SOArrTar[proceso].SOBCTFctUtil;

       if (SOFctUtil > 1.0) {
              SOFctUtil = SOFctUtil - SOArrTar[proceso].SOBCTFctUtil;
              return (FALSO);
       }
       else
              return (VERDAD);
}
```

```
/*-------------------------------------------------------------------------
---------*/
/* SOTarStkIni - Inicializa el Stack de una tarea                          */
/*-------------------------------------------------------------------------
---------*/
//


/*-------------------------------------------------------------------------
---------*/
/* SOTarCrear - Crea una tarea y la pone en el estado DORMIDO
        */
/*-------------------------------------------------------------------------
---------*/

Int16  SOTarCrear( void (*SOTarDireccion)(void *pd), void *pdato, SO_STK *ptp,
                                Int16 SOTarTipo, Int32 SOTarPer, Int32
SOTarTmpEje)
{
Int16 proceso, err;
void *psp;                /*Puntero al SP*/

       SOIntDshbl();                    /* Deshabilitar las interrupciones */

       proceso = SOLstObtPrmr(&SOColBCTLbr);
       if (proceso == NULO)             /* Error si no hay BCT disponibles*/
       {
              err = NO_BCT;
              SOAbortar(err);
              SOIntHbl();                       /* Habilitar las Interrupciones */
              return (err);
       }

       /* Se inicializa el stack de la tarea */
       psp = (void *) SOTarStkIni(SOTarDireccion, pdato, ptp);

       SOArrTar[proceso].SOBCTStkTar = (SO_STK *) psp;

       /* Se continua con la inicializacion de los parametros de la tarea */
       SOArrTar[proceso].SOBCTTipo = SOTarTipo;
       SOArrTar[proceso].SOBCTEdo = LISTO;
       SOArrTar[proceso].SOBCTTmpEje =  SOTarTmpEje;

       if (SOTarTipo == TARCRT)
       {
              /* se actualizan los datos para garantizar la planificacion */
              SOArrTar[proceso].SOBCTPer =  SOTarPer;
              SOArrTar[proceso].SOBCTFctUtil = (Frac) SOTarTmpEje / SOTarPer;

              /* Se agrega el plazo para la tarea */
              SOArrTar[proceso].SOBCTPlz = SOTmpSist + (UInt32) SOTarPer;

              if (!SOTarGarantia(proceso))
              {
                     err = NO_GARANTIA;
                     //Regresamos el indice del BCT a la lista de BCT's Libres
                     SOLstInsertar(proceso, &SOColBCTLbr);
                     SOIntHbl();                       /* Habilitar las
Interrupciones */
                     return(err);
              }
       }
```

```
	if (SOTarTipo == TARNTR)
	{
		// Se suma la cantidad de 1000 para poder diferenciar las
prioridades
		// de las tareas de no tiempo real con las criticas.
		SOArrTar[proceso].SOBCTPrio = (Int16) ( SOTarPer + 1000);
		SOArrTar[proceso].SOBCTPlz = TMPVIDA + (UInt32) SOTarPer;

	}

	/* Insercion del BCT de la tarea a la lista de Tareas Listas*/
	SOLstInsertar(proceso, &SOColTarLst);

	/* Invocacion del Planificador para escoger la tarea en ejecucion */
	if (SOCorriendo)
	{
		SOPlanificar();
	}

	SOIntHbl();			/* Habilitar las Interrupciones */

	return (proceso);
}


/*-------------------------------------------------------------------------
---------*/
/* SOTarDormir - Suspende ella misma en el estado DORMIDO
*/
/*-------------------------------------------------------------------------
---------*/

void SOTarDormir (void)
{
	SOIntDshbl();

	SOArrTar[SOTarEje].SOBCTEdo = DORMIDO;
	//SODespacha();
	SOTarEje = SOLstObtPrmr (&SOColTarLst);
	SOArrTar[SOTarEje].SOBCTEdo = EJECUCION;
	SOBCTColLst = &SOArrTar[SOTarEje];
	SOTARCAMBIO();

	SOIntHbl();
}


/*-------------------------------------------------------------------------
---------*/
/* SOTarTermCcl - Inserta una tarea en la lista de tareas ociosas
*/
/*-------------------------------------------------------------------------
---------*/

void SOTarTermCcl (void)
{
UInt32 plazo;


	plazo = SOArrTar[SOTarEje].SOBCTPlz;
```

```
	SOIntDshbl();
	if (SOTmpSist < plazo) {
		SOArrTar[SOTarEje].SOBCTEdo = OCIO;
		SOLstInsertar(SOTarEje, &SOColTarOcs);
	}
	else {
		plazo = plazo + (UInt32) SOArrTar[SOTarEje].SOBCTPer;
		SOArrTar[SOTarEje].SOBCTPlz = plazo;
		SOArrTar[SOTarEje].SOBCTEdo = LISTO;
		SOLstInsertar(SOTarEje, &SOColTarLst);
	}

	// Realiza el despacho de la tarea q esta lista a ejecutarse

	//SODespacha();			/* Selecciona la sig tarea a ejecutarse */
	SOTarEje = SOLstObtPrmr (&SOColTarLst);
	SOArrTar[SOTarEje].SOBCTEdo = EJECUCION;
	SOBCTColLst = &SOArrTar[SOTarEje];
	SOTARCAMBIO();

	SOIntHbl();

}


/*---------------------------------------------------------------------------
---------*/
/* SOTarTermProc - Termina la tarea en ejecucion
*/
/*---------------------------------------------------------------------------
---------*/

void SOTarTermProc (void)
{
	SOIntDshbl();

	if (SOArrTar[SOTarEje].SOBCTTipo == TARCRT) {
		SOLstInsertar(SOTarEje, &SOColTarZmb);
	}
	else {
		SOArrTar[SOTarEje].SOBCTEdo = LIBRE;
		SOLstInsertar(SOTarEje, &SOColBCTLbr);
	}

