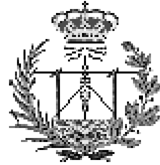
UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN



ORK (*Open Ravenscar Real Time Kernel*)

Sistema Operativo de Tiempo Real para Aplicaciones Espaciales

Implementación sobre Máquinas ERC32

Rodrigo García García

Promoción de 1995 – Septiembre de 2001

Título:

ORK (*Open Ravenscar Real Time Kernel*)
Sistema Operativo de Tiempo Real para Aplicaciones Espaciales
Implementación sobre máquinas ERC32

Resumen:

El proyecto ORK se enmarca dentro de una iniciativa de la Agencia Espacial Europea para potenciar el uso de software libre en la industria aeroespacial. ORK es un núcleo de tiempo real destinado principalmente al desarrollo de aplicaciones críticas multitarea con Ada 95 y las restricciones impuestas por el perfil de Ravenscar.

Tribunal:

Tutor : D. Juan Antonio de la Puente Alfaro
Vocal : D. Tomás Pedro de Miguel Moro
Secretario : D. Alejandro Antonio Alonso Muñoz

Suplente : D. Joaquín Seoane Pascual

Acuerdan otorgar la **calificación** de:

Madrid, de de 2001

Agradecimientos

Quisiera agradecer a mi familia todo el apoyo y la ayuda que me han prestado para que este proyecto haya podido salir finalmente adelante. Sin ellos nunca lo habría logrado. En especial, me gustaría destacar el esfuerzo y empeño que pusieron los últimos días para que todo saliera bien. A pesar de la distancia y de las prisas, consiguieron darme el ánimo necesario para terminar el trabajo y a ellos les debo el poder tener una copia impresa del mismo. Muchas gracias a todos. Os quiero de veras, aunque no haga falta que lo diga.

Quisiera agradecer también a los profesores Juan Antonio de la Puente y Alejandro Alonso el haberme dejado trabajar con ellos en la realización de este proyecto que tantas puertas me ha abierto. Gracias a vosotros he podido viajar y trabajar en el extranjero y conocer gente muy interesante. Aquí incluyo a todo el conjunto personas que participaron en el desarrollo, validación y soporte de ORK, con una mención especial al equipo de Remo. Gracias José por estar siempre ahí cuando te necesitaba, a Juan por ese buen humor y por ponerme las pilas cuando había que hacerlo y a Ramón porque me resolvía cualquier duda que tuviera sobre Linux.

También mando un recuerdo a mis compañeros de la Escuela que no me olvidan. Gracias por inundarme de mensajes el correo todos los días. Aunque ahora no esté con vosotros, me hacen sentir todavía como uno del grupo. Algún día tendremos que repetir esas partidas de mus en los descansos de las clases. Y a ver si conseguimos de una vez que Sergio no se vuelva a casa a las diez de la noche cada vez que salimos por ahí (le secuestramos si hace falta). Nos vemos pronto.

Por último, dar las gracias a todos mis nuevos compañeros del EPFL. Sobre todo a aquéllos que me han insistido una y mil veces en que terminara el proyecto para poder quedarme aquí un tiempo más. Gracias por los partidos de baloncesto, los patines, el volley-playa, las meriendas, el futbolín, el Uno, los Makumbazos... y todas las demás actividades "extra-escolares" que hacemos juntos. En fin, como diría Francesc: ¡ Qué chungos que sois !

Índice General

1 Introducción **1**
1.1 Los Sistemas de Tiempo Real 1
1.2 El Lenguaje de Programación Ada 5
1.3 El Perfil de Ravenscar 8
1.4 El Sistema Operativo ORK 8
1.5 Objetivos 11
1.6 Material 11
1.6.1 Recursos físicos 11
1.6.2 Recursos software 12

2 El ERC32 **13**
2.1 Arquitectura y Componentes del ERC32 13
2.1.1 La Arquitectura SPARC v7 15
2.2 La unidad de enteros TSC691E 16
2.2.1 Los Registros 17
2.2.2 Las Instrucciones 20
2.2.3 El Contador de Programa 23
2.2.4 El *pipeline* 23
2.2.5 Interrumpiendo al procesador: los *traps* 26
2.2.6 El Registro de Estado del Procesador (PSR) 31
2.2.7 El Registro de Ventana Inválida (WIM) 33
2.3 Las Subrutinas 34
2.4 Las instrucciones sintéticas 40
2.5 La Unidad de Coma Flotante TSC692E 42
2.6 El Controlador de Memoria MEC 43

3 El Cambio de Contexto de Bajo Nivel **45**
3.1 El cambio de contexto en el ERC32 46
3.1.1 Guardando las Ventanas de Registros 46
3.1.2 Salvando el Estado de la Unidad de Coma Flotante 48
3.2 La Rutina del Cambio de Contexto en ORK 51

3.2.1 ORK versión 2.1 y anteriores 51
3.2.2 ORK versión 2.2 y posteriores 52
3.3 Medición del tiempo de cambio de contexto 54
3.4 Las interrupciones y los cambios de contexto 57
3.5 La protección de pilas y los cambios de contexto 57

4 Las Interrupciones 60
4.1 Las Interrupciones en el ERC32 60
4.2 El procesamiento de las interrupciones en ORK 62
4.2.1 Uso de la FPU en manejadores de interrupción 65

5 La Protección de Pilas 71
5.1 Introducción 71
5.2 Implementación 73
5.2.1 Primera Opción de Configuración 74
5.2.2 Segunda opción de configuración 75
5.3 Posibles mejoras 79

6 Los periféricos del MEC 80
6.1 Programación de bajo nivel con Ada 95 80
6.2 El puerto serie 80
6.3 Temporización 85
6.4 La protección de memoria 86
6.5 Generación de interrupciones 87

7 Herramientas GNU en Sistemas Embarcados 88
7.1 El fichero de órdenes al enlazador 88
7.2 El fichero de especificaciones de GCC 90
7.3 El fichero de inicio 93
7.4 Las Rutinas de Apoyo en C 95

A Presupuesto 99
A.1 Presupuesto de Ejecución Material 99
A.1.1 Coste de Personal 99
A.1.2 Coste de Material 100
A.1.3 Coste Total de Ejecución Material 100
A.2 Gastos Generales y Beneficio Industrial 100
A.3 Presupuesto de Ejecución por Contrata 101
A.4 Honorarios 101
A.5 Presupuesto Total 101

Bibliografía **103**

Capítulo 1

Introducción

El presente proyecto se engloba dentro de otro más general, consistente en el diseño e implementación de un núcleo de sistema operativo de tiempo real. Dicho sistema operativo se conoce por las siglas ORK [11]. A lo largo de esta introducción, presentaremos el contexto en el que se sitúa ORK y las motivaciones que llevaron a su desarrollo. Finalmente, indicaremos cuál es la parte del proyecto conjunto que se pretende tratar en este proyecto fin de carrera, para posteriormente dedicarle toda nuestra atención en los siguientes capítulos.

1.1 Los Sistemas de Tiempo Real

Se conoce como sistemas de tiempo real a aquéllos cuyo buen funcionamiento depende no sólo de la corrección de su respuesta sino también del tiempo empleado en obtenerla. Estos sistemas deben entregar sus resultados antes de que se cumplan unos plazos de tiempo predeterminados. Existe una clasificación de los sistemas de tiempo real en función de la gravedad de la situación que se produciría caso de que el sistema sobrepasara los plazos que se le asignaron.

Sistemas de tiempo real acrítico (*Soft Real Time Systems*):

> En este tipo de sistemas, el incumplimiento de plazos conlleva una degradación de la calidad del servicio ofrecido. Si el incumplimiento es continuado, la degradación se puede hacer perceptible por el usuario, llegando a ser intolerable en el peor de los casos. Por lo tanto, la peor consecuencia del hecho de que el sistema sobrepase los plazos es la de provocar al usuario un cierto grado de malestar. Dentro de este grupo de sistemas podemos encontrar los servicios de audio y vídeo en tiempo real a través de Internet. Estos servicios pretenden transmitir un flujo constante de información a través de una red de

Sistemas de Tiempo Real	**Críticos**	**Acríticos**
Plazos	Estrictos	Relajados
Consecuencias de un retraso	Graves	Molestas
Comportamiento	Determinista	Variable

Figura 1.1: Diferencias entre los sistemas de tiempo real críticos y acríticos

paquetes que no garantiza la entrega y menos aún la entrega en un plazo determinado. Centrémonos por un momento en el ejemplo del servicio de vídeo. Para dar sensación de movimiento continuo al ojo humano, el sistema deberá proporcionar 25 imágenes por segundo. Si, por cualquier motivo, la red perdiera paquetes o no los entregara a tiempo, el sistema no podría disponer de la información necesaria para generar cada segundo las 25 imágenes mencionadas. Aunque existen técnicas para minimizar los efectos de la pérdida de paquetes, un usuario con cierta agudeza visual podría notar la diferencia. Si la pérdida de paquetes se acentuara, el usuario percibiría “saltos” de una imagen a otra, hasta hacer la situación insoportable (dependiendo siempre del nivel de exigencia del usuario).

Sistemas de tiempo real crítico (*Hard Real Time Systems*):

Los sistemas pertenecientes a este grupo deben cumplir estrictamente los plazos de tiempo asignados, de lo contrario pueden producirse pérdidas económicas importantes o incluso pérdida de vidas humanas. Ejemplos representativos de esta clase son los sistemas empleados en control industrial y de centrales nucleares o los utilizados en los sectores ferroviario, automovilístico y aeroespacial, por citar algunos de los más importantes. El más mínimo fallo en este tipo de sistemas puede ser fatal y, debido a sus características, tener repercusión internacional, como demuestran los ejemplos del primer vuelo del cohete Ariane 5 [1] o la más reciente misión Mars Polar Lander de la NASA [2]. Hay que señalar, sin embargo, que ambos fracasos se debieron a la incorrección de los resultados ofrecidos por el sistema y no a un incumplimiento de los plazos de ejecución.

Vemos que los límites de tiempo que se imponen a un sistema de tiempo real no suelen ser arbitrarios, sino que responden a algún parámetro físico del mundo

[1] `http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html`

[2] `http://mars.jpl.nasa.gov/msp98/lander`

que le rodea. El mundo real, por su propia naturaleza, es concurrente. En todo momento, se producen una multitud de fenómenos físicos distintos que se superponen en el tiempo. Para interactuar correctamente con el mundo exterior, es muy útil la división del trabajo en tareas. El ser humano está habituado a pensar y actuar en función de la tareas que tenga encomendadas, por eso también le es más fácil programar sistemas teniendo en cuenta las distintas tareas que han de realizar. Estas tareas pueden ser periódicas, si han de repetirse cada cierto intervalo de tiempo o esporádicas, si se deben llevar a cabo como respuesta a un suceso determinado. Puede ocurrir que algunas tareas deban esperar a que otras terminen para poder cumplir su función. También es posible que varias tareas tengan que sincronizarse entre sí porque compartan un recurso que no puedan o no deban usar todas a la vez.

Para programar sistemas de tiempo real existen dos planteamientos básicos: usar un ejecutivo cíclico o un sistema software que soporte multitarea [5] [15]. El ejecutivo cíclico es simplemente un bucle con el código de todas las tareas que se repite periódicamente. Con el ejecutivo cíclico, es el programador el que decide cómo se repartirá el tiempo de procesamiento entre las distintas tareas. Se escoge como periodo del ciclo el máximo común divisor del periodo de cada tarea y luego se decide qué partes de las distintas tareas se ejecutarán en cada ciclo. Este método es trabajoso y poco flexible, ya que no permite tareas esporádicas y se comporta mal con tareas periódicas de periodos muy diferentes. Además, exige una replanificación de todo el sistema si se añade o se cambia alguna de las tareas. A cambio, se asegura por construcción que el sistema va a cumplir los plazos. Mucho más flexible es usar un sistema software que soporte la multitarea directamente. Puede ser el propio sistema operativo sobre el que se vaya a programar el que proporcione la multitarea o puede que ésta venga dada por una biblioteca software que contenga procedimientos mediante los cuales se puedan crear, ejecutar y terminar tareas en tiempo de ejecución. También serán necesarios mecanismos de sincronización entre tareas, y es que en este caso el programador no tiene el control absoluto de la situación, porque es el sistema multitarea el encargado de hacer la planificación de las tareas y asignarles tiempo de procesador. Gracias a los sistemas multitarea, se libera al programador del trabajo de planificación y se consigue una mayor flexibilidad en el diseño: se pueden incluir tareas esporádicas y periódicas de cualquier periodo y añadir o eliminar tareas del sistema fácilmente. Como contrapartida, se pierde la certeza del cumplimiento de los plazos: aparece el concepto de analizabilidad.

Muchos de estos sistemas multitarea no fueron diseñados para que se programaran aplicaciones de tiempo real sobre ellos, por lo que no se pensó en que fueran analizables temporalmente, es decir, que se pudiera saber de antemano si una tarea iba a ejecutarse en un plazo determinado o no. Normalmente, lo que hacen estos sistemas es tratar de ser justos y repartir el tiempo de procesador entre

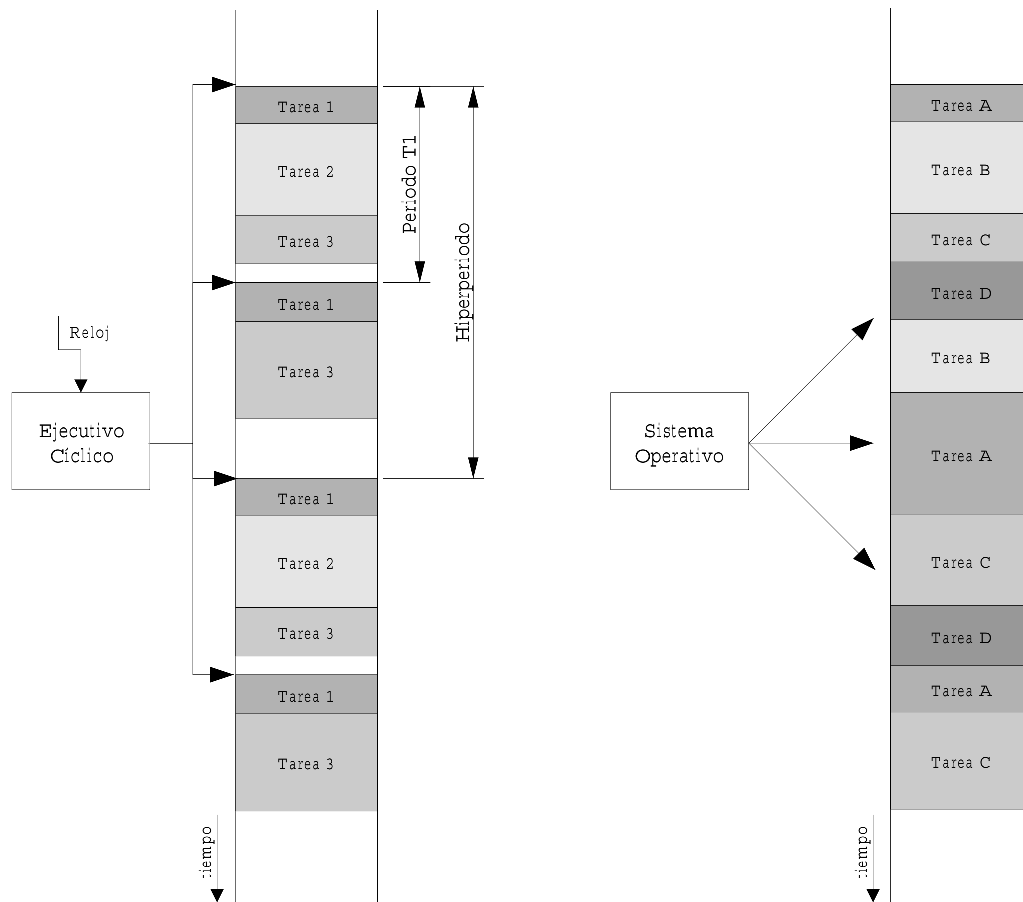


Figura 1.2: Comparativa: ejecutivo cíclico vs. sistema operativo multitarea

todas las tareas por igual. Durante bastante tiempo, la multitarea ha sido desechada de los proyectos de tiempo real crítico debido a esta falta de analizabilidad. Sin embargo, el estudio de los métodos de planificación ha demostrado la posibilidad de construir sistemas de tiempo real totalmente analizables y seguros usando modelos de multitarea restringidos [4]. Estos modelos se basan en dar prioridades de ejecución a las tareas, ejecutándose en cada momento la tarea más prioritaria.

