MÁSTER DE AUTOMÁTICA
E INFORMÁTICA INDUSTRIAL

Implementación de un Kernel Basado en la Especificación ITRON para Arquitectura Blackfin

Autor
Jordi Sánchez Peñarroja

Director
José Enrique Simó Ten

UNIVERSIDAD
POLITECNICA
DE VALENCIA

22 de Septiembre de 2010

Índice general

Índice de figuras

Parte I

Introducción

Capítulo 1

Introducción

La tendencia bien marcada hacia la estandarización ha permitido migrar aplicaciones entre plataformas sin necesidad de realizar grandes cambios en el código. Hablando en términos de sistemas operativos se pueden destacar dos grandes estándares de programación: POSIX y TRON. POSIX es el acrónimo de *Portable Operating System Interface based on UNIX*, acuñado por Richard Stallman para IEEE. Éste es un estándar americano, y se suele utilizar en sistemas operativos de propósito más general, aunque también existen implementaciones para sistemas más específicos. En cuanto a TRON, acrónimo de *The Real-time Operating system Nucleus*, se trata de una especificación japonesa abierta, y se diseñó orientada al uso en sistemas empotrados. Pese a ser un estándar relativamente desconocido en la vida cotidiana, TRON tiene una presencia dominante en la industria tecnológica a nivel mundial. Se encuentra en el software destinado a todo tipo de aparatos electrónicos, electrodomésticos, equipos multimedia, etc, además de aplicaciones industriales y automoción.

En este proyecto se realizará una implementación de ITRON, acrónimo de *Industrial TRON*, que es una subarquitectura de TRON enfocada a aplicaciones industriales. La implementación se realizará sobre un procesador digital de señales.

Diseñada por Ken Sakamura, con esta especificación se perseguía crear una arquitectura de sistema operativo ideal. Al estar enfocada a todo tipo de sistemas, incluyendo los más pequeños, la interfaz se puede implementar de forma muy compacta. Además, también debe ser capaz de soportar tareas de tiempo real.

En este proyecto se ha desarrollado ObelISK: *Open ITRON Specification based Kernel*. ObelISK se ha diseñado siguiendo de forma rigurosa la especificación ITRON. Este núcleo se ha desarrollado tomando como referencia otros sistemas operativos de tiempo real. Entre ellos, principalmente PaRTiKle, RTEMS y uClinux.

1.1. Motivación y Objetivos

El objetivo del presente proyecto es crear una plataforma de desarrollo que sirva como base en la investigación de algoritmos de visión por computador. Este proyecto pretende sustituir un entorno de ejecución propietario y con deficiencias, por uno con mejores prestaciones y basado en licencia GPL.

El desarrollo de este núcleo viene a renglón seguido de la finalización del proyecto SENSE[1]. Durante la implementación del software relativo al proyecto SENSE, se detectaron deficiencias importantes en el software utilizado, que motivaron la creación de un núcleo con mejores prestaciones. En este proyecto se utilizó el núcleo VDK, que ofrece Analog Devices, la empresa fabricante de los procesadores Blackfin. Entre las deficiencias que se han encontrado en el núcleo se tiene:

- En primer lugar, el entorno de desarrollo es cerrado, y todas las librerías que se enlazan en las aplicaciones se encuentran precompiladas. Por lo tanto, no se tiene ningún control sobre el software de la aplicación final.
- Mala gestión de la memoria dinámica. La librería de manejo de memoria dinámica daba problemas, y sucesivas secuencias de reserva y liberación de memoria hacían que el gestor fallara. Por otra parte, el gestor de memoria dinámica que ofrece VDK no se puede utilizar en aplicaciones de tiempo real, al no estar acotado el tiempo de asignación de memoria.
- Mala gestión de la comunicación entre tareas. Este núcleo no proporciona ningún mecanismo de comunicación entre tareas, por lo que la gestión es responsabilidad del usuario. La falta de estos mecanismos obliga utilizar variables compartidas que hay que gestionar de forma relativamente compleja.
- La API que ofrece VDK no es estándar, por lo que resulta necesario aprenderla para poder desarrollar aplicaciones. ITRON es una API abierta, con lo que existen múltiples implementaciones para muchas plataformas, por lo que es más probable que el desarrollador la conozca.

En relación a estas deficiencias, el objetivo global del proyecto es mejorar la plataforma de desarrollo en los siguientes puntos:

- Mejora de la gestión de la memoria dinámica mediante la implantación del TLSF [2].
- La especificación ITRON es mucho más potente que la API de VDK.

[1] *Smart Embedded Network of Sensing Entities* [1].

- Al ser un desarrollo propio, el control sobre el código del núcleo es mayor, ya que el código fuente está disponible.

El uso de una plataforma GPL ofrece ventajas e inconvenientes en el desarrollo de software empotrado. Entre otras ventajas, se destaca la disponibilidad del código fuente de las librerías que se utilizan, lo que incrementa el control sobre el código de la aplicación final, así como el ahorro de costes al evitar el pago de licencias. Por supuesto, también existen desventajas. La principal es que, al utilizar software libre, el entorno de desarrollo no es específico de la plataforma, por lo que resulta necesario utilizar aplicaciones de propósito más general. Esto desemboca en un entorno de trabajo heterogéneo y poco integrado. Es decir, la aplicación privativa que se puede adquirir (en el caso de la arquitectura objetivo, VisualDSP), reúne todas las herramientas necesarias en una única aplicación: un editor de texto, el compilador, el depurador, así como otras que facilitan el trabajo del desarrollador.

Durante todo el desarrollo del proyecto, se ha tenido siempre presente que la finalidad de la plataforma será la de servir como entorno de ejecución en sistemas de visión. Con este fin, algunas partes de la gestión del núcleo se han diseñado a fin de facilitar el uso de la plataforma.

Capítulo 2

Estado del arte

Actualmente, el mercado de los sistemas empotrados se encuentra poblado por una gran variedad de sistemas operativos de tiempo real. En esta sección se describirán aquellos que más han afectado al desarrollo de ObelISK, así como los SOTR más extendidos en el mercado de sistemas empotrados.

2.1. VDK

VDK es el acrónimo de *Visual DSP Kernel*. Este es el núcleo que proporciona Analog Devices para el desarrollo de aplicaciones que utilicen cualquiera que sus procesadores. VDK se tomará como referencia a lo largo de todo el documento, ya que es el software que se ha sustituido en beneficio del nuevo núcleo implementado.

La API que ofrece VDK es propietaria, por lo que no sigue ningún estándar. Éste núcleo es muy básico, ofreciendo servicios de multitarea, semáforos, eventos y una forma de paso de mensajes en arquitecturas con varios procesadores. También ofrece servicios de gestión de memoria dinámica; en este caso, la gestión de bloques de tamaño fijo se realiza a través del núcleo y la de bloques de tamaño variable a través de la librería estándar de C *stdlib*.

VDK es un nanokernel, lo que significa que las aplicaciones solo se ejecutan en un modo de operación del procesador, que es el modo supervisor. Típicamente, los procesadores tienen dos modos de ejecución: modo usuario y modo supervisor.

2.2. uClinux

Este proyecto ha portado el sistema operativo Linux a una gran variedad de microcontroladores y procesadores que no disponen de MMU. En el caso de los

Blackfin, se dispone de una MMU muy básica que permite controlar los permisos de acceso a diferentes páginas de memoria. Sin embargo, esta MMU no permite direccionamiento virtual de memoria.

El proyecto uClinux contiene la mayor parte de las herramientas necesarias para poder desarrollar aplicaciones para Blackfin en entorno GPL. Este proyecto ha crecido en colaboración con la empresa Analog Devices para la implementación del compilador GCC para esta plataforma. La documentación es muy extensa y describe la puesta a punto del sistema donde desarrollar las aplicaciones. Aunque las herramientas de desarrollo están enfocadas principalmente a la implementación de aplicaciones para uClinux en Blackfin, el grupo proporciona herramientas para poder desarrollar aplicaciones *bare metal*, es decir, aplicaciones que funcionan sin un sistema operativo. Se ha utilizado esta parte de las herramientas para desarrollar el núcleo ObelISK como una aplicación bare metal, que proporcionará servicios a otras aplicaciones.

2.3. RTEMS

RTEMS son las iniciales de *Real Time Executive for Multiprocessor Systems*. En principio, este operativo de tiempo real está enfocado a sistemas empotrados en aplicaciones militares. El proyecto es muy amplio, y dispone de una amplia variedad de características, entre ellas:

- Soporte a multiprocesamiento (simétrico o asimétrico).
- Múltiples políticas de planificación.
- Herencia de prioridad.
- Memoria dinámica.

Destaca también el soporte que ofrece RTEMS a varios estándares, entre ellos, precisamente, POSIX e ITRON. Por otro lado, es posible ejecutar RTEMS en multitud de CPU que existen en el mercado, entre ellas, Intel, PowerPC, ARM, SPARC y Blackfin.

De RTEMS se ha adoptado la definición de las estructuras de datos de ITRON.

2.4. PaRTiKle

Este núcleo fue desarrollado en la Universidad Politécnica de Valencia [3]. Se trata de un núcleo de tiempo real que implementa la interfaz POSIX. Este es el

primer núcleo que utilizó el gestor de memoria dinámica TLSF, uno de los motivos por los que se ha desarrollado ObelISK, y se creó como sustituto de RTLinux.

Debido al pequeño tamaño que tiene PaRTiKle, en un principio se tomó como referencia para el desarrollo del núcleo. La principal característica que se ha adoptado de PaRTiKle es la gestión de la cola de hilos preparados. Esta gestión resulta muy eficiente y satisface las necesidades del planificador especificadas en ITRON.

2.5. QNX

QNX es un sistema operativo cerrado, propiedad de la empresa *QNX Software Systems*. Como característica que lo diferencia de otros núcleos, cada uno de los bloques de ejecución se encuentra aislado y protegido, de modo que la política de contención de fallos es muy rígida. Si fallara una de las partes, por ejemplo un driver, el fallo quedaría contenido y el sistema seguiría funcionando.

2.6. VxWorks

Este sistema operativo de tiempo real se ha creado para funcionar principalmente en sistemas empotrados. Ofrece los servicios de cualquier sistema operativo de tiempo real, incluyendo sistemas multiprocesador y gestión de memoria dinámica en tiempo real.

De entre los sistemas donde se ha instalado este núcleo destaca el Mars Reconnaissance Orbiter, un satélite de la NASA en órbita alrededor de Marte.

2.7. ThreadX

ThreadX es un núcleo también muy extendido en muy diversos campos, desde aplicaciones médicas, hasta aplicaciones industriales y militares. Obviando los servicios que ofrece, que son idénticos al resto de sistemas operativos de tiempo real, este núcleo se ha desarrollado para que la ocupación espacial sea mínima, y los tiempos de ejecución de las llamadas al sistema muy rápidos.

Capítulo 3

Herramientas

3.1. Especificación ITRON

TRON fue concebido inicialmente en 1984 por Ken Sakamura, de la Universidad de Tokio. Propuso una nueva arquitectura de sistema operativo, derivando el nombre TRON de *The Real-time Operating system Nucleus*. Sakamura se reunió con personalidades académicas y de la industria tecnológica para colaborar conjuntamente en el desarrollo de esta especificación.

Entre las políticas de diseño de esta especificación se destacan:

- Adaptabilidad al hardware. La especificación evita estandarizar aquellos elementos que atañen al hardware, ya que podría incurrir en una reducción de las prestaciones.
- Adaptabilidad a la aplicación. ITRON puede considerarse como un conjunto de librerías, las cuales se utilizarán o no en función de la aplicación, optimizando así el tamaño del núcleo para cada caso.

3.1.1. El Perfil Estándar

Para mejorar la portabilidad del software, se han definido una serie de requisitos, en cuanto a funciones implementadas. Este conjunto de funciones se conoce como *Standard Profile* o *Perfil Estándar*. La portabilidad está directamente relacionada con la escalabilidad; sistemas más grandes suelen necesitar de código más portable. Así, el Perfil Estándar de ITRON pretende que las aplicaciones desarrolladas sean portables, a la vez que se mantiene la escalabilidad del sistema. El conjunto de funciones definidas en el Perfil Estándar es el que se ha implementado

en este núcleo, y proporcionan una funcionalidad más que suficiente para desarrollar una amplia variedad de aplicaciones. De entre las funciones implementadas se tiene:

- Tareas.
- Semáforos.
- Buzones.
- Eventflags.
- Pools de memoria de tamaño fijo.
- Gestión del tiempo.
- Gestión de interrupciones.
- Gestión del planificador.

Una de las características que destaca de ITRON es la definición de reglas para los nombres de las funciones de la API. Estas reglas facilitan en gran medida la utilización de las llamadas al sistema, lo que supone uno de los puntos fuertes de esta especificación. Cada llamada a un servicio del núcleo tiene la forma *xxx_yyy*, done *xxx* representa un procedimiento operacional y *yyy* representa el objeto de la operación. Por ejemplo, la llamada cre_tsk sirve para crear (*cre*) una tarea nueva (*tsk*). Así, se define la tabla 3.1 de abreviaturas para operaciones y objetos. Como es lógico, no existen todas las combinaciones, y, de todas las posibles, sólo se han implementado las que pertenecen al Perfil Estándar. Adicionalmente, se puede colocar un prefijo *z* (*zxxx_yyy*) a algunas de las llamadas para indicar una forma concreta de ejecutar dicha llamada. Por ejemplo, para crear una tarea con asignación automática de identificador se invocaría acre_tsk.

xxx	English origin
acp	accept
act*	activate
att	attach
cal	call
can	cancel
chg	change
clr	clear
cre	create
def	define
del	delete
dis	disable
dly	delay
ena	enable
exd	exit and delete
ext	exit
fwd	forward
get	get
loc*	lock
pol	poll
ras	raise
rcv	receive
ref	reference
rel	release
rot	rotate
rpl	reply
rsm	resume
set	set
sig	signal
slp	sleep
snd	send
sns	sense
sta	start
stp	stop
sus	suspend
ter	terminate
unl	unlock
wai*	wait
wup*	wake up

yyy	English origin
alm	alarm handler
cfg	configuration
cpu	CPU
ctx	context
cyc	cyclic handler
dpn	dispatch pending
dsp	dispatch
dtq	data queue
exc	exception
flg	eventflag
inh	interrupt handler
ini	initialization
int	interrupt
isr	interrupt service routine
mbf	message buffer
mbx	mailbox
mpf	fixed-sized memory pool
mpl	memory pool
mtx	mutex
ovr	overrun handler
por	port
pri	priority
rdq	ready queue
rdv	rendezvous
sem	semaphore
sys	system
svc	service call
tex	task exception
tid	task ID
tim	time
tsk	task
tst	task status
ver	version

z	English origin
a	automatic ID assignment
f	force
i	interrupt
p	poll
t	timeout

* Abbreviations with asterisks (*) are also used as a *yyy* abbreviation.

Cuadro 3.1: Abreviaturas utilizadas en ITRON para la definición de las llamadas a servicios.

3.1.2. Tipos de datos

ITRON especifica también la notación para los tipos de datos básicos[1]. Existe una gran variedad de compiladores en el mercado, que siguen reglas diferentes

[1]Los tipos de datos pueden ser básicos o compuestos. Una variable tiene un tipo de datos básico si, por sí sola, sirve para referenciar un valor. Una variable es de tipo compuesto cuando está formada por un conjunto de variables del tipo que sea (en C, estructuras o vectores).

según cada procesador. El lenguaje C define unos tipos de datos que no tienen por qué ser equivalentes entre arquitecturas. Por ejemplo, el tipo de dato *int* sirve para definir una variable entera. El problema es que esta palabra reservada no describe la longitud del tipo de dato; *int* podría referirse a una variable de 32 o de 16 bits, según la arquitectura.

Lo conveniente para evitar estas situaciones es envolver los tipos de datos que ofrece C, sustituyéndolos por otros más descriptivos. La forma de sustituir los tipos de datos en C puede verse a continuación:

```
typedef char B;
```

La definición anterior se realiza una vez se conoce la arquitectura. En Blackfin, se sabe que el tipo de datos que define una variable de 8 bits es *char*[2]. Por lo tanto, este tipo de datos se envuelve con el tipo *B*. De este modo, cada vez que se defina una variable del tipo *B*, lo que en realidad se estará haciendo es definir una variable de tipo *char*.

Las definiciones proporcionadas por ITRON son demasiado cortas y se ha evitado su uso dentro del núcleo. Los tipos de datos están a disposición del usuario, pero dentro del código del núcleo se han utilizado los tipos de datos definidos en MISRA [4]. MISRA C define un conjunto de reglas para evitar problemas durante el desarrollo de software. En lugar de utilizar el tipo *B*, se utiliza el tipo *int8_t*. Esta forma de definir los tipos de datos resulta mucho más descriptiva que la definida por ITRON, y se utiliza mucho en desarrollos para sistemas empotrados.

A parte de los tipos básicos, ITRON define una serie de estructuras que tienen como objetivo servir de paquetes de datos en las llamadas a servicios del sistema. Por ejemplo, cuando se crea una tarea hay que inicializar sus parámetros, como la prioridad, el identificador o la dirección de inicio. Cada uno de los paquetes se ha definido en su fichero de cabecera correspondiente.

3.1.3. Constantes

Por último, ITRON define toda una serie de directivas para su uso dentro del código. Existe una gran cantidad de definiciones de constantes, que se utilizan en la creación de los objetos del núcleo y en los códigos de error. Además, cada función se identifica con una constante que, como se verá en la sección 4.9, se ha utilizado en el log del sistema.

[2] En el caso del tipo *char*, tiene 8 bits en todas las arquitecturas.

3.2. Blackfin BF548

El desarrollo del núcleo se ha realizado utilizando un procesador digital de señales Blackfin BF548 de Analog Devices. Este es un DSP segmentado de 32 bits, que tiene una arquitectura Harvard modificada con memoria interna de datos y de programa separadas. El desarrollo de este tipo de procesador lo realizó Analog Devices junto con Intel y Marvell. El resultado es un procesador de coma fija de alto rendimiento y bajo consumo, ideal para sistemas de visión. El procesador puede funcionar a un máximo de 600 MHz, dependiendo de qué configuración se elija para el reloj. La figura 3.2.1 muestra el diagrama de bloques del BF548.

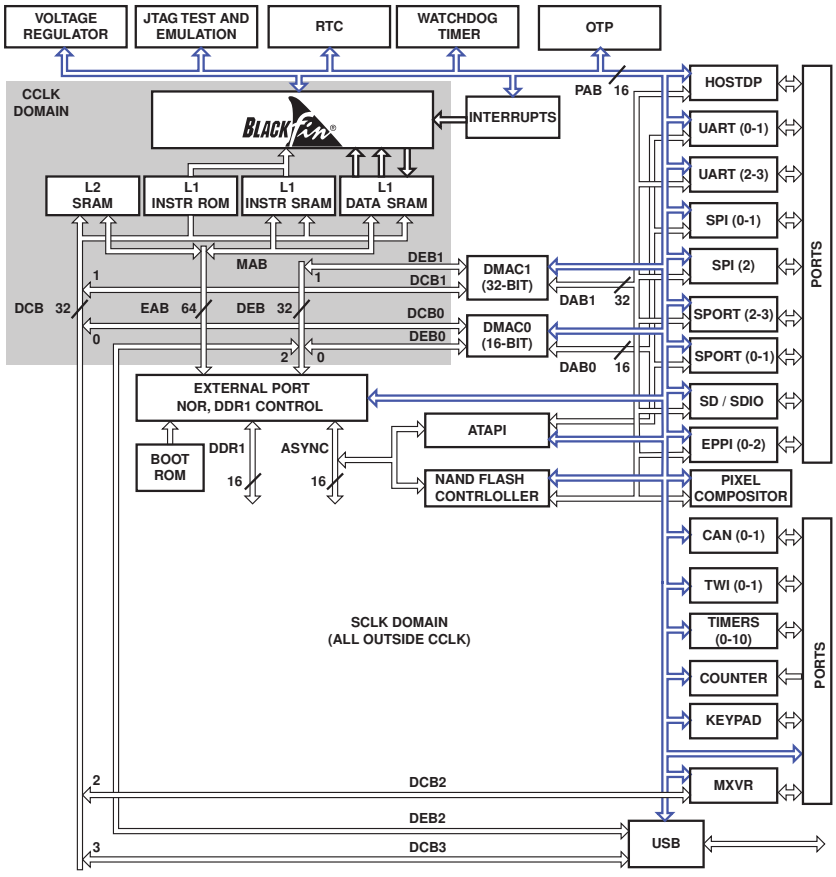

Figura 3.2.1: Diagrama de bloques del BF548.

En los Blackfin pueden distinguirse dos dominios de ejecución. El dominio interno está conducido por el reloj del núcleo, y es el dominio más rápido de ejecución. Este reloj se obtiene de multiplicar la frecuencia del reloj de entrada por un factor. En la figura 3.2.1 puede verse los dispositivos que funcionan a la frecuencia marcada por este reloj en gris oscuro (memorias internas y CPU). El segundo dominio es el dominio externo, que tiene una frecuencia máxima de funcionamiento inferior a la del dominio interno. En este dominio se ubican todos los dispositivos de entrada salida, así como las memorias externas que se instalen.

3.2.1. Eventos

Un evento es una condición que cambia el flujo de ejecución del código. Entre los distintos tipos de eventos dentro de la arquitectura Blackfin se tiene:

- Emulación.
- Reset.
- Interrupción no enmascarable.
- Excepción.
- Interrupción.

Las interrupciones responden a un evento externo; típicamente, un periférico solicitando atención. Una interrupción es un evento asíncrono que produce un salto en el código hacia un manejador de interrupción. Una excepción es un evento síncrono, que se dispara como resultado de una finalización incorrecta de una instrucción, por ejemplo, un acceso desalineado a memoria o el intento de ejecutar una instrucción no válida.

3.2.1.1. Interrupciones

En la arquitectura Blackfin existen 15 niveles de interrupción, siendo más prioritarias las de nivel más bajo. Así, los niveles 0 a 6 están reservados a los eventos más críticos del procesador, los niveles 7 a 13 para los periféricos y, por último, los niveles 14 y 15 están destinados a interrupciones software. Cada nivel de interrupciones se denomina *Interrupt Vector Group* (IVG) y, en el caso de los periféricos, se puede asociar a un IVG interrupciones de más de un dispositivo.

La activación de las interrupciones se controla mediante el registro de máscara IMASK. Un 1 indica que la interrupción está enmascarada y, por lo tanto, puede activarse. Las interrupciones activas (que se han disparado) se mantienen en el

registro IPEND como bits a nivel alto. Para saber en qué nivel de interrupción se encuentra el procesador, basta con comprobar el bit activo de menor peso del registro IPEND. La posición de dicho bit indicará el IVG que se está sirviendo.

Las interrupciones son la forma de activar el modo supervisor en el procesador. Al activarse cualquier interrupción, el procesador pasa a modo supervisor y se tiene acceso a todos los dispositivos sin restricciones. Si no hay ninguna interrupción activa, entonces el procesador se encuentra en modo usuario. La figura 3.2.2 presenta de forma más detallada los distintos modos del procesador.