	//SODespacha();
	SOTarEje = SOLstObtPrmr (&SOColTarLst);
	SOArrTar[SOTarEje].SOBCTEdo = EJECUCION;
	SOBCTColLst = &SOArrTar[SOTarEje];
	SOTARCAMBIO();


	SOIntHbl();			/* Se habilitan las interrupciones */
}


/*---------------------------------------------------------------------------
---------*/
/* SOTarFinalizar - Termina una tarea
*/
/*---------------------------------------------------------------------------
---------*/

void SOTarFinalizar (Int16 proceso)
```

```
{
       SOIntDshbl();

       if (SOTarEje == proceso) {
              SOTarTermProc();
              return;
       }

       if (SOArrTar[proceso].SOBCTEdo == LISTO)
              SOLstExtraer(proceso, &SOColTarLst);
       if (SOArrTar[proceso].SOBCTEdo == OCIO)
              SOLstExtraer(proceso, &SOColTarOcs);
       if (SOArrTar[proceso].SOBCTTipo == TARCRT)
              SOLstInsertar(proceso, &SOColTarZmb);
       else {
              SOArrTar[proceso].SOBCTEdo = LIBRE;
              SOLstInsertar(proceso, &SOColBCTLbr);
       }

       SOIntHbl();
}



/*-------------------------------------------------------------------------------
---------*/
/* SOTarOcio - Toma el CPU cuando no hay tareas listas para ejecutarse
                                    */
/*-------------------------------------------------------------------------------
---------*/

void SOTarOcio (void)
{
       while (SOTmpSist < TMPVIDA)
       {
              SOIntDshbl();
              SOContOcio++;
              SOIntHbl();
       }

}


/*-------------------------------------------------------------------------------
---------*/
/* SOTarEstd - Realiza estadisticas de las tareas
         */
/*-------------------------------------------------------------------------------
---------*/

#if SO_TAR_ESTD_AC
void SOTarEstd(void *dato)
{

       UInt32 ejecucion;
       Int16  uso;

       dato = dato;                                    /* Previene un warning del
compilador por no usarla */

       while (SOEstdLista == FALSO) {
              SOTmpEsp(2 * SOTics_X_Seg);             /* Espera hasta q la tarea
este lista*/
```

```
	}

	while (1) {
		SOIntDshbl();

		SOContOcioEje = SOContOcio;		/* Obtiene el contador de
ocio del ultimo segundo*/
		ejecucion	= SOContOcio;
		SOContOcio	= 0L;				/* Reinicia el
contador de ocio para el siguiente segundo */

		SOIntHbl();

		if (SOContOcioMax > 0L) {

			uso = (Int16) (100L - 100L * ejecucion / SOContOcioMax);

			if (uso > 100){
				SOCPUUso = 100;
			} else if (uso < 0) {
				SOCPUUso = 0;
			} else {
				SOCPUUso = uso;
			}
		} else {
			SOCPUUso = 0;
		}

		SOTmpEsp(SOTics_X_Seg);		/* Acumula SOContOcio para el
sig. segundo */
	}
}
#endif
```

Código del Archivo SO_TIEMPO.C

```
#include "inc_maestro.h"


/********************** Manejo del Tiempo
*********************************************/

/*****************************************************************************
************************/
/*  SOTmpEsp, Coloca la tarea que la invoca, en una lista de espera, y se reanuda
despues del numero     */
/*            de Tics indicado.
                                                                          */
/*****************************************************************************
************************/

void   SOTmpEsp (UInt16 tics)
{
	if (tics > 0){

		SOIntDshbl();                                                    //
Deshabilita las interrupciones
```

```
            SOArrTar[SOTarEje].SOBCTEdo = ESPERA;               // Cambia el
edo. de la tarea
            SOArrTar[SOTarEje].SOBCTEsp = (UInt32) tics;

            SOLstInsEsp (SOTarEje, &SOColTarEsp);               // Inserta la
tarea en la lista usando la funcion propia

            SOIntHbl();
      // Habilita interrupciones

            SODespacha();

      }

}


/*************************************************************************
************************/
/*  SOTmpEspRndr, Reanuda una tarea que este en espera de tics    */
/*************************************************************************
************************/

Int16  SOTmpEspRndr (Int16 proceso)
{

      SOIntDshbl();                          // Deshabilita las interrupciones

      if (proceso != NULO)
      {
            if ( (SOArrTar[proceso].SOBCTEdo==ESPERA) &&
                        (SOArrTar[proceso].SOBCTEsp != 0 ) )
            {

                  proceso = SOLstExtraer (proceso, &SOColTarEsp);           //
Extrae la tarea de la lista



                  SOArrTar[proceso].SOBCTEdo = LISTO;
      // Cambia el edo. de la tarea
                  SOArrTar[proceso].SOBCTEsp = 0;
      // Limpia el tiempo de espera

                  SOLstInsertar(proceso, &SOColTarLst);                     //
Inserta la tarea a la lista de tar. listas

                  SOIntHbl();
                  // Habilita las interrupciones

                  SOPlanificar();
                  // Invoca el planificador

                  return (OK);
            // Regreso q No error
            }
      }
      else {
            SOIntHbl();                     // Habilita las interrupciones
            return (NO_TMP_ESP);       // Regresa el error
```

```
      }



}

/*
*************************************************************************************
************************
Esta rutina se incluye por compatibilidad con el SDK
*************************************************************************************
************************
*/
```

```
void timerSleep(Word32 Ticks);

void timerSleep(Word32 Ticks)
{
      while (Ticks > 32767) {
                  SOTmpEsp (32767);
                  Ticks -= 32767;
            }
            SOTmpEsp (Ticks);
}
```

APÉNDICE D - CÓDIGO DE LAS APLICACIONES

Como nota cabe mencionar que todas las aplicaciones usaron el mismo código del archivo **inc_maestro.h**. Sólo se modificaron los archivos **main.c**, y **so_cnfg.h** en cada uno de los casos.

Código del Archivo INC_MAESTRO.H