Los sistemas de tiempo real están experimentando un gran auge en los últimos tiempos y nada hace suponer que la tendencia vaya a cambiar a corto plazo. Más bien al contrario, podemos aventurarnos a decir que su crecimiento será todavía mayor en los próximos años. Consideremos algunas de las causas que han propiciado o que pueden propiciar este crecimiento:

- El aumento de electrónica en los automóviles, dando lugar a sistemas de antibloqueo de ruedas y antiderrapaje, mecanismos pretensores del cinturón de seguridad y de disparo del "airbag" o nuevos sistemas de navegación por satélite.

- El número creciente de sistemas informáticos embarcados en pequeños electrodomésticos, que realizan funciones de control sobre los mismos o que sirven para la comunicación personal, donde merece una mención especial el gran éxito de la telefonía móvil.

- La enorme expansión de Internet, con una fuerte demanda de servicios multimedia en tiempo real y estudios sobre calidad de servicio en la Red.

- La próxima puesta en marcha de operadores de telecomunicaciones de banda ancha, que permitirán una mayor velocidad en la transmisión de datos, haciendo posibles servicios como la videoconferencia de alta calidad.

Todas estas nuevas áreas, junto con las ya tradicionales de los sistemas de tiempo real, nos permiten augurar un futuro prometedor a todo lo relacionado con este tipo de sistemas.

1.2 El Lenguaje de Programación Ada

Ada es un lenguaje de programación de alto nivel. Lleva el nombre de Ada en honor a la que fue condesa de Lovelace, Augusta Ada Byron, hija de Lord Byron y considerada la primera programadora de la historia, dado que trabajó junto a Charles Babbage en el desarrollo de su Motor Analítico. Aunque Ada es un lenguaje de propósito general, está especialmente indicado para la construcción de aplicaciones de gran tamaño o de programas críticos que requieran altos niveles de seguridad y fiabilidad.

El lenguaje Ada surgió como un proyecto del Departamento de Defensa de los Estados Unidos para reducir las enormes sumas de dinero que destinaba a software. En 1983 apareció el primer estándar ANSI (*American National Standards Institute*) que definía el lenguaje Ada: el *Manual de Referencia para Ada 83* [3]. Más tarde, el documento se propuso a ISO, que lo adoptó como estándar número 8652 en 1987 [14]. La última versión del lenguaje, conocida como Ada 95, es el resultado de la evolución tecnológica del software desde que Ada 83 vio la luz. La revisión del estándar se hizo con la colaboración de la comunidad de usuarios del lenguaje Ada, incluyendo finalmente nuevos paradigmas de programación como la orientación a objetos y, debido a la diversidad de necesidades, forzando la división del lenguaje en lo que se conoce como el Núcleo y los Anexos especializados [1].

Figura 1.3: Lady Ada Byron

La seguridad y robustez de Ada vienen dados por el tipado fuerte de los datos y por el elevado número de comprobaciones que se realizan tanto en tiempo de compilación como en tiempo de ejecución, incluyendo el tratamiento de excepciones. La biblioteca en tiempo de ejecución (*Run-Time Library*) de Ada [13] está compuesta de una serie de rutinas que se cargan junto con el programa ejecutable y es responsable, entre otras cosas, de hacer las comprobaciones en tiempo de ejecución. Puede detectar, por ejemplo, el desbordamiento de un número entero o la asignación de valores a una matriz con el índice fuera del rango declarado. Ada trata también de forma especial a los tipos de datos que referencian a otros, conocidos generalmente como punteros, ya que su uso incontrolado se considera peligroso. El alcance de los punteros está restringido en Ada y se evitan, en la medida de lo posible, los punteros que no referencien a nada. Además, la sintaxis del lenguaje Ada, parecida a la de Pascal, facilita su lectura y las estructuras del lenguaje facilitan la programación modular permitiendo aplicar conceptos de ingeniería del software al desarrollo de programas grandes.

Las facilidades que ofrece el lenguaje Ada para programar sistemas de tiempo real son muchas y variadas. La multitarea está integrada en el propio lenguaje, lo cual hace mucho más fácil la programación concurrente. Dispone de mecanismos para la declaración, creación y destrucción de tareas, así como otros para la asignación de prioridades, sincronización y elección de métodos de planificación. La biblioteca en tiempo de ejecución de Ada se encarga de traducir estos mecanismos

del lenguaje a las operaciones de bajo nivel correspondientes. Estas operaciones pueden ser llamadas al sistema operativo subyacente, si éste es multitarea, o pueden ser operaciones que generen un entorno multitarea independiente del sistema operativo sobre el que se ejecute la aplicación Ada. Por ejemplo, hay versiones de compiladores Ada para GNU/Linux y MS-DOS. GNU/Linux es un sistema operativo multitarea, por lo que la biblioteca de ejecución de Ada puede hacer llamadas a las funciones del sistema operativo que controlen la multitarea. Sin embargo, MS-DOS [3] es un sistema operativo secuencial. En este caso, la biblioteca Ada deberá usar unos mecanismos propios que le permitan simular la multitarea sin ayuda del sistema operativo. Esto no quiere decir que no pueda usar ningún servicio de MS-DOS, incluso puede que necesite apoyarse en algunos de ellos para implementar la multitarea (la temporización, por ejemplo), pero no podrá hacer llamadas directas al sistema que creen tareas, las ejecuten o las paren, ya que simplemente no existen.

Otra gran ventaja que ofrece Ada para programar sistemas de tiempo real es, precisamente, el Anexo de Tiempo Real. El núcleo del lenguaje Ada no dice nada acerca de cómo se debe hacer la planificación de las tareas ni sobre sus prioridades. Cada implementación particular de Ada es libre de escoger el modelo de planificación que desee. Es en este anexo especializado donde se especifica cómo escoger un modelo de planificación determinado, definiéndose uno estándar, y permitiendo que cada implementación pueda incluir otros. También explica la forma de dar prioridades a las tareas y define políticas de interacción entre las tareas y los objetos que se usan para sincronizarlas. Para poder cumplir con los requisitos temporales más estrictos, se define un reloj de tiempo real monótono de alta resolución, dando una gran importancia a los límites de precisión, exactitud y granularidad que ha de tener dicho reloj. Por último, pero de vital importancia para el proyecto que nos ocupa, se definen una serie de restricciones a la multitarea de Ada, aplicables mediante configuración, que eliminen ciertas características del lenguaje para facilitar la construcción de bibliotecas en tiempo de ejecución muy eficientes. Hay que añadir además que, para ser conforme con el Anexo de Tiempo Real, la implementación debe cumplir también las especificaciones del Anexo de Programación de Sistemas, donde se describen algunas utilidades para la programación a bajo nivel.

A todas estas ventajas que ofrece el lenguaje Ada, hay que sumarle las que se derivan de la existencia de una implementación libre: GNAT [18] [4]. Cuando

[3] Quizás el ejemplo de MS-DOS no sea muy afortunado, ya que existe una implementación de *Pthreads* para dicho sistema operativo. Para hacer más portable la biblioteca en tiempo de ejecución, GNAT aprovecha que la mayoría de sistemas operativos son compatibles con las tareas POSIX (*Pthreads*)

[4] `http://www.gnat.com`

hablamos de libre, nos referimos a que se distribuye bajo licencia GPL [5]. Básicamente, dicha licencia da al usuario la libertad de usar, copiar, distribuir, estudiar y modificar el software, siempre que se concedan a los demás los mismos derechos en caso de redistribución. El hecho de poder modificar el compilador de Ada para adaptarlo a las necesidades del proyecto hizo que GNAT fuera la única opción aceptable. Además, el compilador GNAT está integrado con el resto de las herramientas de desarrollo GNU, como el depurador GDB, que también son software libre y, por lo tanto, susceptibles de ser utilizadas y adaptadas al proyecto.

1.3 El Perfil de Ravenscar

En el apartado anterior vimos que el Anexo de Tiempo Real permitía la aplicación de restricciones a la multitarea de Ada. El Perfil de Ravenscar [6] es, fundamentalmente, un conjunto de dichas restricciones. Fue elaborado durante la celebración del octavo IRTAW (International Real Time Ada Workshop) que tuvo lugar en la localidad de Ravenscar (North Yorkshire, Inglaterra), de ahí el nombre del perfil.

El Perfil de Ravenscar tiene como objetivo aumentar el determinismo y la eficiencia de las aplicaciones Ada que hagan uso de la multitarea, así como disminuir el tamaño de las mismas y permitir la implementación de bibliotecas en tiempo de ejecución eficientes (objetivo inicial de las restricciones). Este determinismo permitirá que los sistemas diseñados conformes al perfil puedan someterse a los procesos de certificación requeridos para la operación de sistemas de tiempo real crítico.

Las exigencias de analizabilidad hicieron que, para la completa definición del Perfil de Ravenscar, fuera necesario añadir nuevas restricciones a las ya existentes en el Anexo de Tiempo Real. Sin embargo, el perfil no sólo se compone de las mencionadas restricciones. También dicta cuáles han de ser las políticas de planificación y de sincronización de las tareas. Aunque las políticas escogidas son, precisamente, las únicas definidas por el Anexo de Tiempo Real.

1.4 El Sistema Operativo ORK

El sistema operativo ORK [6] es el resultado de un contrato de la Agencia Espacial Europea con la Universidad Politécnica de Madrid, en el que también han participado activamente, como empresas o instituciones subcontratadas, la Universidad Rey Juan Carlos, la Universidad de York y la División Espacio de Construcciones Aeronáuticas SA.

El proyecto ORK entra dentro de una iniciativa de la ESA destinada a fomentar

[5] http://www.gnu.org/copyleft/gpl.html
[6] http://www.openravenscar.org

el uso de software libre en aplicaciones espaciales. Las ventajas del software libre no han pasado desapercibidas a los ojos de la ESA, que ha visto la oportunidad de suprimir el coste por licencias y de reducir su dependencia de los suministradores convencionales de software. Aunque todavía se muestran reticencias a la adopción de soluciones libres debido a la inexistencia de programas de mantenimiento claros. La aparición de nuevas empresas que trabajan con softaware libre y que, precisamente, basan su estrategia de negocio en servicios de consultoría y mantenimiento, puede hacer que el panorama cambie. En el caso que nos ocupa, es la propia UPM, junto con CASA, la encargada del mantenimiento a largo plazo.

ORK es un núcleo de sistema operativo de complejidad y tamaño reducidos. Está diseñado para que sobre él se puedan desarrollar aplicaciones críticas de tiempo real escritas en Ada 95 y conformes al perfil de Ravenscar. Debido a la popularidad del lenguaje C, también se impuso, mediante contrato, la condición de que se pudieran desarrollar programas para ORK en dicho lenguaje. Para cumplir con este requisito y aprovechando las capacidades de Ada 95 para interactuar con otros lenguajes de programación, ORK dispone de un interfaz C que permite a los programas escritos en este lenguaje hacer llamadas al sistema operativo. La mayor parte de ORK está programada en Ada 95, exceptuando algunas rutinas de bajo nivel que están escritas en lenguaje ensamblador y otras de apoyo que lo están en C.

ORK está integrado dentro del sistema de compilación libre GNAT. Para su adaptación, se ha aprovechado la estructura en capas del compilador GNAT:

GNARL (*GNU Ada Run Time Library*) Esta es la capa que implementa la biblioteca en tiempo de ejecución de Ada. Contiene todo el modelo de tareas del lenguaje: planificación, sincronización, etc... Y es independiente, en su mayoría, del sistema operativo subyacente, lo cual facilita su adaptación a distintos sistemas. Los programas Ada compilados con GNAT hacen llamadas a esta capa a través de su interfaz, conocida como GNARLI. Este interfaz no debe modificarse, de lo contrario también habría que alterar el nombre de las llamadas que genera el compilador. Algunos paquetes de GNARL, en cambio, sí tuvieron que modificarse para adaptarlos a las características de ORK y se añadieron otros nuevos, específicos para ORK.

GNULL (*GNU Low-Level Library*) Esta biblioteca se encarga de traducir las llamadas a la biblioteca Ada (GNARL) en llamadas de bajo nivel al sistema operativo. GNARL accede a los servicios de esta capa a través de su interfaz: GNULLI. Al igual que pasaba en el apartado anterior, dicho interfaz no deberá ser modificado, ya que esto requeriría cambiar también la capa superior (GNARL en este caso). Al tratarse GNULL de una capa dependiente del sistema operativo, los paquetes incluidos en ella han sido

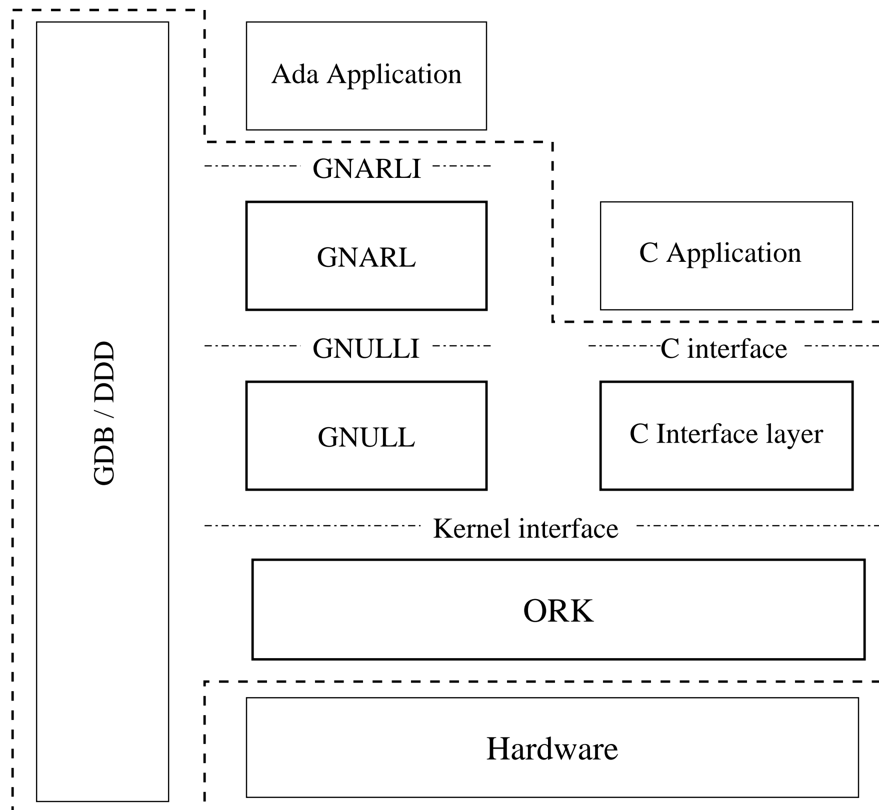


Figura 1.4: Arquitectura en capas de GNAT/ORK

convenientemente adaptados a ORK. A pesar de lo dicho, esta capa también es bastante independiente del sistema operativo, ya que la mayoría de sistemas modernos implementan la biblioteca *Pthreads*, es decir, el conjunto de especificaciones POSIX para tareas. La capa GNULL se diseñó como un subconjunto de las normas POSIX, por lo que su implementación más popular hace llamadas a las funciones de *Pthreads*, al ser ésta la forma más fácil de implementarlas. Sin embargo, no siempre es la más eficiente (ver [9]).