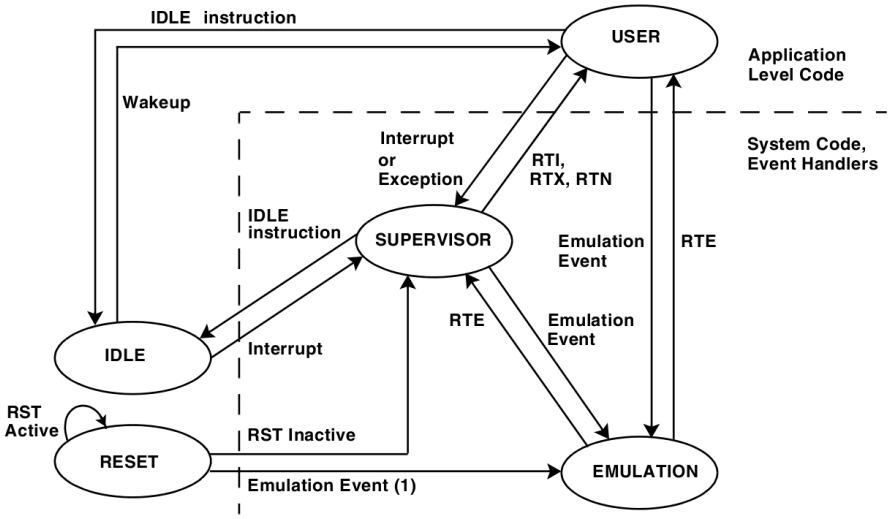

Figura 3.2.2: Modos y estados del procesador.

Uno de los principales problemas que se ha tenido en el desarrollo de este núcleo ha sido las importantes limitaciones del modo usuario. Al encontrarse el procesador en modo usuario, no se tiene acceso ni a los periféricos, ni a algunos registros (como el registro de pila), ni a algunas instrucciones. Estas limitaciones han sido la principal razón de que el núcleo se haya implementado para funcionar en modo supervisor. En concreto, la clave se encuentra en la instrucción *raise*. Esta instrucción permite activar una interrupción a voluntad, pero está reservada para el modo supervisor, lo que implica que un usuario no puede salir del modo usuario a no ser que se produzca una interrupción. Así pues, la aplicación no tiene ninguna posibilidad de acceder a la interfaz del núcleo, lo cual limita seriamente las posibilidades de las aplicaciones. Por otro lado, se ha comprobado que el núcleo VDK (*VisualDSP Kernel*), desarrollado por el propio fabricante del DSP, también funciona como un nanokernel, manteniendo el modo supervisor. En conclusión,

el núcleo funciona manteniendo activa de forma permanente la interrupción 15, que es la de menor prioridad, para mantener el modo superusuario. Esta decisión acarrea una serie de consecuencias en el desarrollo de las aplicaciones.

Cada nivel de interrupción tiene su entrada correspondiente en una tabla denominada Event Vector Table (EVT). Esta tabla mantiene, por cada IVG, la dirección de memoria donde empieza el código del manejador para ese nivel de interrupción. El funcionamiento de las interrupciones es el siguiente: cuando se produce una interrupción, se vacía el pipeline de instrucciones (mediante la inserción de instrucciones *nop*) y se carga en el PC la dirección almacenada en la entrada del EVT asociada a la interrupción. El procesador tiene un registro, RETI, que almacena la dirección de retorno de la interrupción. Para retornar de un manejador de interrupción se tiene la instrucción *rti (return from interrupt)*. Al ejecutarla, el procesador pone a 0 el bit de menor peso del registro IPEND, por ser la interrupción de mayor prioridad que se está sirviendo, y devuelve el flujo de ejecución a la dirección apuntada por RETI.

3.2.1.2. Excepciones

Las excepciones son un tipo de evento más prioritario que las interrupciones. Las excepciones se analizan a través del registro SEQSTAT, mostrado en la figura 3.2.3. La mayoría de las excepciones están relacionadas con problemas con la memoria, por ejemplo, accesos desalineados o fallos de CPLB.

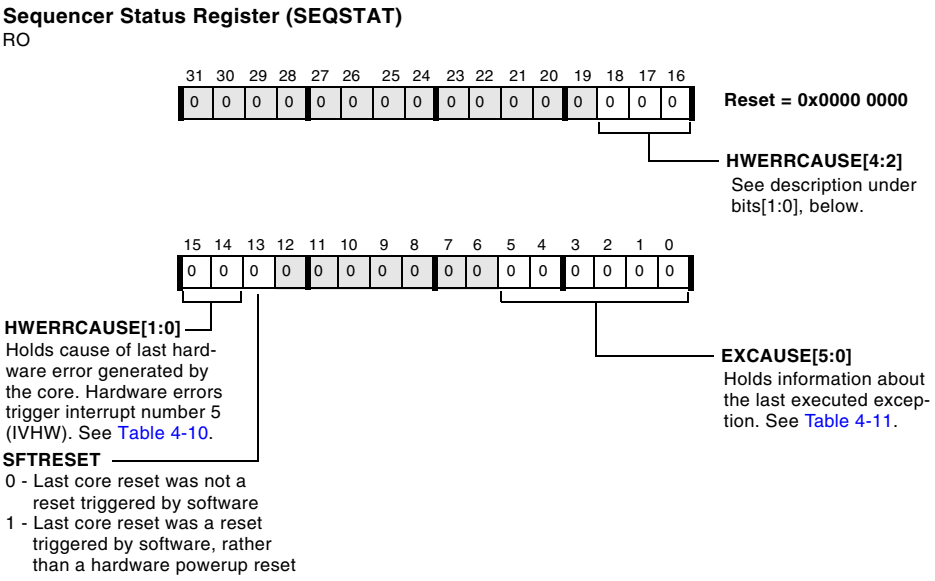

Figura 3.2.3: Registro SEQSTAT.

Cuando se produce una excepción, se guarda en el registro RETX el valor de la dirección de la siguiente instrucción a la que produjo la excepción. Asimismo, el registro SEQSTAT contiene la información relativa al error que produjo la excepción, en el campo EXCAUSE. Este campo indica el número de excepción que se ha producido. Es de 6 bits, con lo que el número total de excepciones diferentes que pueden producirse es de 64. Para el manejo de las mismas, se ha definido una tabla de 64 entradas, para almacenar los punteros a los 64 posibles manejadores de excepción.

3.2.2. Memoria

La estructura de memoria del BF548 es un espacio unificado de direcciones de 4 GB utilizando direccionamiento de 32 bits. Todos los recursos, incluyendo memoria interna, memoria externa y los dispositivos, se controlan a través de secciones separadas de este espacio común de direcciones. El procesador no soporta direcciones virtuales, por lo que todas las direcciones de memoria que emite el procesador tienen una correspondencia física directa. La figura 3.2.4 muestra el mapa de memoria.

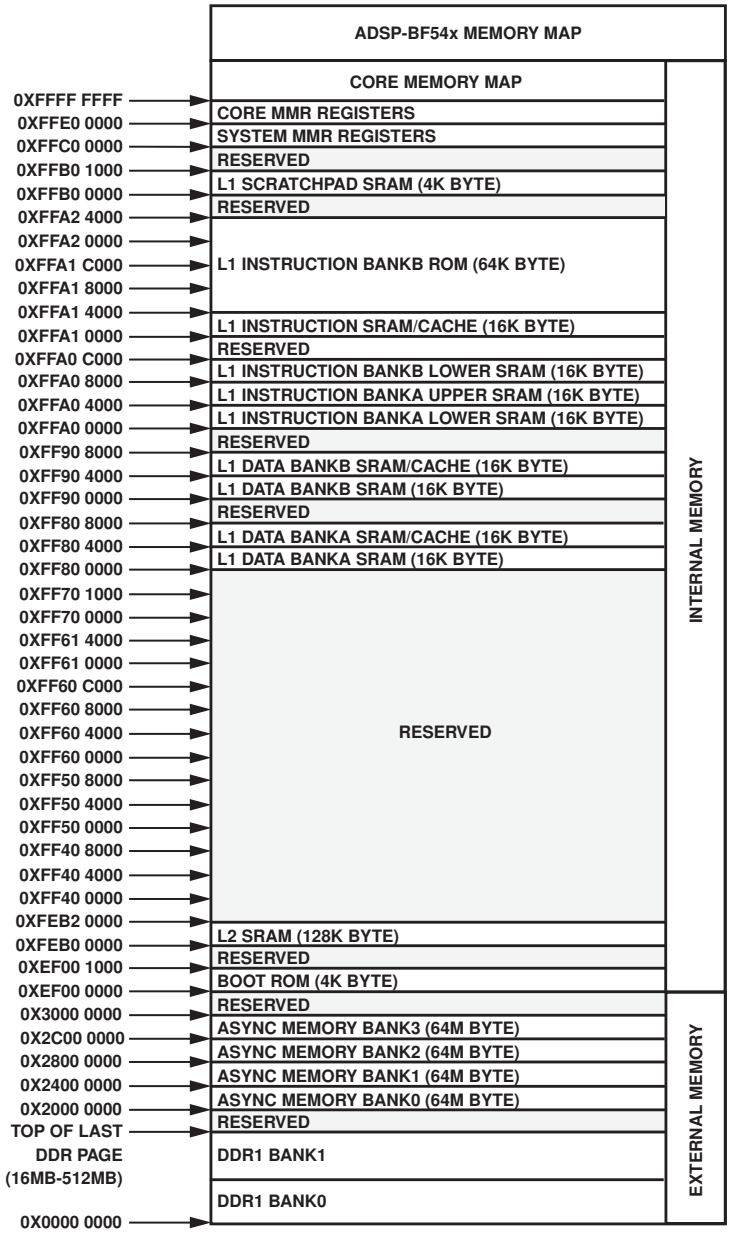

Figura 3.2.4: Mapa de memoria.

En este procesador el manejo de la memoria es complejo, y es crucial tener

una buena distribución de las aplicaciones en memoria, ya que puede significar un incremento importante en el rendimiento de las mismas.

3.2.2.1. Memory Management Unit (MMU)

El BF548 dispone de una MMU con la que se puede gestionar los permisos de acceso a las distintas páginas de la memoria, así como la *función de cache*[3] de las páginas de memoria. La MMU está implementada como dos tablas (una de instrucciones y una de datos) de 16 entradas. Cada entrada es referida como un descriptor CPLB (*Cache Protection Lookaside Buffer* ó buffer de inspección de protección de cache). Cada descriptor consta de dos MMR (*Memory Mapped Registers*) de 32 bits. Uno de los MMR mantiene la dirección de inicio de la página para la cual se quieren definir los niveles de protección y el otro controla los atributos de dicha página. Entre otras cosas, gestiona el tamaño de la página, los permisos de lectura/escritura y la función de cache.

Debido al tamaño de la memoria externa en el CM-BF548 (64 MB), no es posible controlar la protección de todas las páginas de la memoria a la vez, ya que no existen suficientes descriptores CPLB en la MMU. Las 16 entradas permiten controlar, como máximo, 64 MB de memoria. Por otro lado, una vez se active la MMU, cualquier acceso a una dirección de memoria no descrita en ningún CPLB (siempre y cuando se hayan activado las CPLB) generará una excepción, acuñada con el término *fallo de CPLB*. Por lo tanto, no solo se debe mantener descriptores para la memoria externa, sino también para todos los MMR, la memoria interna L1 y L2 y los bancos de memoria asíncrona que se pudieran utilizar. Para poder proteger todo el espacio disponible, se debe mantener la tabla de descriptores completa como un buffer en la memoria de datos. En el momento en que se genere un fallo de CPLB, el manejador de excepción deberá sustituir alguno de los descriptores instalados por el que controla la protección de la página que generó dicha excepción. Lógicamente, la tabla completa de descriptores CPLB debe estar en una página de memoria siempre controlada a su vez por un CPLB. De lo contrario, si la tabla quedara desprotegida en algún momento y se produjera un fallo de CPLB, se produciría una doble excepción al acceder a la dirección de memoria donde está almacenada, y el procesador quedaría bloqueado. Este problema se resuelve ubicando la tabla de descriptores de cache en la parte baja de la memoria *L1 Data A* (ver figura 3.2.4).

[3] En los documentos de Analog se utiliza el término *cacheability* para mencionar la posibilidad de activar la *función de cache* en ciertas páginas. Es decir, controlar si las páginas de L2 y memoria externa, en el evento de un fallo de página, se cargarán en cache o no. El autor reconoce que no ha encontrado una mejor traducción para este término, por lo que es el que utilizará a lo largo del documento.

3.2.2.2. Cache

La activación de la función de cache en el Blackfin permite cargar páginas de memoria externa en L1 o L2. Activar esta funcionalidad puede significar un enorme incremento en el rendimiento aplicación. La cache del Blackfin es de 4 vías asociativas por set, con posibilidad de activar política de victimización LRU (*Least Recently Used*). Las páginas de memoria que se manejan son de 32 bytes, y es posible seleccionar la asociatividad de las páginas en relación a los dos bits utilizados para seleccionar cada una de las vías. Al activar la cache, la mitad superior de las memorias de instrucciones y datos internas queda inhabilitada para almacenar datos de forma explícita.

3.2.3. Relojes

En el BF548 se pueden distinguir dos dominios de reloj: el del núcleo (CCLK, del inglés *Core Clock*) y el de sistema (SCLK, del inglés *System Clock*). El CCLK marca la temporización tanto de la ejecución de las instrucciones como de las memorias internas del procesador. El CCLK se considera el dominio rápido del sistema, ya que es el reloj que, generalmente, funcionará a mayor frecuencia. El SCLK es el sistema que marca la temporización de los periféricos, y se considera el dominio lento.

Ambos relojes están gobernados por un reloj de entrada CLKIN. Esta señal se conecta a un oscilador externo para crear la señal de reloj que temporiza el sistema. El CCLK se obtiene multiplicando el reloj de entrada, CLKIN, por un factor *cmult*. El SCLK depende del CCLK, ya que se calcula dividiendo este último por un factor *cdiv*. La elección de estos factores determinará las frecuencias a las que funcionará el sistema. Las ecuaciones (3.2.1) y (3.2.2) muestran la forma de calcular los factores necesarios para configurar el procesador a la frecuencia deseada. La forma de presentar las ecuaciones refleja la forma de tratar la frecuencia del procesador. Las frecuencias de reloj CCLK y SCLK las configura el usuario, por lo que lo que hay que calcular realmente son los factores de multiplicación y división (ecuaciones de la parte izquierda). La forma en que el procesador establece la frecuencia de las señales queda reflejada en las ecuaciones de la parte derecha.

$$cmult = \frac{CCLK}{CLKIN} \rightarrow CCLK = cmult \cdot CLKIN \tag{3.2.1}$$

$$cdiv = \frac{CCLK}{SCLK} \rightarrow SCLK = \frac{cdiv}{CCLK} \tag{3.2.2}$$

3.3. Core Module CM-BF548

El procesador de Analog por sí solo no es una herramienta que se pueda utilizar para desarrollar una aplicación. El procesador se monta en una placa base que contiene una serie de dispositivos básicos para la ejecución de programas. La empresa Bluetechnix monta este y otros procesadores en *core modules*. La figura 3.3.1 muestra los componentes principales del Core Module CM-BF548, utilizado para el desarrollo de ObelISK.

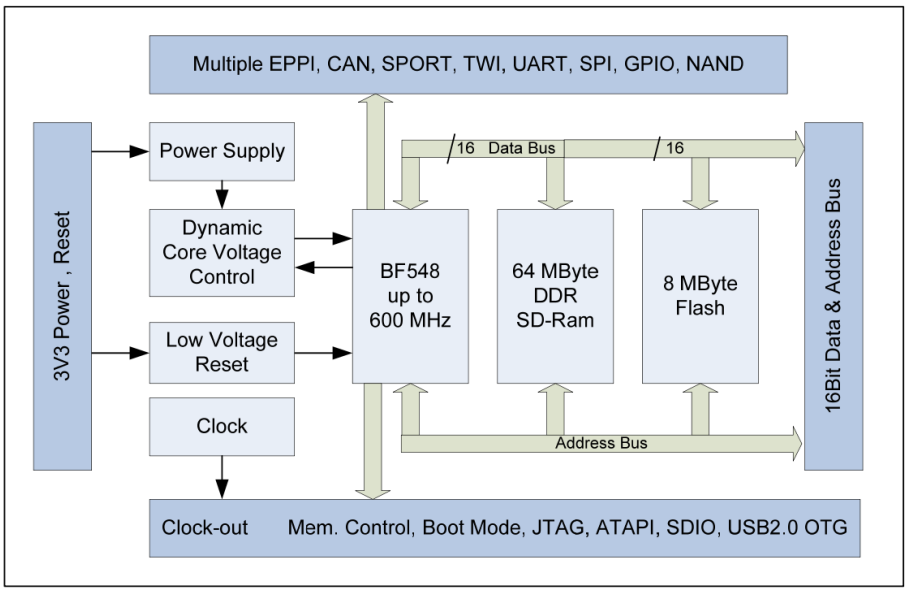

Figura 3.3.1: Diagrama de bloques del CM-BF548

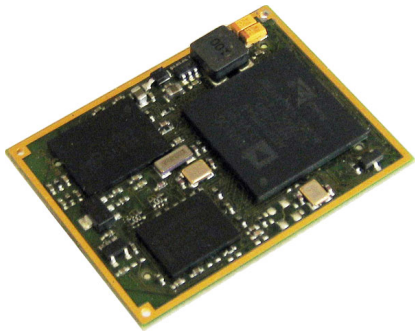

Figura 3.3.2: Fotografía del módulo CM-BF548.

El abanico de dispositivos responde a las típicas demandas de los sistemas empotrados: un sistema de memoria limitado para la ejecución del programa, una memoria de estado sólido para almacenar el programa y arrancarlo, y una serie de dispositivos de comunicación y tratamiento de señales. En concreto, el CM-BF548 dispone de los siguientes elementos:

- Oscilador de 25 MHz.
- Memoria DDR SDRAM de 64 MB.
- 8 MB de memoria flash.
- Dos conectores de expansión de 100 pines, donde se puede conectar el resto de dispositivos.

3.4. Placa de desarrollo DEV-BF548DA-Lite

La placa de desarrollo DEV-BF548DA-Lite se ha diseñado para servir como entorno de desarrollo para el CM-BF548. La figura 3.4.1 muestra el diagrama de bloques de la placa de desarrollo.

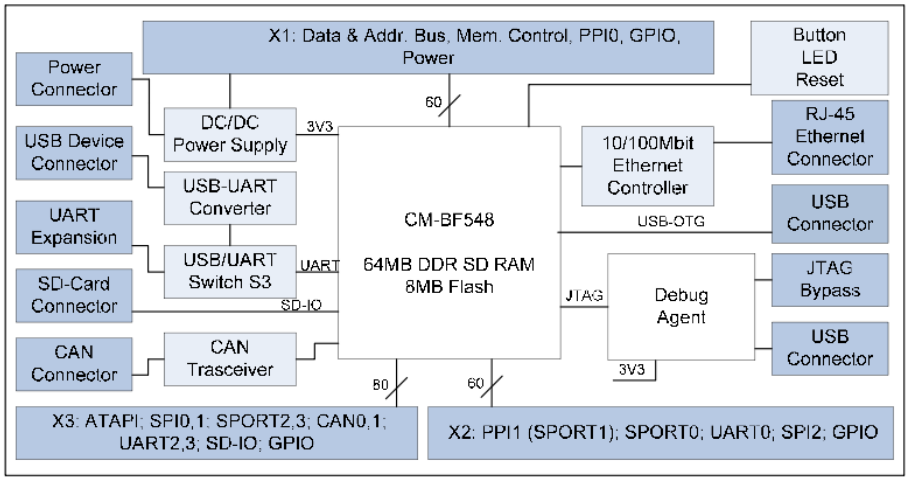

Figura 3.4.1: Diagrama de bloques de la placa de desarrollo DEV-BF548DA-Lite.

La placa de desarrollo utilizada contiene el resto de dispositivos necesarios para poder desarrollar las aplicaciones. Por ejemplo, el conector de alimentación y la interfaz de JTAG se encuentran integradas en esta placa.

Figura 3.4.2: Fotografía de la placa de desarrollo DEV-BF548DA-Lite.

3.5. Toolchain

El toolchain se compone de todas las herramientas necesarias para poder desarrollar una aplicación para una plataforma determinada. Para el desarrollo del núcleo, se ha utilizado una amplia variedad de herramientas, así como hardware específico para poder cargar y depurar las aplicaciones en la plataforma. Principalmente, se han seguido los pasos descritos en [5]. Esta es la página del proyecto de uClinux para Blackfin. Este proyecto está parcialmente financiado y mantenido por Analog Devices.

3.5.1. GCC

El compilador de GNU [6] utilizado es la herramienta fundamental. En este caso, se ha trabajado con lo que se conoce como un *cross compiler*. Un *cross compiler* es un compilador que produce código ensamblador para una máquina diferente de la que se ha utilizado para compilar el código. El compilador se ha obtenido del proyecto uClinux para Blackfin, el cual se proporciona en dos versiones diferentes: una para compilar aplicaciones para uClinux y otra para compilar aplicaciones para Blackfin sin ningún sistema operativo subyacente. Este segundo compilador es el que se ha utilizado para el desarrollo del núcleo.

3.5.2. Binutils

Binutils es una colección de herramientas para el procesamiento de ficheros binarios. También se proporcionan desde el proyecto uClinux. Se destacan dos herramientas indispensables:

- **LD**. El enlazador de GNU [7]. Esta herramienta recoge un fichero de entrada conocido como *Linker Description Script* (LDS) y, en base a la configuración descrita en él, se enlaza el código de forma adecuada. En la sección 5.4 se describirá de forma más detallada el funcionamiento de esta herramienta.

- **AS**. En ensamblador de GNU [8]. La herramienta *as* analiza un fichero de texto con código ensamblador y genera un fichero de código binario que es el que realmente puede interpretar y ejecutar el procesador. El ensamblador fue la primera forma de escribir código de “alto nivel”. En un principio, los programas se escribían perforando en tarjetas los bits de las instrucciones. Con la aparición de esta herramienta, la sintaxis de la programación se hizo más sencilla. Hoy en día sigue siendo una herramienta muy necesaria, ya que muchas partes de los sistemas operativos se escriben en ensamblador.

3.5.3. Eclipse

Existe una gran variedad de editores para poder escribir el código. Se ha utilizado Eclipse [9] debido a que es un entorno bastante avanzado, con funcionalidades que facilitan la programación y, por lo tanto, incrementan la productividad mientras se escribe código.

3.5.4. Make

La herramienta Make [10] resulta indispensable a la hora de realizar compilaciones complejas. Automatiza el proceso de compilación, mediante el uso de una cierta sintaxis. Sin esta herramienta la compilación del núcleo sería prácticamente inabordable, debido a la gran cantidad de ficheros que hay que compilar.

3.5.5. JTAG

Diseñado originalmente para circuitos impresos, actualmente es utilizado para la prueba de submódulos de circuitos integrados, y es muy útil también como mecanismo para depuración de aplicaciones empotradas, puesto que provee una puerta trasera hacia dentro del sistema. Cuando se utiliza como herramienta de depuración, un emulador en circuito que usa JTAG como mecanismo de transporte permite al programador acceder al módulo de depuración que se encuentra integrado dentro de la CPU.

En este proyecto se ha utilizado un JTAG provisto por la empresa Section 5, llamado *ICEBear JTAG* [11]. La elección de este JTAG se debe a que es la mejor interfaz de JTAG existente para Linux en cuanto a velocidad de transferencia de datos. La figura 3.5.1 muestra el aspecto del dispositivo JTAG utilizado.

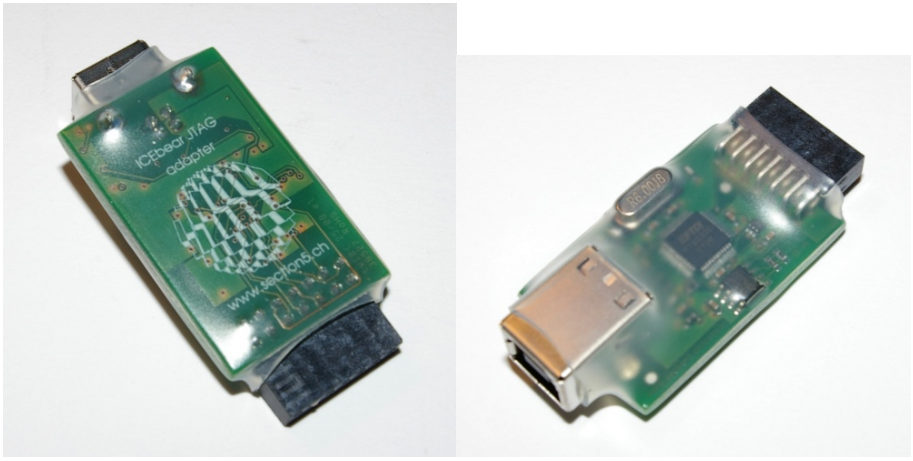

Figura 3.5.1: Vistas superior e inferior del dispositivo JTAG ICEBear de Section 5.