```
/* SDK specific include files */

#include    "port.h"
#include    "mempx.h"

/* Application specific include files */
#include    "so_cnfg.h"

/* Processor independent include files */
#include    "ktr.h"
```

Código de los Archivos de la Aplicación 1

Archivo SO_CNFG.H para la Aplicación 1

```
/*
*********************************************************************************************************
*
                                          CONFIGURACION DEL KTR
*********************************************************************************************************
*/
#ifndef __SO_CNFG_H
#define __SO_CNFG_H


#ifndef SDK_LIBRARY
       #include "configdefines.h"

       #ifndef INCLUDE_UCOS
              #error INCLUDE_UCOS deberia estar definido en appconfig.h para
inicializar el SDK para el KTR
       #endif
#endif


#define SO_MAX_TARS              13    /* Max. numero de tareas en la aplicacion
...                  */
                                       /* ...debe ser >= 2
*/



#define SO_PRIO_MASBAJA          16    /* Define la prioridad mas baja que puede
ser asignada ...    */
                                       /* ... nunca debe ser mayor que 63!
*/

#define SO_TAR_OCIO_TAMSTK              128    /* Tamaño del stack de la tarea
ociosa (# entradas SO_STK ) */

#define SO_TAR_ESTD_AC                  1    /* Activa (1) o Desactiva(0) la tarea
de estadisticas          */
#define SO_TAR_ESTD_TAMSTK              128    /* Tamaño del stack de la tarea de
estadisticas          */


#define SO_SEM_AC                       1    /* Activa (1) o Desactiva(0) el uso
de semaforos          */
#define SO_BAC_AC                       0    /* Activa (1) o Desactiva(0) el uso
de semaforos          */
#endif
```

Archivo MAIN.C para la Aplicación 1

```
/* Archivo: Aplicacion1.c */

#include "inc_maestro.h"
```

```
#include "stdio.h"

#include "port.h"
#include "arch.h"


/*
*********************************************************************************
  CONSTANTES
*********************************************************************************
*/


#define N_TAREAS    10                                          // Numero de tareas
identicas


/*
*********************************************************************************
  VARIABLES
*********************************************************************************
*/

SO_STK       PilaTareas[N_TAREAS][TAMSTACK];         // Pilas de las tareas

SO_STK       PilaTarIni[TAMSTACK];

UInt32       TareaDato[N_TAREAS];                            // Parametro para
pasar a cada tarea



sem          semaforo;


/*
*********************************************************************************
  PROTOTIPOS DE FUNCIONES
*********************************************************************************
*/

void configFinalize (void);                                  /* Salida del Programa
SDK */

void Tarea(void *dato);
void TareaIni(void *dato);


/*
*********************************************************************************
  PRINCIPAL
*********************************************************************************
*/
void main (void)
{

       SOIniSist();

       semaforo = SOSemCrear(1);

       SOTarCrear(TareaIni,(void *)0, &PilaTarIni[0], TARNTR, 0L, 0L);

       SOIniciar();
```

```
}



/*
*********************************************************************************
  TAREA INICIAL
*********************************************************************************
*/
void TareaIni(void *dato)
{

UInt16       i;
Int16  MaxCPUUso = 0, Tiempo = 0;
UInt32 TiempoIni, TiempoFin;


       TiempoIni =  SOInfoTmp();                       // Tiempo del timer en ms.

       SOIniEstd();                                           // Inicializa las
estadisticas p/ KTR

       for (i=0; i < N_TAREAS; i++) {

              TareaDato[i] = '0' + i;
              SOTarCrear(Tarea, (void *)&TareaDato[i], &PilaTareas[i][0], TARNTR,
i+1 ,  0L );
       }


              while (1)
              {

                     SOTmpEsp(2);

                     if (SOCPUUso > MaxCPUUso)
                     {
                            MaxCPUUso = SOCPUUso;
                     }

                     for (i=0; i < N_TAREAS; i++)
                     {
                            if (TareaDato[i] < 10000)
                            {
                                   break;
                            }

                            TiempoFin =  SOInfoTmp();

                            Tiempo = (Int16) (TiempoFin - TiempoIni) /1000L ;    //
Este es tiempo que se tardan las tareas

                            // desde su creacion hasta la finalizacion

                            archDisableInt();

                            /* Todas las tareas estan hechas */

                            printf ("La maxima utilizacion del Procesador fue:
%d\%\n", MaxCPUUso);
```

```
                        printf ("El tiempo del procesamiento de las tareas fue
de %d seg.\n", Tiempo);

                        printf ("\nListo...\n");

                        while (1)
                        {
                               /* Salida del programa */
                               configFinalize();
                        }

                  }
            }


}



/*
*********************************************************************************
  TAREAS
*********************************************************************************
*/

void Tarea(void *dato)
{

UInt32 *pDato = (UInt32 *)dato;

       while (++(*pDato) < 10000) {

              SOSemAdquirir(semaforo);         // Adquiere el semaforo
              SOSemLiberar(semaforo);                 // Libera el semaforo

              SOTmpEsp(1);                            // Espera 1 tic del timer
       }


       SOTarTermProc();

}
```

Código de los Archivos de la Aplicación 2

Archivo SO_CNFG.H para la Aplicación 2

```
/*
*********************************************************************************************
************************
*                                          CONFIGURACION DEL KTR
*********************************************************************************************
************************
*/
#ifndef __SO_CNFG_H
#define __SO_CNFG_H


#ifndef SDK_LIBRARY
```

```
       #include "configdefines.h"