Por debajo de GNULL se encuentra el sistema operativo. En nuestro caso, el sistema operativo es ORK. También se puede acceder a ORK a través del interfaz C, como se puede observar en la figura 1.4. Más abajo aún, se encuentra la máquina desnuda: el *hardware*.

Si nos adentramos en la estructura de ORK, veremos que no todos sus módulos acceden directamente al hardware. Una gran parte de ORK es independiente de la máquina sobre la cual se ejecute. De esta manera, la transición de ORK hacia otras arquitecturas requerirá menos esfuerzo, teniendo que modificarse solamente

las partes que dependan del hardware.

1.5 Objetivos

Este proyecto trata sobre las partes de ORK que sí dependen del hardware, en especial, las programadas por el autor de este documento. La versión del sistema operativo que comentaremos será la que trabaja sobre la máquina para la que se desarrolló ORK inicialmente: el ERC32.

Se pretende que el escrito sea de utilidad a todo aquél que vaya a dedicarse a trabajar sobre las partes de ORK dependientes de la máquina, ya sea para portarlo a otras arquitecturas o para perfeccionar la implementación sobre ERC32. También puede servir al usuario avanzado de ORK que esté interesado en los detalles de bajo nivel o, simplemente, a la persona que tenga curiosidad por conocer la estructura interna del sistema operativo.

A lo largo de la introducción se ha presentado el sistema operativo ORK desde un punto de vista generalista, sin suponer ningún conocimiento previo al lector y exclusivamente para situarlo dentro de su contexto. A partir de este momento, debido al contenido técnico de los siguientes capítulos, se necesitará tener cierta familiaridad con el lenguaje Ada y el Perfil de Ravenscar. En caso de que se considere necesario, se pueden consultar los excelentes libros y artículos incluidos en la bibliografía . También es aconsejable tener nociones de hardware y de programación en ensamblador del ERC32, especialmente si se quieren estudiar y comprender los listados de ORK escritos en dicho lenguaje.

1.6 Material

Los recursos materiales que se han empleado para la realización de este proyecto se pueden dividir en dos grupos. Aquéllos que tienen una entidad física (los recursos materiales propiamente dichos) y otros caracterizados por su intangibilidad (los programas, utilidades y software en general).

1.6.1 Recursos físicos

Estos recursos han sido aportados por el Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid.

- Ordenador personal PC compatible con procesador Intel Pentium III y frecuencia de reloj de 500 Mhz, 128 Mbytes de memoria principal y 8,5 Gbytes de disco duro.

- Red local tipo Ethernet. Velocidad de transmisión de datos: 10 Mbps.
- Conexión a Internet a través de Red Iris.
- Libro sobre programación en ensamblador de máquinas SPARC v7 [16].

1.6.2 Recursos software

Los programas usados pertenecen a sus respectivos dueños. Si bien se ha tratado de usar software libre en la medida de lo posible, dado el carácter y objetivos del proyecto.

- GNU/Linux como sistema operativo sobre el cual trabajar en el ordenador personal.
- El compilador de Ada 95 GNAT v3.13p (aunque a lo largo del proyecto se han usado diferentes versiones) junto con su código fuente.
- El emulador de ERC332 para Linux: TSIM [7] un simulador del conjunto de instrucciones de la arquitectura SPARC v7 y los periféricos propios del ERC32.
- Los editores de texto Kedit, Kwrite y Emacs para la escritura del código de ORK y de la presente memoria.
- El programa de mantenimiento de versiones CVS, para el control del desarrollo del proyecto y el trabajo en equipo.
- El programa de dibujo Xfig para el trazado de las figuras que aparecen en la memoria.
- El sistema de preparación de documentos LaTeX para la confección de este escrito.

[7] http://www.gaisler.com

Capítulo 2

El ERC32

Se llama ERC32 a un núcleo de computación fabricado y comercializado por la empresa Atmel Wireless and Microcontrollers [1], anteriormente conocida como Temic Semiconductors. El conjunto de elementos que componen el ERC32 forman una implementación de la séptima versión de la arquitectura SPARC. Esta implementación posee ciertas cualidades que la hacen especialmente adecuada para su uso en aplicaciones espaciales. Cabe destacar, por ejemplo, que sus componentes son tolerantes a radiación y que el contenido de los circuitos de memoria está protegido mediante códigos de redundancia cíclica. Estas características son indispensables para que el ERC32 pueda soportar las grandes cantidades de radiación cósmica que se reciben en el espacio exterior, sin que por su causa se produzcan fallos de funcionamiento.

A lo largo de este capítulo presentaremos los conceptos básicos de la arquitectura y programación del ERC32, haciendo un mayor hincapié en aquéllos que han sido de especial interés para la realización de este proyecto.

2.1 Arquitectura y Componentes del ERC32

Los tres elementos fundamentales del ERC32 son la unidad central de proceso o unidad de enteros, el coprocesador matemático o unidad de coma flotante y el controlador de memoria. Estos tres elementos, inicialmente separados físicamente, han sido integrados en la misma pastilla en las últimas versiones del ERC32. Se comunican entre sí y con los circuitos de memoria a través de un bus de datos y otro de direcciones, ambos de 32 bits. La disposición de estos elementos se muestra en la figura 2.1 (antiguos ERC32). Las explicaciones que siguen tomarán como referencia los modelos antiguos de ERC32, ya que la separación física de los componentes permite una mejor exposición de los diferentes conceptos.

[1] `http://www.atmel.com`

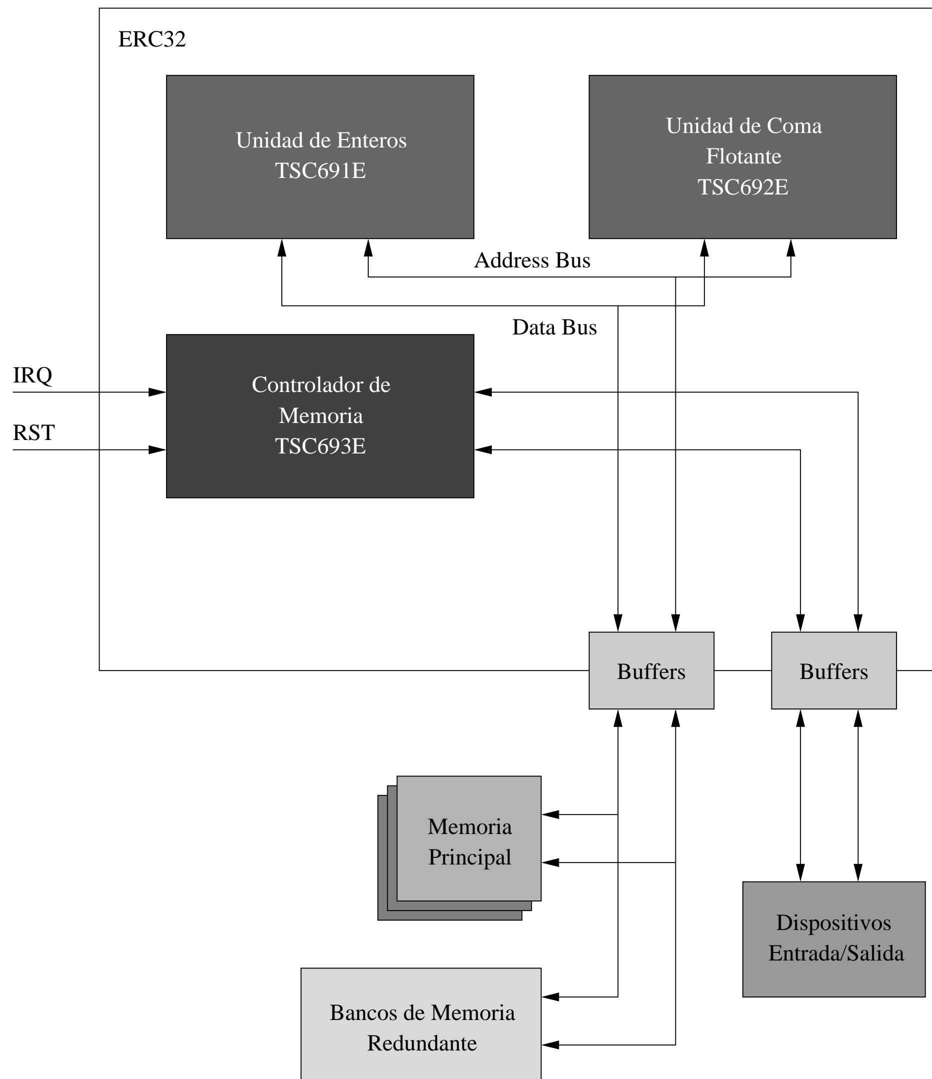


Figura 2.1: Esquema de bloques del ERC32

En la figura se puede apreciar que, además de estar conectados mediante los buses de direcciones y de datos, los tres componentes principales disponen de otras señales para comunicarse entre ellos. La explicación detallada del propósito de todas estas señales está fuera de los objetivos de este documento. Por el momento, sólo diremos que tienen misiones de sincronización (e.g. que el procesador deba esperar la finalización de una operación de coma flotante) y de comunicación con los periféricos (e.g. interrupciones), además de otras funciones específicas del controlador de memoria, alguna de las cuales se verán en la sección dedicada a este dispositivo.

2.1.1 La Arquitectura SPARC v7

SPARC [2] es el nombre que recibe una arquitectura de ordenadores abierta que fue originalmente propuesta y desarrollada por Sun Microsystems, existiendo otros fabricantes que la han adoptado en la construcción de sus productos. En ella se basan, por ejemplo, las conocidas estaciones de trabajo Ultra de Sun. Y en ella está basada también la máquina que nos ocupa: el ERC32. Concretamente, el ERC32 está implementado siguiendo las especificaciones de la versión 7 de dicha arquitectura.

Esta versión define, básicamente, una arquitectura de computadores RISC de 32 bits. El conjunto de instrucciones y el comportamiento que un sistema basado en SPARC debe tener quedan completamente determinados por la arquitectura. Sin embargo, no se dice nada acerca de cómo se debe implementar un sistema SPARC, dejando libertad en este aspecto a los fabricantes de hardware. A pesar de todo, hay ciertos aspectos de la arquitectura que están especialmente pensados para ser implementados sobre mecanismos hardware concretos que permiten una ejecución más eficiente.

El desarrollo de la arquitectura SPARC se encuentra actualmente en su novena versión, que supone un gran cambio respecto a las versiones precedentes, puesto que en ella se habla ya de máquinas de 64 bits, en lugar de los 32 bits de las versiones séptima y octava. Además de estar desfasado en este aspecto, observamos también que el ERC32 dispone de una velocidad de reloj en torno a los 10 megahercios, frente a las frecuencias del orden de gigahercios que alcanzan las máquinas actuales más potentes. El lector podrá preguntarse por qué se emplean microprocesadores tan antiguos en un área de tecnología punta como es la aeroespacial. La respuesta es que las arquitecturas de reciente aparición no están lo suficientemente probadas para alcanzar los niveles de fiabilidad requeridos por este tipo de aplicaciones (recuérdese, por ejemplo, el caso del error de

[2] `http://www.sparc.com`

división del Pentium [3]). Otra razón es que, como hemos señalado anteriormente, los microprocesadores necesitan ser adaptados a las duras condiciones de trabajo de las misiones espaciales, causadas principalmente por la ausencia de atmósfera protectora. Esta adaptación requiere de una cantidad de tiempo no despreciable, en comparación con la rapidez de la evolución del mercado informático. También requiere una inversión de capital importante en un producto que, por otra parte, no se beneficia de la economía de escala de los procesadores dedicados al consumo doméstico, ya que el número de potenciales clientes de procesadores de este tipo es reducido.

Veremos ahora un resumen de las características de los dos componentes del ERC32 compatibles con la arquitectura SPARC v7: la unidad de enteros y la unidad de coma flotante.

2.2 La unidad de enteros TSC691E

El TSC691E es una unidad central de proceso RISC de 32 bits compatible con la arquitectura SPARC v7. Con el objetivo de obtener el máximo rendimiento posible, el TSC691E utiliza varias técnicas que le permiten aumentar el ritmo de ejecución de las instrucciones, acercándose al límite de una instrucción por cada ciclo de reloj. Estas técnicas se pueden resumir en los siguientes puntos:

- El uso de un conjunto reducido de instrucciones simples (de ahí la denominación RISC) que, en condiciones normales, no necesiten más de un ciclo de reloj para la ejecución de cada una de sus fases. Si exceptuamos aquéllas que necesiten ocupar el bus para acceder a la memoria.

- La longitud de las instrucciones en memoria es fija (32 bits), lo que ayuda a una sencilla, rápida y eficiente decodificación de las mismas. No es, por lo tanto, necesario procesar el primer byte de la instrucción (su código de operación) para conocer su longitud.

- Las operaciones aritmético-lógicas se realizan enteramente en los registros del procesador. Si un operando requerido está en memoria, primero hay que cargar su valor en un registro, luego se opera sobre el registro y posteriormente se guarda el resultado en memoria (arquitectura de carga y almacenaje *load/store*).

- El uso del *pipeline*. Al dividir el ciclo de ejecución de las instrucciones en sus distintas fases, se pueden ejecutar las fases de distintas instruccio-

[3] `http://epubs.siam.org/sam-bin/dbq/article/29395`

nes al mismo tiempo, permitiendo así un cierto grado de paralelismo. Este importante mecanismo se tratará en detalle más adelante.

2.2.1 Los Registros

Como acabamos de ver, el uso de los registros es esencial en la optimización del rendimiento de la máquina. Para disminuir en lo posible el número de accesos a memoria (más lentos), el ERC32 dispone de una gran cantidad de registros donde ir almacenando los datos que se usarán más tarde. Éstos se organizan en ventanas de registros, tal y como se contempla en la definición de la arquitectura SPARC v7.

Una ventana de registros se compone de un total de 24 registros:

- Ocho registros *in*: %i0, %i1, ... , %i7.
- Ocho registros *local*: %l0, %l1, ... , %l7.
- Ocho registros *out*: %o0, %o1, ... , %o7.

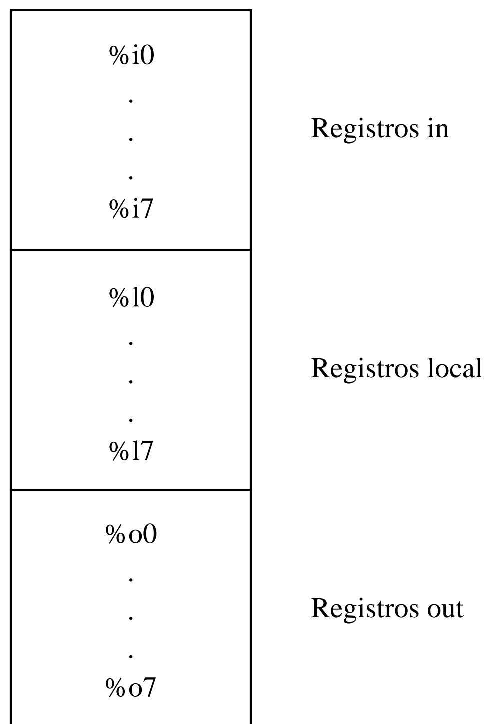


Figura 2.2: Ventana de registros

El ERC32 implementa 8 ventanas de registros como la arriba descrita. Sin embargo, esto no quiere decir que en conjunto tenga 192 registros (8×24), ya que los registros *out* de una ventana se solapan con los registros *in* de la contigua, por lo que en realidad el número total de registros es 128 (8×16).

Por si esta última apreciación no le ha quedado suficientemente clara al lector, la figura 2.3 muestra la disposición de las ventanas de registros en el ERC32.