3.5.6. GDB

GDB es el acrónimo de *GNU Debugger*. El depurador de GNU [12] ha sido fundamental para poder corregir los errores de implementación. Un depurador es una herramienta que permite conocer el estado interno de un programa durante su ejecución.

3.5.7. Insight

Insight es un entorno gráfico para la depuración de código. En sí, esta aplicación no es un depurador, sino que funciona como una capa encima del depurador para ofrecer un entorno gráfico. Facilita la depuración, ya que GDB por sí solo funciona en modo consola, lo cual dificulta la tarea.

3.5.8. Gdbproxy

Sirve para crear un puente entre un sistema local y un depurador remoto. Junto con Insight y GDB, conforman las herramientas necesarias para la depuración del código en el sistema empotrado.

Parte II

Desarrollo

Capítulo 4

El núcleo de ObelISK

El objetivo clave de todo núcleo es proporcionar los elementos necesarios para poder ejecutar varias tareas de forma concurrente en un mismo procesador. Como es lógico, la concurrencia es sólo virtual; el sistema operativo se encarga de asignar el procesador a cada una de las tareas, de acuerdo con algún criterio preestablecido que se conoce como *planificación*.

El núcleo que se ha desarrollado es de tiempo real, lo que implica que todas y cada una de las llamadas al sistema y librerías que ofrece tienen un tiempo de ejecución *determinista*. Esto quiere decir que es posible estimar el tiempo de ejecución en el peor caso (WCET[1]) de todas y cada una de las funciones del núcleo. El determinismo en los tiempos de ejecución no implica que el tiempo de ejecución sea constante. En algunos casos, la llamada siempre responde en tiempo constante; en otros, simplemente no es posible asegurar tiempo constante. Por ejemplo, puede ser necesario recorrer una lista, por lo que el tiempo de ejecución no es constante aunque sí está acotado, en tanto el tamaño de la lista esté acotado. Este matiz es especialmente importante, ya que el WCET de ciertas llamadas depende de la aplicación que se ha implementado.

La concurrencia implica que el sistema operativo debe proporcionar una serie de mecanismos de sincronización de forma que no se produzcan conflictos entre las distintas tareas en el acceso a los recursos. Asimismo, también se proporcionan mecanismos que permiten comunicar las tareas entre sí. Así, dentro del sistema operativo, se puede distinguir dos elementos básicos: las *tareas*, y los *objetos de sincronización*.

El núcleo de ObelISK se ha construido en torno a la especificación ITRON. Esta especificación no indica la forma en que se ha de gestionar el núcleo de forma interna. Simplemente, se dan unas pautas para la gestión de los servicios, dejan-

[1] Del inglés, *Worst Case Execution Time.*

do al desarrollador libertad para configurar el núcleo. Por esa razón, el núcleo de ObelISK se ha desacoplado de la especificación, en términos de las estructuras de datos y las construcciones utilizadas. La figura 4.0.1 presenta la estructura de ObelISK. Como puede verse, ITRON cuelga del núcleo de ObelISK, del que utiliza la infraestructura necesaria para proporcionar sus servicios. La utilización de esta arquitectura deja abierta la posibilidad de implementar una especificación diferente a ITRON sobre el núcleo, lo cual lo hace escalable y versátil.

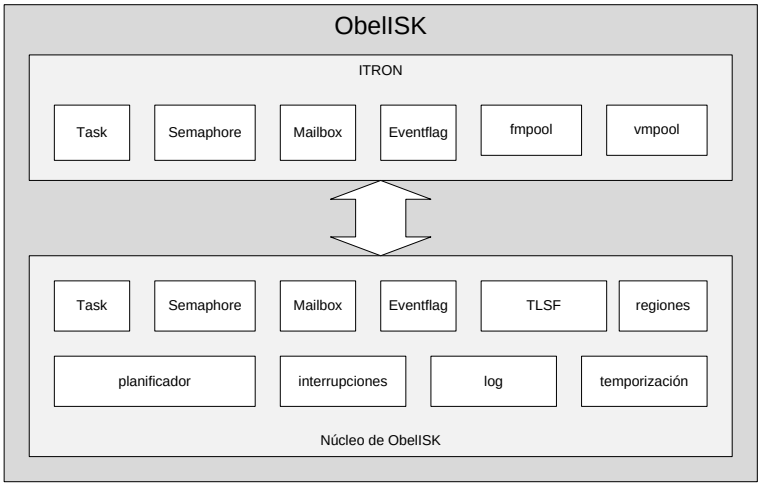

Figura 4.0.1: Diagrama de bloques de ObelISK.

El desacoplamiento entre ITRON y el núcleo de ObelISK es todavía débil. Si se observa la figura 4.0.1, ésta no refleja una distribución de código, sino más bien el marco conceptual bajo el que funciona el núcleo. De hecho, es la interfaz de ITRON la que implementa el código de las funciones de gestión y sincronización de tareas, y no el propio núcleo. El núcleo simplemente proporciona las estructuras de datos que contienen las propiedades de los hilos y los objetos de sincronización. Si, por ejemplo, se pretendiera implementar POSIX, sería conveniente que fuera el núcleo de ObelISK el que implementara los objetos de sincronización, con el fin de evitar tener dos implementaciones diferentes para el mismo tipo de objeto de sincronización.

4.1. La pila de ejecución

El elemento fundamental que maneja un sistema operativo en relación a las tareas en ejecución es la pila. Una *pila* es una estructura de datos compuesta por un

puntero, que apunta a la *cima* de la pila, y dos operaciones, *apilar* y *desapilar*. La operación de apilamiento se realiza copiando el dato que se desea apilar en la cima de la pila e incrementando el valor del puntero de cima. Para desapilar, se extrae el dato de la cima y se decrementa el puntero. Únicamente se permite el acceso al dato en la cima de la pila; cualquier acceso al resto de datos se considera una operación ilegal. Por convención, la pila crece hacia direcciones decrecientes de memoria.

La pila cobra especial importancia cuando se pretende ejecutar código implementado en un lenguaje de alto nivel. Normalmente, en las inicializaciones de los procesadores, uno de los primeros pasos suele ser el establecimiento de la pila. De este modo, se reduce la cantidad de código en ensamblador que hay que implementar y se puede saltar pronto a código de alto nivel. Este salto resulta especialmente interesante ya que programar en lenguaje máquina, pese a que se produce el código de mayor calidad, es lento y complejo.

El siguiente ejemplo en lenguaje C ilustra el papel de la pila en la ejecución de cualquier código.

```
1  int resultado;
2
3  int factorial(int n) {
4          int x;
5          x = n;
6          if (x <= 1) {
7                  return 1;
8          }
9          return x*factorial(x-1);
10 }
11
12 int main(void) {
13         int x = 10;
14         resultado = factorial(x);
15         return resultado;
16 }
```

Analizando el código expuesto, en primer lugar hay que distinguir entre código y datos. El código son las instrucciones que debe ejecutar el procesador, resultado de la traducción del programa anterior de C al código ensamblador de la plataforma objetivo. Este código ocupará un espacio en memoria diferente del de los datos. En cuanto a los datos, se pueden distinguir dos tipos: locales y globales. En el ejemplo, la variable `resultado` es un dato global de tipo entero, y tiene una ubicación específica y única en memoria. Por otro lado, las variables `x` y `n` son

locales, también de tipo entero, pero no pueden almacenarse estáticamente, ya que puede haber múltiples ocurrencias de cada una. Nótese, además, que la variable `x` está replicada en dos funciones diferentes por lo que, de almacenarla en memoria, existiría un conflicto. Por otro lado, la función `factorial` es una función que se conoce como recursiva; la función se llama a sí misma. La condición `x<=1` (línea 6) se conoce como *caso base*, y marca la condición de parada de la recursividad. El caso de la recursividad es muy interesante; la función factorial se llama a sí misma y, mientras no se haya llegado al caso base, existirá un número de ocurrencias en memoria igual al número de llamadas.

La forma de gestionar las variables locales es mediante la utilización de una pila. Cuando se llama a una función, se debe apilar una serie de datos. Dependiendo del compilador, el apilamiento se realiza forma diferente. Básicamente, se debe almacenar:

- La *dirección de retorno*. En una llamada a una subrutina, el flujo de ejecución da un salto, por lo que se debe almacenar la dirección de retorno. Es decir, la dirección a la que debe regresar el flujo de ejecución una vez terminada de ejecutar la subrutina. En Blackfin, cuando se realiza una llamada a una función, se genera la instrucción en ensamblador `call`. Esta instrucción salta al código de la subrutina y almacena la dirección de retorno en el registro RETS. Dado que una subrutina podría llamar a otra subrutina, el valor del registro RETS se podría sobreescribir. Para evitar perder la dirección de retorno, el compilador apila el registro RETS a la entrada de cada subrutina.

- Los *parámetros de la función*. En el ejemplo anterior, la función factorial recibe un parámetro `n`. Este parámetro se debe almacenar en la pila, ya que podría darse el caso de anidamiento de llamadas. Cuando se invoca una función, debido a la ABI[2] de Blackfin, los parámetros se pasan a través de los registros de datos `[r0-r3]`. Si se produce anidamiento de llamadas, los parámetros deben apilarse para que no se pierdan al retornar a la función invocadora.

- Las *variables locales*. En el caso de las variables locales no se utilizan los registros de datos. Para las variables locales se utiliza la instrucción `LINK`. Esta instrucción reserva un cierto espacio de pila, que se utilizará para almacenar los datos locales.

- El *valor de retorno*. En el caso de que la función devolviera algún valor, éste se pasa a través del registro `r0`. En realidad, este valor no es necesario

[2] Application Binary Interface. Define la forma en que el compilador debe utilizar los registros.

almacenarlo en la pila, puesto que es un dato de salida, de la función invocada a la invocadora.

Un aspecto muy importante en la utilización de la pila es que los datos locales que se manejan no ocupan espacio en memoria mientras la función no entre en ejecución. Este factor es crítico en la reducción de la ocupación espacial de todo programa. Si cada una de las variables locales se almacenara de forma explícita en memoria, el coste espacial sería varios órdenes de magnitud mayor que utilizando una pila.

En el momento en que una secuencia de instrucciones haga uso de datos locales, se debe asociar una pila de ejecución. De ahí que, si se pretende tener varios flujos de ejecución, lo usual es necesitar varias pilas[3]. El sistema operativo asigna un espacio de pila a cada tarea. Cuando se realiza un cambio de contexto, se debe almacenar el puntero a pila de la tarea saliente, para poder recuperar su estado en el momento en que vuelva a ejecutarse.

En los Blackfin, existe un registro que mantiene el puntero a pila. En este caso, el uso de un registro del procesador para mantener la pila es muy interesante, y permite que los Blackfin implementen operaciones en ensamblador para el manejo de la pila. Dichas operaciones mejoran la semántica de la programación. Por otro lado, la implementación de estas instrucciones hace que el manejo de la pila desde código máquina sea muy cómodo. Un apilamiento consta de una guarda y un decremento, que son dos operaciones diferentes. Estas dos operaciones se realizan en un único ciclo de reloj gracias a las instrucciones de manejo de pila. Un ejemplo del problema que supone que el procesador no tenga pila explícita se tiene en la arquitectura PowerPC. Este procesador no dispone de un registro específico para mantener la pila. En lugar de ello, IBM ha definido un estándar conocido como *Embedded Application Binary Interface* (EABI) que define la forma en que se deben utilizar los registros de propósito general. Así, uno de los registros queda reservado para ser utilizado como puntero a pila.

4.2. Mapas de bits

Dentro del núcleo, los *mapas de bits* o *bitmaps* juegan un papel crucial en la reducción de los tiempos de ejecución de las llamadas al sistema. Mediante un bitmap, se indexa cada posible instancia de un tipo de elemento del sistema operativo. A través de operadores binarios, se puede reducir el tiempo de búsqueda a $\Theta(log_2(n))$, siendo n el número máximo de instancias del mismo tipo (o bien,

[3] En [13] se describe una política de planificación para sistemas de tiempo real que permite compartir la pila entre las tareas en ejecución.

el número de instancias mapeadas). En concreto, el operador busca el primer bit activo dentro de un bitmap. La figura 4.2.1 muestra la forma en que funcionaría la búsqueda del primer bit activo de un bitmap, para un mapa de 4 bits. El procedimiento sería exactamente el mismo para cualquier tamaño de mapa de bits potencia de 2, dividiendo cada vez el bitmap, e incrementando el índice de forma adecuada.

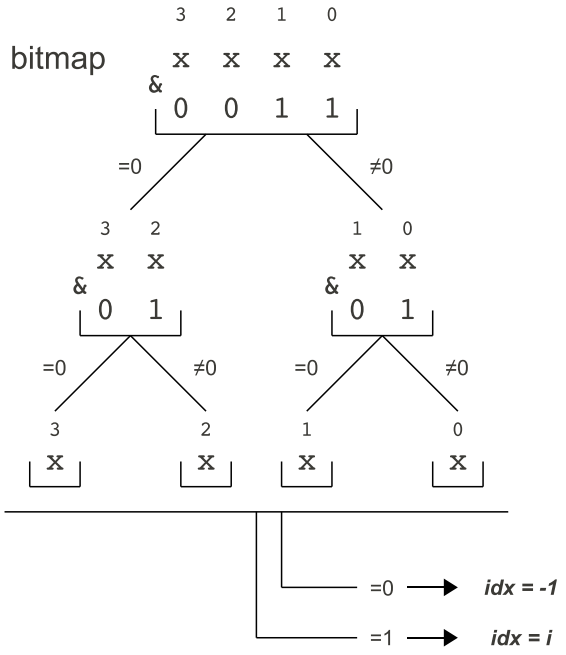

Figura 4.2.1: Búsqueda del primer bit activo en un bitmap.

Los elementos que se indexan con mapas de bits son tanto las tareas como los objetos de sincronización. Los elementos del mismo tipo se mantienen en un vector, indexado por un bitmap. Este bitmap sirve para dar cuenta de qué instancias se encuentran reservadas y cuáles libres. La especificación ITRON define que cada uno de estos elementos ha de ser referenciado mediante un identificador. De acuerdo con la especificación de ITRON, el identificador debe estar comprendido entre 1 y 256, con lo que se interpreta que pueden haber hasta 256 instancias de cada elemento. Sin embargo, debido a las restricciones del procesador, en la implementación se ha acotado a 32 identificadores, con lo que el máximo de elementos que pueden crearse en el sistema está limitado respecto de la especificación estándar. Se considera que, a pesar de esta limitación, el diseño básico sigue respetándose, ya que el sistema no sería capaz de almacenar un número de instancias tan elevado

como el que se quiere representar. En cualquier caso, está pendiente de desarrollo la posibilidad de dar soporte al número completo de instancias del sistema.

4.3. Tareas[4]

El objetivo de todo sistema operativo es el de gestionar múltiples tareas, asignando a cada una un tiempo de procesador cuando procede. Una tarea se gestiona a través de una estructura de datos que contiene todos los parámetros relativos a su estado. A través de la interfaz ITRON, se gestionan algunas de estas variables.

Además de las tareas de usuario, el núcleo utiliza dos hilos adicionales: el hilo ocioso y el principal. El hilo ocioso entra en ejecución únicamente cuando no hay ninguna tarea en la cola de hilos preparados. El hilo principal es el que contiene la rutina `main`, que es la primera que se ejecuta en toda aplicación. El usuario deberá tener en cuenta el espacio que ocupan en memoria estos dos hilos.

Para crear una tarea en ObelISK, en primer lugar se reserva e inicializa la estructura que contiene toda la información relativa a la misma. A continuación se reserva el espacio de pila configurado en el paquete de creación de la tarea. Es de crucial importancia que el espacio de pila de una tarea sea suficientemente grande como para mantener el peor caso de apilamiento, es decir, el momento en que la pila contenga la mayor cantidad posible de datos apilados para esa tarea. De lo contrario, la pila podría crecer por debajo de la frontera de su espacio reservado y destruir datos del sistema operativo o de otras tareas.

Cuando se crea una tarea, el primer código que ejecuta es el código de inicialización de la tarea. Esta función envuelve a la función principal de la tarea por dos razones:

1. Pese a que en ITRON la función principal de una tarea no recibe parámetros, ha resultado necesario incluir esta posibilidad, ya que la pila TCP/IP instalada lo requería. Así pues, la función principal de una tarea puede recibir o no argumentos. La función de inicialización se encarga de detectar este hecho para invocar a la tarea con los argumentos adecuados.

2. Al finalizar una tarea se debe ejecutar necesariamente una de las dos funciones de finalización `ext_tsk()` o `exd_tsk()`. Estas funciones finalizan las tareas y liberan los recursos que están ocupando. En ITRON se deja a responsabilidad del usuario la invocación de estas funciones. Sin embargo, se ha considerado prudente que el código de inicialización también incluya la llamada, por si acaso el usuario olvidara liberar recursos. Esto no afecta en

[4] Se utilizará de forma indistinta los términos *hilo* y *tarea*.

modo alguno a la ejecución del código ya que, al invocar una de estas funciones, la tarea termina su ejecución de forma inmediata.

4.4. Objetos de sincronización

ITRON define una variedad de objetos de sincronización. De acuerdo con el perfil estándar, el sistema operativo sólo debe incluir semáforos, buzones y eventflags. Con estos tres elementos de sincronización se pueden implementar la mayoría de aplicaciones. Aún así, el problema de la inversión de prioridad no queda resuelto y, de acuerdo con ITRON, la corrección de este problema sería una característica extendida no requerida. ObelISK no implementa ningún mecanismo de evitación de inversión de prioridad. En relación al núcleo privativo VDK, los buzones son una característica adicional muy útil de comunicación entre tareas. Una de las limitaciones importantes de VDK es, precisamente, que carece de un mecanismo adecuado de comunicación entre hilos. Esto tiene cierta repercusión, como podrá verse en el capítulo 7.

4.4.1. Semáforos

El *semáforo* es la construcción básica de sincronización de un sistema operativo. Más que garantizar exclusión mutua, un semáforo suele utilizarse para gestionar la disponibilidad de los recursos. El Perfil Estándar de ITRON sólo contempla la implementación de semáforos, y no de mutexes. En cualquier caso, el semáforo es una construcción básica útil para garantizar exclusión mutua en secciones críticas.

Un semáforo consta de un contador y dos operaciones. Al adquirir un semáforo, el contador se decrementa en una unidad. Si el contador es mayor que cero, entonces la tarea continuará su ejecución. Si el contador ya valía cero en el instante en que se intenta adquirir el semáforo, la tarea quedará bloqueada a la espera de que alguien libere dicho semáforo.

4.4.2. Buzones

Un *buzón* es un mecanismo de comunicación entre hilos. Un hilo puede dejar un mensaje en un buzón. Otro puede leerlo y, si hay un mensaje, lo recogerá; en caso contrario el hilo consumidor quedará bloqueado a la espera de un mensaje. La funcionalidad es la de un modelo productor-consumidor.

La figura 4.4.1 muestra el funcionamiento de un buzón cuando una tarea inserta un mensaje y hay varios hilos bloqueados en espera. Un buzón tiene una cola de hilos que están esperando la llegada de un mensaje. Cuando una tarea inserta un mensaje en un buzón, se recoge la tarea en la cabeza de la cola y se le entrega

el mensaje, con lo que la tarea continúa su ejecución. El resto de tareas deberán esperar la llegada de más mensajes.

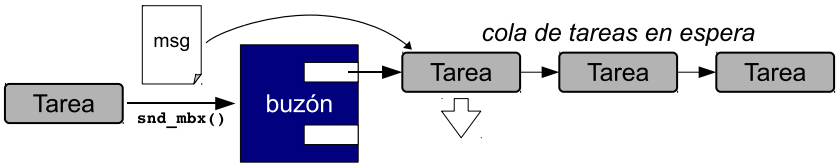

Figura 4.4.1: Buzón con tareas en espera.

Si en el momento de enviar un mensaje a un buzón no hubiera ninguna tarea en espera, el buzón lo guarda en la cola de mensajes. Cuando una tarea lee de un buzón que tiene mensajes, recoge el mensaje en la cabeza de la cola y continúa su ejecución (figura 4.4.2).

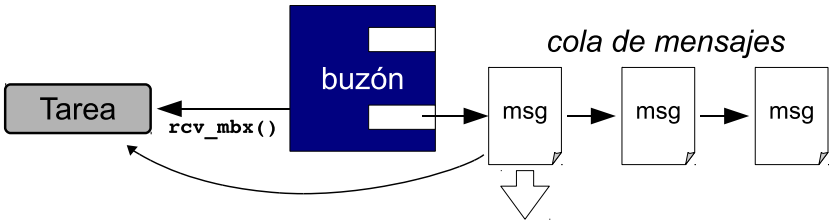

Figura 4.4.2: Buzón con mensajes en espera.

4.4.3. Eventflags[5]

Un *eventflag* es un mapa de bits que produce el disparo de un evento cuando su valor se corresponde con un cierto patrón. Los eventflags son muy útiles como mecanismo para reflejar el estado de un elemento del sistema, o bien la ocurrencia de un evento. Por ejemplo, si se ha creado una tarea para adquirir imágenes de una cámara, debe esperar a que la cámara haya sido inicializada. Aquí, la tarea esperaría a un eventflag que reflejaría el estado de inicializada/no inicializada de la cámara. Mientras la cámara no se hubiera inicializado, el eventflag seguiría a cero.

Un eventflag funciona tal y como puede verse en la figura 4.4.3. Cada tarea espera un patrón de eventflag específico, por lo que, dependiendo de qué bits se activen en un mismo eventflag, serían unas tareas u otras las que cambiarían su estado. Esta es la principal diferencia con el resto de objetos de sincronización.

[5]El término *eventflag* tiene una traducción bastante pobre, por lo que se utilizará su versión inglesa.

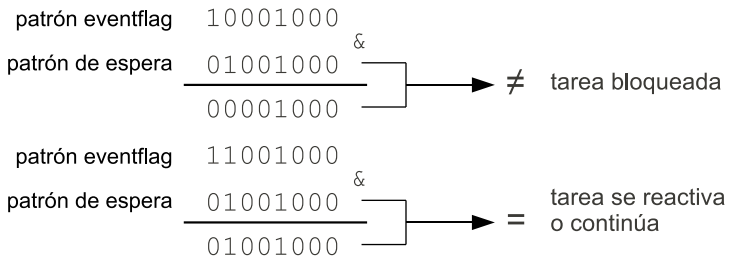

Figura 4.4.3: Funcionamiento de un eventflag

4.5. Planificador

El planificador es el encargado de gestionar la cola de hilos preparados y activar el hilo de mayor prioridad para su ejecución. La cola de hilos preparados se gestiona por niveles de prioridad. En cada nivel de prioridad hay una FIFO[6], de modo que, de entre las tareas encoladas en un mismo nivel, tiene más prioridad la primera que se activó.

Esta forma de gestionar la cola de hilos preparados está definida en la especificación ITRON. Permite planificación con prioridades tanto estáticas como dinámicas. Sin embargo, esta forma de gestionar la cola de hilos preparados no es la más eficiente en algunos casos. Por ejemplo, si se implementara un planificador EDF, sería mejor utilizar un heap para la gestión de la cola de hilos preparados. Aunque ITRON no utiliza otros algoritmos de planificación, en ObelISK se ha contemplado la posibilidad de implementar un planificador EDF.