       #ifndef INCLUDE_UCOS
              #error INCLUDE_UCOS deberia estar definido en appconfig.h para
inicializar el SDK para el KTR
       #endif
#endif


#define SO_MAX_TARS              6    /* Max. numero de tareas en la aplicacion
...                 */
                                       /* ... debe ser >= 2
*/

#define SO_PRIO_MASBAJA          12    /* Define la prioridad mas baja que puede
ser asignada ...    */
                                       /* ... nunca debe ser mayor de  63!
*/

#define SO_TAR_OCIO_TAMSTK              128    /* Tamaño del stack de la tarea
ociosa (# entradas SO_STK ) */

#define SO_TAR_ESTD_AC                  0    /* Activa (1) o Desactiva(0) la tarea
de estadisticas         */
#define SO_TAR_ESTD_TAMSTK              128    /* Tamaño del stack de la tarea de
estadisticas         */

#define SO_SEM_AC                       0    /* Activa (1) o Desactiva(0) el uso
de semaforos         */
#define SO_BAC_AC                       0    /* Activa (1) o Desactiva(0) el uso
de semaforos         */
#endif
```

Archivo MAIN.C para la Aplicación 2

```
/* Archivo: Aplicacion2.c */

#include "inc_maestro.h"

#include "port.h"
#include "arch.h"
#include "bsp.h"
#include "led.h"

/*
*********************************************************************************
************************
*                                                 CONSTANTS
*********************************************************************************
************************
*/

#define  TAM_STK_TAR                  60       /* Size of each Tarea's stacks (#
of WORDs)            */
#define  NUM_TARS                      6       /* Number of identical tasks
*/

/*
*********************************************************************************
************************
```

```
*                                                       VARIABLES
*******************************************************************************
************************
*/

typedef struct {
      int  Led;
      int  Retardo;  /* in (1 ms) ticks */
} tDatoTar;

SO_STK           PilaTar[NUM_TARS][TAM_STK_TAR];     /* Tasks stacks
*/

tDatoTar        DatoTars[NUM_TARS]                    /* Parameters to pass to
each Tarea               */
                  = {
                           {
#ifdef LED_RED
                              LED_RED, 2000
#else
                                                          0, 0
#endif
                                    },
                           {
#ifdef LED_YELLOW
                              LED_YELLOW, 1000
#else
                                                          0, 0
#endif
                                    },
                           {
#ifdef LED_GREEN
                              LED_GREEN, 500
#else
                                                          0, 0
#endif
                                    },
                           {
#ifdef LED_RED2
                              LED_RED2, 1000
#else
                                                          0, 0
#endif
                                    },
                           {
#ifdef LED_YELLOW2
                              LED_YELLOW2, 500
#else
                                                          0, 0
#endif
                                    },
                           {
#ifdef LED_GREEN2
                              LED_GREEN2, 250
#else
                                                          0, 0
#endif
                                    }
                               };



/*
```

```
*********************************************************************************
************************
*                                          FUNCTION PROTOTYPES
*********************************************************************************
************************
*/

void   Tarea(void *data);                                /* Function prototypes of
tasks                  */
void   TareaPer(void *data);
/*
*********************************************************************************
************************
*                                                MAIN
*********************************************************************************
************************
*/

void main (void)
{
       int i, ret = 0;


    SOIniSist();                                                 /* Initialize
uC/OS-II                     */

       for (i=0; i<NUM_TARS/2; i++)
       {
              ret = DatoTars[i].Retardo;
       SOTarCrear(Tarea, (void *)&DatoTars[i], &PilaTar[i][0], TARNTR,
(SO_PRIO_MASBAJA - 3 - i), ret);
    }

    for (i=NUM_TARS/2; i<NUM_TARS; i++)
       {
              ret = DatoTars[i].Retardo;
       SOTarCrear(TareaPer, (void *)&DatoTars[i], &PilaTar[i][0], TARCRT, ret,
25);
    }

    SOIniciar();                                                /* Start
multitasking                       */
}

/*
*********************************************************************************
************************
*                                                TASKS
*********************************************************************************
************************
*/

void Tarea (void *data)
{
       int         LedFD     = -1;
    tDatoTar * pDatoTars = (tDatoTar *)data;

       if (pDatoTars -> Retardo > 0)
       {
              LedFD = ledOpen (BSP_DEVICE_NAME_LED_0, 0);
       }

       while (1)
```

```
	{
		if (LedFD != -1)
		{
			ledIoctl(LedFD,  LED_TOGGLE, pDatoTars -> Led,
BSP_DEVICE_NAME_LED_0);
		}

		SOTmpEsp(pDatoTars -> Retardo);
	}
}


void TareaPer (void *data)
{
	int         LedFD     = -1;
    tDatoTar * pDatoTars = (tDatoTar *)data;

	if (pDatoTars -> Retardo > 0)
	{
		LedFD = ledOpen (BSP_DEVICE_NAME_LED_0, 0);
	}

	while (1)
	{
		if (LedFD != -1)
		{
			ledIoctl(LedFD,  LED_TOGGLE, pDatoTars -> Led,
BSP_DEVICE_NAME_LED_0);
		}

		SOTarTermCcl();
	}

}
```

Código de los Archivos de la Aplicación 3

Archivo SO_CNFG.H para la Aplicación 3

```
/*
*********************************************************************************
************************
*                                            CONFIGURACION DEL KTR
*********************************************************************************
************************
*/
#ifndef __SO_CNFG_H
#define __SO_CNFG_H


#ifndef SDK_LIBRARY
	#include "configdefines.h"

	#ifndef INCLUDE_UCOS
		#error INCLUDE_UCOS deberia estar definido en appconfig.h para
inicializar el SDK para el KTR
	#endif
#endif
```

```
#define SO_MAX_TARS              7    /* Max. numero de tareas en la aplicacion
...                  */
                                       /* ... debe ser >= 2
*/