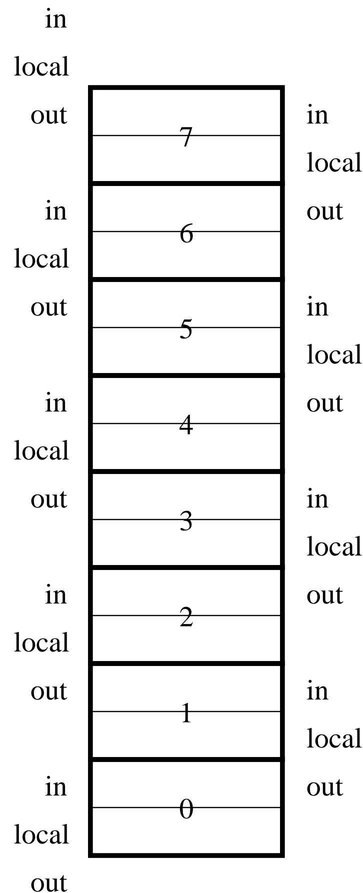


Figura 2.3: Pila circular de registros

Esta disposición recibe el nombre de pila circular de registros. La explicación a esta denominación es la siguiente. Como ya dijimos más arriba, los registros *out* de una ventana, se superponen con los *in* de la anterior. De esta forma, los registros *out* de la ventana uno se corresponden con los *in* de la ventana cero, los *out* de ventana dos son los *in* de la número tres... y así sucesivamente hasta que llegamos a la última ventana. Los registros *in* de la última ventana cierran el círculo y se solapan con los *out* de la ventana cero. Para reflejar este comportamiento cíclico,

también se suele representar la pila circular de registros de la forma indicada en la figura 2.4.

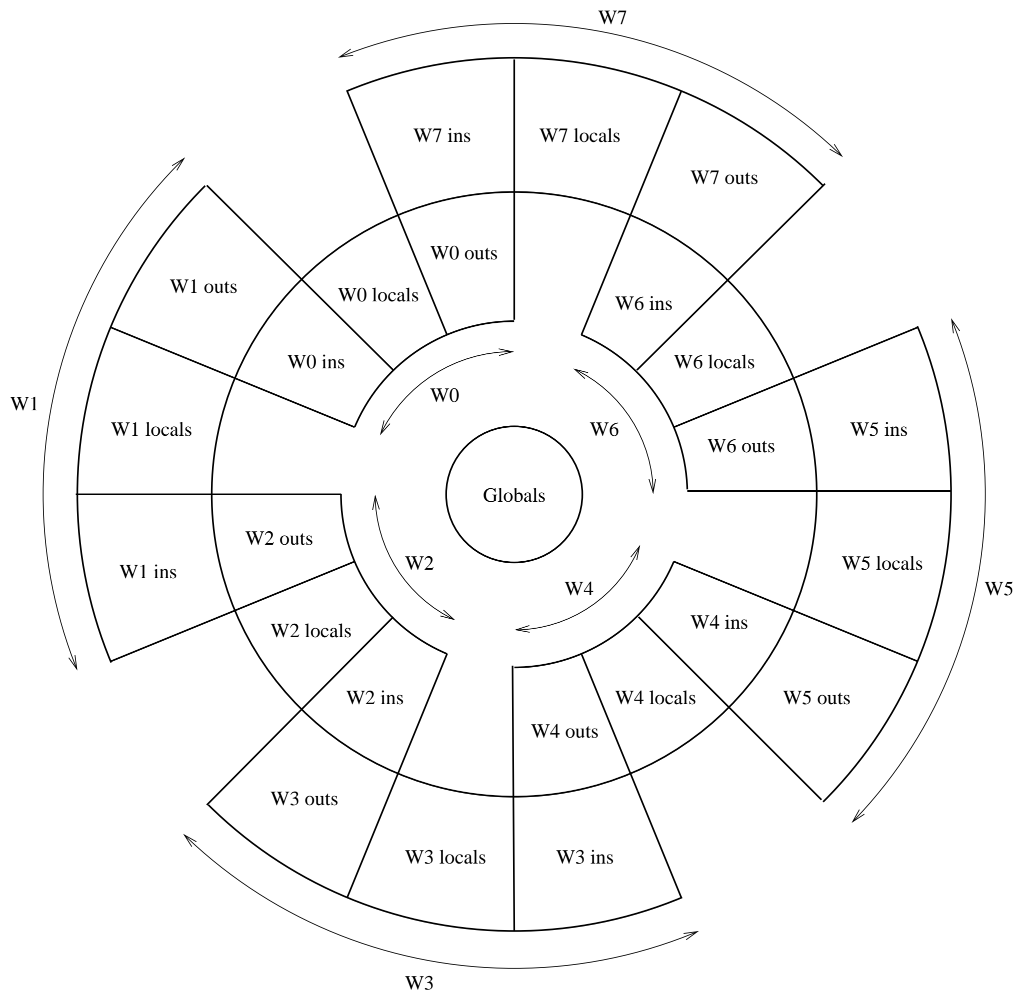


Figura 2.4: Representación circular de la pila de registros

En un momento determinado de la ejecución de un programa sólo se tiene acceso a una ventana de registros: la indicada por el campo CWP (*Current Window Pointer*, indicador de la ventana actual) del registro de estado del procesador (PSR), registro muy importante que discutiremos más adelante. El valor del CWP se puede cambiar directamente escribiendo en el registro de estado o indirectamente usando ciertas instrucciones que cambian su valor automáticamente. La escritura del registro de estado es posible únicamente cuando el procesador se encuentra en modo supervisor.

Además de los registros de la pila circular, el ERC32 posee 8 registros más de

propósito general. Son los registros globales %g0, %g1, ... , %g7. Estos registros son visibles siempre, independientemente de la ventana seleccionada por el CWP, por lo que se puede acceder a ellos en cualquier momento. De entre los registros globales hay uno especial: el registro %g0. Este registro mantiene continuamente el valor cero. En realidad, el registro %g0 no suele existir como tal, sino que, por construcción del hardware, está cableado directamente a tierra. Contrariamente a lo que pueda parecer a primera vista, el registro %g0 tiene varias e interesantes aplicaciones, como descubriremos en algunos de los siguientes apartados.

2.2.2 Las Instrucciones

El TSC691E reconoce todo el conjunto de instrucciones definido por la arquitectura SPARC v7. Todas las instrucciones tienen, por motivos de eficiencia, 32 bits de longitud, es decir, ocupan 4 octetos de memoria. También por motivos de eficiencia, las instrucciones deben estar alineadas en el espacio de memoria, de tal modo que su primer octeto comience en una dirección divisible entre 4. Esto hace, por ejemplo, que las instrucciones de salto absoluto no necesiten almacenar la dirección completa a la que saltar, sino que basta con que conozcan los 30 bits más significativos: ya saben que los dos bits restantes van a ser cero. De esta manera, las instrucciones de salto absoluto pueden codificarse con 32 bits, como todas las demás, siendo la longitud de su código de operación la más corta de todas las instrucciones: dos bits.

El formato general de una instrucción es el siguiente:

```
mnem    reg_fuente1, reg_fuente2 o inmediato, reg_dest
```

Donde *mnem* es el mnemónico de la instrucción. Por ejemplo, la instrucción para sumar el contenido de los registros %o0 y %o2 y guardar el resultado de la operación en %o3 se escribiría así:

```
add     %o0, %o1, %o2
```

Hay algunas instrucciones que no siguen este formato general. Tenemos el caso de las instrucciones encargadas de transferir el contenido de una posición de memoria a un registro y viceversa, que sólo tienen dos operandos:

```
ld      mem, reg_dest
st      reg_fuente, mem
```

Las instrucciones de transferencia de memoria usan la indirección para poder acceder a cualquier posición del espacio de memoria del ERC32. Si usaran un valor inmediato de 32 bits para este propósito, sólo este valor ocuparía toda la

longitud de la instrucción, haciendo imposible su codificación. Por eso, primero cargan la dirección a la que quieren acceder en un registro y luego se realiza la operación de transferencia mediante direccionamiento indirecto sobre dicho registro. El problema ahora consiste en cargar un valor inmediato de 32 bits (la dirección de memoria en cuestión) en un registro del procesador sin que ocurra lo mismo que antes. Dicho problema se resuelve cargando el valor inmediato en dos etapas. Para ello, se usa la instrucción *sethi*, que explicaremos mediante un ejemplo:

```
sethi   %hi(0x2000220), %o0
```

Esta instrucción carga los 22 bits más significativos de %o0 con el valor inmediato indicado, dejando a cero los restantes 10 bits menos significativos. La pseudo-operación %hi es una directiva del ensamblador que sirve para seleccionar los 22 bits más significativos del valor inmediato que se le aplica. Suele usarse siempre junto con la instrucción *sethi*. Para cargar los 10 bits menos significativos se utiliza una instrucción *or* lógico:

```
or      %lo(0x2000220), %g0, %o0
```

De la misma manera que la directiva %hi seleccionaba los 22 bits más significativos del valor inmediato, %lo selecciona los 10 bits menos significativos del parámetro que se le pasa, ejerciendo de función complementaria. En esta instrucción, podemos ver además una de las aplicaciones del registro %g0. Como lo que queremos es cargar directamente el valor inmediato en %o0, el *or* debe efectuarse con un operando que tenga valor cero para obtener el efecto deseado, ya que el *or* lógico de cualquier número con cero devuelve la cifra de partida. La manera más rápida de conseguir este valor nulo es acudir al registro %g0.

Ahora ya podemos utilizar el registro %o0 para direccionar una instrucción de transferencia cualquiera:

```
ld      [%o0], %o1
```

Para ahorrar una instrucción en este proceso (la carga de los 10 bits menos significativos mediante el *or* lógico), también se puede usar el indireccionamiento indexado:

```
ld      [%o0 + %lo(0x2000220)], %o0
```

Las instrucciones de transferencia vistas tienen una variante que permite trabajar con dos registros y dos direcciones de memoria a la vez, en lugar de usar un registro y una dirección solamente. Estas variantes se conocen como *cargar*

doble y *guardar doble*. Utilizan el registro indicado en la instrucción y el que le sigue junto con la dirección de memoria dada y la consecutiva. Pero para que estas variantes puedan ejecutarse correctamente (sin que se produzca un error de alineamiento) se deben cumplir dos condiciones: que el número asociado al registro elegido sea divisible por dos y que la dirección de memoria especificada sea múltiplo de ocho. Por ejemplo:

```
sethi   %hi(0x2000000), %o0

ldd     [%o0], %l0           ! Carga el contenido de la direccion
                             ! de memoria 0x2000000 en %l0 y el
                             ! de la direccion 0x2000004 en %l1

ldd     [%o0], %g1           ! Error: %g1 no es un registro par.

std     %g2, [%o0 + 0x2]     ! Error: La direccion de memoria no
                             ! esta alineada correctamente:
                             ! 0x2000002 no es divisible por ocho.
```

También existen instrucciones de transferencia que permiten trabajar con operandos de menor tamaño, pero no se han usado en la realización de este proyecto. Nos referimos aquí a la parte programada manualmente en ensamblador. El compilador, por su parte, ha podido generar muchas de estas instrucciones al traducir el código en Ada a lenguaje de máquina.

Por el contrario, las instrucciones de transferencia doble se han utilizado extensivamente en el código ensamblador perteneciente a este proyecto, ya que permiten guardar y restaurar el conjunto de los registros del procesador más rápidamente que las instrucciones de transferencia simples.

Además de las instrucciones de transferencia, el TSC691E posee un conjunto de instrucciones aritmético-lógicas típico; incluyendo instrucciones de adición, substracción, desplazamientos, lógica booleana ("y" lógico, "o" lógico, ...). Sin embargo, las instrucciones de multiplicación y división se consideraron demasiado complicadas para una máquina RISC de estas características y se implementan a base de ejecutar multiplicaciones y divisiones parciales. A partir de la versión 8 de la arquitectura SPARC, la multiplicación y la división se efectúan con una única instrucción.

Para una descripción detallada del conjunto de instrucciones se puede consultar el manual [19]. En cualquier caso, en los siguientes apartados se irán describiendo las instrucciones que guarden relación con el tema de discusión.

2.2.3 El Contador de Programa

El contador de programa (PC) es un registro especial del procesador que apunta a la instrucción que se está ejecutando en ese mismo instante de tiempo. Contiene, por lo tanto, una dirección de 32 bits. El programador no tiene acceso directo a este registro, que cambia de valor después de cada ciclo de reloj, recogiendo el contenido del registro contador de programa siguiente (nPC) cada vez que se termina de ejecutar la instrucción actual.

El nPC (*next Program Counter*) contiene la dirección de memoria donde se encuentra la próxima instrucción a ejecutar. Normalmente, esta dirección coincide con la de la instrucción consecutiva a la que está en ejecución, es decir, el nPC contiene el mismo valor que el PC más cuatro. Cada vez que el procesador termina de interpretar una instrucción, el contenido del nPC se incrementa en cuatro unidades, apuntando de nuevo a la siguiente instrucción. Este comportamiento se repite constantemente, excepto cuando la instrucción interpretada es una instrucción de salto. En ese caso, si la condición para efectuar el salto se cumple, se efectúa un cambio en el flujo del programa, escribiendo el nPC con la dirección indicada por la instrucción de salto. El valor del PC se modifica indirectamente a través de esta operación, ya que en el siguiente ciclo recibirá el valor del nPC.

Los *traps* también modifican el contenido del PC y del nPC. Este mecanismo de interrupción de la ejecución normal de un programa se tratará en detalle más adelante (ver apartado 2.2.5).

2.2.4 El *pipeline*

El *pipeline* es un paradigma de trabajo que la informática ha heredado del campo de la producción industrial. Es un símil de lo que en la industria se conoce como cadena de montaje. Para comprender fácilmente el concepto del *pipeline*, lo mejor es considerar el ejemplo de una planta de fabricación de automóviles:

Anteriormente a la implantación en las fábricas de la cadena de montaje (iniciada por Henry Ford en 1913), los coches se fabricaban uno por uno. Es decir, en el proceso de fabricación, se le iban añadiendo al automóvil todas las piezas necesarias (el motor, el chasis, las ruedas...), hasta que finalmente se daban por terminados. Era sólo entonces cuando se procedía a la fabricación del siguiente coche.

Con la llegada de la cadena de montaje (*pipeline* en inglés), el trabajo se divide en sus distintas fases. Los trabajadores se colocan formando una línea en la que cada uno tiene su misión específica: uno pone las ruedas, otro coloca el motor, otro pinta el coche, etc... De esta forma, se pueden tener varios coches en producción al mismo tiempo. Y más importante aún, y como consecuencia de lo dicho, una vez que el primer coche ha salido de la cadena de montaje, el tiempo que se tarda

en producir un nuevo vehículo se reduce al tiempo que requiere una sola de las etapas de la cadena (suponiendo que todas duren lo mismo).

En el dominio de la informática se han hecho varios usos del paradigma del *pipeline*. En el campo de los sistemas operativos, el lector que haya utilizado sistemas UNIX recordará que usando el símbolo "|" puede encadenar distintos programas. De tal suerte que la salida producida por el programa a la izquierda del "|" se propaga hacia la entrada del programa situado a la derecha del símbolo del *pipeline*. Por ejemplo, en la orden:

```
$ ls | sort
```

El programa *ls* envía la lista de ficheros en el directorio actual al programa *sort*, que los ordena alfabéticamente y los muestra por pantalla.

En el campo de los microprocesadores, la metáfora del *pipeline* se lleva más lejos, siendo mucho más parecida al ejemplo automovilístico presentado. Los microprocesadores sin *pipeline* deben ejecutar completamente una instrucción antes de empezar a procesar la siguiente. En los que sí tienen *pipeline*, sin embargo, lo que se hace es dividir la ejecución de una instrucción en cada una de sus fases fundamentales. En el procesador que nos ocupa, se identifican cuatro fases por instrucción:

1. Obtención (Fetch): El procesador usa la dirección donde se encuentra la instrucción para obtenerla de la memoria

2. Decodificación: La instrucción se carga en el registro de instrucciones y se decodifica. El procesador lee los operandos de la instrucción y calcula la dirección de la siguiente instrucción a ejecutar.

3. Ejecución: El procesador ejecuta la instrucción y guarda el resultado en unos registros temporales. Los *traps* que haya pendientes son tenidos en cuenta por su orden de prioridad y es en esta etapa donde se toman los *traps* internos.