La inversión de prioridad es un problema que, actualmente, es susceptible de ocurrir, ya que el Perfil Estándar de ITRON no contempla ningún mecanismo para corregirla. Para ello habría que utilizar los mutexes contemplados en la definición del conjunto completo de ITRON. En cualquier caso, se ha dejado abierta una puerta para poder implementar una gestión de recursos basada en *Stack Resource Policy* (*SRP*) [13], capaz de resolver éste y otros problemas de gestión de recursos en sistemas de tiempo real.

4.5.1. La cola de hilos preparados

La cola de hilos preparados es una estructura formada por un vector de colas de tareas (figura 4.5.1). Cada posición del vector representa un nivel de prioridad, de forma que las más prioritarias tienen valores más bajos. Dos tareas pueden tener

[6]Del inglés, *First In First Out*. Este término se utiliza para referirse a una cola.

el mismo nivel de prioridad; en este caso, el orden de ejecución viene determinado por el orden en que las tareas se activaron para su ejecución.

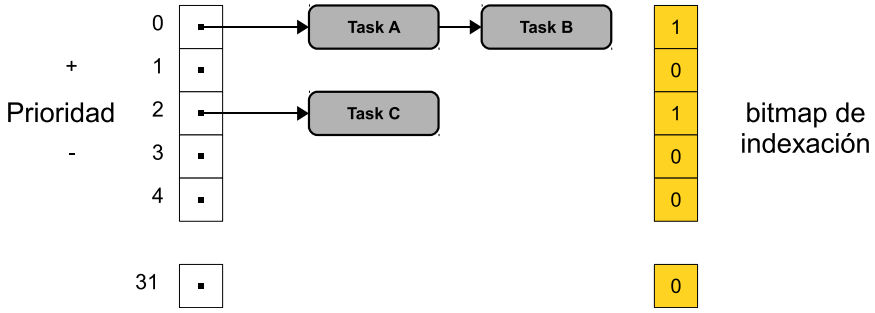

Figura 4.5.1: Cola de hilos preparados.

Cuando se invoca al planificador, éste debe buscar en la cola de hilos preparados a la tarea de mayor prioridad. Para ello, debería recorrer todas las posiciones del vector de colas de tareas en orden creciente para encontrar la tarea más prioritaria. Para evitar buscar cada vez entre 31 elementos, se ha colocado el bitmap de indexación, que sirve para reducir el tiempo de búsqueda, tal y como se ha comentado en la sección 4.2.

4.5.2. Cambio de contexto

La implementación del cambio de contexto es muy interesante y se realiza en tres pasos, como puede verse en las figuras 4.5.2, 4.5.3 y 4.5.4.

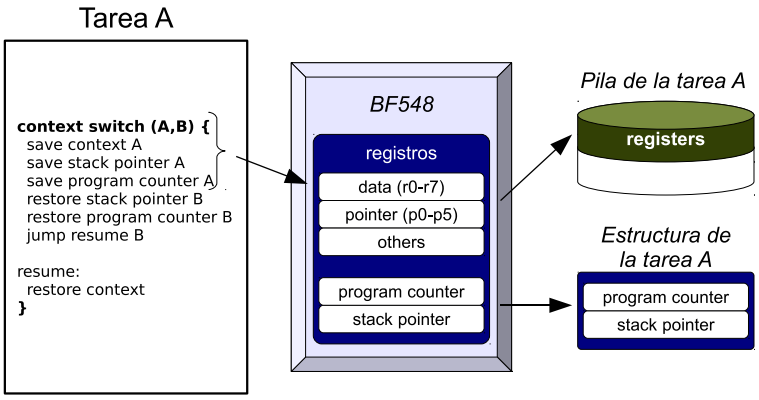

Figura 4.5.2: Cambio de contexto, paso 1. Guardar contexto y punteros de pila y contador de programa de la tarea saliente.

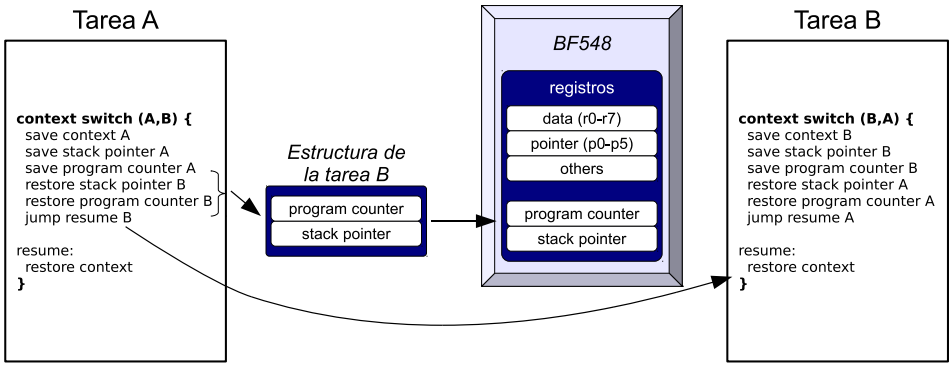

Figura 4.5.3: Cambio de contexto, paso 2. Restaurar los punteros de pila y contador de programa de la tarea entrante y saltar a su código.

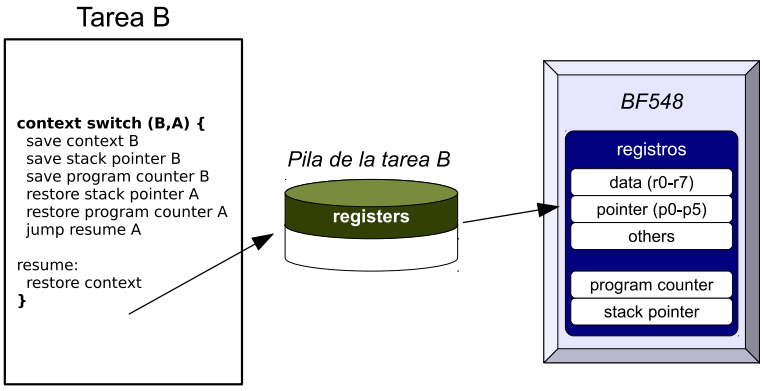

Figura 4.5.4: Cambio de contexto, paso 3. Restaurar el contexto de la tarea entrante.

Cuando una tarea sale del procesador, al guardar el contador de programa, almacena la dirección de la etiqueta `resume`. Es precisamente desde esta etiqueta desde donde continúa la ejecución de la tarea, restaurando el contexto.

4.6. Memoria

Existen dos paradigmas de gestión de la memoria en el desarrollo de toda aplicación: memoria *estática* y memoria *dinámica*.

4.6.1. Memoria estática

La memoria *estática* es toda aquella memoria asignada en tiempo de compilación. Desde el punto de vista del lenguaje C, la memoria estática se corresponde con todos los buffers que se han definido en el código fuente. Por ejemplo, una declaración del tipo

```
int vector[256];
```

es una forma de gestión de memoria estática, puesto que este buffer no cambiará su ubicación o tamaño durante toda la ejecución del programa.

Toda la memoria estática se distribuye en tiempo de compilación, mediante el uso de unos ficheros de configuración llamados Linker Description Scripts (sección 5.4). Debido a las características de la arquitectura de los Blackfin, es necesario realizar una distribución de forma inteligente entre las memorias ubicadas en los dos dominios de reloj. Para obtener el máximo rendimiento, lo ideal sería ubicar los buffers a los que se accede más frecuentemente en las memorias internas, que funcionan a la frecuencia más rápida. Es responsabilidad del desarrollador de la aplicación final obtener el máximo rendimiento del procesador, por lo que en el desarrollo del sistema operativo se ha dado menor importancia a la distribución en memoria de los datos y el código del núcleo. Generalmente, los sistemas operativos funcionan con un esquema de memoria rígido; se ubica el código del núcleo en un trozo reservado de la memoria y las aplicaciones del usuario se enlazan en otras zonas preparadas a tal efecto. Sin embargo, debido a las características de las aplicaciones que se van a desarrollar, se ha dejado a disposición del usuario el LDS que enlaza la aplicación final, haciendo posible reubicar el código del sistema operativo si la aplicación así lo requiriese. De esta forma, el desarrollador puede obtener el máximo rendimiento en las aplicaciones, sin tener que hacer frente a un esquema rígido de memoria.

4.6.2. Memoria dinámica

El núcleo de ObelISK da soporte a memoria dinámica principalmente basado en el TLSF [2]. Este gestor de memoria dinámica permite servir bloques de memoria de tamaño variable en tiempo acotado. Un gestor de memoria dinámica es básicamente un algoritmo de gestión de huecos de memoria libre. Ante una petición de un bloque de memoria, se debe buscar entre todos los bloques libres uno que satisfaga al menos el tamaño de la petición. Se puede refinar esta búsqueda y reservar la memoria en uno de los pedazos libres que cumpla una serie de condiciones.

A la elección de qué bloque de memoria se utiliza para servir la petición se

denomina política, estando el TLSF basado en una política *good-fit*[7]. Por norma general, el rango de tamaños de bloque que utiliza una aplicación de tiempo real está acotado, por lo que una política best-fit tiende a producir la menor fragmentación posible [14]. Este hecho es de suma importancia, y es una de las razones que han llevado a la implementación de este núcleo. La librería de gestión de memoria dinámica que ofrece Analog Devices se basa en la política *next-fit*[8], lo cual supone que ObelISK ofrece una mejor gestión de memoria, no sólo en términos de fragmentación, sino también al ser capaz de manejar memoria dinámica en tiempo predecible. La búsqueda del siguiente bloque libre que realiza la librería de Analog no está acotada en tiempo, por lo que vulnera las restricciones de tiempo real.

En el campo de la gestión de memoria dinámica, se utiliza el término *heap* para definir el bloque de memoria física que se utiliza para servir peticiones de bloques de tamaño variable. Asimismo, se utiliza el término *memory pool* para definir una región de memoria en la que se servirán peticiones de bloques de tamaño fijo. Sin embargo, en algorítmica existe una estructura de datos también conocida como *heap* (ver sección 4.8), por lo que se puede producir cierta confusión en la utilización de estos términos. Es más, en la implementación de este núcleo, se utiliza un heap para mantener las tareas que se encuentran suspendidas con un timeout. Para evitar ambigüedades en el uso de estos términos, se define la siguiente terminología, que se utilizará a lo largo de todo el documento:

- Se utilizará el término *vmpool* (del inglés, *variable-size block memory pool*) para referir la región de memoria que se utilizará para servir bloques de tamaño variable.
- Se utilizará el término *fmpool* (del inglés, *fixed-size block memory pool*) para referir la región de memoria que se utilizará para servir bloques de tamaño fijo.
- Se utilizará el término *heap* para referir a la estructura de datos en forma de árbol binario.

Tanto las tareas como los objetos de sincronización tienen una representación similar en el núcleo, y se gestionan de forma similar. Por cada elemento, se mantiene

[7] *Good-fit* es una política en la que, ante una petición, se intenta utilizar el bloque libre de mínimo tamaño que satisface la petición, aunque no siempre lo conseguiría. La política se considera una aproximación de *best-fit*, que se basa en servir el bloque libre de mínimo tamaño siempre.

[8] En la política *next-fit* se mantiene una lista enlazada de bloques libres. Cuando se hace una petición, se recorre la lista desde la posición actual para encontrar el primer bloque de tamaño mayor o igual que el solicitado. Debido a que el número de bloques en que puede quedar dividida la memoria no se conoce, un gestor de memoria dinámica basado en next-fit no puede acotar sus búsquedas en el tiempo.

una estructura que da constancia de su estado. Actualmente, el núcleo se ha desarrollado de forma que se define un número máximo de instancias de cada elemento. Se hace así, entre otras cosas, para tener un comportamiento predecible del sistema. Durante la inicialización, se reserva memoria para todas las instancias de todos los elementos que el usuario ha preconfigurado. Teniendo en cuenta la forma en que se gestiona la memoria dinámica por parte del TLSF, las instancias quedan apiladas en las direcciones más bajas de la memoria reservada para el vmpool de sistema (figura 4.6.1). Esto permite que el núcleo arranque con la mínima fragmentación en el vmpool de sistema, a costa de un cierto desperdicio de memoria.

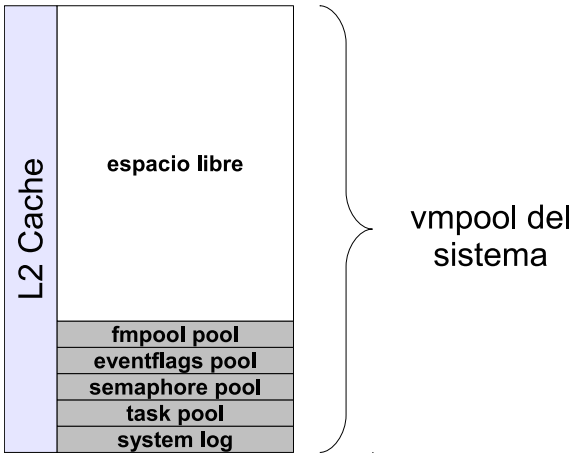

Figura 4.6.1: Estado del vmpool del sistema tras la inicialización.

4.7. Eventos

Desde el punto de vista del sistema operativo, pueden distinguirse dos dominios de operación: el de los hilos y el de las interrupciones (figura 4.7.1). El dominio de los hilos es el modo de operación normal. Al tratarse de un nanokernel, la interrupción software se encuentra siempre activa, de modo que el sistema se ejecutará en el contexto de los hilos mientras no haya ninguna otra interrupción activa. Como es la interrupción de menor nivel, cualquier otra interrupción que se produzca será más prioritaria y, por lo tanto, será su manejador el que se ejecute. Se pasa al dominio de las interrupciones al activarse una interrupción y ejecutarse su manejador.

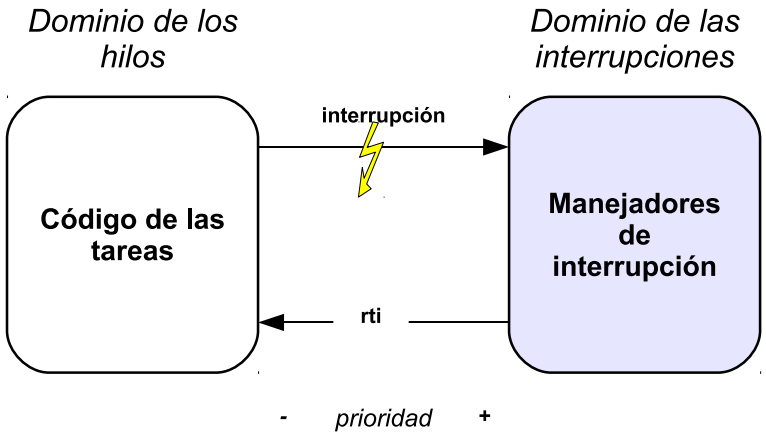

Figura 4.7.1: Dominios de ejecución del núcleo.

4.7.1. Interrupciones

El núcleo gestiona varios tipos de interrupciones. El manejo de las interrupciones críticas es específico de cada tipo de interrupción. En cuanto a las interrupciones que generan los periféricos, el sistema operativo virtualiza la tabla de vectores de eventos. En lugar de acceder directamente a los registros del hardware, el manejador de interrupción de usuario se instala en un vector de punteros a función. En cada una de las entradas del vector de eventos, se instala una *rutina envoltorio*[9] común a todas las interrupciones. En la figura 4.7.2 puede verse en detalle cómo se manejan las interrupciones.

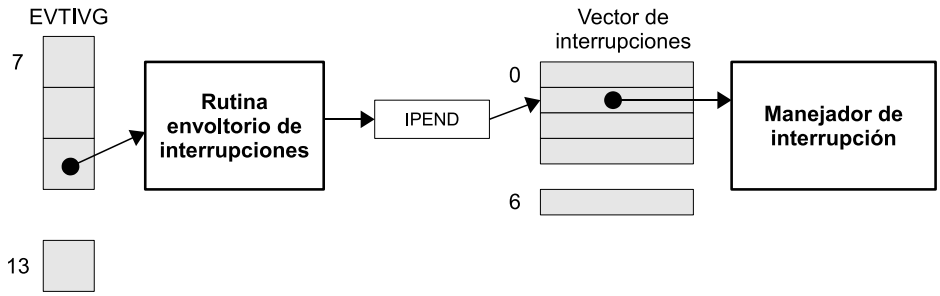

Figura 4.7.2: Estructura del código de gestión de interrupciones.

La rutina envoltorio tiene dos objetivos: gestionar el contexto y proteger el do-

[9] Se suele utilizar el término inglés *wrapper*.

minio de las interrupciones. La gestión del contexto es necesaria, aunque responde, más bien, a una cuestión de usabilidad. Al ser el sistema operativo el que proporciona estas dos funciones, el usuario simplemente instala la función para manejar la interrupción y no se preocupa del protocolo subyacente de gestión del contexto. Por supuesto, guardar el contexto es un paso necesario, tanto si se utilizan registros del procesador que puedan afectar al retorno del manejador, como si se van a utilizar interrupciones anidadas[10].

La protección frente a cambios de contexto en el dominio de las interrupciones es una funcionalidad crítica. Al producirse una interrupción, el procesador entra en un modo más prioritario. En este momento, hay que desactivar el planificador. Si el manejador de interrupciones invoca una función que produce un cambio de contexto, se produciría un salto de dominio, y el flujo de ejecución pasaría a estar en el dominio de los hilos, pero con prioridad elevada debido a la interrupción que sigue activa.

Para proteger el sistema de los saltos de dominio, se utiliza una variable global que se modifica de forma atómica. A la entrada del manejador, se incrementa esta variable en una unidad y, a la salida, se decrementa. De este modo, en tanto el valor de la variable sea mayor que cero, el planificador no realizará ningún cambio de contexto.

Una ventaja adicional de realizar la gestión de las interrupciones con una misma rutina envoltorio es la reducción del tamaño del código. Cada vector apunta a una función común, en lugar de tener cada uno su propio manejador. Al tener todas las interrupciones el mismo punto de entrada, la rutina envoltorio tiene que determinar la causa de la interrupción antes de llamar al manejador. Esto supone una carga computacional extra en la gestión de las interrupciones. Sin embargo, y como es habitual, hay que llegar a un compromiso entre latencia de ejecución y ocupación en memoria. El usuario tiene acceso a la tabla de eventos durante la ejecución de las aplicaciones, por lo que podría utilizar una gestión propia si así lo creyera conveniente. Por último, es totalmente desaconsejable esta práctica, a no ser que el usuario disponga de los suficientes conocimientos como para evitar situaciones como la de salto de dominio, descrita anteriormente.

4.7.2. Excepciones

El manejo de las excepciones por parte del núcleo se realiza de idéntica forma al manejo de interrupciones, tal y como puede verse en la figura 4.7.3. A diferencia

[10]Cuando se produce una interrupción, el procesador inhibe todas las interrupciones. El usuario puede reactivarlas de forma que, mientras se ejecuta un manejador de interrupción, se permita servir interrupciones de mayor prioridad en el caso de que se produjeran. El término anidada se refiere precisamente a este hecho.

de las interrupciones, las excepciones se producen en situaciones de error, por lo que el control de los saltos de dominio no se tiene en cuenta. Actualmente, no se ha implementando el mecanismo de gestión de manejadores de excepción, es decir, una API para que el usuario pueda instalar manejadores. No obstante, se han implementado los manejadores de excepción para las excepciones de fallo de CPLB[11], los cuales reemplazan los descriptores cuando se accede a una página de memoria no descrita en ningún CPLB.

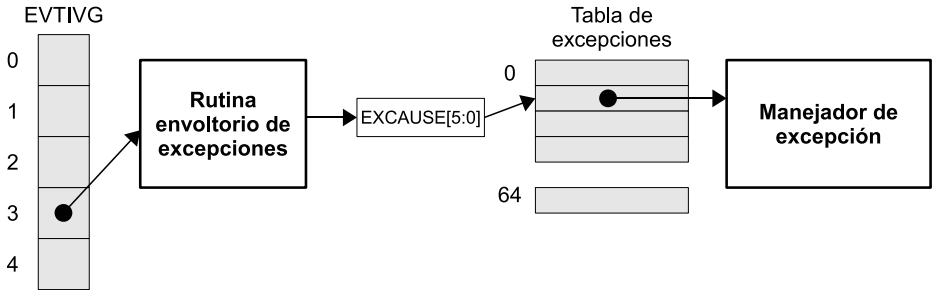

Figura 4.7.3: Manejo de excepciones en el núcleo.

4.8. Timeout heap

Un *heap* es una estructura de datos en forma de árbol binario, que tiene una cierta ordenación. Este árbol consta de nodos, cada uno de los cuales tiene una clave con un valor numérico. En general, suponiendo que se tiene dos nodos, A y B, y que $A = padre(B)$, entonces $clave(A) \leq clave(B)$. La definición anterior describe un *minheap*[12], que es un heap en el que el nodo cuya clave tenga el valor más pequeño se encontrará siempre en la raíz. La figura (4.8.1) muestra un ejemplo de heap. Un heap es un árbol binario completo, lo que significa que cada nivel del árbol, excepto el último posiblemente, se encuentra totalmente ocupado. Debido a la implementación de esta estructura, el orden de lectura de los nodos es de arriba a abajo y de izquierda a derecha[13].

[11] Los fallos de CPLB se manejan por separado para páginas de código y páginas de datos.

[12] De igual forma, si se cambia la desigualdad, se podría construir un *maxheap*, en el que la raíz sería el nodo cuya clave tendría el valor más alto de todo el árbol.

[13] En la figura 4.8.1 el orden de lectura sería 1, 5, 25, 18, 10, 32, 63.

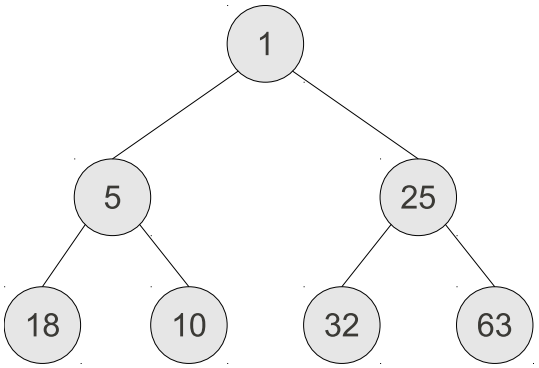

Figura 4.8.1: Ejemplo de minheap.

Un heap soporta las operaciones de inserción y extracción. La inserción se realiza colocando el nuevo nodo en la última posición del heap. En el ejemplo anterior, el nuevo nodo se insertaría como una nueva hoja debajo del nodo cuya clave es 18. Cuando se inserta un nodo en un heap, debe realizarse una operación que se conoce como *heapify*. La operación de *heapify* sirve para recuperar la condición de minheap en el árbol. Por ejemplo, si el nuevo nodo insertado tuviera un valor de 20, entonces no sería necesario reordenar el heap. En cambio, si la clave del nodo insertado es 3, se debe proceder a reordenar el heap. La forma de reordenar la estructura consiste en intercambiar cada nodo por su nodo padre, en caso de que se incumpla la propiedad de heap, en dirección a la raíz. Volviendo al ejemplo, si se inserta un nodo con clave 3, entonces, la operación de heapify actuará tal y como puede verse en la figura 4.8.2.