#define SO_PRIO_MASBAJA          8    /* Define la prioridad mas baja que puede
ser asignada ...    */
                                       /* ... nunca debe ser mayor que 63!
*/

#define SO_TAR_OCIO_TAMSTK              128    /* Tamaño del stack de la tarea
ociosa (# entradas SO_STK ) */

#define SO_TAR_ESTD_AC                  0    /* Activa (1) o Desactiva(0) la tarea
de estadisticas         */
#define SO_TAR_ESTD_TAMSTK              128    /* Tamaño del stack de la tarea de
estadisticas         */

#endif
```

Archivo MAIN.C para la Aplicación 3

```
/* Archivo: ejemplobac.c */

/*
       Este ejemplo trata de mostrar el uso de los buffers asincronos ciclicos
       para la comunicacion entre procesos.
*/

#include "inc_maestro.h"

#include "stdio.h"

#include "port.h"
#include "arch.h"

/*
       VARIABLES
 */

SO_STK       PilaTar1[TAMSTACK];
SO_STK       PilaTar2[TAMSTACK];

BUFBAC       bufbacs[3];
BAC    bac1;

Int16  suma=0;

sem          semaforo;

Int16  dato1, *dato2;

/*
       PROTOTIPO DE FUNCIONES
 */
void configFinalize (void);                                  /* Salida del Programa
SDK */

void Tarea1(void *dato);
```

```
void Tarea2(void *dato);
void SalidaProg ();


/*
*********************************************************************
  PRINCIPAL
*********************************************************************
*/
void main (void)
{
       /* Inicializacion del Kernel */
       SOIniSist();

       SOBacCrear(&bac1, bufbacs, 3, 3);

       semaforo = SOSemCrear(1);

       /* Creacion de las tareas */
       SOTarCrear(Tarea1,(void *)0, &PilaTar1, TARNTR, 0L, 0L);
       SOTarCrear(Tarea2,(void *)0, &PilaTar2, TARNTR, 0L, 0L);

       /* Inicio de la multitarea */
       SOIniciar();

}



/*
       TAREA1 (TIPO: NO TIEMPO REAL)

       DESCRIPCION: Coloca 100 datos enteros (0-99), en el buffer del BAC.
 */

void Tarea1(void *dato)
{
BUFBAC *bb1;
//Int16      dato1;

       dato = dato;                     // Solo para evitar un mensaje de prec.
del compilador

       for (dato1 = 0; dato1 < 100; dato1++)
       {

              SOSemAdquirir(semaforo);         // Adquiere el semaforo

              bb1 = SOBacReservar(&bac1);
              bb1->SOBACDato = (void *) &dato1;
              SOBacPonerMsj(&bac1, bb1);

              SOSemLiberar(semaforo);                // Libera el semaforo



              SOTmpEsp(1);                           // Espera 1 tic del timer
       }


       SOTarTermProc();
}
```

```
/*
       TAREA2 (TIPO: NO TIEMPO REAL)

       DESCRIPCION: Extrae 100 datos enteros (0-99), del buffer del BAC, y los
suma.
 */

void Tarea2(void *dato)
{
BUFBAC       *bb2;
//Int16      *dato2;
Int16   i;

       dato = dato;                     // Solo para evitar un mensaje de prec.
del compilador

       for (i = 0; i < 100; i++)
       {

              SOSemAdquirir(semaforo);         // Adquiere el semaforo

              bb2 = SOBacObtMsj(&bac1);
              dato2 = bb2->SOBACDato;
              SOBacLiberar(&bac1, bb2);

              suma = suma + (Int16) *dato2;

              SOSemLiberar(semaforo);                // Libera el semaforo

              SOTmpEsp(1);                           // Espera 1 tic del timer
       }


       //SOTarTermProc();
       SalidaProg();
}


void SalidaProg()
{

       archDisableInt();

       /* Todas las tareas estan hechas */
       printf ("Ultimo dato enviado por Tarea1 a Tarea2: %d \n", dato1-1);
       printf ("Ultimo dato recibido por Tarea2 de Tarea1: %d \n", (*dato2)-1);
       printf ("La suma es total es: %d", suma);

       while (1)
       {
                              /* Salida del programa */
              configFinalize();
       }


}
```

Código de los Archivos de la Aplicación 4

Archivo SO_CNFG.H para la Aplicación 4

```
/*
*********************************************************************************
************************
*                                          CONFIGURACION DEL KTR
*********************************************************************************
************************
*/
#ifndef __SO_CNFG_H
#define __SO_CNFG_H


#ifndef SDK_LIBRARY
       #include "configdefines.h"

       #ifndef INCLUDE_UCOS
              #error INCLUDE_UCOS deberia estar definido en appconfig.h para
inicializar el SDK para el KTR
       #endif
#endif


#define SO_MAX_TARS              6    /* Max. numero de tareas en la aplicacion
...                  */
                                       /* ... debe ser >= 2
*/

#define SO_PRIO_MASBAJA          63    /* Define la prioridad mas baja que puede
ser asignada ...    */
                                       /* ... nunca debe ser mayor que 63!
*/

#define SO_TAR_OCIO_TAMSTK              128    /* Tamaño del stack de la tarea
ociosa (# entradas SO_STK ) */

#define SO_TAR_ESTD_AC                  0    /* Activa (1) o Desactiva(0) la tarea
de estadisticas          */
#define SO_TAR_ESTD_TAMSTK              128    /* Tamaño del stack de la tarea de
estadisticas          */

#define SO_SEM_AC                1      /* Activa o Desactiva el uso de semaforos
*/
#define SO_BAC_AC                0      /* Activa o Desactiva el uso de los
BAC's*/

#endif
```

Archivo MAIN.C para la Aplicación 4

```
/* Archivo: main.c */
/* Comprobacion de los semaforos, resolviendo el problema de la cena de
   los filosofos */
#include "inc_maestro.h"
```