4. Escritura: Si no se tomó ningún *trap*, el resultado se guarda en el registro de destino especificado en la instrucción.

Cada una de estas fases es lo suficientemente simple como para poder llevarse a cabo en un solo ciclo de reloj. Por lo tanto, una instrucción que pase por las cuatro fases consecutivamente tardará cuatro ciclos de reloj en terminar su ejecución. Sin embargo, cuando la primera instrucción haya salido del *pipeline*, ya habrá otras tres instrucciones dentro del mismo. Éstas irán saliendo a un ritmo de una por ciclo de reloj; es por eso que estas instrucciones se conocen como de *ciclo único*.

Existen otras instrucciones que deben retrasar el *pipeline* porque tienen que ocupar el bus de datos para obtener información de la memoria (por ejemplo, las instrucciones *load/store*). Estas instrucciones se denominan de *multiciclo* ya que, al ocupar el bus, evitan que el procesador pueda obtener la siguiente instrucción inmediatamente, necesitando ciclos de reloj adicionales. Otra causa que puede producir retrasos en el *pipeline* es el uso, por parte de una instrucción, de un registro que fue modificado por la instrucción anterior, dando lugar a problemas de carreras. En cualquiera de estos casos, el procesador inserta automáticamente los ciclos de espera adecuados para que no se produzcan resultados "sorpresa". Hay que señalar que otros procesadores de arquitectura RISC no realizan este trabajo, siendo el propio programador (o compilador) el que debe insertar los ciclos de espera adecuados en forma de instrucciones *nop*.

Por todo lo dicho, no se deberían usar instrucciones que usen un registro que haya sido modificado por la instrucción anterior, ya que ello conlleva una disminución de la eficiencia. El TSC691, a pesar de todo, gestiona automáticamente el *pipeline* para obtener un resultado correcto. El único caso en el que el programador debe obligatoriamente ocuparse del *pipeline* es el de la ejecución de una instrucción de salto. El *pipeline* se hace visible al programador durante la ejecución de este tipo de instrucciones, para que se pierda el menor número de ciclos de reloj posible.

El problema viene dado porque el procesador ignora la dirección a la que debe saltar hasta el momento en el que ejecuta la instrucción de salto. Pero, según se ha explicado, en ese momento ya hay otras instrucciones dentro del pipeline esperando a ser terminadas. Si la condición para que se produjera el salto se cumplió, las instrucciones que se hallen dentro del *pipeline* no serán válidas, ya que el contador de programa habría sido modificado y apuntaría a otra dirección de la memoria (la indicada por la instrucción de salto). El *pipeline* debería ser descargado para volverse a llenar con las instrucciones correctas, lo que supondría un retraso intolerable del programa por cada instrucción de salto que hubiera en él.

Para solucionarlo, el *pipeline* se hace aparente al programador en sus dos primeras etapas. Como se puede ver en la tabla de fases del *pipeline*, es en la segunda etapa cuando se calcula la dirección de la siguiente instrucción a ejecutar. Esto hace que la instrucción situada justo después de una instrucción de salto entre en la primera etapa del *pipeline* antes de que el procesador sepa si se va a efectuar el salto o no. Para no tener que sacar a esta instrucción del *pipeline*, la instrucción que se coloca después de una instrucción de salto normal se ejecuta siempre, tanto si el salto se produce como si no.

Esto puede parecer un poco extraño a simple vista, ya que no suele ser habitual un comportamiento de este tipo en programación secuencial. Lo normal sería que la instrucción que sigue a la de salto no se ejecutara si el salto no se toma. Para

clarificar este punto, veamos un ejemplo:

Dada la siguiente secuencia de instrucciones, ¿Qué ocurre si el salto se toma?

```
subcc   %l0, 1, %l0  ! restar 1 al contenido de %l0
bz      fin          ! si el resultado es cero, saltar a fin
add     %l1, 1, %l1  ! sumar 1 al contenido de %l1
```

1. Se ejecuta la instrucción de substracción. Si el resultado es cero, la instrucción lo indicará en los códigos de condición del registro de estado.

2. Se procesa la instrucción de salto y, al mismo tiempo, el nPC apunta ya a la instrucción siguiente, haciendo que entre en la primera fase del *pipeline*.

3. Suponiendo que se cumplió la condición de salto, la instrucción *bz* habrá cambiado el contenido del nPC para continuar la ejecución del programa en el punto indicado por la etiqueta “fin”.

El lugar que es ocupado por una instrucción situada tras otra de salto se conoce con el nombre de espacio de retardo (*delay slot*), debido a que el salto no se efectúa inmediatamente, sino que es retardado, permitiendo la ejecución de la instrucción colocada en este lugar.

Existe una variante de las instrucciones de salto que permite anular la ejecución de la instrucción que se encuentra en el *espacio de retardo*. Las variantes usan el mismo mnemónico que sus correspondientes instrucciones de salto normal añadiendo el sufijo “,a”. Por ejemplo, para la instrucción de salto *bz*, la variante con anulación es *bz,a*.

Para el uso correcto de estas variantes, es importante señalar que la instrucción en el espacio de retardo se anula sólo cuando el salto no se toma. Por el contrario, si la condición de salto sí se cumple, la instrucción en el espacio de retardo se ejecutará normalmente. También es importante notar que la anulación de la instrucción supone la pérdida de ciclos de reloj al tener que retirarla del *pipeline*. La utilidad de estas variantes se limita a la optimización de bucles.

La única variante que no sigue este comportamiendo descrito es la correspondiente a la instrucción de salto incondicional: *ba*, lo cual es bastante lógico. La instrucción *ba,a* hace que la instrucción en el espacio de retardo no se ejecute nunca. La anulación de la instrucción provocará una pérdida de eficiencia en el *pipeline* como se explica más arriba, por lo que esta instrucción no debe usarse salvo por motivos especiales.

2.2.5 Interrumpiendo al procesador: los *traps*

La mayor parte de los procesadores modernos disponen de un mecanismo mediante el cual se puede interrumpir la ejecución normal del programa para atender

algún suceso especial. En la terminología de la arquitectura SPARC, estos sucesos reciben el nombre de *traps*.

En esta misma arquitectura, se distinguen dos clases de traps:

Traps síncronos: Se producen como consecuencia de la ejecución de una instrucción que dispara el trap, por eso son síncronos con el procesador. La instrucción que los provoca puede tratarse de una instrucción específica para generar traps (las de formato **txx**) o alguna que implícitamente esté realizando una operación anómala, como por ejemplo, que esté accediendo a una posición de memoria no implementada o mal alineada.

Traps asíncronos: Son producidos por elementos hardware externos que demandan de esta manera la atención del procesador. Son lo que se conoce habitualmente en la literatura por el nombre común de interrupciones. La señal emitida por el dispositivo externo para interrumpir al procesador puede ser recibida por éste en cualquier momento, de ahí el nombre de traps asíncronos.

Cuando el procesador recibe un trap del tipo que sea, ejecuta los siguientes pasos:

1. Cambia a estado supervisor, en caso de no encontrarse ya en dicho estado, indicando así que el procesamiento que sigue es excepcional

2. Decrementa el CWP en una unidad (módulo el número de ventanas implementadas) y, ya en la nueva ventana reservada, guarda el contenido del PC y del nPC en los registros %l1 y %l2 (ver figura 2.5).

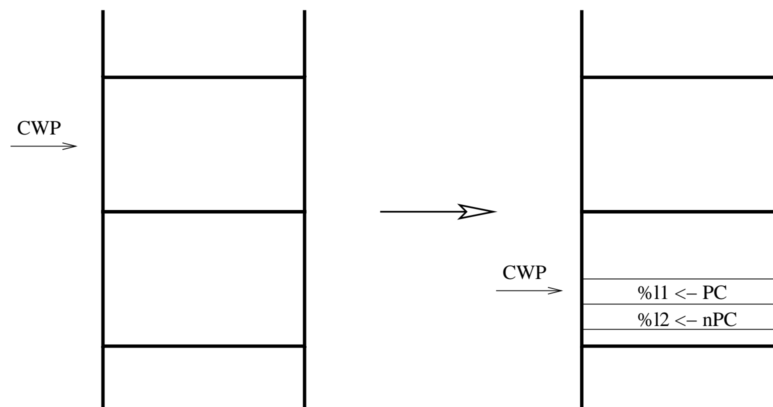


Figura 2.5: Reserva de ventana durante el procesamiento de un trap

3. Salta a la dirección donde se halle el manejador del trap. Para calcular dicha dirección, utiliza el registro de la tabla de traps (TBR) y el tipo de trap (tt) que se haya producido.

El registro base de la tabla de traps TBR (*Trap Base Register*) indica la posición de memoria en la que comienza la tabla donde se almacenan las cuatro primeras instrucciones de cada manejador de trap. Lo habitual es que estas cuatro instrucciones provoquen un salto a otra dirección de memoria donde se halla el manejador de trap completo, ya que es muy raro que un manejador se componga de cuatro instrucciones solamente.

La presencia de instrucciones en la tabla de traps es una particularidad de la arquitectura SPARC. La mayoría de procesadores convencionales (como la familias Intel 80x86 y Motorola 68000) almacenan en la tabla de vectores de interrupción la dirección de los manejadores y no las propias instrucciones del manejador en sí.

Además de la dirección de la base de la tabla de traps, el TBR tiene un campo en el que se registra el tipo del último trap que se produjo. El tipo de trap (tt) es un número comprendido entre 0 y 255 que identifica unívocamente la causa de la interrupción.

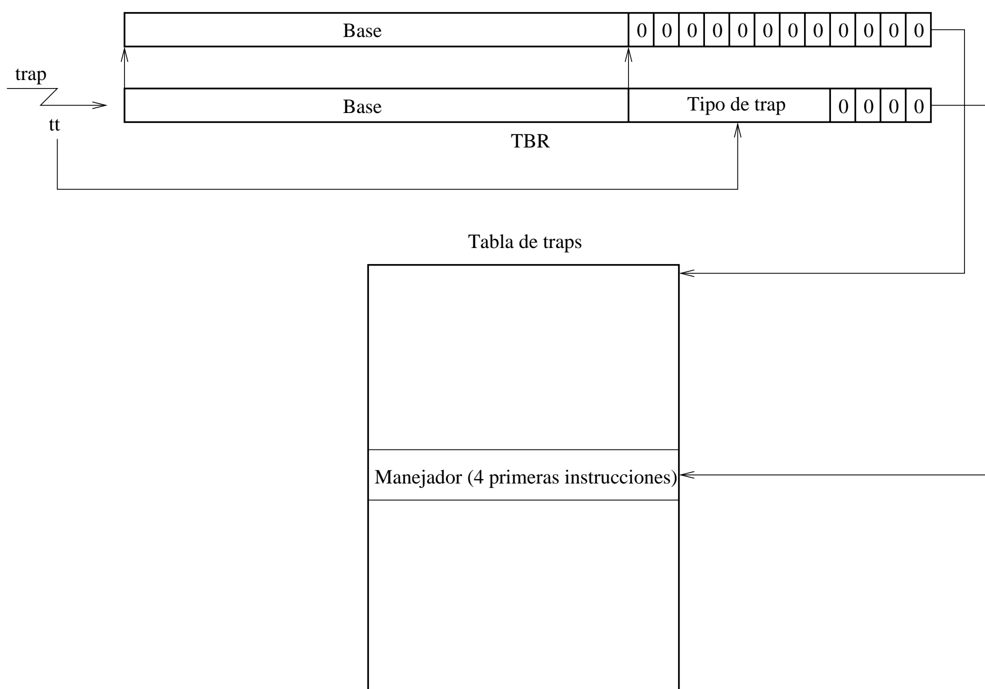


Figura 2.6: Búsqueda del manejador de trap

Una vez se han seguido estos pasos y se ha completado la ejecución del manejador, el procesador debe volver al punto en el que se interrumpió el programa justo antes de recibirse el trap. Este punto es una dirección de memoria que depende de la clase de trap que se haya producido:

- Si el trap es síncrono, la dirección que se almacenó en %l1 es la que tenía el contador de programa cuando apuntaba a la instrucción que provocó el trap.

- Si el trap es asíncrono, el procesador no se interrumpe inmediatamente, sino que espera a que se complete la instrucción actual antes de atender el trap. Por lo tanto, el valor del contador de programa almacenado en %l1 corresponde, en este caso, a la dirección de memoria de la instrucción siguiente a la que se estaba ejecutando en el momento en el que llegó el trap.

Esta diferencia entre trap síncronos y asíncronos hay que tenerla muy en cuenta a la hora de programar el retorno desde un manejador de trap. Las dos instrucciones que se han de emplear para volver al programa principal desde un manejador son el salto con enlace y la instrucción especial “retorno de trap”.

El salto con enlace: Esta instrucción sirve para saltar a cualquier dirección del espacio de memoria del ERC32, guardando en el registro indicado la dirección desde la cual se salta (el enlace). Este mecanismo es el que se usa en las llamadas a subrutina para que se pueda conocer la dirección de retorno. Ejemplo:

```
jmpl    %o0, %o7  ! Salta a la direccion contenida en %o0 y
                  ! guarda la direccion de retorno en %o7.
```

La instrucción rett: La instrucción de retorno de trap realiza las siguientes funciones:

1. Carga en el nPC la dirección indicada por los operandos fuente.
2. Incrementa el CWP en una unidad (módulo el número de ventanas implementadas).
3. Devuelve al procesador al estado en el que se encontraba antes de producirse el trap, ya sea supervisor o usuario.

Aunque la instrucción rett es una instrucción de salto retardada, en la práctica no se pone ninguna instrucción detrás de ella, ya que siempre está precedida de una instrucción de salto con enlace.

La combinación de dos instrucciones de salto retardado provocan la anulación del espacio de retardo de la segunda instrucción de salto.

Para ilustrar el uso conjunto de estas dos instrucciones veremos algunos ejemplos:

- Cómo volver a la instrucción que provocó el trap, si éste fue síncrono

  En este caso, la dirección a la que hay que volver estará almacenada en %l1 y el nPC en %l2. La secuencia de instrucciones que se necesita es ésta:

```
jmpl    %l1, %g0
rett    %l2
```

  Hay que recordar que la instrucción de salto con enlace es retardada, por lo que el *rett* se ejecutará siempre. El efecto de estas instrucciones se puede ver en la figura 2.7.

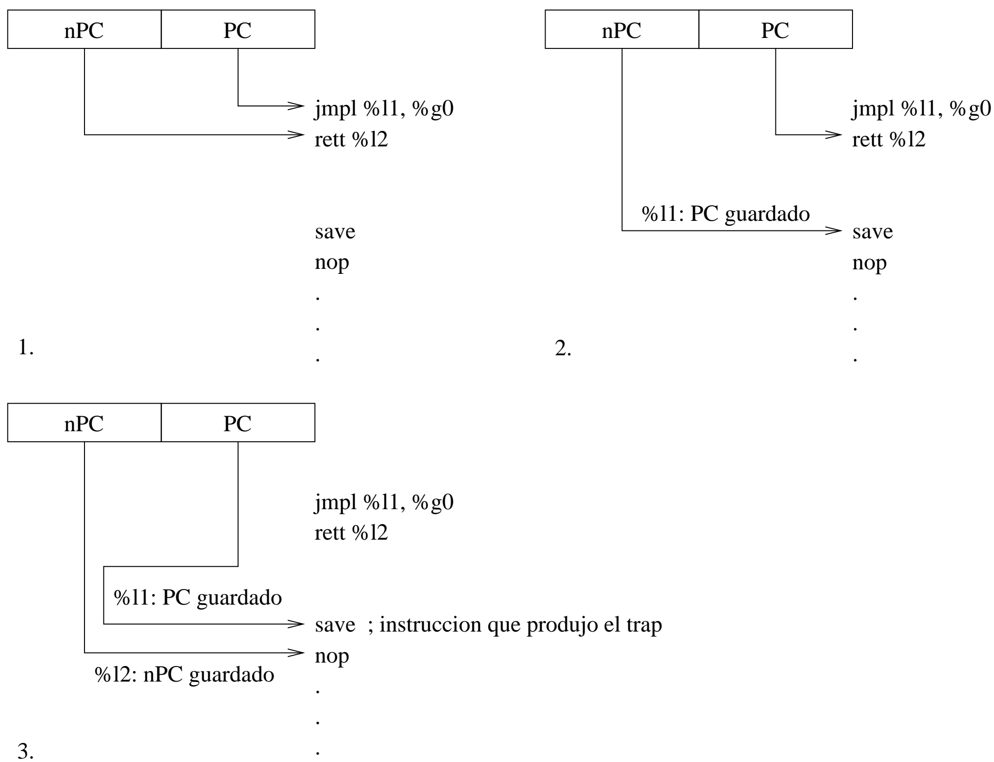


Figura 2.7: Retorno de trap

- Cómo volver a la instrucción siguiente a la que se estaba ejecutando cuando se produjo el trap.