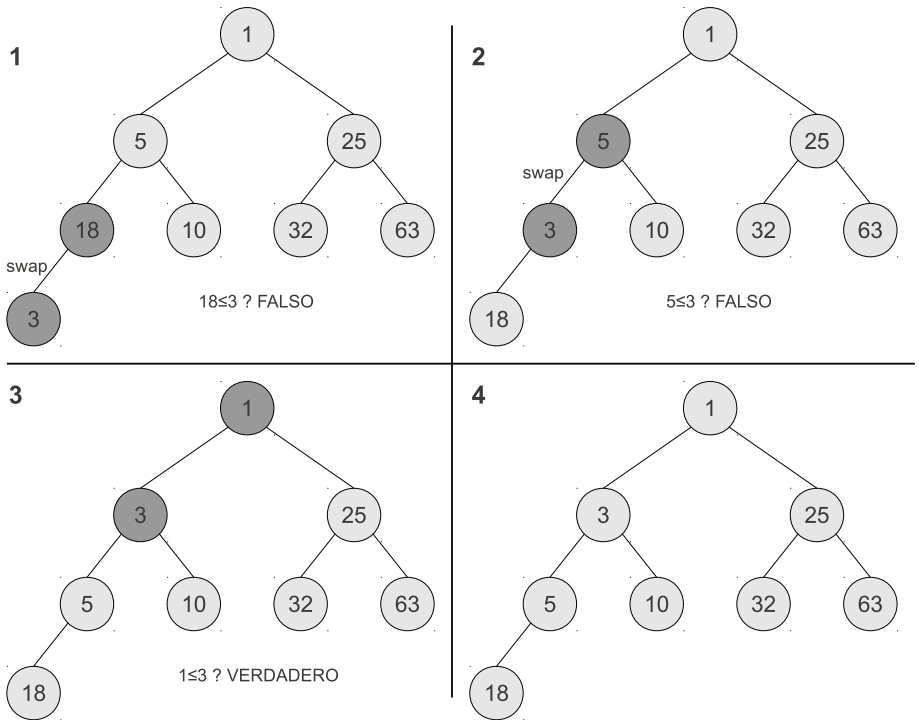

Figura 4.8.2: Inserción en heap y *heapify*.

Si se observa el árbol anterior, ha sido posible insertar un nodo en un árbol de 8 elementos y ordenarlo en tan solo 3 pasos. Debido a su carácter binario, el coste computacional de las operaciones de inserción y extracción es de $\Theta(log_2(n))$.

De forma análoga, para la operación de extracción del heap se recoge la raíz y se reordena, pero esta vez recorriendo el heap hacia las hojas del árbol. Esta operación puede verse en la figura 4.8.3. La operación de extracción del heap tiene el mismo coste que la operación de inserción, por lo que es también una operación muy eficiente.

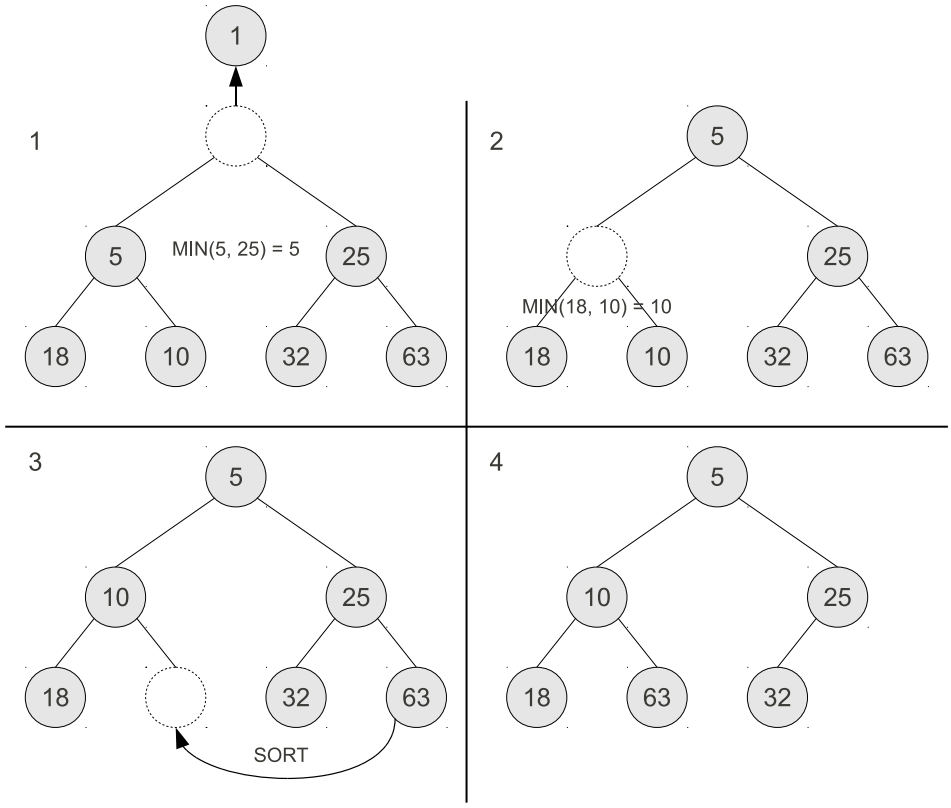

Figura 4.8.3: Extracción del heap y *heapify*.

En ObelISK se utiliza un minheap para acelerar el tratamiento de las tareas que se encuentran suspendidas con un cierto timeout. Cada nodo representa una tarea, y su clave es el tiempo que queda para que el temporizador de la tarea expire. De este modo, cada vez que se ejecuta el temporizador del núcleo, se comprueba la tarea en la raíz del timeout heap para ver si ha expirado su temporizador. De ser así, la tarea se despierta, se extrae del timeout heap y se inserta en la cola de hilos preparados.

4.9. Log del sistema

Se ha instalado un pequeño logger de eventos de sistema. Este logger está pensado para poder seguir el flujo de ejecución en depuración, lo cual puede resultar imprescindible para encontrar errores en las aplicaciones. Una de las premisas en

el desarrollo del log ha sido minimizar la ocupación espacial de la tabla de eventos.

El log no mantiene constancia a cada instante del estado del sistema. Únicamente se almacenan eventos que pueden producir un cambio de estado en el sistema como, por ejemplo, adquirir un semáforo o bien crear una nueva tarea. Cada mensaje consta de una serie de parámetros, y no se utiliza texto, sólo datos binarios. Los campos de cada mensaje pueden verse en la tabla 4.1.

	Descripción	Tamaño (B)
time	Instante en que se produjo el evento	4
event	Código de evento	2
arg1	Objeto de sincronización o tarea a que afecta el evento	2
arg2	Parámetro del evento	2
arg3	No utilizado	2

Cuadro 4.1: Estructura de un mensaje de log.

El log del sistema es un vector circular de mensajes. El número total de mensajes que contiene el log se configura junto con el núcleo. Actualmente, se ha limitado a 128 eventos, aunque este número debería variar en función del número de eventos que se espera por unidad de tiempo. Si en una aplicación la tasa de eventos es muy elevada, el espacio temporal que representará el log del sistema será menor. Esto hay que tenerlo en cuenta puesto que, si el número de eventos que puede registrar el log es demasiado pequeño, podría no ser útil. De forma análoga, un log que registre demasiados eventos podría provocar un desperdicio de memoria.

4.10. Protección de secciones críticas

Una *sección crítica* es una secuencia de instrucciones que modifica datos globales utilizados por dos o más hilos de ejecución. El principal problema de la concurrencia son las condiciones de carrera. Una *condición de carrera* se produce cuando el resultado de la ejecución de un conjunto de instrucciones pertenecientes a una sección crítica no es el esperado. Esto se debe a que este resultado depende de la ejecución secuencial de las instrucciones, que se puede haber roto debido a la concurrencia.

Para ilustrar este problema, se muestra el ejemplo de la figura 4.10.1. En este ejemplo, *a* y *b* son variables compartidas a las que acceden las tareas M y N, por lo que las zonas gris oscuro de ambas tareas son secciones críticas. En (1), la tarea A se encuentra en ejecución. De acuerdo con la secuencia de instrucciones, la tarea M espera que al finalizar la sección crítica *a=5* y *b=10*. Sin embargo, en (2), justo

después de asignar su valor a *a*, se produce una interrupción y comienza a ejecutarse la tarea N. En (3), esta tarea modifica los valores de las variables compartidas, dejándolas en *a=3* y *b=8*. Al volver a ejecutarse la tarea M en (4), continúa modificando el valor de *b*, pero como ya había modificado *a*, no hace nada con esta variable. En este ejemplo, la tarea N ha obtenido el resultado correcto de la secuencia de instrucciones de la sección crítica. En cambio, al salir de la sección crítica, la tarea M ha obtenido los valores de *a=3* y *b=10*, los cuales no son los esperados.

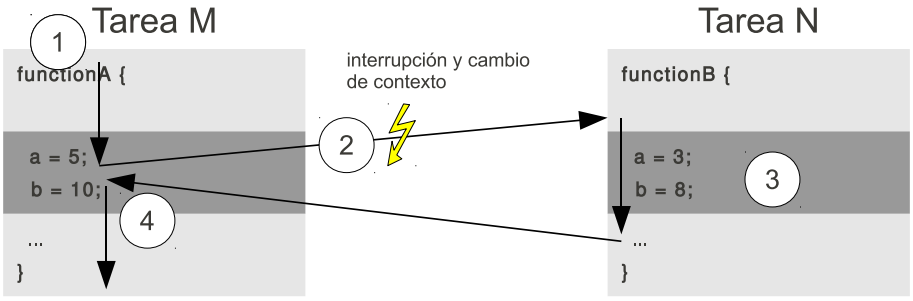

Figura 4.10.1: Condición de carrera.

Una condición de carrera es sumamente difícil de detectar. Es más, pueden existir secciones críticas realmente sutiles. Blackfin es una arquitectura de 32 bits. El tiempo del sistema se mide con una variable de 64 bits llamada system_time. Cuando se incrementa esta variable, el código C necesario es:

```
system_time = system_time + 1;
```

El problema surge cuando este código se compila y se genera las instrucciones reales que ejecuta el procesador. Debido a que es una arquitectura de 32 bits, las variables de mayor tamaño deben operarse mediante registros de 32 bits. En el caso de Blackfin, la anterior instrucción genera el siguiente código ensamblador:

```
1  R0 = [FP+-8];
2  R1 = [FP+-4];
3  R0 += 1;
4  cc = ac0;
5  R2 = cc;
6  R1 = R1 + R2;
7  [FP+-8] = R0;
8  [FP+-4] = R1;
```

Como puede verse en el listado anterior, la instrucción que parecía única en C se compila como una secuencia de instrucciones relativamente larga en ensambla-

dor. En las líneas 1 y 2 se recoge el valor de la variable y se guarda en dos registros de 32 bits. En la línea 3 se incrementa en 1 el valor del registro R0, que representa los 32 bits de menor peso de la variable. En las líneas 4, 5 y 6 se comprueba si, al incrementar el valor del registro R0, se ha producido un desbordamiento, en cuyo caso hay que incrementar el valor del registro R1, que contiene los 32 bits de mayor peso. Finalmente, se vuelve a guardar el valor en memoria[14].

A nivel de usuario, la forma de proteger secciones críticas es mediante el uso de semáforos. Sin embargo, dentro del código del núcleo también existen secciones críticas, pero no se pueden proteger con semáforos, ya que no existen a este nivel. Una condición de carrera se produce porque una interrupción ha cambiado el flujo de ejecución dentro de una sección crítica. Por lo tanto, para proteger los accesos a memoria compartida, se desactivan las interrupciones del procesador a la entrada de la sección y se reactivan a la salida. De esta forma, el flujo secuencial de ejecución no puede romperse dentro de las secciones críticas.

Este mecanismo de protección tiene una contrapartida. Cuando una tarea invoca una función del núcleo, existe un lapso de tiempo en el cual las interrupciones se desactivan. Mientras esto sea así, cualquier tarea, por prioritaria que sea, deberá esperar a que las interrupciones vuelvan a estar activas. Por ello, la ejecución de instrucciones del núcleo puede incrementar los tiempos de ejecución de las tareas. En ObelISK se ha intentado reducir las secuencias de instrucciones que se ejecutan de forma atómica. Por ejemplo, cuando se invoca una función de la API de ITRON, el primer paso es comprobar que los argumentos son correctos. Los argumentos residen en la pila de la tarea y en los registros del procesador, con lo que no están accediendo a regiones críticas. Por lo tanto, no es necesario bloquear la llegada de interrupciones mientras se comprueban los parámetros.

Por último, la desactivación de las interrupciones debe realizarse de forma cuidadosa. Debido a los cambios de contexto, el núcleo puede ejecutar secuencias de instrucciones muy variopintas que, si no están correctamente gestionadas, pueden conducir a un bloqueo del sistema. De no ser así, podría quedar activo el hilo ocioso con las interrupciones desactivadas, quedando el sistema bloqueado.

[14] En este ejemplo, el dato es una variable local que reside en la pila.

Capítulo 5

Fase de Compilación

La compilación del núcleo es uno de los pasos más complejos. En este capítulo se comentarán las técnicas empleadas, que atañen al conjunto de herramientas utilizadas. Es necesario resaltar que algunas de estas técnicas tienen que ver con el lenguaje de programación C, que ha sido el empleado para programar el núcleo.

5.1. Ocultación de estructuras y símbolos

Una de las primeras cosas que hay que evitar es que el usuario tenga visibilidad sobre los símbolos y estructuras del núcleo. Esto es de vital importancia, ya que hay que evitar que el usuario modifique información del núcleo que podría dañar al sistema. Por ejemplo, una tarea se mantiene a través de una estructura. Esta estructura debería ser un tipo de dato oculto para el usuario, de forma que no pueda hacer uso de él. De lo contrario, podría modificar esta estructura y causar inestabilidad en el sistema. Por otro lado, si tuviera acceso a las variables que maneja el núcleo, podría modificarlas, con resultados catastróficos para el sistema.

Así pues, es necesario ocultar, en la medida de lo posible, toda la información sensible que maneja el núcleo. Para ello, se utiliza el preprocesador de C en combinación con las opciones del compilador GCC. El preprocesador se utiliza de dos formas con el mismo objetivo. La primera forma permite ocultar un fichero al usuario, normalmente un fichero de cabecera, que oculta una librería utilizada por el núcleo:

```
#ifndef _KERNEL_
#error  Internal file. Do not include it in your sources.
#endif
```

En el código anterior, si la directiva _KERNEL_ no está definida, entonces el compilador mostrará un error. Para que al compilar el código del núcleo el compilador no muestre este error, se utiliza el mismo con la opción -D. Esta opción permite definir una directiva en la compilación de un fichero. Así, para la compilación de todos los ficheros del núcleo, se utilizará un comando parecido al siguiente:

```
gcc [opciones de compilación] –D_KERNEL_
```

De este modo, al compilar los ficheros del núcleo, se indica al preprocesador de C que debe definir la macro _KERNEL_, con lo que el error mostrado anteriormente no se producirá.

La segunda forma de utilizar esta técnica es como sigue a continuación:

```
#ifdef _KERNEL_
/* Definición de símbolos ocultos */
#endif
```

A diferencia de la primera de las técnicas, en este caso se permite al usuario que haga uso del fichero de cabecera del núcleo, pero se oculta cierta información sensible, definida después de la directiva #ifdef.

5.2. Generación de desplazamientos para uso en ensamblador

El núcleo se implementa con una mezcla de código ensamblador y código C. Generalmente, es posible combinar ambos códigos sin problemas. Sin embargo, puede darse la situación en la que se necesite hacer uso de estructuras de alto nivel, definidas en C, desde código ensamblador. Para hacer más comprensible el procedimiento de generación de desplazamientos, se comentará con un ejemplo. Se tiene la siguiente estructura de datos:

```
struct ejemplo {
   int uno;
   int dos;
};
```

Suponiendo que cada entero es de 32 bits (4 bytes), esta estructura tiene una representación en memoria de 64 bits. Si la estructura se define en la dirección de memoria M, la variable uno estará alojada en la posición de memoria *M+0* y la variable dos estará en *M+4*. Cuando se accede desde C a esta estructura, no es necesario conocer estos detalles; simplemente se accede al campo deseado. El compilador se encarga de generar los desplazamientos necesarios en cada acceso.

El código ensamblador que se genera a partir de estos accesos lee y escribe en las posiciones de memoria *M+0* y *M+4*. En definitiva, para acceder a cada uno de los campos de una estructura desde código ensamblador, es necesario conocer los desplazamientos de cada uno.

5.2.1. Cálculo de los desplazamientos

Conocer los desplazamientos dentro de una estructura a simple vista no siempre es posible. Existen restricciones de alineamiento para los campos que dependen de la arquitectura que pueden modificar los desplazamientos. Para conocer de forma exacta los valores de los desplazamientos, es necesario conocer el compilador. Como esto supondría una tarea excesivamente compleja, los desplazamientos se calculan utilizando el propio compilador:

```
((struct ejemplo *)(0))->uno
((struct ejemplo *)(0))->dos
```

Lo que se hace es ubicar virtualmente la estructura en la posición de memoria *M=0*, y acceder al campo deseado. De este modo, al compilar y generar código ensamblador, el compilador se encargará de realizar los cálculos necesarios para acceder a las posiciones de memoria de cada uno de los campos, en este caso *0+0* y *0+4*.

5.2.2. Generación de desplazamientos

En el punto anterior, se ha analizado la forma de poder calcular los desplazamientos dentro de una estructura. El siguiente paso consiste en poder generar los valores de forma que se puedan utilizar desde código ensamblador. La generación de desplazamientos consiste en crear de forma automática un fichero de cabecera con la siguiente estructura:

```
#define EJEMPLO_UNO    0
#define EJEMPLO_DOS    4
```

En arquitectura Blackfin, se podría hacer uso de estas directivas de la siguiente forma utilizando el código de la arquitectura:

```
[p0+(EJEMPLO_UNO)] = p1;
```

En el ejemplo anterior, `p0` y `p1` son dos registros del procesador. La instrucción guarda el contenido de `p1` en la dirección de memoria `p0+EJEMPLO_UNO`. Sería equivalente a guardar un valor en el campo `uno` de la estructura `ejemplo`.

Para la generación de los desplazamientos se utilizan dos herramientas: el compilador y la herramienta *awk*. El compilador dispone de una opción (-S) que permite generar código máquina no en binario, sino la representación en texto de dicho código máquina. Por otro lado, se define una directiva (asm) para generar código en ensamblador con un formato específico. Nótese en el siguiente fragmento que la macro offsetof(st, m) es la que calcula el desplazamiento tal y como se ha explicado en el punto anterior.

```
#define offsetof(st, m) (((st *)(0))->m)
#define DEFINE(sym, val) \
        asm volatile("\n-> " #sym " %0 " #val : : "i" (val))
```

A continuación, se implementa una función en la cual se utilizarán las directivas anteriores.

```
1 void gen_offsets(void)
2 {
3     DEFINE(EJEMPLO_UNO, offsetof(struct ejemplo, uno));
4     DEFINE(EJEMPLO_DOS, offsetof(struct ejemplo, dos));
5 }
```

El significado salta a la vista. Se está definiendo el símbolo EJEMPLO_UNO como el desplazamiento calculado del campo uno de la estructura ejemplo (ídem con el campo dos). Esta función no implementa código ejecutable. Es una función vacía que se compila con la opción -S para generar código en ensamblador en formato texto con una estructura concreta. Al compilarla, se genera la siguiente salida:

```
-> EJEMPLO_UNO 0 offsetof(struct ejemplo, uno)
-> EJEMPLO_DOS 4 offsetof(struct ejemplo, dos)
```

En este momento es cuando entra en juego la aplicación awk. *awk* es una aplicación de tratamiento de cadenas de caracteres. Permite analizar cadenas de texto y aplicar ciertos tratamientos. La salida anterior se puede dividir en campos utilizando awk. Cada campo está delimitado por un carácter separador que, en este caso, es un espacio en blanco. Los campos que interesa recoger son el primero y el segundo. Estos campos se recogen, con el objetivo de generar el fichero de desplazamientos, y se imprimen en un fichero de cabecera, que será el que finalmente se utilice, con el formato deseado.

```
#define EJEMPLO_UNO    0
#define EJEMPLO_DOS    4
```

5.2.3. Uso de desplazamientos

La generación de desplazamientos tiene como objetivo poder hacer uso de algunas estructuras de datos definidas en C desde código ensamblador. Esta necesidad surge, por ejemplo, cuando se produce un cambio de contexto. Cuando se produce un cambio de contexto, dos entidades quedan involucradas: la tarea saliente y la tarea entrante. El cambio de contexto consiste en guardar el estado actual de la máquina para la tarea saliente y restaurar el estado en el que se encontraba la última vez que se ejecutó la tarea entrante.

Cada tarea mantiene el contexto con dos variables: el *puntero de pila* y el *contador de programa*. Cuando una tarea es expulsada, guarda el estado de todos los registros del procesador en la pila y a continuación se guarda el puntero de pila para poder restaurarlo cuando entre de nuevo en ejecución. Además, se guarda el contador de programa, es decir, la dirección de memoria desde donde debe continuar la ejecución del código de la tarea saliente.

Las dos variables mencionadas, el puntero de pila y el contador de programa, se almacenan en la estructura de datos que representa la tarea. Esta es una de las razones por las que resulta necesario generar los desplazamientos; el cambio de contexto se implementa en ensamblador y hay que acceder a la estructura de la tarea para poder guardar y recuperar el estado de la máquina.

5.3. Configuración del núcleo

El núcleo de ObelISK permite configurar distintas funcionalidades. La configuración del núcleo se aplica a través de directivas. Por ejemplo, la frecuencia del procesador se controla mediante las siguientes directivas:

```
#define CONFIG_CLKIN 25000000
#define CONFIG_CCLK 500000000
#define CONFIG_SCLK 125000000
```

Estas declaraciones se utilizan después en el código para calcular la frecuencia de reloj. En este caso, la configuración por defecto establece el reloj interno de la CPU a 500 MHz, y el reloj de sistema a 125 MHz. El reloj de entrada lo produce un oscilador a 25 MHz (del CM-BF548).

La configuración se almacena en un fichero de cabecera que contiene todas las declaraciones que configuran el núcleo. Este fichero de cabecera, denominado `autoconf.h`, se genera de forma automática a través de una aplicación denominada *menuconfig*. Esta aplicación se creó originalmente para configurar el núcleo de Linux. Para utilizarla se crear unos ficheros basados en un lenguaje de scripting

muy básico con el que se definen los menús y se pone nombre a las declaraciones. Debido a su facilidad de uso, se ha decidido utilizarla para generar el fichero de configuración de ObelISK. En la figura 5.3.1 puede verse el aspecto del menú principal de configuración de la aplicación menuconfig para ObelISK.

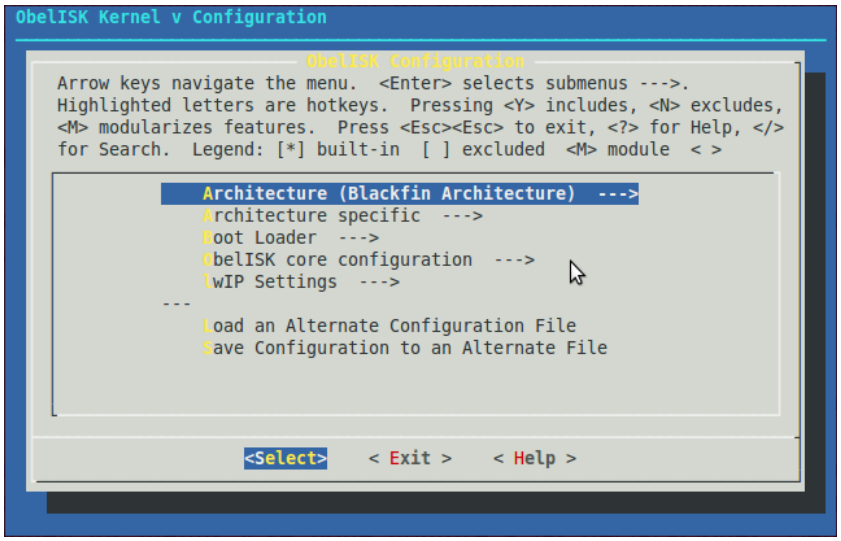

Figura 5.3.1: Ventana principal de la aplicación menuconfig para la configuración de ObelISK.