```
#include "port.h"
#include "arch.h"
#include "led.h"

#define      NUM_FIL             5                         /* numero de filosofos
*/
#define      IZQ          (num-1)%NUM_FIL            /* numero del vecino
izquierdo de num */
#define      DER          (num+1)%NUM_FIL            /* numero del vecino derecho
de num */
#define      PENSANDO     1                   /* el filosofo esta pensando */
#define      HAMBRIENTO   2                   /* el filosofo intenta conseguir
los
                                                              tenedores */
#define             COMIENDO     3                   /* el filosofo esta comiendo
*/

#define      TAM_STK_TAR   64          /* Tamaño de la pila de cada tarea (# de
WORDs)            */


int    estadoFil[NUM_FIL];                    /* arreglo para llevar un registro
del
                                                              estado de cada
quien */
sem    semMutex = 1;                   /* exclusion mutua para las regiones
                                                              criticas */
sem    idSemFil[NUM_FIL];                     /* un semaforo por filosofo */


typedef struct {
       int  idFil;
       int  Led;
       int  RetPensar;  /* in (1 ms) ticks */
       int     RetComer;
} tDatoTar;

SO_STK PilaTar[NUM_FIL][TAM_STK_TAR];     /* Tasks stacks
*/

tDatoTar datoTar[NUM_FIL] = {
       {0,LED_RED,1000,500},
       {1,LED_YELLOW,1000,500},
       {2,LED_GREEN,1000,500},
       {3,LED_RED2,1000,500},
       {4,LED_YELLOW2,1000,500}
};

/* Prototipo de las funciones */

void tomarTenedores (int num);
void dejarTenedores (int num);
void probar (int num);
void pensar (int tiempo, int ledFD, int idLed);
void comer (int tiempo, int ledFD, int idLed);

void filosofo (void *dato);

/* Funciones */

void filosofo(void *dato)
{
       int         LedFD     = -1;
```

```
    tDatoTar * pDatoTars = (tDatoTar *)dato;

       LedFD = ledOpen (BSP_DEVICE_NAME_LED_0, 0);

       while (VERDAD){                                                    /*
se repite por siempre */
             pensar(pDatoTars -> RetPensar,LedFD, pDatoTars -> Led);
             /* el filosofo esta pensando */
             tomarTenedores (pDatoTars -> idFil);          /* obtiene dos
tenedores o se bloquea */
             comer(pDatoTars -> RetComer, LedFD, pDatoTars -> Led);
             /* filosofo comiendo */
             dejarTenedores (pDatoTars -> idFil);                /* coloca ambos
tenedores en la mesa*/
       //    SOTmpEsp(2);
       }
}


void tomarTenedores(int num)
{
       SOSemAdquirir(semMutex);               /* entra a la region critica */
       estadoFil[num] = HAMBRIENTO;           /* registra el hecho de que el
filosofo
                                                                       tiene
hambre */
       probar(num);                                  /* intenta tomar 2 tenedores
*/

       SOSemLiberar(semMutex);                       /* sale de la region critica
*/
       SOSemAdquirir(idSemFil[num]);          /* se bloquea si no consiguio los
tenedores */
}

void dejarTenedores(int num)
{
       SOSemAdquirir(semMutex);               /* entra a la region critica */
       estadoFil[num] = PENSANDO;             /* el filosofo ha terminado de
comer */

       probar((num+1)%NUM_FIL);                                   /* ve si el
vecino izquiedo puede comer
                                                                       ahora */
       probar((num+NUM_FIL-1)%NUM_FIL);                           /* ve si
el vecino derecho puede comer
                                                                       ahora */
       SOSemLiberar(semMutex);                       /* sale de la region critica
*/
}


void probar(int num)
{
       if (estadoFil[num] == HAMBRIENTO && estadoFil[(num+1)%NUM_FIL] != COMIENDO
             && estadoFil[(num+NUM_FIL-1)%NUM_FIL] != COMIENDO) {

             estadoFil[num] = COMIENDO;
             SOSemLiberar(idSemFil[num]);

             }
}
```

```
void pensar (int tiempo, int ledFD, int idLed)
{
      if (ledFD != -1)
            {
                  ledIoctl(ledFD, LED_OFF, idLed, BSP_DEVICE_NAME_LED_0);
            }

            SOTmpEsp(tiempo);
}

void comer (int tiempo, int ledFD, int idLed)
{
      if (ledFD != -1)
            {
                  ledIoctl(ledFD, LED_ON, idLed, BSP_DEVICE_NAME_LED_0);
            }

            SOTmpEsp(tiempo);
}


void main (void)
{

int i, ret;

      SOIniSist();                                          /* inicia estructuras
de datos del KTR */

      for (i=0; i<NUM_FIL; i++)                     /* crea un semaforo por
filosofo */
      {
            idSemFil[i] = SOSemCrear(0);
            estadoFil[i] = PENSANDO;
      }

      semMutex = SOSemCrear(1);                     /* crea semaforo para la
region critica */

      for (i=0; i<NUM_FIL; i++)              /* crea una tarea por filosofo */
      {
            ret = datoTar[i].RetPensar + datoTar[i].RetComer;
      SOTarCrear(filosofo, (void *)&datoTar[i], &PilaTar[i][0], TARNTR,
(SO_PRIO_MASBAJA - 3 - i), ret);
    }