1. Traps asíncronos.

   Se aplica la misma secuencia de instrucciones que en el apartado anterior. Como hemos visto, la dirección de la instrucción que sigue a la que estaba en ejecución cuando se produjo el trap se halla también, en este caso, en el registro %l1. Ésta es la única forma de retorno que se usa en los manejadores de trap asíncronos.

2. Traps síncronos.

   La dirección de la instrucción siguiente a la que provocó el trap se halla guardada en %l2 (nPC en el momento de producirse el trap). Las instrucciones que necesitamos para volver del manejador en son, en este caso:

   ```
   jmpl    %l2, %g0  ! Salta al contenido del nPC almacenado.
   rett    %l2 + 4   ! Regresa al nPC guardado mas cuatro.
   ```

Hay que remarcar la coherencia del método de retorno de trap con el mecanismo del *pipeline*. El hecho de guardar tanto el PC como el nPC cuando se produce un trap, hace que el retorno desde el manejador funcione aunque la instrucción interrumpida sea una instrucción de salto.

2.2.6 El Registro de Estado del Procesador (PSR)

Este registro alamcena información relativa al estado en el que se encuentra el procesador en un momento determinado de la ejecución de un programa.

El registro en sí está dividido en varios campos que sirven diferentes propósitos. Se pueden observar en la figura 2.8.

Número de implementación Campo que contiene un número identificativo del procesador. No interviene en la ejecución y su contenido no cambia.

Número de versión Similar al número de implementación.

Códigos de condición Indican si el resultado de una operación aritmética o lógica ha sido cero, negativo, provocó desbordamiento o acarreo.

Reservados Este campo no se usa. Una instrucción de escritura del registro de estado deberá escribir siempre ceros en este campo.

Coprocesador habilitado Permite el uso del coprocesador.

Unidad de coma flotante habilitada Permite el uso de la unidad de coma flotante. Este campo se ha empleado en ORK para detectar aquellas tareas que usen instrucciones de coma flotante durante su.

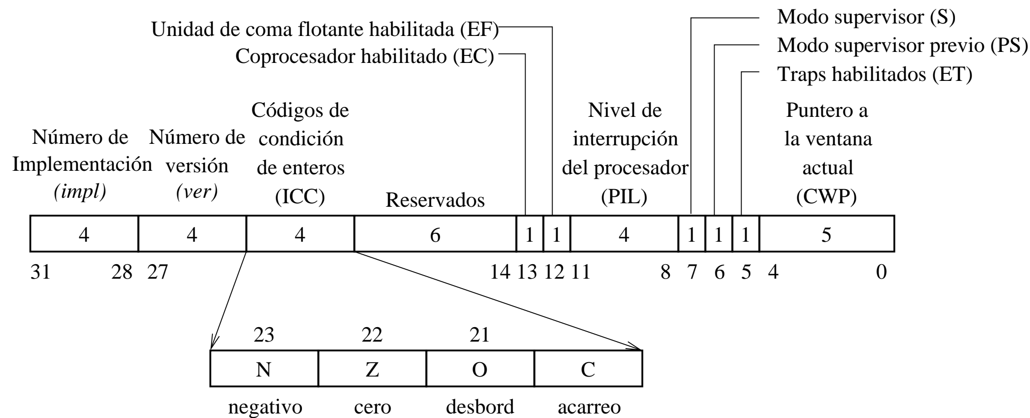


Figura 2.8: Registro de estado del procesador

Nivel de interrupción del procesador Sólo las interrupciones cuya prioridad supere el nivel que marca este campo serán atendidas.

Modo supervisor Este bit indica si el procesador se encuentra en modo supervisor. En este modo, se pueden ejecutar las instrucciones privilegiadas. ORK corre todo el tiempo bajo modo supervisor.

Modo supervisor previo En los cambios de modo supervisor a usuario y viceversa, este bit indica el modo en el que se encontraba el procesador en el momento anterior al cambio.

Traps habilitados Este bit informa sobre la aceptación de traps. Si los traps están inhabilitados, los asíncronos se ignoran; caso de producirse un trap síncrono, el procesador se detiene, quedándose en modo error.

Puntero a la ventana actual Este campo contiene un número que identifica la ventana de la pila de registros que está activa en cada momento.

El contenido del registro se puede leer y modificar a través del uso de dos instrucciones privilegiadas (sólo pueden ejecutarse en modo supervisor):

- Lectura del registro de estado (Read PSR):

```
rd      %psr, %g1  ! Guarda el contenido del PSR en %g1
```

- Escritura en el registro de estado (Write PSR):

```
wr      %g1, %g2, %psr  ! Realiza el XOR logico de %g1 y %g2
                        ! guardando el resultado en el PSR
```

La operación de escritura del registro de estado requiere de un cuidado especial. La modificación del valor del PSR mediante esta instrucción se realiza de forma retardada. Concretamente han de pasar tres ciclos de reloj para que la escritura se complete. Las instrucciones que se ejecuten durante dichos tres ciclos de reloj pueden causar problemas si necesitan o cambian indirectamente el valor de los campos del PSR que están siendo modificados por la instrucción de escritura. Para más detalles ver la descripción completa de dicha instrucción en [19].

El formato de las instrucciones que se utilizan para leer y escribir el registro base de la tabla de traps TBR, el registro de ventana inválida WIM y el registro %y es el mismo que tienen las instrucciones que acabamos de ver de lectura y escritura del registro de estado. De igual forma que en el caso del PSR, las instrucciones de escritura sobre estos otros registros son retardadas. Consecuentemente, habrá que tomar las precauciones necesarias para evitar que las tres instrucciones que siguen inmediatamente a éstas de escritura interfieran con ellas. La solución más segura para evitar este tipo de interferencias es colocar tres *nop* a continuación de la instrucción de escritura, si bien esta solución no suele ser la más eficiente.

2.2.7 El Registro de Ventana Inválida (WIM)

El registro de ventana inválida marca con un bit la accesibilidad de cada ventana de registros del procesador. Si el bit está a cero, la ventana es accesible y sus registros pueden ser modificados. Si, por el contrario, el bit correspondiente contiene un uno, la ventana de registros no es accesible (ver figura 2.9).

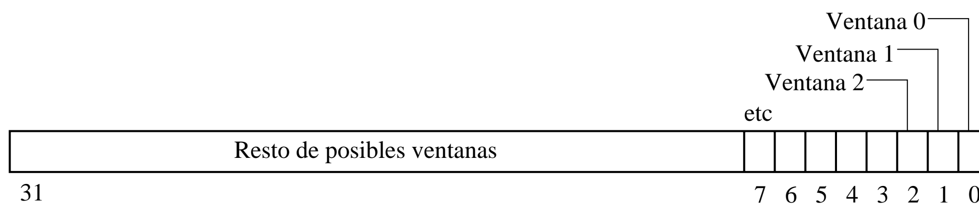


Figura 2.9: Registro de ventana inválida

En el caso de que una instrucción de las que cambian el valor del CWP (*save*, *restore* o *rett*) haga que éste apunte a una ventana inválida, se producirá un trap síncrono.

El ERC32 sólo implementa ocho ventanas de registros de las 32 posibles. Es por eso que sólo se utilizan los ocho primeros bits del registro WIM, los otros no tienen significado alguno para este procesador en concreto.

2.3 Las Subrutinas

En los apartados anteriores, hemos visto que la arquitectura SPARC v7 define el concepto de la pila circular de registros, cuyo control se efectúa a través del registro WIM y del campo CWP del registro de estado. Sin embargo, la arquitectura no dice nada acerca del uso que se ha de dar al mecanismo de control de las ventanas de registros, dejando libertad de elección al programador en este aspecto.

En [20] se describen posibles usos de la pila circular de registros. Uno de ellos consiste en que la aplicación pueda acceder a cualquier registro en cualquier momento, cambiando directamente (a mano) el valor del CWP si el registro no se encuentra en la ventana actual. Otro consiste en asignar una ventana de registros a cada proceso o tarea de la aplicación, siendo los registros globales comunes a todas las tareas. De esta manera el cambio de contexto sería muy rápido: simplemente cambiando el valor del CWP se entraría en el contexto de otra tarea. Este mecanismo no se ha usado en ORK porque sólo demuestra una mayor eficiencia en sistemas pequeños cuyo número de tareas sea menor o igual al número de ventanas de registros disponibles. Además, el mecanismo favorece el cambio de contexto, pero penaliza las llamadas a procedimiento (subrutinas).

Sin embargo, el convenio más extendido, y para el que probablemente se pensó inicialmente el mecanismo de la pila circular de registros, es el de usar las ventanas de registros para favorecer el paso rápido de parámetros en las llamadas a subrutina. El generador de código de GCC, usado en la etapa final del compilador de Ada GNAT, sigue este convenio por defecto, cuando se configura para máquinas SPARC v7. Por esta razón, la parte de ORK programada en ensamblador también lo sigue, siendo así compatible con la parte del sistema operativo programada en Ada.

Veamos ahora en qué consiste dicho convenio. Para simplificar los ejemplos, utilizaremos una máquina SPARC v7 impaginaria con sólo tres ventanas de registros en la pila circular (recuérdese que el ERC32 dispone de ocho ventanas de registros). Inicialmente, se marca una ventana como inválida y se introduce un valor en el CWP, de tal forma que la ventana activa sea la inmediatamente inferior a la inválida.El escenario que podría presentarse en nuestro procesador reducido sería el expuesto en la figura 2.10.

En este estado, el procesador puede usar directamente los registros *in*, *local* y *out* de la ventana dos. Supongamos ahora que el programa en ejecución se encuentra con un salto a subrutina. Es en este momento cuando aparece el problema del paso de parámetros. En muchas arquitecturas se resuelve este problema pasando los argumentos a través de la pila. Sin embargo, la arquitectura SPARC v7 usa un mecanismo más eficiente aprovechando la superposición parcial de las ventanas de registros. Los parámetros que se quieren pasar al subprograma se colocan en los registros *out* de la ventana actual. Posteriormente se decrementa el valor del

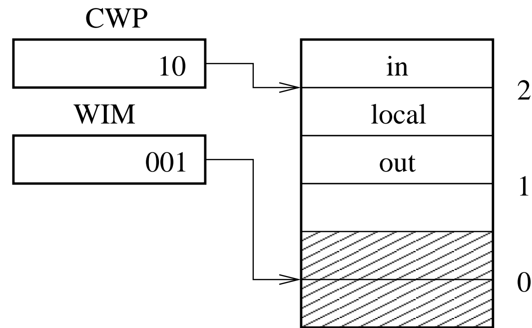


Figura 2.10: Escenario inicial

CWP (siempre módulo el número de ventanas implementadas). Esta operación hace que la ventana actual cambie, quedándose los parámetros proporcionados en los registros *in* de esta nueva ventana. Así, la subrutina tiene a su disposición toda una ventana de registros, de entre los cuales los *in* contienen los argumentos con los que se la llamó (figura 2.11).

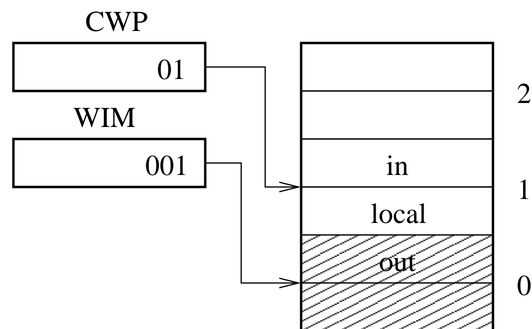


Figura 2.11: Paso de parámetros a la primera ventana

Este mecanismo tiene además la ventaja de permitir que cada subprograma pueda guardar su estado en los registros *local*, ya que ningún otro subprograma puede acceder a estos registros (no están solapados).

Para los subprogramas que son funciones y deben devolver un valor (o varios), también se usa el mismo mecanismo pero a la inversa: los valores a devolver se colocan en los registros *in* de la ventana actual y cuando se vuelve a la ventana del subprograma llamante (incrementando en una unidad el valor del CWP), este los recibe en sus registros *out*.

Examinemos ahora con más detenimiento cuáles son las instrucciones que se emplean para las llamadas a subrutina y la reserva de ventanas de registros. Para

realizar una llamada a subrutina se pueden usar dos instrucciones: *call* o *jmpl*.

La instrucción *call* tiene como único operando una etiqueta de 32 bits con la dirección de comienzo de la subrutina. Cuando se ejecuta esta instrucción, la dirección de retorno se guarda automáticamente en %o7. La instrucción *call* se codifica con dos bits únicamente, dejando los restantes 30 bits para almacenar la dirección de la subrutina. Los dos bits menos significativos de dicha dirección no necesitan almacenarse. El motivo es que las instrucciones deben colocarse en direcciones de memoria que sean múltiplo de cuatro, por lo que estos bits siempre valdrán cero.

La instrucción *jmpl* se puede usar para efectuar una llamada a subrutina cuando la dirección de la misma está contenida en un registro. Además, con esta instrucción se puede guardar la dirección de retorno en cualquier registro. De cualquier manera, en las llamadas a subrutina utilizaremos siempre el registro %o7 para guardar dicha dirección. De este modo, la instrucción *jmpl* es compatible con el convenio que usa la instrucción *call*.

En la arquitectura SPARC, el puntero de pila no es un registro especial del procesador. Su valor en cada instante está reflejado en el registro %o6 de la ventana actual. De hecho, el ensamblador reconoce el nombre de registro %sp (*stack pointer*, puntero de pila) como un nombre alternativo a %o6.

Por lo tanto, cuando vayamos a hacer una llamada a subrutina, hay que tener en cuenta que los registros %o6 y %o7 están ocupados por el puntero de pila y la dirección de retorno, respectivamente. Esto deja al resto de los registros *out* libres para poder ser usados en el mecanismo de paso de parámetros descrito más arriba. Disponemos, por tanto, de seis registros (%o0 - %o5) para pasar otros tantos argumentos al subprograma.

En el caso de que la subrutina requiera más argumentos o que dichos argumentos superen en tamaño a los registros disponibles, no queda más remedio que utilizar la pila, aunque esto reduzca la eficiencia. A pesar de todo, estos casos no son muy habituales y el espacio proporcionado por la ventana de registros suele ser suficiente para el paso de parámetros. En el código en lenguaje ensamblador de ORK no ha sido necesario usar la pila para el paso de parámetros en ninguna ocasión.

Retomemos ahora el punto en el que habíamos dejado la ejecución del salto a subrutina. Una vez se ejecuta la instrucción de salto (*call* o *jmpl*) y la instrucción retardada que la sigue (ver apartado 2.2.4), le toca el turno a la primera instrucción de la subrutina, que suele ser una instrucción *save* como la del siguiente ejemplo:

```
save    %sp, -112, %sp
```

Esta instrucción reserva una nueva ventana de registros decrementando el valor del CWP en una unidad. Además, suma el contenido de los dos primeros

operandos, guardando el resultado en el registro destino. El ejemplo presentado muestra la operación más habitual que se suele realizar con esta instrucción: crear un marco de pila para la subrutina. Para ello se resta la cantidad de memoria que se quiera reservar al puntero de pila actual y el resultado se guarda en el puntero de pila de la nueva ventana reservada. El cambio de ventana ocurre a mitad de la instrucción, produciéndose el efecto deseado.