5.4. Enlace del código del núcleo

El procesador no interpreta símbolos, únicamente entiende datos binarios y posiciones de memoria. Debido a esto, cuando en una instrucción en ensamblador hace una llamada a una subrutina, debe conocer la dirección de memoria donde se inicia dicha subrutina. Por otro lado, el programador implementa las funciones utilizando nombres, no posiciones de memoria. Así, cada símbolo hace referencia a la posición de inicio de una función o de un buffer de datos. Estos símbolos no pueden resolverse (es decir, sustituirse por posiciones de memoria) hasta conocer la dirección de memoria donde se van a almacenar. El enlace de código consiste precisamente en esto, en ubicar el código y los datos en memoria y resolver los símbolos cuando se conocen las direcciones finales.

Normalmente, la compilación se hace en varios pasos. En primer lugar, se compila el código pero no se resuelven los símbolos. El código resultante no es código máquina ejecutable. Para resolver las referencias incompletas, se hace uso de la herramienta *ld*. ld es el enlazador de código de GNU. Esta herramienta hace uso de

un fichero de descripción de enlace (LDS), que sirve para definir la distribución del código y datos en memoria. La figura 5.4.1 muestra la forma en la que se enlaza el código.

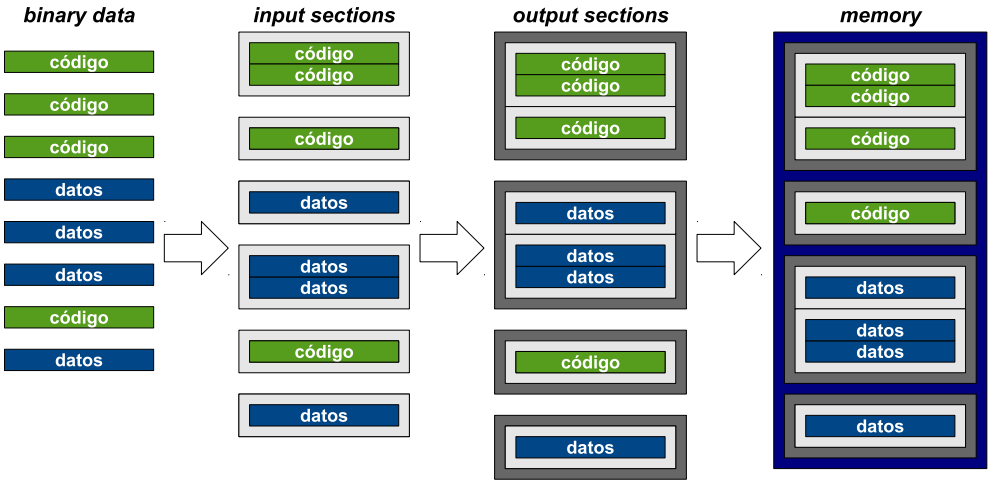

Figura 5.4.1: Enlace de código y datos en memoria.

Cada función y cada buffer se ubica en una *input section* o sección de entrada. Si no se define una sección de entrada para un trozo de código o un buffer determinados, se ubican en una sección de entrada por defecto. El código que no tiene ninguna indicación de dónde debe ubicarse se coloca en la sección de entrada *.text*. De forma análoga, los datos se pueden ubicar en tres secciones por defecto distintas:

- *.data*, para datos que se han inicializado a un valor en concreto.
- *.bss*, para datos para los cuales no se ha especificado un valor inicial.
- *.rodata*, que almacena datos de sólo lectura.

Las secciones de entrada se distribuyen entre las distintas *output sections* o secciones de salida, que son definidas por el usuario a conveniencia. Una sección de salida sólo puede contener código o datos, pero no una mezcla de ambos.

Uno de los retos que ha sido necesario superar es la posibilidad de establecer una configuración válida por defecto del enlace del código que, a su vez, se pudiera modificar de forma sencilla. Debido al tipo de aplicaciones objetivo a que va dirigido el procesador, la ubicación de los buffers en memoria influye en el rendimiento y, por lo tanto, es dependiente de la aplicación. Se ha establecido un mapa,

más o menos general y estándar, válido para todas las aplicaciones. La figura 5.4.2 muestra el mapa de memoria diseñado.

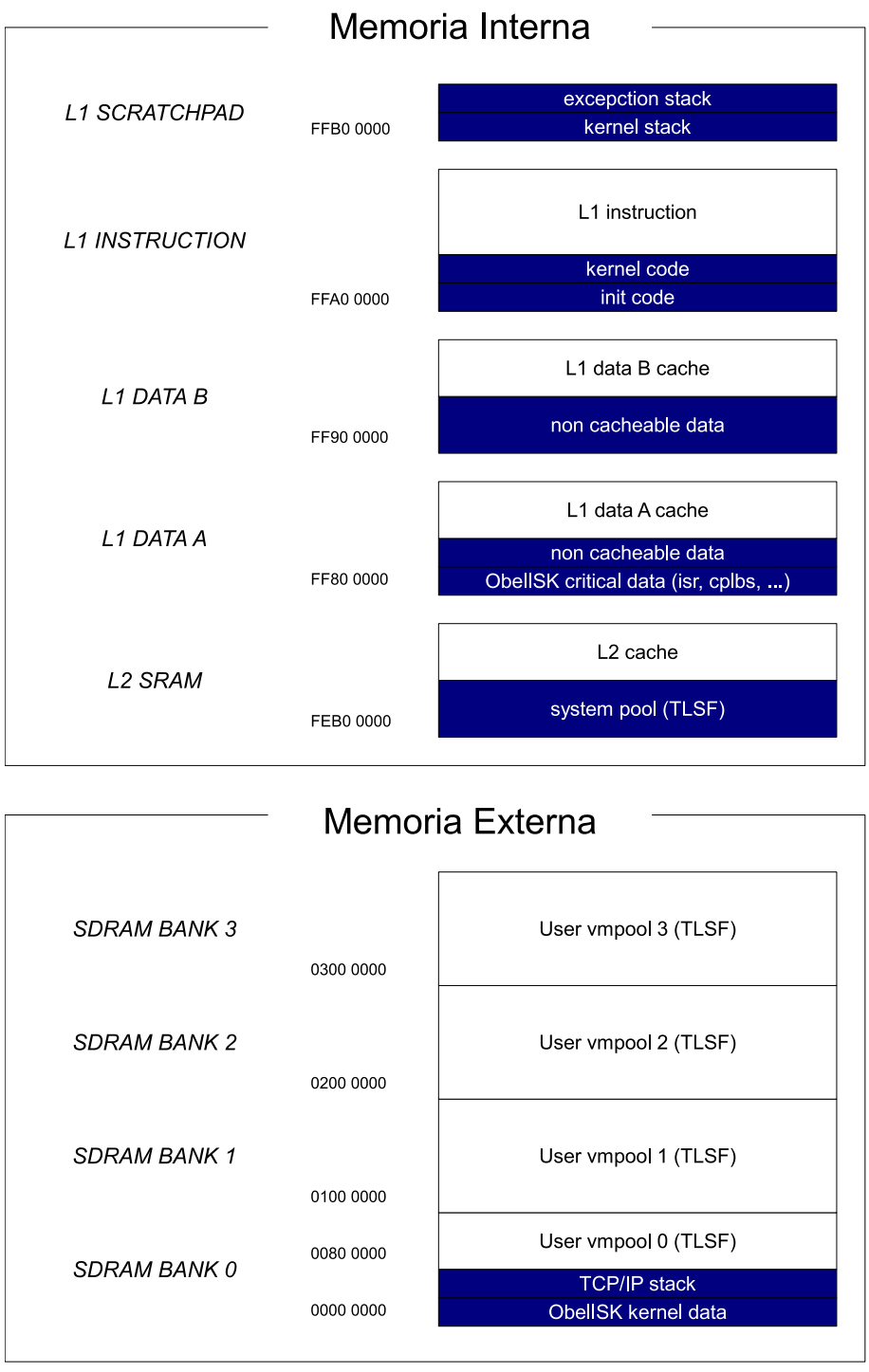

Figura 5.4.2: Distribución por defecto de código y datos en memoria.

El núcleo consta de un vmpool de sistema, necesario para el sistema operativo, y varios vmpools de usuario. Como requisito no necesario, se aconseja que el vmpool de sistema se ubique en la memoria interna del procesador. Este vmpool gestiona información crítica del sistema operativo, con lo que colocarlo en memoria externa, es decir, memoria lenta, provocaría un decremento importante de las prestaciones del sistema.

La distribución en memoria externa responde a la arquitectura de la memoria SDRAM instalada por Bluetechnix en el CM-BF548. La SDRAM consta de 4 bancos de 16 MB accesibles en paralelo. Por ejemplo, en un escenario donde se ha implantado un sistema de visión. Lo normal es que las imágenes lleguen al sistema a través de una transferencia DMA. Este DMA escribirá la imagen en uno de los bancos. Si, a su vez, el procesador está escribiendo en el mismo banco, los accesos serán el doble de lentos, ya que hay que arbitrarlos y secuenciarlos. Por lo tanto, interesa que, mientras el motor DMA trabaja con un banco, la CPU lo haga con otro. Con la configuración por defecto proporcionada, el usuario puede realizar peticiones de memoria dinámica en cualquiera de los cuatro bancos, distribuyendo así los buffers según su conveniencia. Como contrapartida, existen dos desventajas. La primera es que se está limitando el máximo de memoria reservada en los vmpools. Esto provoca que el usuario deba controlar el estado de cada uno (en el caso de aplicaciones que hagan uso intensivo de memoria). La segunda desventaja es que la división de la memoria limita el máximo tamaño de bloque que es posible reservar de una vez. Aún así, siendo realistas, es improbable que una aplicación necesite un buffer de más de 16 MB.

5.5. Compilación de ObelISK

Un makefile es un fichero que se utiliza como entrada a la herramienta *make*. Estos ficheros permiten automatizar la compilación de ficheros, y son muy útiles, especialmente cuando el tamaño del proyecto es grande. Durante la compilación del núcleo, cada fichero de código fuente se compila por separado, generando una serie de ficheros objeto. Estos ficheros objeto son las funciones y librerías utilizadas en el núcleo. Una vez se tienen todos los ficheros objeto, se enlazan en un único fichero que representa el código fuente de ObelISK, igual que componer un puzzle. En el momento de redactar este documento, el núcleo constaba de 138 ficheros de código fuente y 147 ficheros de cabecera. Esta cantidad de ficheros pone de manifiesto la necesidad de automatizar el proceso de compilación del núcleo.

La compilación del núcleo se hace tal y como se muestra en la figura 5.5.1. Como puede observarse, de cada fichero de código fuente se genera su código objeto. Una vez se tiene todos los ficheros compilados por separado, se someten a un

primer enlace. Este enlace es temporal, y sirve únicamente para agrupar y ordenar el código en un único fichero de acuerdo con sus secciones de salida. Nótese que ficheros distintos pueden contener código y datos que se ubiquen en la misma sección de salida (ver figura 5.4.1). En la figura 5.5.1, `itron.o` es un fichero en formato *Executable and Linking Format* (ELF) [15], el cual es un formato estándar para ficheros ejecutables, que contiene una imagen del núcleo.

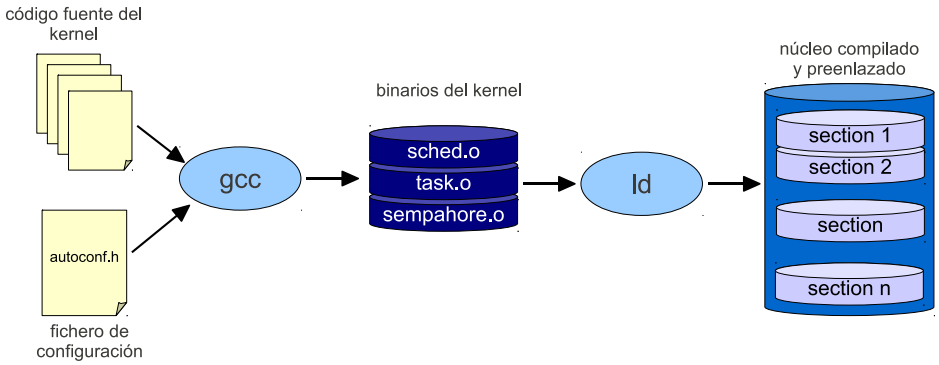

Figura 5.5.1: Procedimiento de compilación del núcleo.

El fichero objeto resultante de la compilación del núcleo es un fichero que se conoce como *reenlazable*. Es decir, es posible reubicar el código si fuese necesario, resolviendo nuevamente los símbolos. El enlace temporal se realiza ubicando todo el código y datos en las secciones de salida estándar, tal y como se ha comentado en la sección 5.4. A continuación, y a falta de un script LDS, todas las secciones de salida se ubican a partir de la dirección de memoria 0x00000000.

5.6. Desarrollo de aplicaciones con ObelISK

El desarrollo de una aplicación para ObelISK se asemeja al procedimiento de compilación del núcleo. El usuario debe compilar su aplicación y generar un fichero con el código objeto. Este código objeto contendrá llamadas al núcleo de ObelISK que no estarán resueltas. El paso final en la generación de una aplicación es enlazar el código de la aplicación y el código del núcleo, mediante el uso de un script LDS y la herramienta ld. El usuario puede decidir, a través de este LDS dónde desea ubicar el código de la aplicación, tanto el código del núcleo como el de las funciones que haya implementado. La figura 5.6.1 muestra el procedimiento de enlace de una aplicación con ObelISK.

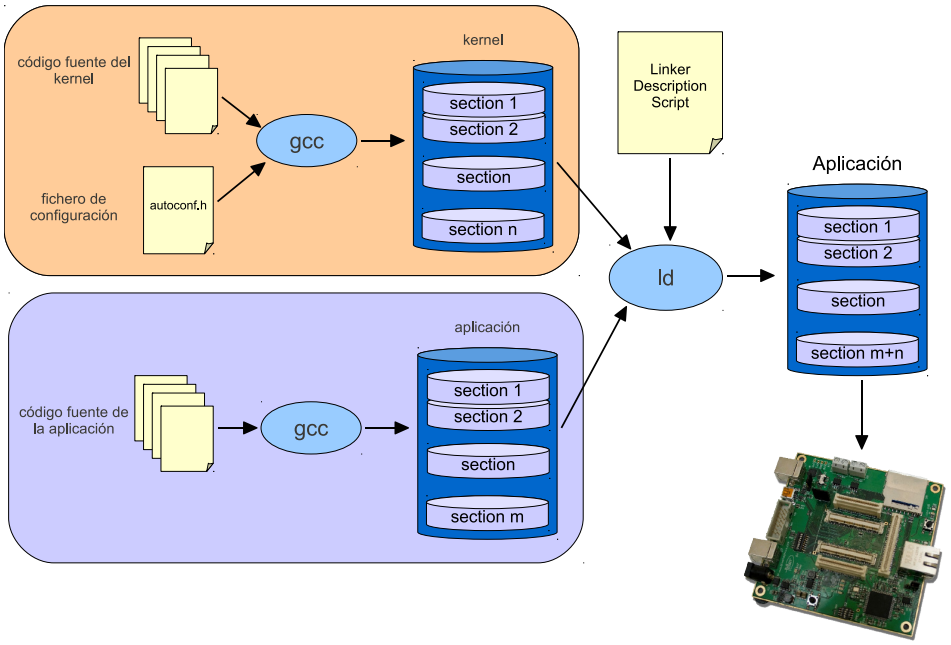

Figura 5.6.1: Procedimiento de enlace del código de la aplicación con el código del núcleo.

Capítulo 6

El núcleo en tiempo de ejecución

6.1. Inicialización del procesador

El inicio de la programación de un núcleo no resulta fácil si no se tiene experiencia. Uno puede quedar desbordado por la gran cantidad de elementos que hay que tener en cuenta. En el desarrollo de este núcleo, se ha partido de un conocimiento previo de la plataforma, en relación a los procedimientos de inicialización necesarios. Éste ha sido el punto de partida, establecer un sistema base que pueda ejecutar código a partir del cual comenzar el desarrollo.

La referencia durante la construcción del sistema base ha sido la plataforma hardware, pese a que una parte del código del núcleo se ha desarrollado de forma que sea portable sin necesidad de realizar cambio alguno en el mismo. Por lo tanto, el primer paso ha sido crear la rutina de inicialización del sistema para el BF548. Típicamente, los procesadores Blackfin inician la ejecución del código en la dirección base de la memoria de código en L1. Esto es, el *loader* carga en el contador de programa (PC) la dirección física 0xFFA00000. Es precisamente en esta dirección donde se enlaza el código de inicialización necesario (rutina `__start`).

Cuando el procesador se conecta a la alimentación, o se activa la señal de reset en el pin correspondiente, su interrupción asociada (EVT1) se activa. Esto se debe a que ésta es la interrupción de máxima prioridad, lo que permite ejecutar el código de inicialización sin temor a que una interrupción no deseada rompa la ejecución secuencial del código. La interrupción permanecerá activa hasta recibir una instrucción *rti* (return from interrupt). La inicialización de la CPU se realiza en los siguientes pasos:

1. **Inicialización de todos los registros del procesador a 0**. Esto evita posibles anomalías en el arranque.

2. **Apagado de la MMU**. El uso de MMU está relegado, en principio, a la existencia de memoria externa. Esta conocimiento es parte del núcleo y no es correcto que se incluya en la inicialización del procesador.

3. **Inicialización de la pila**. Esto es necesario para la utilización de variables locales. Es más, permite llamar a las rutinas escritas en C, con lo que se puede implementar algunas de las funciones de inicialización en código de alto nivel.

4. **Instalación de los manejadores de interrupción por defecto**. En el caso de que una interrupción descontrolada se activara, el comportamiento del sistema estaría definido.

5. **Configuración de la interrupción software (de mínima prioridad)**. Como ya se ha mencionado en el apartado 3.2.1, las aplicaciones que corran bajo este núcleo funcionarán en modo supervisor, con acceso total a todos los dispositivos del sistema. Así, en EVT15 se instala la rutina de inicialización del núcleo y se activa la interrupción asociada.

6. **Salto al código de inicialización del núcleo mediante la instrucción *rti***. Al ejecutar esta instrucción, se limpia la interrupción de reset pero, al haber activado la interrupción software, el código salta a la rutina de inicialización para servir la interrupción activa (IVG15).

Con este procedimiento, se ha configurado un sistema estable que podría utilizarse para ejecutar aplicaciones. Lógicamente, esto es sólo la inicialización más básica. A partir de este momento comienza la configuración del núcleo propiamente dicha.

6.2. Inicialización del núcleo

6.2.1. Inicialización de bajo nivel

Con un sistema funcional, es el momento de comenzar la inicialización del núcleo. Tras la compilación, en el fichero objeto, el código y los datos quedan repartidos por lo que se conoce como secciones. Posteriormente, las secciones se ubican en regiones de memoria a través de un fichero de configuración y se inicia el procedimiento de enlace del código y los datos, que es básicamente ubicar los mismos en memoria. En concreto, hay que inicializar la sección *.bss*, que almacena datos no inicializados por el usuario. Aquí se incluirían las variables globales y las variables estáticas, definidas dentro de funciones, no inicializadas. Es muy común que los usuarios asuman que estas variables estén inicializadas a cero. Por otro

lado, dado que la sección *.bss* es contigua, resulta muy cómodo inicializarla de una atacada, y no tener que hacerlo variable a variable en el código.

El siguiente paso es configurar temporización del sistema. En base a la configuración introducida por el usuario, se inicializan los relojes del sistema CCLK y SCLK. Aunque no es necesario, parece coherente configurar en primer lugar la frecuencia de reloj. Esto no siempre tendría que ser así. Por ejemplo, en ciertas aplicaciones, resulta conveniente reducir el consumo del procesador, por lo que la frecuencia de reloj se configura en tiempo de ejecución en base a la carga del procesador.

Al no formar parte la memoria externa del procesador, es necesario establecer una configuración adecuada para la misma. Junto con la configuración de memoria, se activará la MMU. La activación de la función de cache se ha dejado a elección del usuario. Además, también existe la posibilidad de crear una tabla de descriptores CPLB personalizada, lo que da al usuario bastante flexibilidad para configurar el sistema.

Un ejemplo de la importancia de dejar al usuario esta posibilidad es una aplicación donde el motor DMA (*Direct Memory Address*) y la CPU accedan a páginas de memoria compartidas. El dispositivo DMA se ideó para evitar que la CPU quede bloqueada en accesos a memoria lenta. Este dispositivo permite copiar datos entre memoria y periféricos o entre distintas páginas de memoria. Así pues, si DMA y CPU accedieran a la misma página podría aparecer un problema de incoherencia de memoria si la función de cache se encuentra activada. Es decir, que la página que ha accedido el DMA se encuentra todavía en cache L1, por lo que bien los datos leídos por el DMA no serían válidos, bien existirían dos valores para la misma variable. En una política de cache Write Through[1], este problema no existiría. En cambio, si se decide utilizar una política Write Back[2], sí que podría surgir dicho evento. En conclusión, al usuario puede interesarle manejar la tabla de descriptores CPLB para desactivar la función de cache en ciertas páginas de memoria donde va a haber tráfico de DMA.

A continuación, se activa el temporizador del núcleo, que proporciona al núcleo una forma de tomar el control de la CPU. La definición del período del temporizador del núcleo permitirá configurar la granularidad del sistema. Un período más rápido permitirá capturar la CPU con mayor frecuencia y tener un control del sistema más fino. A cambio, se inducirá una mayor sobrecarga por parte del manejador del temporizador, que tendrá que ejecutarse más a menudo.

La figura 6.2.1 muestra, en escala logarítmica, el uso de CPU inducido por la

[1] Una modificación de datos en cache modifica los datos en memoria externa en ese momento, con lo que se mantiene la coherencia en todo momento.

[2] Las páginas sólo se escriben en memoria externa cuando se victimizan de cache.

elección de un período más pequeño en el temporizador del núcleo. Las medidas de tiempo se han tomado en ciclos de reloj, habiendo configurado el reloj del procesador a 500 MHz. La línea azul representa el uso de CPU en relación al tiempo mínimo de ejecución del manejador de interrupciones del temporizador. La línea negra representa el uso teórico de CPU en el caso en que el manejador tardara más en ejecutarse, ya que el tiempo de ejecución depende del estado del sistema. Por ejemplo, en el momento en que una tarea despierta de un *sleep*, el manejador del temporizador debe extraer la tarea del timeout heap e insertarla en la cola de hilos preparados. Todo este proceso hace que el tiempo de ejecución del manejador del temporizador sea variable, aunque siempre acotado por el número de tareas en el sistema. Esta situación es la que se pretende reflejar con la pendiente negra de la figura. En cualquier caso, el manejador de interrupciones realizará estas tareas más pesadas de forma muy esporádica (en relación al número de ejecuciones), de modo que se puede concluir que el uso de la CPU, de acuerdo con la figura 6.2.1, se situará en el área entre la pendiente negra y la pendiente azul, aunque muy próxima a la pendiente azul.

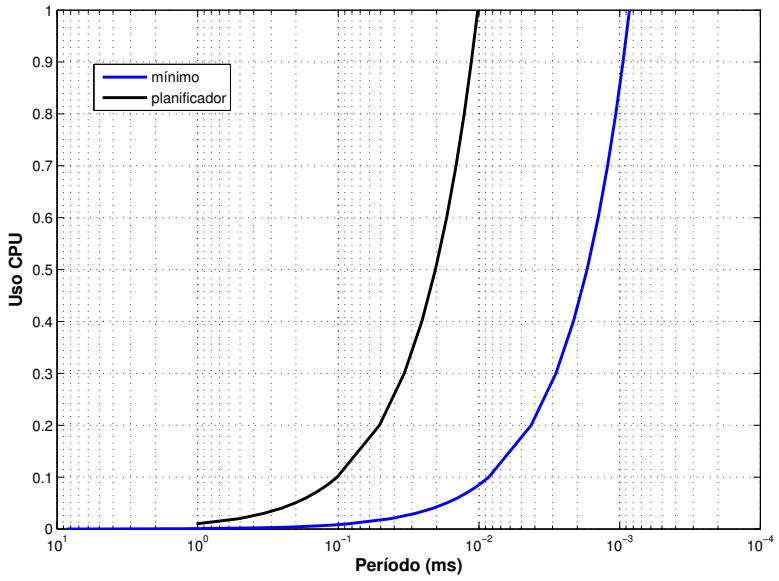

Figura 6.2.1: Uso de CPU por parte del temporizador del núcleo en función del período de interrupción.