      SOIniciar();                                          /* inicia multitarea
del KTR */
}
```

APÉNDICE E - ARTÍCULOS PUBLICADOS EN CONGRESOS SOBRE LA TESIS

A lo largo del desenvolvimiento de la tesis fue necesaria la publicación de varios artículos para mostrar a los demás colegas los resultados de la investigación realizada hasta el momento.

El resultado de todo ello fue la publicación de tres artículos que enfatizan las partes del desarrollo de la tesis, el primero de ellos trata del diseño del propio Kernel de Tiempo Real presentado en esta tesis, el segundo se enfoca más hacia el Algoritmo de Planificación que se emplea en el Kernel y el último artículo hace énfasis en el mecanismo de comunicación implementado en el Kernel.

Los títulos de los Artículos y los Congresos en los cuales se presentaron dichos artículos se mencionan a continuación:

- "Diseño de un kernel de Tiempo Real para el dsp 56827", presentado de forma oral en el 4to. Congreso Internacional de Control, Instrumentación Virtual y Sistemas Digitales, celebrado los días 26-30 de Agosto de 2002 en la ciudad de Pachuca, Hidalgo, México.
- "Implementación de Buffers Cíclicos Asíncronos en un Kernel de Tiempo Real para el DSP56827", presentado en forma de póster en el XII Congreso Internacional de Computación, celebrado los días 13-17 de Octubre de 2003 en la ciudad de México, México.
- "Kernel de Tiempo Real aplicando el algoritmo de planificación EDF para el DSP 56827", presentado de forma oral en el XVI Congreso Nacional y II Congreso Internacional de Informática y Computación de ANIEI, celebrado los días 22-24 de Octubre de 2003 en la ciudad de Guadalupe, Zacatecas, México.

La realización de estos artículos ha sido gracias al apoyo y los recursos proporcionados por el Instituto Politécnico Nacional a través del Centro de Investigación en Computación y las dependencias relacionadas para el apoyo de proyectos de investigación.

REFERENCIAS BIBLIOGRÁFICAS

[1] Aho Alfred V., ***"Data Structures and Algorithms"***, 1a. Edic. 1983, Editorial Addison-Wesley, USA, ISBN 0-201-00023-7.

[2] Burns Alan, Wellings Andy, ***"Real-Time Systems and Programming Languages"***, 1a. Edic. 1989, 2da. Edic. 1996, Reimp. 1997, Ed. Addison-Wesley, England, ISBN 0-201-40365-X.

[3] Buttazzo Giorgio C., ***"Hard Real-Time Computing Systems",*** 1a. Edic 1997, Ed. Kluwer Academic Publishers, USA, ISBN 0-7923-9994-3.

[4] Douglas Comer, ***"Operating System Design, The XINU Approach"***, 1a. Edic. 1984, Ed. Prentice Hall, ISBN 0-13-637539-1.

[5] Joffrain Chris, ***"Using CodeWarrior for DSP56800E to Take Advantage of Motorola's New Digital Signal Processors"***, 2001, USA.

[6] Labrosse J. Jean, ***"MicroC/OS II - The Real-Time Kernel"***, 1ra. Edic. 1999, 2da. Edic. 2002, Ed. CMP Books, USA, ISBN 1-57820-103-9

[7] Laplante Philip A., ***"Real-Time Systems Design and Analysis: An Engineer's Handbook"***, 2da. Edic., 1997, Ed. IEEE Press, ISBN 0-7803-3400-0.

[8] Liu, C. and Layland, J. (1973). ***"Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment"***. En Journal of the ACM, Vol. 20, No. 1, pp. 46-61.

[9] Motorola, ***"DSP56800 Family Manual - Detailed description of the DSP56800 family architecture, and 16-bit DSP core processor and the instruction set"***

[10] Motorola, ***"DSP56F827 Technical Data Sheet - Electrical and timing specifications, pin descriptions, and package descriptions "***

[11] Motorola, ***"DSP56F826/F827 User's Manual - Detailed description of memory, peripherals, and interfaces of the DSP56F826 and DSP56F827"***

[12] Motorola, ***"DSP56F827EVM User's Manual"***

[13] Nutt Gary J., ***"Kernel Projects for Linux"***, 1a. Edic. 2001, Ed. Addison Wesley, ISBN 0-201-61243-7.

[14] Pons Vicanco Ramon, ***"Tipos de Kernel - http://laurel.datsi.fi.upm.es/~rpons/personal/trabajos/lpractico/node60.html",*** *Junio 2003.*

[15] Rajkumar Ragunathan, *"**Synchronization in Real-Time Systems - A Priority Inheritance Approach**"*, 1a. Edic. 1991, Ed. Kluwer Academic Publishers, USA, pp. 175, ISBN 0-7923-9211-6.

[16] Ramos R. Rafael, ***"Diseño de un Kernel de Tiempo Real para el DSP 56827"***, 4to. Congreso Internacional de Control, Instrumentación Virutal y Sistemas Digitales, Agosto 2002, Pachuca, Hidalgo, México.

[17] Ramos R. Rafael, Barrón F. Ricardo, ***"Implementación de Buffers Cíclicos Asíncronos en un Kernel de Tiempo Real para el DSP56827"***, XII

Congreso Internacional de Computación, Octubre 2003, Cd. De México, México.

[18] Ramos R. Rafael, Barrón F. Ricardo, ***"Kernel de Tiempo Real aplicando el Algoritmo de Planificación EDF para el DSP 56827"***, XVI Congreso Nacional y II Congreso Internacional de Informática y Computación de ANIEI, Octubre 2003, Guadalupe, Zacatecas, México.

[19] Rendón Gallón Álvaro, ***"Sistemas de Tiempo Real-http://dtm.unicauca.edu.co/esptelematica/sist-tiempo-real.html"***, Agosto 2003, Instituto de Postgrado en Electrónica y Telecomunicaciones, Departamento de Telemática, Universidad del Cauca, Colombia.