El registro %i6 de la nueva ventana se conoce como el puntero del marco de pila. El ensamblador reconoce el nombre de %fp (*frame pointer*, puntero al marco) para referirse al mismo registro. Contiene el valor del puntero de pila de la anterior ventana y sirve para acceder a las variables locales cuyo espacio en memoria ha sido creado por la instrucción *save*.

El formato del marco de pila también sigue un convenio. La figura 2.13 muestra la organización del espacio de pila de una subrutina cualquiera.

Esta estructura contiene el espacio necesario para almacenar, entre otras cosas, las variables locales de la subrutina y la ventana de registros actual en caso de que se produzca un desbordamiento (*overflow*) de la pila circular de registros.

El desbordamiento ocurre cuando el nivel de anidamiento de subprogramas es mayor que el número de ventanas que posee el procesador. Si seguimos con el ejemplo del procesador de tres ventanas, vemos que el desbordamiento se producirá si se efectúa otra llamada a una subrutina que reserve para sí una nueva ventana de registros. En tal caso (ver figura 2.12), el CWP apuntaría a la ventana marcada como inválida por el registro WIM. Los registros *out* de la ventana inválida se superponen con los *in* de la ventana número dos, cuyo valor no se debe cambiar, ya que pueden estar siendo usados por nuestro primer subprograma.

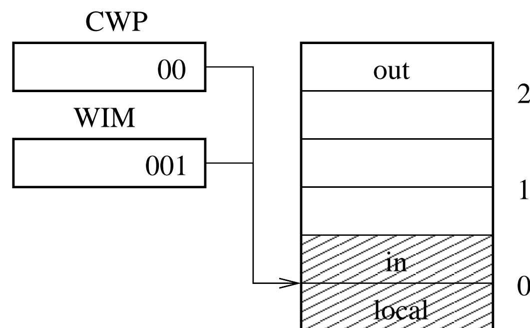


Figura 2.12: Desbordamiento de ventana

Cuando el CWP apunta a la ventana inválida, el procesador lanza un trap de desbordamiento. El manejador de este trap debe guardar el contenido de la ventana que sigue a la inválida en el espacio destinado para ello que se le había reservado en el marco de pila de la subrutina. Así, el paso de parámetros se realiza

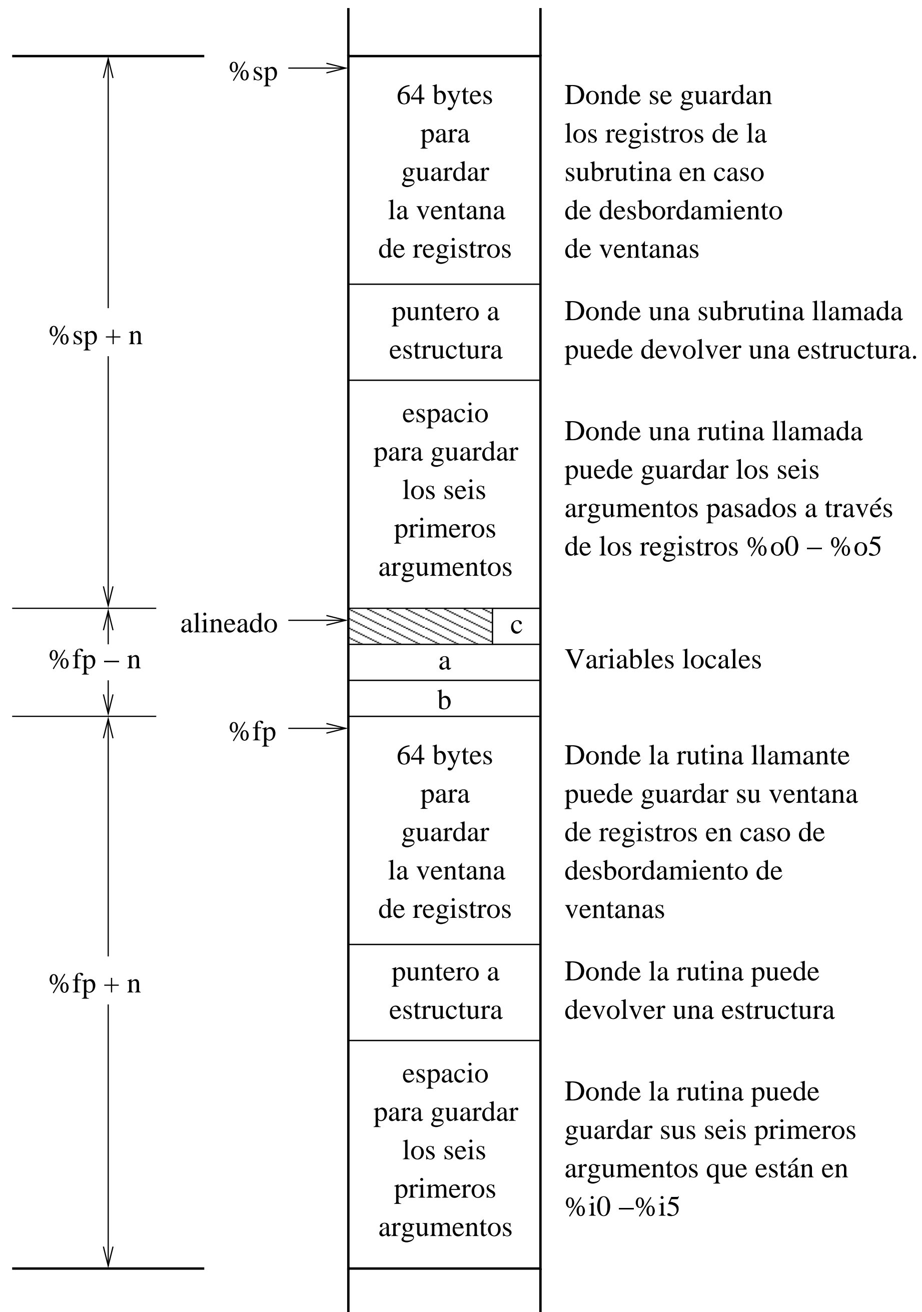


Figura 2.13: La pila

habitualmente a través de los registros de la pila circular, sin necesidad de usar la memoria principal. El acceso a memoria, más lento, se retrasa hasta el momento en que no quedan más ventanas libres.

Una vez el manejador de desbordamiento ha guardado el contenido de la ventana en memoria, la marca como inválida, dejando libre la antigua ventana inválida, para que pueda ser usada por la subrutina llamada. Ver figura 2.14.

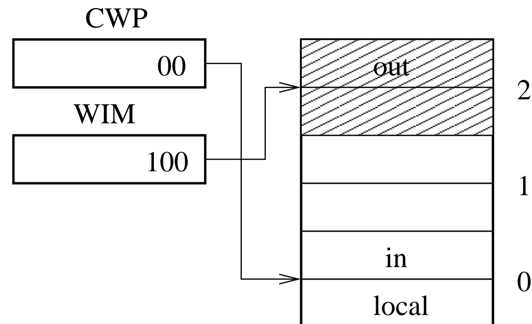


Figura 2.14: Procesado del desbordamiento

El mismo mecanismo descrito se aplicaría sucesivamente en el caso de que se produjeran nuevas llamadas a subrutinas que conllevaran la reserva de ventanas de registros. La eficiencia de todo el conjunto se basa en el supuesto de que el nivel de anidamiento local de un programa no es nunca demasiado alto. Es decir, supone que la profundidad de las llamadas a subrutina no superará normalmente al número de ventanas de registros disponibles, por lo que la cantidad de desbordamientos será escasa.

Cuando una subrutina termina de ejecutarse devuelve el control a la que la llamó mediante la instrucción:

```
jmpl    %i7, %g0
```

Al hacer la llamada, se guardó la dirección de retorno en %o7. Con el cambio de ventana que genera cada subrutina, el registro %o7 pasa a ser el %i7 de la nueva ventana. Es por eso que en el salto se utiliza el registro %i7 para volver al subprograma llamante.

Además de devolver el control a la subrutina que realizó la llamada, también hay que recuperar su ventana de registros. Para ello, la instrucción de salto se acompaña habitualmente de una instrucción *restore*, situada justo después de la de salto (en su espacio de retardo). La instrucción *restore* tiene un efecto complementario al de *save*. Mientras que *save* reserva una nueva ventana decrementando

el valor del CWP, *restore* incrementa el CWP para volver a la ventana se la subrutina llamante (insistimos en que todas las operaciones sobre el CWP se calculan módulo el número de ventanas implementadas).

La instrucción *restore* también tiene la posibilidad de realizar una operación aritmética (de suma) al mismo tiempo que incrementa el CWP, como ocurría con la instrucción *save*. Recordemos que, en el caso de la instrucción *save*, lo habitual era usar esta suma para reservar un marco de pila a la subrutina entrante. En el caso de *restore* no es necesario realizar ninguna operación para eliminar el marco de pila creado por *save*, por lo que esta instrucción se suele emplear sin operandos adicionales. El cambio de ventana basta para recuperar el puntero de pila original y descartar el marco de la subrutina saliente (ver figura 2.15).

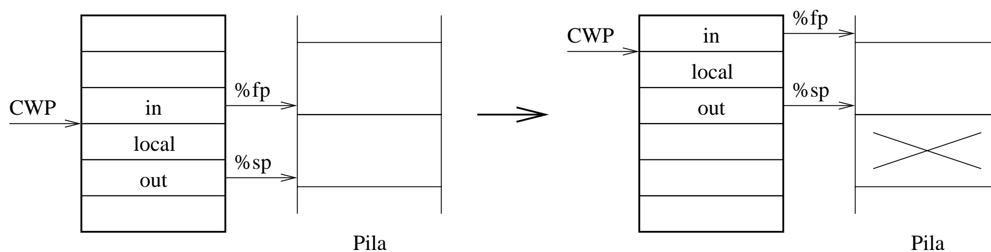


Figura 2.15: Recuperar la ventana de la rutina llamante

De igual forma que la instrucción *save* podía provocar el desbordamienteo (*overflow*) de la pila circular, la instrucción *restore* puede producir un desbordamiento por abajo (*underflow*) de la pila circular de registros. Este comportamiento se da en el caso de que la ventana a la que se vaya a volver sea inválida. En esta situación, el procesador genera un *trap* de *underflow*, cuyo manejador debe recuperar de la memoria los registros que corresponden a dicha ventana, marcando la anterior como inválida (ver figura 2.16). Como los registros de una ventana sólo se guardan en memoria en los desbordamientos, la situación de *underflow* sólo podrá producirse en caso de que haya habido un *overflow* previo.

2.4 Las instrucciones sintéticas

Aparte del conjunto completo de instrucciones de la arquitectura SPARC v7 (ver [19]), el programa ensamblador[4] reconoce otras pseudo-instrucciones llamadas sintéticas. Reciben este nombre porque no son instrucciones propias del procesador,

[4]Para la realización de este proyecto se ha usado el ensamblador de GNU: GAS (GNU Assembler), que sí entiende las instrucciones sintéticas

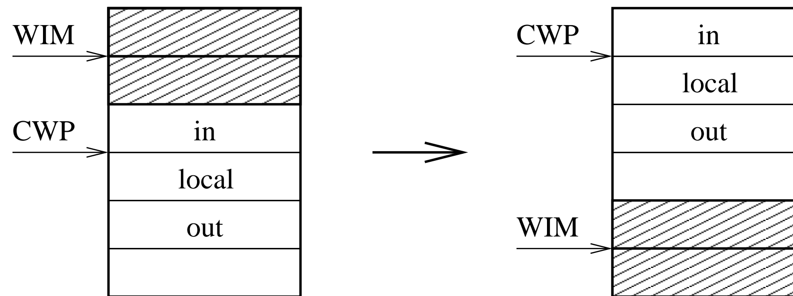


Figura 2.16: Desbordamiento hacia abajo (*undeflow*)

sino que están sintetizadas a partir de otras instrucciones que sí lo son. Las instrucciones sintéticas están pensadas para hacerle la vida más fácil al programador que utilice lenguaje ensamblador. Sus mnemónicos son fáciles de recordar y tratan de evitar detalles de muy bajo nivel.

Ilustraremos el uso de las instrucciones sintéticas con algunos ejemplos. Para una lista completa de las mismas, consultar [16].

En la arquitecteura SPARC v7 no existe como tal una instrucción del tipo *mover el contenido de un registro a otro*, tan habitual en otras arquitecturas. Sin embargo, sí que existe una instrucción sintética *mov*:

```
mov     %orig, %dest
```

Para conseguir el efecto deseado, el programa ensamblador traduce esta pseudo-instrucción por:

```
or      %orig, %g0, %dest
```

Como se puede observar, la instrucción sintética es mucho más descriptiva, más fácil de usar y de recordar que la estándar (en la que, por cierto, podemos ver uno de los usos del registro %g0).

Otro ejemplo es el retorno de subrutina. En la sección anterior hemos visto que es necesario un salto para volver de la subrutina. Para efectuar este salto, existe una instrucción sintética *ret*, que se traduce por la instrucción de salto vista más arriba:

```
jmpl    %i7, %g0
```

Volvemos a ver que la instrucción sintética esconde al programador los detalles de muy bajo nivel.

Como último ejemplo, observemos la instrucción sintética:

```
set     inm32, %reg
```

Esta instrucción tiene el efecto de cargar un valor inmediato de 32 bits en un registro. Ya dijimos que una instrucción así sería imposible de codificar en una arquitectura en que las instrucciones tienen 32 bits de longitud. Pero ésta no es una instrucción propiamente dicha. En realidad, el ensamblador se encarga de traducir esta pseudo-instrucción por dos instrucciones estándar que producen el resultado deseado:

```
sethi   %hi(inm32), %reg
or      %lo(inm32), %g0, %reg
```

En este último caso hay que tener en cuenta que la instrucción sintética se traduce en dos, por lo que no conviene usarla en el espacio de retardo de las instrucciones de salto.

2.5 La Unidad de Coma Flotante TSC692E

Para simular operaciones con números reales [5], el ERC32 dispone de una unidad de coma flotante (FPU). La FPU se compone de 32 registros de 32 bits cada uno, más un registro de estado, también de 32 bits. Los registros pueden concatenarse para realizar operaciones de doble precisión (64 bits).

Los módulos en lenguaje ensamblador que forman parte de este proyecto no han necesitado aprovechar las capacidades de cálculo de la FPU. Simplemente se han usado las instrucciones de carga y almacenamiento de registros, para poder guardar y restaurar el contexto de la unidad de coma flotante. De esta manera, se permite el uso de la FPU a los programas multitarea que se ejecuten sobre ORK.

Dichas instrucciones pueden emplearse de la misma manera y usando la misma sintaxis que las instrucciones de carga y almacenamiento de la unidad de enteros. El programa ensamblador detecta si la instrucción usa registros de la unidad de coma flotante y codifica la instrucción en consecuencia. Ejemplos:

```
ld      [%sp + 4], %f1  ! carga simple
ldd     [%g6], %f2      ! carga doble

st      %f3, [%fp]      ! almacenamiento simple
std     %f0, [%o2 + 8]  ! almacenamiento doble
```

Las instrucciones de carga doble y almacenamiento doble tienen las mismas restricciones que las explicadas para sus homónimas de la unidad de enteros.

Para una definición completa de la unidad de coma flotante, ver [21]. Consúltese [16] para aprender su manejo.