Los resultados muestran que, para el manejador implementado, se podría con-

trolar el procesador con una granularidad de unos 10 microsegundos con soltura. A partir de ahí, el uso de CPU se dispara, lo que deja menor margen de ejecución a las aplicaciones. Para tareas extremadamente rápidas, sería conveniente desacoplar su funcionamiento del propio sistema operativo. El BF548 dispone de varios temporizadores, con lo que se podría implementar dichas tareas como manejadores de interrupción de esos temporizadores.

6.2.2. Inicialización de alto nivel

La inicialización de alto nivel consiste en configurar las estructuras de datos necesarias para poder comenzar la ejecución de tareas. El procedimiento consiste en inicializar cada uno de los módulos que dispone el núcleo:

1. **Memoria**. El primer paso, es necesario inicializar los vmpool que maneja el TLSF. La distribución se corresponde con la mostrada en la figura 5.4.2. El resto de módulos del núcleo necesitan utilizar memoria dinámica durante la inicialización, por lo que es indispensable poner a punto el gestor de memoria.

2. **Planificador**. Inicializar el planificador supone limpiar la memoria reservada para la cola de hilos preparados y crear la estructura de datos para el timeout heap.

3. **Tareas y objetos de sincronización**. A continuación se inicializan los pools de tareas y objetos de sincronización, quedando tal y como se muestra en la figura 4.6.1.

4. **Hilo ocioso**. Al inicializar el pool de tareas, ya se puede reservar memoria para el hilo ocioso. Como ya se ha comentado, este hilo se ejecuta cuando ninguna otra tarea se encuentra lista para entrar en ejecución.

5. **Pila TCP/IP**. Si el usuario decide utilizar la pila TCP/IP, se inicializa en este momento. La pila TCP/IP se comentará en más detalle en el capítulo 7.

6. **Hilo principal**. El penúltimo paso es crear la tarea principal, la que contiene la rutina `main()`. De momento, esta tarea se ha creado con máxima prioridad, para permitir inicializar el resto de tareas sin que haya ningún tipo de interferencia.

7. **Salto al código de usuario**. Finalmente, se invoca al planificador. Como se ha creado la tarea principal, se producirá un cambio de contexto y el sistema arrancará desde la rutina principal.

El código del hilo ocioso es diferente del código del resto de tareas. Se encuentra al final del código de inicialización del sistema, donde queda ejecutando en un bucle infinito la instrucción idle. Esta instrucción deja al procesador en un estado ocioso, en el cual se reduce al mínimo el consumo energético. Dependiendo de las implementación, dentro de este bucle infinito se pueden incluir rutinas para la liberación de recursos, similar a un recolector de basura. Por ejemplo, el hilo ocioso en VDK contiene una rutina que sirve para purgar los recursos que está ocupando un hilo que ha sido finalizado. En ObelISK no se ha considerado necesario que el hilo ocioso ejecute ninguna tarea, de modo que simplemente se deja al procesador en estado de bajo consumo.

6.3. Manejo de interrupciones

Las interrupciones son la forma de capturar el procesador y cambiar el flujo de ejecución de forma asíncrona. El manejo de las interrupciones se realiza en los siguientes pasos:

1. **Guardar contexto**. El contexto de ejecución, es decir, los registros del procesador, se guardan.

2. **Incrementar cuenta de anidamiento**. El procesador maneja una variable contador para mantener el nivel de anidamiento de interrupción. En el dominio de los hilos, el nivel de anidamiento es cero. Al producirse una interrupción, se incrementa el nivel de anidamiento en uno. Si durante el manejo de dicha interrupción se produce otra interrupción, el nivel de anidamiento vuelve a incrementarse y así sucesivamente. Cuando la cuenta de anidamiento es mayor que cero, el planificador no realizará ningún cambio de contexto.

3. **Detectar el nivel de interrupción**. Se analiza el registro IPEND para detectar el nivel de interrupción actual.

4. **Llamar al manejador de interrupciones**. Una vez se conoce el nivel de interrupción, se llama a la rutina instalada en la posición correspondiente del vector de interrupciones.

5. **Decrementar cuenta de anidamiento**.

6. **Restaurar contexto y volver a código de usuario**.

Los manejadores de interrupción pueden realizar llamadas al sistema que cambien las condiciones de planificación. Por ejemplo, existen dos formas de detectar la llegada de datos desde un periférico. La primera es comprobar constantemente la

llegada de algún dato; este método se conoce como *polling*. La segunda es tener un hilo bloqueado en un semáforo (podría servir cualquier objeto de sincronización). Cuando llegan datos al periférico, éste eleva una interrupción, el manejador la recoge y desbloquea el semáforo correspondiente. Al desbloquearse el semáforo, el hilo sabe que han llegado datos y realiza las tareas pertinentes. El polling es una forma poco eficiente de recoger datos de un periférico, ya que se hace uso intensivo de la CPU para comprobar constantemente la llegada de datos. La segunda forma es más eficiente, ya que sólo se hace uso de la CPU cuando realmente han llegado los datos.

En relación al ejemplo anterior, se pone de manifiesto que el manejador de interrupciones podría realizar una llamada al sistema que cambiara la planificación del sistema. Cuando esto sucede, se podría actuar de dos formas. Primero, registrar el hecho de que la planificación ha cambiado, para realizar un cambio de contexto a la salida del manejador de interrupciones. Este método sería el más eficiente. Sin embargo, su implementación es extremadamente compleja e incrementa la latencia de los manejadores de interrupción. Esto es debido, sobretodo, a la gestión del contexto que hay que realizar en el manejador. La segunda forma de actuar es volver del manejador normalmente y esperar a que la interrupción del temporizador del núcleo invoque al planificador. En ObelISK se procede de esta segunda forma ya que, de este modo, se reduce la complejidad del código de los manejadores de interrupción.

Esperar la ocurrencia de la interrupción del temporizador del núcleo tiene una desventaja. La latencia máxima para un manejador de interrupción será igual a la granularidad escogida para el sistema, es decir, el período del temporizador del núcleo. Cuanto menor sea el período, menor será la latencia, pero mayor el uso de CPU por parte del manejador de interrupción del temporizador (figura 6.2.1).

Capítulo 7

Integración de la pila TCP/IP lwIP

La placa de desarrollo dispone de una variedad de dispositivos de comunicación: bus CAN, USB y Ethernet, además de otros dispositivos serie y paralelo. De entre todos, el más interesante a la hora de depurar aplicaciones y controlar el procesador es el puerto de Ethernet. Ethernet es un tipo de red muy extendida, y está disponible para toda clase de máquinas. Especialmente interesante resulta su disponibilidad en los PC de sobremesa, con los que se desarrollan aplicaciones para la plataforma. Así, si se activan las comunicaciones por Ethernet, será posible comunicarse con la plataforma, lo que permite realizar diagnóstico, depuración y configuración de la plataforma de un modo muy sencillo.

En el momento de redactar este documento, la integración de la pila todavía se encuentra en desarrollo. El driver implementado tiene algunas deficiencias, y las comunicaciones vía Ethernet no son del todo estables. Sin embargo, el grueso de la integración de lwIP ya se ha realizado y se ha conseguido con éxito comunicar con la plataforma.

Entre las características de Ethernet destacan su velocidad, de hasta 100 Mbps, y su alcance[1]. En el proyecto SENSE, la activación de las comunicaciones permitió la depuración de los algoritmos de visión que, de otro modo, hubiera sido prácticamente imposible. En SENSE, los algoritmos de visión que ejecutaba el Blackfin podían configurarse con multitud de parámetros. Sin las comunicaciones, la depuración consistía en ejecutar el código hasta llevarlo al punto deseado, introducir un punto de ruptura[2] y descargarse las imágenes para comprobar lo que había pasado. En primer lugar, esta forma de trabajo es demasiado tediosa y lenta. Además, dado

[1] Es decir, que prácticamente en cualquier sitio se puede encontrar un dispositivo con Ethernet.
[2] *Breakpoint*.

que el sistema tenía que funcionar en tiempo real, resultaba imposible depurar los algoritmos para secuencias de imágenes. Así pues, la solución apareció al instalar una pila TCP/IP y desarrollar una aplicación en el PC que descargara el vídeo en tiempo real[3]. Este ejemplo intenta poner de manifiesto las razones por las que resulta importante instalar algún mecanismo de comunicación. Ciertamente, existen mecanismos de comunicación más simples de implementar como, por ejemplo, el bus CAN de que dispone la placa de desarrollo. Aún así, es imposible utilizar CAN para la transmisión de vídeo en tiempo real.

Para este sistema se ha escogido la pila TCP/IP lwIP [16]. Este software está muy extendido entre los sistemas empotrados por varias razones. En primer lugar, lwIP es el acrónimo de *low weight TCP/IP stack* o pila TCP/IP de bajo peso. Esto significa que el software está preparado para funcionar cuando se dispone de una cantidad limitada de memoria, ya que su coste espacial es bajo. La segunda razón es que esta pila es muy configurable, con lo que es posible hacerla funcionar en cualquier procesador, tanto si se utiliza un sistema operativo como si no. Además de estas dos razones, la documentación existente en [17] ha facilitado la integración de la pila.

Para que la pila lwIP pueda funcionar con un sistema operativo subyacente, es necesario implementar una interfaz que permita a la pila comunicarse con el sistema operativo y hacer uso de él. Además, esta pila no dispone de drivers para el dispositivo de Ethernet específico que se vaya a utilizar; esto debe proporcionarlo el responsable de la plataforma.

La integración de la pila lwIP es similar a componer un puzzle. La figura 7.0.1 muestra el diagrama de bloques de los elementos necesarios para la integración de la pila TCP/IP. En la figura puede verse, en amarillo, las funcionalidades que ha sido necesario implementar y, en negro, los componentes hardware involucrados.

[3]Nótese que, en este caso, en el uso del término tiempo real no se está refiriendo a un sistema que debe entregar sus resultados antes de un cierto deadline, sino al hecho de que las imágenes se recibían en el PC prácticamente a la vez que se capturaban y procesaban en la plataforma.

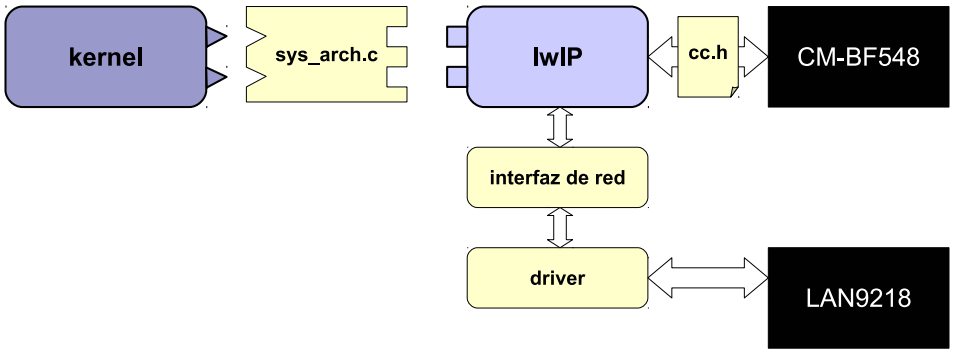

Figura 7.0.1: Integración de lwIP.

Entre las piezas que componen la integración de la pila se tiene:

1. **Descripción de la arquitectura**. Se realiza a través de un fichero de cabecera, `cc.h`, que proporciona definiciones para los tipos de datos que debe utilizar lwIP. A través de este fichero de cabecera también se proporcionan mecanismos de protección de regiones críticas, como desactivar y reactivar las interrupciones, y directivas para el compilador.

2. **Interfaz del sistema operativo**. Cuando se instala lwIP sobre un sistema operativo, hace uso de tareas y objetos de sincronización. Para ello, lwIP llama a unas funciones vacías que debe implementar el desarrollador. Estas funciones se implementan en un fichero llamado `sys_arch.c`.

3. **Interfaz de Red**. Un mismo sistema puede contener diferentes interfaces de red. Diferentes interfaces de red pueden hacer uso del mismo dispositivo hardware.

4. **Drivers**. Debido a la cantidad de dispositivos de red existentes, la implementación de los drivers de dichos dispositivos es responsabilidad del usuario.

Además de esto, se han generado varios ficheros para la inicialización de la pila, así como el registro de sucesos de lwIP.

7.1. Interfaz del sistema operativo

La primera de las piezas necesarias para la integración de la pila es la interfaz con el sistema operativo. Como ya se ha comentado, lwIP utiliza una API de sistema operativo propia pero vacía. La implementación de las funciones debe proporcionarla el diseñador del sistema. El siguiente listado es un ejemplo de cómo se ha implementado la función de creación de semáforos.

```
1  sys_sem_t sys_sem_new(u8_t count)
2  {
3      T_CSEM pk_csem;
4      sys_sem_t sem;
5
6      pk_csem.sematr = 0;
7      pk_csem.isemcnt = count;
8      pk_csem.maxsem = MAX_SEM_COUNT;
9
10     sem = acre_sem(&pk_csem);
11     return sem;
12 }
```

Se procede a desgranar el ejemplo anterior:

- sys_sem_t. Es el tipo de datos que se define en lwIP para mantener la referencia a un semáforo. En este caso, sys_sem_t es un entero de 32 bits.
- sys_sem_new(). Es la función que invocará la pila TCP/IP cada vez que necesite crear un semáforo.
- T_CSEM. Este tipo de datos forma parte de la especificación ITRON. Es un paquete de datos con la información necesaria para la creación de un semáforo, y se inicializa en las líneas 6, 7 y 8 del listado de programa anterior.
- acre_sem(). Esta función es parte de la API de ITRON. Sirve para crear un nuevo semáforo con asignación automática de identificador. Recibe como parámetro el paquete de inicialización pk_csem.

En la mayoría de los casos, la implementación de las funciones de la interfaz sys_arch será muy parecida a ésta. Sin embargo, lwIP hace uso de ciertas funciones de sistema operativo que no están implementadas de forma explícita en ObelISK. Por ejemplo, en el uso de buzones, lwIP hace uso de una función llamada sys_arch_mbox_tryfetch(). Esta función la invoca una tarea para comprobar si se ha recibido un mensaje en el buzón y, de no ser así, la tarea no queda bloqueada, sino que continúa su ejecución. En ITRON no existe tal función, y cualquier tarea que intente leer de un buzón vacío quedará bloqueada. Por lo tanto, es necesario simular dicha funcionalidad implementando los mecanismos necesarios en la interfaz del sistema operativo.

7.2. Drivers

La placa de desarrollo dispone de un dispositivo de Ethernet LAN9218 de la empresa SMSC [18]. Este dispositivo es un controlador 10/100 de Ethernet[4] enfocado a servir como sistema de comunicación en sistemas empotrados que ejecutan aplicaciones multimedia (transmisión de audio y vídeo).

Para el control de este dispositivo se ha implementado un driver muy básico, todavía incompleto, que permite enviar mensajes y recibirlos vía Ethernet. Este driver inicializa el dispositivo, asignando una dirección física (o dirección MAC) e inicializando las interrupciones del dispositivo. Ante una recepción de mensaje, el dispositivo enviará una interrupción al procesador, el cual la capturará y realizará las tareas pertinentes para recoger el mensaje. Para el envío del mensaje no es necesario controlar ninguna interrupción; simplemente se copian los datos en el buffer de salida.

Este dispositivo se encuentra mapeado en el CM-BF548, en la dirección base 0x24000000, que pertenece al inicio de uno de los bancos de memoria asíncrona. El mapa de memoria de este dispositivo puede verse en la figura 7.2.1. El control del dispositivo se realiza escribiendo en los registros, que se encuentran en las direcciones de memoria *base* + *R*.

[4]Lo que significa que es posible configurarlo para que funcione tanto a 10 Mbps como a 100 Mbps.

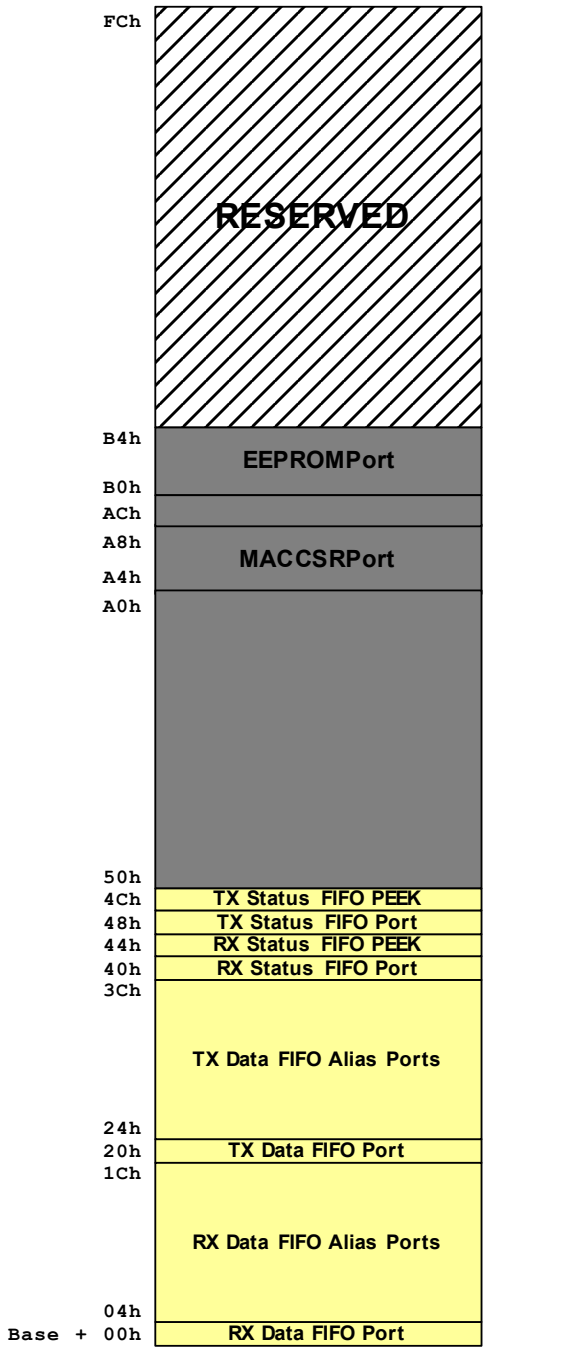

Figura 7.2.1: Mapa de memoria del LAN9218.

7.3. Interfaz de red

La interfaz de red contiene la información de alto nivel relativa a un dispositivo de red. Entre la información que maneja se tiene el nombre de la interfaz, la dirección IP, la máscara de subred y la puerta de enlace predeterminada. La interfaz de red implementa las funciones de entrada/salida de alto nivel, y se comunica con el driver del dispositivo para transmitir y recibir paquetes.

El papel de la interfaz de red en la transmisión es recoger una lista de paquetes de datos, generada por la pila TCP/IP, e ir entregando uno a uno al dispositivo para que los vaya mandando por red. En el caso de la implementación realizada, la interfaz de red implementa también el punto de entrada al manejador de interrupciones del dispositivo LAN9218. Del mismo modo, en la recepción de datos, la interfaz recoge un conjunto de paquetes de datos del driver y los pasa a la pila TCP/IP.

Parte III

Resultados

Capítulo 8

Trabajo Experimental

Durante el desarrollo del núcleo, se ha tenido que comprobar la funcionalidad de gran cantidad de algoritmos. Debido al tamaño del código que conforma el núcleo, cada una de las partes ha tenido que ser probada por separado antes de poder ejecutar la primera aplicación. Así, se han desarrollado tests para la comprobación del buen funcionamiento de la gestión de las listas, las colas, el planificador, la distribución por defecto en secciones de salida, la inicialización de la pila lwIP, etc. Como es lógico, todos los tests no han funcionado con éxito en la primera ejecución. Han servido para corregir problemas y depurar las funcionalidades del núcleo.

El trabajo experimental se ha basado, fundamentalmente, en la comprobación de cada una de las funcionalidades del núcleo, mediante la implementación de aplicaciones de ejemplo básicas. En total, se han generado 8 aplicaciones de ejemplo con las que se ha comprobado con éxito el buen funcionamiento del núcleo. Todos los tests se han basado en el uso de un led de depuración que hay instalado en la placa de desarrollo. El objetivo de cada test ha sido hacer parpadear este led con una cierta frecuencia.

8.1. Test de Tareas

El primer test de todos los realizados ha sido el de creación de una tarea. En este test, el código entra en la rutina main y se crea una tarea con un identificador específico. A continuación, la tarea main finaliza, con lo que debe entrar en ejecución la tarea recién creada. Esta tarea entra en un bucle que activa y desactiva el led en ciclos de un segundo. Con esta aplicación se ha comprobado:

1. **Inicialización del código de entrada a la rutina principal**. Una vez se ha inicializado el núcleo, el código debe saltar a la rutina main.

2. **Gestión del vmpool de sistema a través de la API de ITRON**. Las aplicaciones no hacen uso de forma directa de las funciones de librería del TLSF. En lugar de ello, el núcleo implementa las funciones descritas en ITRON para la gestión de vmpools. La librería TLSF gestiona de forma individual cada vmpool; es el sistema operativo el que gestiona el conjunto de vmpools instalado.

3. **Creación de tareas**. Se ha comprobado que la tarea se inicializa de forma correcta, estableciéndose su estado inicial como *durmiendo*. También se ha comprobado que se reserva la estructura y se actualiza el bitmap de indexación de forma correcta.

4. **Activación de tareas**. La activación inserta la tarea en la cola de hilos preparados y establece su estado como *preparada*.

5. **Hilo ocioso**. La tarea ejecuta un bucle en el cual cambia el estado del led cada medio segundo. Para ello, cambia el estado y después invoca a la función `tslp_tsk()`, para suspenderse durante medio segundo. Con este ejemplo se ha comprobado que la suspensión funciona correctamente, así como la entrada en ejecución del hilo ocioso.

6. **Finalización de tareas**. Por último se ha comprobado que las tareas finalizan correctamente y que, al invocar la función `ext_tsk()`, la tarea se extrae de la cola de hilos preparados y se deja en estado durmiendo.

8.2. Test de Semáforos

Se ha comprobado el buen funcionamiento de los semáforos mediante la creación de un segundo test. En este test, se han creado dos tareas, además de la principal, las cuales se sincronizaban mediante un semáforo, siguiendo un modelo productor consumidor. Esta aplicación ha comprobado:

1. **Procedimiento de creación de semáforos**. Al igual que las tareas, se ha comprobado que los semáforos se crean de forma adecuada, tanto en la reserva de la estructura como en el uso del bitmap de indexación.

2. **Gestión del valor del semáforo**. La adquisición y liberación del semáforo ha funcionado de forma adecuada.

8.3. Test de Buzones

El test para la comprobación de los buzones es similar al modelo seguido en el test de semáforos. Nuevamente se han creado dos hilos de ejecución, uno de ellos produciendo mensajes vacíos que inserta en un buzón. El otro de los hilos los recoge y cambia el estado del led. Uno de los problemas que se han afrontado con los buzones es la gestión de las cabeceras de los mensajes. Un buzón contiene una cola de mensajes. Los mensajes no se insertan tal cual en el buzón, sino que se inserta una cabecera que contiene el puntero a la dirección de memoria donde está el mensaje y la información relativa a la cola. Así, el test de buzones ha sido, además de comprobar el buen funcionamiento de estos objetos de sincronización, ha servido para desarrollar una metodología de uso de los mismos, en cuando a la gestión de las cabeceras de los mensajes.