[20] Shaw Alan C., ***"Real-Time Systems and Software"***, 1a. Edic. 2001, Ed. John Wiley & Sons, ISBN 0-471-35490-2.

[21] Stankovic John A., Spuri Marco, Ramamritham, Buttazzo Giorgio C., ***"Deadline Scheduling for Real-Time Systems – EDF and Related Algorithms"***, 1a. Edic. 1998, Ed. Kluwer Academic Publishers, ISBN 0-7923-8269-2.

[22] Tanenbaum, Adrew S., ***"Sistemas Operativos Modernos"***, Trad. Oscar Alfredo Palmas Velasco, 1a. Edic. 1993, Ed. Pearson Educación, ISBN 968-880-323-5.

BIBLIOGRAFÍA

Abel Peter, *"Lenguaje Ensamblador y Programación para PC IBM y Compatibles"*, 1a. Edic. Original 1995, 3ra. Edic. 1996, Ed. Prentice Hall Hispanoamericana, México, pp. 594, ISBN 968-880-708-7.

Berger S. Arnold, *"Embedded Systems Design - An Introduction to Processes, Tools, & Techniques"*, 1a. Edic. 2002, Ed. CMP Books, USA, pp. 237, ISBN 1-57820-073-3.

Cooling J.E., *"Software design for Real-Time systems"*, 1a. Edic. 1991, Ed. Chapman & Hall, London, England, ISBN 0-412-63240-3.

Herrera Charles Roberto, *"Diseño del Núcleo de un Sistema Operativo de Tiempo Real (Tesis)"*, CITEDI-IPN, Octubre 1997, Tijuana, B.C., México, pp. 222

Kaisler Stephen H., *"The Design of Operating Systems for Small Computer Systems"*, 1a. Edic. 1983, Ed. John Wiley & Sons, USA, pp. 667, ISBN 0-471-07774-7.

Labrosse J. Jean, *"Embedded Systems Building Blocks"*, 2a. Edic. 2002, Ed. CMP Books, USA, pp. 611, ISBN 0-87930-604-1.

Levine R. John, *"Linkers & Loaders"*, 1a. Edic. 2000, Ed. Academic Press, USA, pp. 256, ISBN 1-55860-496-0.

Liu Jane W. S., *"Real-Time Systems"*, 1a. Edic. 2000, Ed. Prentice Hall, USA, pp. 610, ISBN 0-13-099651-3.

Rus Teodor, *"Data Structures and Operating Systems"*, 1a. Edic. 1979, Ed. John Wiley & Sons, Rumania, pp. 364, ISBN 0-471-99517-7.

Silberschatz A., Galvin P., *"Sistemas Operativos"*, Trad. Roberto L. Escalona, 5ta. Edic. 1999, Ed. Addison Wesley Longman, ISBN 968-444-310-2.

Tanenbaum A., "Sistemas Operativos Distribuidos", Trad. Óscar Alfredo Palmas Velasco, 1a. Edic. 1996, Ed. Prentice Hall, ISBN 968-880-627-7.

GLOSARIO

BCT: Bloque de Control de Tarea, se emplea en el *Kernel* para alojar todos los datos relacionados con las tareas.

BCS: Bloque de Control de Semáforo, contiene los datos que caracterizan el semáforo.

Codec: Coder/DECoder. Parte usada para convertir señales analógicas a digitales (codificador) y señales digitales a analógicas (decodificador).

DSP: Digital Signal Processor o Digital Signal Processing. En español Procesador Digital de Señales o Procesamiento Digital de Señales.

EEPROM: Electrically Erasable Programmable Read Only Memory, Memoria Programable y Borrable Eléctricamente de Sólo Lectura.

FIFO: (First In First Out) Estructura de datos empleada generalmente para administrar listas enlazadas.

GPIO: Puerto de Entrada-Salida de propósito general (General Purpose Input and Output Port) en los DSP's de la Familia Motorola.

IC: Circuito Integrado.

JTAG: Joint Test Action Group. Bus protocolo/interfaz usado para probar y depurar código en el DSP.

Kernel: Un ambiente operativo que permite a un conjunto de tareas ejecutarse concurrentemente dentro de un solo procesador.

MPIO: Puerto de Entrada-Salida Multipropósito (Multi Purpose Input and Output Port) en la Familia de DSP's de Motorola. Comparte grupos de pines con otros periféricos en el chip y pueden funcionar como un GPIO.

OnCE: On-Chip Emulation, puerto y bus de depuración creado por Motorola para permitir a los diseñadores crear una interfaz de bajo costo para un ambiente de calidad para la depuración.

PCB: Printed Circuit Board, Tarjeta de circuito impreso.

Planificación: Actividad del *Kernel* que determina el orden en el cual las tareas concurrentes son ejecutadas en un procesador.

SCI: Serial Communications Interface Port, Puerto de Interfaz para comunicaciones seriales en la Familia de DSP's de Motorola.

Semáforo: Estructura de datos del *kernel* usada para sincronizar la ejecución de las tareas concurrentes.

Sistemas Incrustados: Es una combinación de computadoras (*hardware* y *software*), y quizás de partes mecánicas adicionales a otras, diseñadas para ejecutar una función dedicada.

SPI: Serial Peripheral Interface Port, Puerto de Interfaz para periféricos seriales en la Familia de DSP's de Motorola.

SRAM: Static Random Access Memory, Memoria estática de acceso aleatorio.

SSI: Synchronous Serial Interface Port, Puerto de interfaz serial síncrona en la Familia de DSP's de Motorola.

STRD: Sistema de Tiempo Real Duro o Crítico, sistema en donde es absolutamente imperativo que las respuestas ocurran dentro del plazo requerido.

STRS: Sistema de Tiempo Real Suave o No Crítico, sistema en donde los plazos son importantes pero funcionará correctamente si los plazos son ocasionalmente errados.

Tarea: Cómputo en el cual las operaciones son ejecutadas por el procesador una a la vez. Una tarea puede consistir de una secuencia de instancias idénticas. Las palabras proceso y tarea son usadas a menudo como sinónimos.

WS: Wait State, Estado de Espera en el DSP.