Esta aplicación ha comprobado:

1. **Procedimiento de creación de buzones**. Nuevamente se ha comprobado que los buzones se crean correctamente.

2. **Gestión de la cola de mensajes**. Se ha comprobado que la cola de mensajes se llena y se vacía de forma correcta.

3. **Gestión de la cola de hilos**. Igualmente, se ha comprobado que la cola de hilos funciona bien.

4. **Gestión de las cabeceras de los mensajes**. La necesidad de utilizar cabeceras en los mensajes puede ser una de las razones por las que VDK no haya implementado esta funcionalidad. El uso de buzones conduce al uso de memoria dinámica, debido precisamente a las cabeceras de los mensajes.

8.4. Test de Eventflags

La comprobación del buen funcionamiento de un eventflag es un poco más compleja que la del resto de objetos de sincronización. Esto se debe, principalmente, a que varias tareas pueden esperar patrones diferentes de un mismo eventflag.

1. **Procedimiento de creación de eventflags**.

2. **Gestión de los patrones de espera**. Se ha comprobado que las esperas funcionan correctamente, y que las tareas se despiertan como es debido cuando el patrón de espera se cumple para el eventflag.

8.5. Test de Cache

Además de las funciones básicas del sistema operativo, se ha comprobado que la activación de la función de cache funciona correctamente. A tal efecto, se ha implementado una aplicación específica en la que se han distribuido dos buffers. Uno de estos buffers se ha ubicado en una página de memoria con la función de cache activa y el otro en una página con dicha función desactivada. A continuación se han ejecutado operaciones sobre ambos buffers y se ha comprobado con éxito que la cache ha incrementado el rendimiento de las operaciones para el buffer ubicado en la página de memoria con la función de cache activa.

8.6. Test de Interrupciones

El funcionamiento de las interrupciones se ha testeado mediante la implementación de una aplicación en la que se ha utilizado uno de los timers del CM-BF548. Se ha programado un timer para que active una interrupción cada cierto intervalo de tiempo. Por cada vez que se ha ejecutado el manejador, se ha cambiado el estado del led de depuración de la placa de desarrollo, comprobando el parpadeo de dicho led con éxito. Para este test se ha comprobado:

1. **Correcta instalación del manejador de interrupciones**. Se ha comprobado que la instalación del manejador en el vector virtual de interrupciones ha funcionado de forma correcta.

2. **Gestión del contexto y del dominio**. El dominio y el contexto se han gestionado de forma correcta durante los tests.

8.7. Test del temporizador del núcleo

Este test ha servido para realizar las mediciones sobre el uso de CPU presentadas en la figura 6.2.1. Para este test se ha incluido código adicional dentro del núcleo, que ha permitido medir, en ciclos de reloj, la latencia del manejador de interrupciones del temporizador, en función de la carga.

8.8. Test de lwIP

Se ha implementado una aplicación en la que se ha activado un servicio TCP de eco en el puerto 12345. Con este test se ha comprobado:

1. **Inicialización de la pila TCP/IP**. La inicialización de la pila es un procedimiento complejo. Se ha tenido que comprobar la correcta inicialización de lwIP, las direcciones MAC e IP, así como toda la información relativa a la interfaz de red. También se ha comprobado que el driver implementado para el dispositivo LAN9218 se ha inicializado correctamente (inicialización de registros y establecimiento de la dirección MAC).

2. **Inicialización del socket TCP**. Cuando se inicia un servidor, en primer lugar se crea un socket. Un socket es una abstracción software que representa un canal de comunicación entre dos equipos. Crear un socket consiste básicamente en reservar la memoria necesaria e inicializar los parámetros del mismo como, por ejemplo, el tipo de conexión a que está destinado (en este caso TCP).

3. **Comunicación TCP**. El ejemplo implementado acepta una conexión y realiza un eco sobre un dato de 32 bits que se mande. En este caso, la comunicación todavía es deficiente, puesto que la capa TCP descarta algunos de los paquetes que recibe, por lo que el dato no llega. Esto ocasiona que haya que esperar a que expire el temporizador de ese envío y que se tenga que reenviar los datos. Pese a que la comunicación es posible, y existe un servicio de eco funcionando, los problemas descritos ralentizan la comunicación. Este es el último de los problemas que queda por resolver, antes de tener la pila lwIP completamente operativa en el nodo.

Capítulo 9

Conclusiones

En este trabajo se ha presentado los detalles de implementación de un núcleo de tiempo real basado en especificación ITRON. El núcleo se ha implementado siguiendo esta especificación hasta donde ha sido posible. El único detalle que no ha seguido el Perfil Estándar de ITRON se encuentra en el número de identificadores disponibles para para las tareas y los objetos de sincronización. Sin embargo, este detalle no va en desmedro de las prestaciones del sistema.

Los objetivos se han alcanzado. Se ha construido un entorno de ejecución basado en una especificación potente, configurable y con muy buenas prestaciones. Se ha eliminado las librerías de software propietario, por lo que el control sobre el código de cada aplicación es total. Además, la especificación ITRON es más potente que la API del kernel propietario proporcionado por Analog. Es aquí donde las posibilidades de comunicación entre tareas mejoran notablemente.

Este trabajo únicamente presenta una parte del proyecto, como es el kernel desarrollado. El proyecto sigue desarrollándose, ahora con el foco puesto en el entorno de trabajo. Se están construyendo aplicaciones que permitan desarrollar aplicaciones y cargar los programas en el Blackfin de forma sencilla. Gran parte de las mismas se encuentran finalizadas y se describen en los anexos de este documento.

Aunque el núcleo se encuentra finalizado, la pila TCP/IP aún está en pruebas. La instalación de lwIP es un desarrollo periférico aunque muy ligado al núcleo. Ésta es otra de las ventajas añadidas al entorno de desarrollo. Con el software propietario, el desarrollo de las aplicaciones es bastante rígido en cuanto al uso de la pila lwIP. Si se desarrolla una aplicación que no haga uso de esta pila, no será posible realizar modificaciones a posteriori. En cambio, con el kernel proporcionado se puede reconfigurar el núcleo de forma simple e instalar o desinstalar la pila TCP en cualquier instante del desarrollo.

Una vez la instalación de la pila se finalice, se podría pasar al desarrollo de aplicaciones de visión, que son precisamente el objetivo de este proyecto. Como trabajo futuro, se tiene planificada la creación de una página web para dar visibilidad al proyecto. El código se va a publicar bajo licencia GPL, de forma que esté disponible, y que sea algo vivo y en constante evolución.

Por último, la implementación de este núcleo ha supuesto un reto sumamente interesante. Ha permitido profundizar los conocimientos principalmente de compiladores. El trabajo aquí descrito es sólo una pincelada de las posibilidades que ofrece este campo. Por otro lado, este procesador es relativamente simple. Las capacidades de gestión de memoria por parte de la MMU son ciertamente limitadas. Además, el conjunto de instrucciones también es limitado en relación a otro tipo de procesadores. Lógicamente, este es un procesador digital de señales, y está enfocado a usos específicos, por lo que no necesita soportar mecanismos de gestión de software más complejos. En conclusión, debido a las características de este procesador, el núcleo que se ha implementado será capaz de servir sobradamente las necesidades de cualquier aplicación que se desarrolle para esta plataforma.

Capítulo 10

Anexos

10.1. Menuconfig

El procedimiento de configuración del núcleo se ha automatizado mediante el uso de la aplicación menuconfig. Esta aplicación se desarrolló para configurar las opciones de compilación del núcleo de Linux. Dispone de una interfaz gráfica basada en la librería *Ncurses*. Los menús se configuran haciendo uso de un lenguaje de scripting con una sintaxis bastante simple. La figura 10.1.1 muestra uno de los menús de configuración construidos para ObelISK a partir de menuconfig.

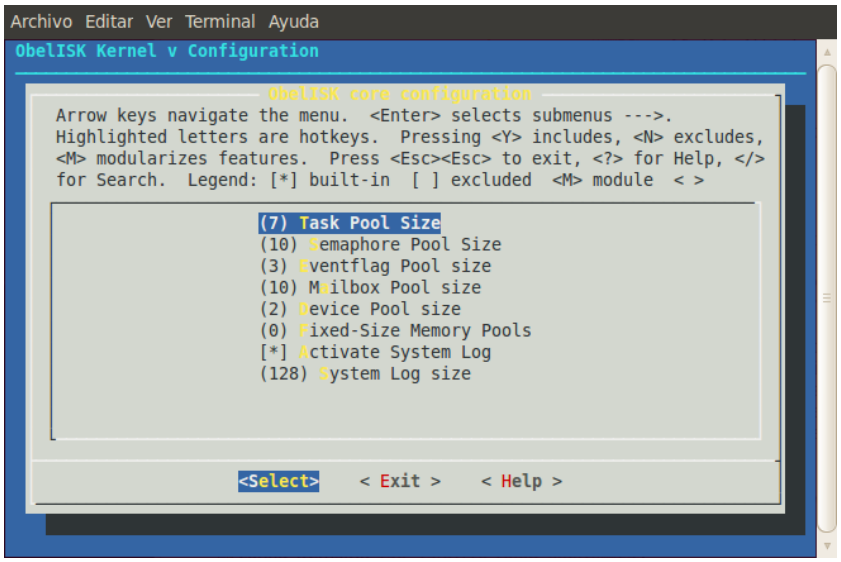

Figura 10.1.1: Vista de menuconfig para la configuración de ObelISK.

Esta aplicación genera un fichero de cabecera llamado autoconf.h, que incluye cada una de las opciones de compilación como una directiva del preprocesador de C. Por ejemplo, la directiva que configura la frecuencia del núcleo del procesador se configura en menuconfig de la siguiente forma:

```
config CCLK
int "Core Clock Frequency (Hz)"
default 500000000
```

De acuerdo con el listado anterior, se está configurando una directiva de tipo entero (int), con el nombre CCLK y el valor por defecto 500.000.000 (500 MHz). *menuconfig* añade un prefijo CONFIG_ a la directiva de configuración, con lo que la anterior configuración, una vez procesada, se imprimiría en el fichero de configuración de la siguiente forma:

```
#define CONFIG_CCLK 500000000
```

El siguiente esquema muestra el árbol de opciones incluidas en la operación de configuración del núcleo:

- **Architecture**
  - Blackfin
- **Architecture specific**
  - Clock Input Frequency
  - Core Clock Frequency
  - System Clock Frecuency
  - Core Timer Period
  - External Memory Size
  - Enable Instruction Cache
  - Enable Data Cache
    - Caching Method
      - ◇ Write Through
      - ◇ Write Back
    - Generate Custom CPLB Table
  - Use Blackfin Optimized String Library Functions
- **Bootloader**

- Flash Memory Size
- Flash Start Address

- **ObelISK Core Configuration**
  - Task Pool Size
  - Semaphore Pool Size
  - Eventflag Pool Size
  - Mailbox Pool Size
  - Device Pool Size
  - Fixed-size Memory Pools
  - Activate System Log
    - System Log Size

- **lwIP Settings**
  - Enable lwIP TCP/IP Stack
    - lwIP Features
      - ◇ Minimal Features
      - ◇ Sequential API
      - ◇ Socket API
    - Ethernet Thread Priority
    - Configure Network
      - ◇ Activate DHCP
      - ◇ IP Address
      - ◇ Network Mask
      - ◇ Default Gateway
      - ◇ MAC Address

10.2. OvLog

Se ha implementado una aplicación en JAVA para visualizar el log del sistema gráficamente. Esta aplicación hace uso de la librería Canvas de JAVA para dibujar el log. La figura 10.2.1 muestra una vista de la representación gráfica de un log correspondiente a una comunicación TCP.

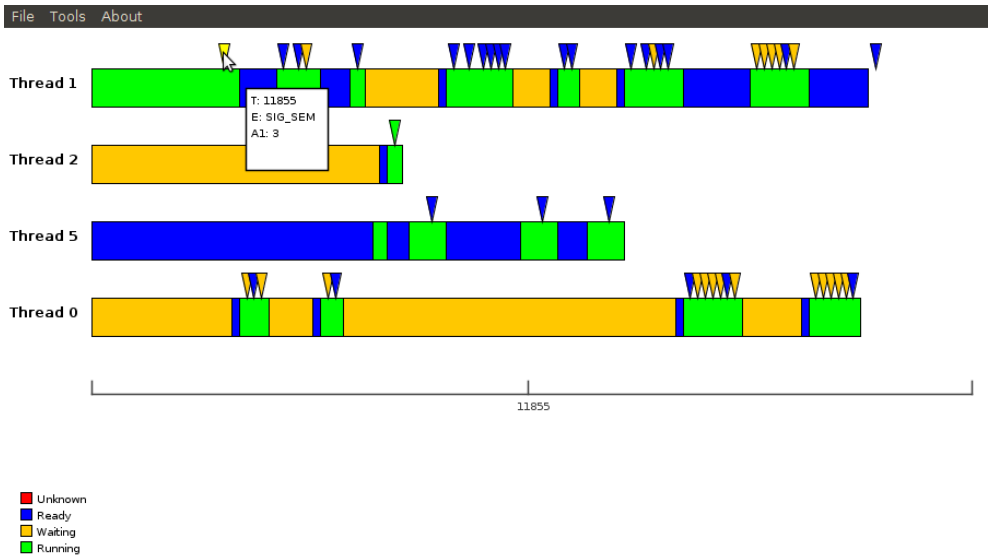

Figura 10.2.1: Vista de la aplicación OvLog.

Esta aplicación recoge un fichero binario con el vector de eventos del núcleo y lo analiza para generar los estados y los eventos que afectan a cada tarea en cada instante. Por el momento, la aplicación se encuentra todavía en desarrollo aunque, como puede verse en la figura anterior, ya se puede utilizar para analizar los sucesos del sistema.

10.3. Packer

La aplicación packer es uno de los últimos desarrollos realizados en este proyecto. El CM-BF548 dispone de una memoria flash de 8 MB, ubicada en la dirección de memoria 0x20000000. Este módulo soporta varias formas de arranque. Mientras se utiliza el JTAG, el sistema arranca desde la posición de memoria 0xFFA00000. Sin embargo, la placa de desarrollo se puede configurar para que la aplicación arranque desde flash. Así, en flash se graba una imagen de la aplicación que tiene el formato que puede verse en la figura 10.3.1. El objetivo del packer es realizar el empaquetado de todos los ficheros en la forma descrita.

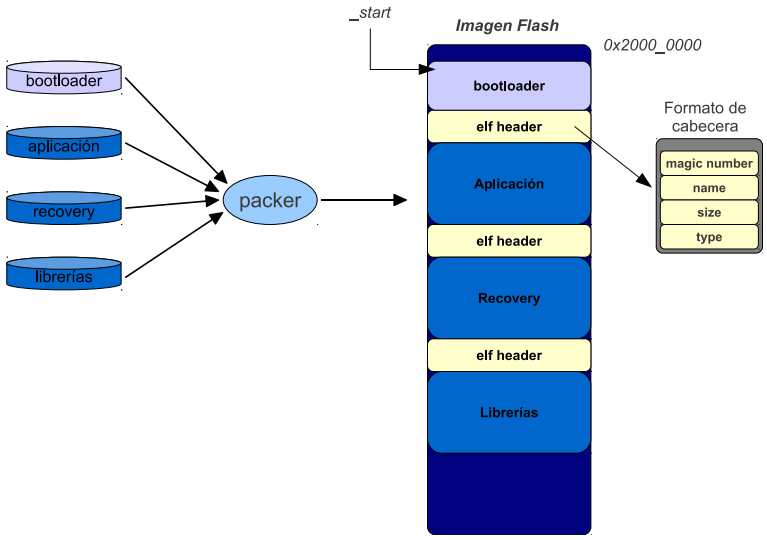

Figura 10.3.1: Formato de la imagen de flash.

Al empaquetar así los ficheros, se pueden cargar diferentes aplicaciones en el BF548. El formato de la imagen es el siguiente:

- **Bootloader**. Se explica en la sección 10.4.
- **Aplicación**. El fichero que contiene la aplicación principal aparece en primer lugar.
- **Recovery**. En caso de que el sistema sufriera un error irrecuperable, se activaría la aplicación de *recovery*. Esta aplicación se ha implementado de forma que arranca un servicio TCP desde el que se puede grabar una nueva imagen de flash.

- **Librerías**. Asimismo, los últimos ficheros que se graban se corresponden a librerías de enlace dinámico. Esta funcionalidad se encuentra todavía en desarrollo.

10.4. Bootloader

El bootloader es la aplicación encargada de cargar los programas de flash a memoria principal. Esta aplicación conoce la estructura de la imagen de flash y la recorre para cargar la aplicación correspondiente. El bootloader es una aplicación pequeña, que realiza una inicialización mínima del procesador y la memoria y ejecuta el cargador de aplicaciones.

La inicialización del bootloader consiste en establecer un puntero de pila adecuado e inicializar la memoria externa, de forma que se pueda utilizar para copiar las aplicaciones.

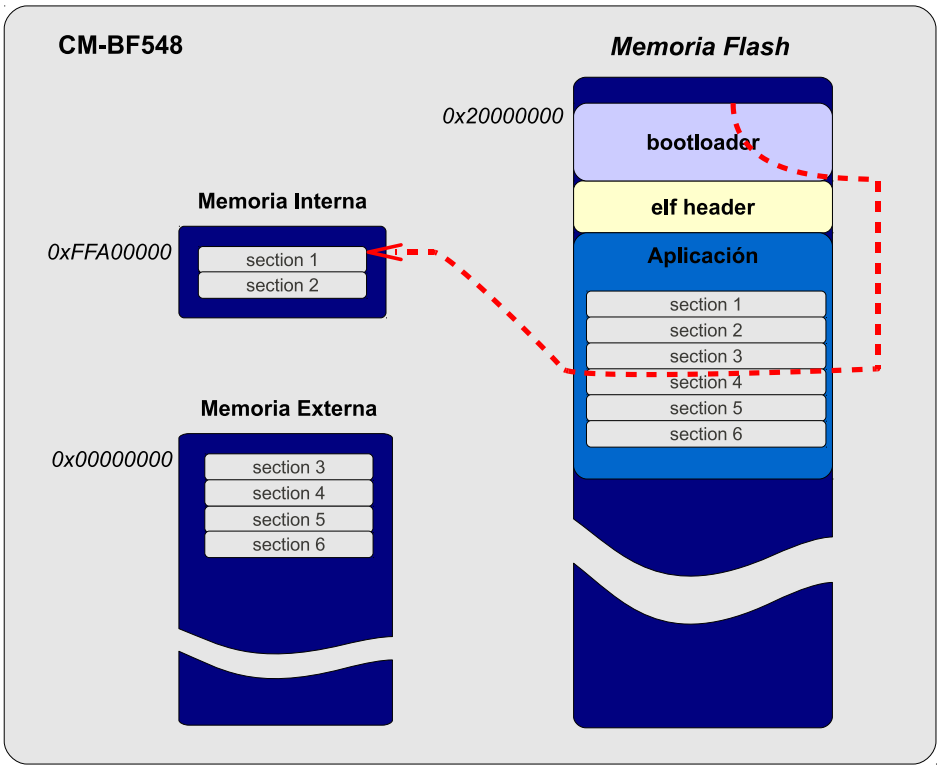

Figura 10.4.1: Funcionamiento del bootloader.

10.5. Recovery

Actualmente se está trabajando en una aplicación llamada *Recovery*. El objetivo de esta aplicación es abrir un servicio TCP a través de un puerto en el nodo y, con un cierto protocolo de comunicación, recibir un paquete de datos para grabarlos en memoria flash. Esta aplicación hace uso de un driver de flash desarrollado hacia el final de proyecto.

Con esta aplicación se persigue una forma cómoda de desarrollar aplicaciones para la plataforma. En el caso de que una aplicación sufriera un fallo crítico, se registraría y se reiniciaría el nodo. El cargador de arranque detectaría que el reset se ha producido por este hecho y activaría la aplicación de recovery, la cual permite grabar una nueva imagen en la memoria flash.

De momento, el desarrollo de la aplicación está relegado a la finalización de la integración de la pila lwIP en la plataforma. Aún así, el código es pequeño y ya se ha implementado. Por otro lado, resta testear el buen funcionamiento del driver de flash, todavía no comprobado.

Bibliografía

[1] "Smart embedded network of sensing entities." www.sense-ist.org. SENSE Consortium.

[2] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: A new dynamic memory allocator for real-time systems," *Real-Time Systems, Euromicro Conference on*, vol. 0, pp. 79–86, 2004.

[3] M. Masmano, "Posix real-time kernel." http://www.e-rtl.org/partikle/. Grupo de Informatica Industrial UPV.

[4] "The motor industry software reliability association." http://www.misra.org.uk/. MISRA C.

[5] "uclinux for blackfin." http://blackfin.uclinux.org/. Analog Devices Open Source Koop.

[6] "Gcc compiler manual." http://gcc.gnu.org/onlinedocs/gcc-4.5.0/gcc.pdf. GNU Project.

[7] "Ld, the gnu linker." http://www.ipp.mpg.de/ dpc/gnu/ld-2.9.1/ps/ld.ps.gz. GNU Project.

[8] "As, the gnu assembler." http://sourceware.org/binutils/docs/as/index.html. GNU Project.

[9] "Eclipse." www.eclipse.org. The Eclipse Foundation.

[10] "Gnu make." http://www.gnu.org/software/make/manual/make.pdf. GNU Project.

[11] "Icebear jtag adapter." http://www.section5.ch/icebear. Section 5.

[12] "Gdb, the gnu debugger." http://sourceware.org/gdb/current/onlinedocs/gdb.pdf.gz. GNU Project.

[13] T. Baker, "A stack-based resource allocation policy for realtime processes," *Real-Time Systems Symposium. Proceedings., 11th*, pp. 191 – 200, 1990.

[14] C. Bays, "A comparison of next-fit, first-fit, and best-fit," *Commun. ACM*, vol. 20, no. 3, pp. 191–192, 1977.

[15] "Executable and linking format." http://pdos.csail.mit.edu/6.828/2005/readings/elf.pdf, 1995. UNIX System Laboratory.

[16] A. Dunkels, "A lightweight tcp/ip stack." http://savannah.nongnu.org/projects/lwip/.

[17] "lwip wiki." http://lwip.wikia.com/wiki/LwIP_Wiki.

[18] "High-performance single-chip 10/100 ethernet controller with hp auto-mdix support." http://www.smsc.us/media/Downloads_Public/Data_Sheets/9218.pdf, 2010. SMSC.

[19] M. Masmano, *Dinamic Memory Management in Real-Time Systems*. PhD thesis, Universidad Politecnica de Valencia, 2006.

[20] "Blackfin processor programming reference." http://www.analog.com/static/imported-files/processor_manuals/blackfin_pgr.ref.man.rev1.3.pdf. Analog Devices.

[21] "Real-time executive for multiprocessor systems." http://www.rtems.com/.

[22] "Adsp-bf54x blackfin processor hardware reference (volume 1 of 2)." http://www.analog.com/static/imported-files/processor_manuals/bf54x_phwr_vol-1.pdf, 2008. Analog Devices.

[23] "Adsp-bf54x blackfin processor hardware reference (volume 2 of 2)." http://www.analog.com/static/imported-files/processor_manuals/bf54x_phwr_vol-2.pdf, 2008. Analog Devices.

[24] "uitron 4.03 specification." http://www.assoc.tron.org/spec/itron/itron403e/mitron-403e.pdf, 2007. TRON Association.

[25] J. Sanchez, G. Benet, and J. E. Simo, "A constant-time region-based memory allocator for embedded systems with unpredictable length array generation," in *Proceedings of the 15th IEEE International Conference on Emerging Techonologies and Factory Automation (ETFA2010)* (M. Marcos and R. Zurawsky, eds.), (Bilbao (Spain)), University of Basque Country, IEEE, 2010.