[5] En realidad, los números que puede manejar una computadora tienen precisión finita

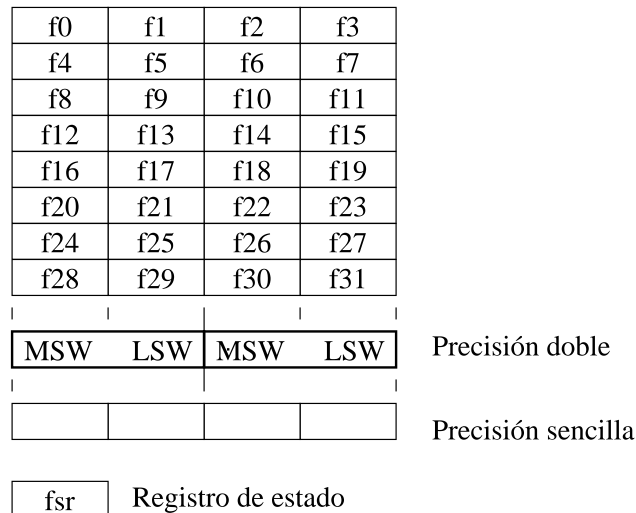


Figura 2.17: Registros de la FPU

2.6 El Controlador de Memoria MEC

El controlador de memoria, conocido abreviadamente como MEC, agrupa una serie de funciones que no se limitan sólo al control de la memoria del sistema, sino que también se dedican al control de ciertos periféricos y de las interrupciones.

El MEC no debe confundirse con lo que habitualmente se conoce como controlador de memoria virtual (MMU). Los controladores de memoria virtual utilizan un dispositivo de almacenamiento secundario del sistema (típicamente el disco duro) para crear la ilusión de que se dispone de una cantidad de memoria mucho mayor de la que se tiene en realidad.

El ERC32, por su parte, no está pensado (en principio) para poseer dispositivos de almacenamiento secundario. Todo el sistema software ha de cargarse en memoria ROM o RAM. El direccionamiento de la memoria es lineal, es decir, las direcciones lógicas se corresponden con las direcciones físicas de la memoria (en un sistema con memoria virtual, las direcciones lógicas son traducidas a direcciones físicas, no coincidiendo en general).

Por otra parte, un bus de direcciones de 32 bits permite direccionar un máximo de 4Gb de memoria, por lo que las necesidades de memoria están cubiertas incluso para las aplicaciones más exigentes. Normalmente, la memoria disponible en el sistema es mucho menor que dicho límite, debido principalmente al coste de los circuitos de memoria tolerantes a radiación. Esto hace que la cantidad de memoria implementada sea la mínima necesaria para que la aplicación que cargue en el ERC32 funcione correctamente.

Una de las funciones del MEC consiste, precisamente, en detectar los accesos a direcciones de memoria no implementadas, generando una excepción en tal caso.

Las características del MEC son muy útiles para el desarrollo de sistemas embarcados de tiempo real. Entre ellas, podemos citar:

1. Control del inicio del sistema
2. Reloj del sistema
3. Temporizadores: de propósito general, de tiempo real, "watchdog"
4. Interfaces con la memoria RAM y ROM
5. Interfaz de entrada/salida para cuatro periféricos
6. Interfaz DMA
7. Árbitro del bus
8. Protección de acceso a áreas de la memoria
9. Detección y corrección de errores
10. Dos canales de comunicaciones serie
11. Control de paridad en el bus del sistema

Entrar a comentar en detalle cada una de las características mencionadas cae fuera de los objetivos de este documento. Referimos al lector interesado en alguna característica en especial al documento de especificaciones del MEC [22].

Las características que sí se verán con más detalle son las que han sido usadas en la implementación de diversas facilidades que ofrece ORK. Cada una de ellas se tratará en el apartado correspondiente.

El MEC y los periféricos que incluye están mapeados en memoria a partir de la dirección `0x01FE000`. Esto quiere decir que se puede acceder a sus registros a través de las instrucciones de carga y almacenamiento habituales. Es importante señalar, no obstante, que los accesos al MEC han de llevarse a cabo empleando operandos del tamaño de una palabra (*word*: 32 bits). De lo contrario, el MEC emite un error de paridad interno. Habrá que asegurarse, por lo tanto, de que las instrucciones que acceden al MEC no usen un tamaño de operando equivocado. Esto es válido tanto para la codificación manual en ensamblador como para el código producido por el compilador de Ada, al cual habrá que decirle de alguna forma que sólo use operandos de 32 bits en las instrucciones en código máquina que genere para acceder al MEC (ver "peripherals").

Capítulo 3

El Cambio de Contexto de Bajo Nivel

En un sistema multitarea, una de las funciones más importantes que ha de llevar a cabo el sistema operativo es, precisamente, la de cambiar de tarea. Esta función recibe el nombre de cambio de contexto. Un cambio de contexto consiste en dejar de ejecutar una tarea para continuar con la ejecución de otra. Este cambio supone tener que almacenar cierta información de la tarea a la que se para, de tal forma que, en un futuro cambio de contexto, se pueda recuperar dicha información y volverla a ejecutar normalmente.

El sistema debe saber en cada momento qué tarea ha de ejecutarse de acuerdo con la prioridad de cada una de ellas. La parte del cambio de contexto que se encarga de decidir cuándo y qué tarea se ejecutará puede implementarse en lenguaje de alto nivel. En el proyecto que nos ocupa, por ejemplo, ha sido programada en Ada 95. La parte a la que dedicaremos nuestra atención en este capítulo, sin embargo, es aquélla que normalmente sólo puede programarse a muy bajo nivel. Nos referimos al proceso de almacenamiento y recuperación del contexto de ejecución de una tarea, tal y como se ha descrito más arriba. La información que se maneja en este caso se mueve a nivel de los registros de la máquina, de ahí que esta parte del cambio de contexto se suela implementar en lenguaje ensamblador.

El tiempo de ejecución de la rutina de cambio de contexto es uno de los factores a minimizar en un sistema multitarea. La razón es que todo el tiempo que se gaste en hacer el cambio entre tareas, no se está empleando en realizar ningún cálculo ni operación útil del sistema. Una alta duración de los cambios de contexto puede afectar incluso a la planificabilidad del sistema; esto es, a que las tareas puedan cumplir los plazos de tiempo que le fueron asignados. Ésta es otra de las razones por las que el cambio de contexto se programa en lenguaje ensamblador

3.1 El cambio de contexto en el ERC32

La definición del contexto de una tarea depende fuertemente de la máquina que se esté usando. En el caso del ERC32, el estado de una tarea queda completamente determinado por los siguientes elementos:

- Los registros globales.
- Las ventanas de la pila de registros que haya usado.
- El registro de estado del procesador.
- El contador de programa y el contador de programa siguiente.
- El registro Y.
- Los registros de la unidad de coma flotante.
- El espacio de memoria de la tarea.

El espacio de memoria de una tarea es, fundamentalmente, su pila. Cada tarea de ORK guarda en su contexto las direcciones de memoria que indican el comienzo y el final de su pila. La pila de una tarea contiene los valores de variables locales o de registros guardados temporalmente. Estos valores deberán ser preservados hasta la siguiente ejecución de la tarea. Por ello, ninguna tarea debería poder acceder al espacio de memoria de otra.

El resto de los elementos arriba mencionados son registros del procesador. Para no perder su contenido durante el cambio de contexto, estos registros se guardan en una porción de memoria destinada a tal efecto, con el propósito de poderlos recuperar más tarde.

3.1.1 Guardando las Ventanas de Registros

Como ya vimos en el capítulo anterior, el ERC32 dispone de una pila circular de 128 registros. Tal cantidad de registros requieren una cantidad de tiempo no despreciable para ser guardados y repuestos. Para hacer más rápido el cambio de contexto, se ha seguido una estrategia que consiste en guardar sólo aquéllas ventanas de registros que estén siendo usadas por la tarea en ejecución.

Así se mejora sensiblemente el tiempo medio que se tarda en realizar el cambio de contexto, puesto que será sólo en el peor de los casos cuando se tengan que guardar los 128 registros de la pila circular; esto es, cuando la tarea en ejecución haya hecho uso de todas las ventanas de registros disponibles.

El problema consiste en saber cuáles de entre las ventanas de la pila circular han sido usadas por la tarea en ejecución y, por lo tanto, han de salvarse. Sin embargo, el algoritmo necesario para hacer esta comprobación es bastante sencillo y no añade una sobrecarga significativa a la rutina de cambio de contexto, siendo el tiempo total en el peor caso muy parecido al que se obtendría con una rutina que guardara siempre todas las ventanas de registros.

El algoritmo se basa en el hecho de que sólo las ventanas que se encuentren situadas entre la ventana inválida y la que está en uso (descendiendo desde la inválida) necesitan ser guardadas. Esto es debido al recorrido del CWP producido por el mecanismo de reserva de ventanas de las subrutinas, como se muestra en la figura 3.1. La ventana en uso, obviamente, también necesita ser guardada, incluyendo sus registros *out*.

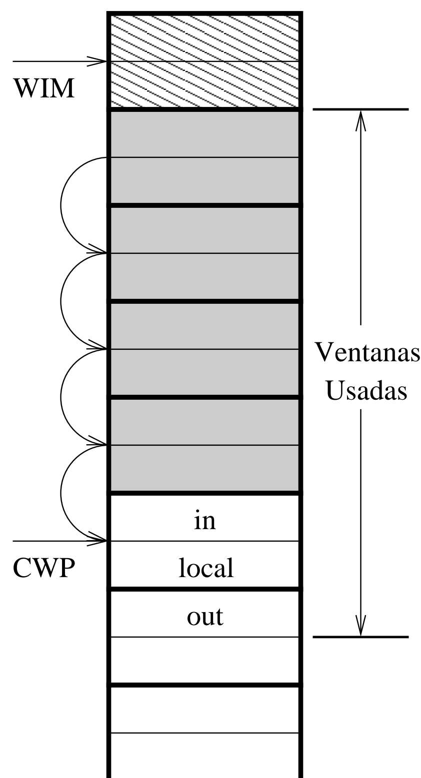


Figura 3.1: Ventanas de Registros en Uso

Así pues, lo único que tiene que hacer el algoritmo es guardar la ventana actual e incrementar el valor del registro CWP. Si, después de sumar uno a su valor, el CWP apunta a la ventana inválida, ya no es necesario continuar. En caso contrario, habría que guardar la ventana a la que apunte el CWP sumándole uno otra vez y

repetir esta operación hasta alcanzar la ventana inválida.

3.1.2 Salvando el Estado de la Unidad de Coma Flotante

La unidad de coma flotante se compone de una cantidad importante de registros que también hay que preservar durante los cambios de contexto. Sin embargo, no todas las tareas necesitan usar la FPU para realizar el trabajo que tienen encomendado. ORK aprovecha este hecho para no guardar el estado de la unidad de coma flotante hasta que no sea estrictamente necesario.

La pregunta es, ¿cuándo es necesario guardar el estado de la FPU? Para resolver esta duda, supondremos que el sistema sabe si las tareas que va a ejecutar utilizan la FPU o no. Estudiaremos todos los casos que se pueden dar en un cambio de contexto dependiendo de si la tarea entrante o la saliente usan el coprocesador matemático. Luego veremos un método para que ORK pueda averiguar automáticamente cuáles son las tareas del sistema que usan la unidad de coma flotante.

1. **Ni la tarea que deja el procesador ni la que entra a ejecutarse usan la FPU.**

   En este caso, está claro que no es necesario guardar los registros de la FPU. Ninguna de las tareas ha podido modificar su valor al no hacer uso de las instrucciones de coma flotante.

2. **Ambas tareas, la entrante y la saliente, utilizan la FPU.**

   Es el caso inverso al anterior. Ahora es necesario guardar el estado de la FPU de la tarea saliente, para que la entrante pueda efectuar sus operaciones matemáticas sin destruir los resultados obtenidos por la que sale. De cualquier manera, antes de empezar a ejecutar la entrante habrá que cargar en la FPU el contenido que tenían sus registros la última vez que entró en ejecución (que también debieron salvarse en algún momento).

3. **La tarea saliente usa la FPU, pero la entrante no.**

   Como la tarea entrante no utiliza la FPU, no hace falta guardar los registros de coma flotante de la tarea saliente en el acto. Sin embargo, sí que hay que señalar de alguna forma que puede ser necesario guardarlos en un próximo cambio de contexto.

4. **La tarea saliente no usa la FPU, pero la entrante sí.**

   Este caso se puede descomponer en tres subcasos:

(a) *Ninguna tarea señaló la necesidad de guardar el estado de la FPU.*

En esta situación, la tarea entrante puede empezar a ejecutar sus operaciones matemáticas sin tener que preocuparse de cargar ni guardar ningún registro de la FPU.

(b) *Una tarea señaló que había que guardar el estado de la FPU, pero coincide que es la tarea entrante.*

Si la tarea que dejo la señal es la misma que va a entrar, quiere decir que nadie ha tocado los registros de coma flotante desde entonces. Por lo tanto, la tarea puede continuar sus operaciones normalmente, sin tener que guardar ni cargar de nuevo ningún registro. Eso sí, se deberá borrar la señal que indica que los registros de la FPU están pendientes de ser guardados.

(c) *Una tarea señaló la necesidad de guardar el estado de la FPU y otra distinta va a entrar en ejecución.*

Este es el caso en el que no se puede retrasar más el cambio de contexto de la FPU. Ambas tareas usan los registros del coprocesador matemático, debiendo guardarse los de la tarea saliente y restaurarse los de la entrante. Además, también habrá que borrar la señal que marca la obligación de tener que guardar los registros de la FPU.

El mecanismo que detecta si una tarea utiliza la unidad de coma flotante se basa en una característica especial del *hardware*. El registro de estado contiene un bit (*EF, Enable Float*) que permite o prohíbe la ejecución de instrucciones de la FPU. En el caso de que su ejecución esté prohibida, el intento de ejecutar una instrucción de coma flotante provocará un trap específico: `fpu_disabled` *(FPU inhabilitada)*.

Cada tarea guarda el valor del registro de estado en los cambios de contexto, por lo que el contenido del PSR y, por extensión, el del bit EF, es particular a cada una de ellas. Para detectar el uso de la FPU por parte de alguna de las tareas del sistema, el método que se ha seguido es el que se detalla a continuación.

Al arrancar el sistema, todas las tareas empiezan con el bit EF del registro de estado a cero. Esto les impide ejecutar instrucciones de coma flotante. Si una tarea no utiliza nunca la FPU, dicho bit permanecerá con el valor cero durante toda la vida de la tarea. En cambio, si la tarea debe realizar cálculos matemáticos que impliquen el uso de la FPU, en algún momento intentará ejecutar una instrucción de coma flotante. En ese preciso instante, se producirá un trap de *FPU inhabilitada*. El manejador de este trap deberá entonces poner el bit EF a uno para que el sistema sepa que la tarea utiliza la unidad de coma flotante. Además, si en el momento en el que se dispara el trap, hubiera pendiente una señal que indicara la

obligatoriedad de guardar el estado de la FPU, el manejador tendría que guardar dicho estado (y borrar la señal) antes de devolver el control a la tarea interrumpida.

En el código de ORK, se utiliza como dicha señal una variable global (llamada *FP_Save_Pending*). Esta variable puede tener valor cero, lo cual quiere decir que no hay ninguna tarea pendiente de guardar el estado de la FPU, o bien puede contener una dirección de memoria. Durante un cambio de contexto en el que se quiera señalar que el estado de la FPU queda pendiente de ser guardado, lo único que hay que hacer es rellenar la variable global mencionada con una dirección de memoria adecuada. La dirección a la que nos referimos es la que apunta al espacio de memoria que tiene cada tarea para almacenar el estado de la FPU. Una vez salvado dicho estado, cuando corresponda según los casos descritos, se borrará la variable global escribiendo el valor cero en ella.

Esta forma de actuar tiene la ventaja de que permite saber siempre dónde ha de guardarse el estado de la FPU en el momento en el que sea necesario. La figura 3.2 muestra un diagrama con los posibles estados que han sido descritos en los párrafos anteriores. Las flechas indican los cambios de contexto, excepto la que se refiere al trap de FPU inhabilitada. En cada cambio de contexto se salvarán los registros de coma flotante o no, según corresponda.

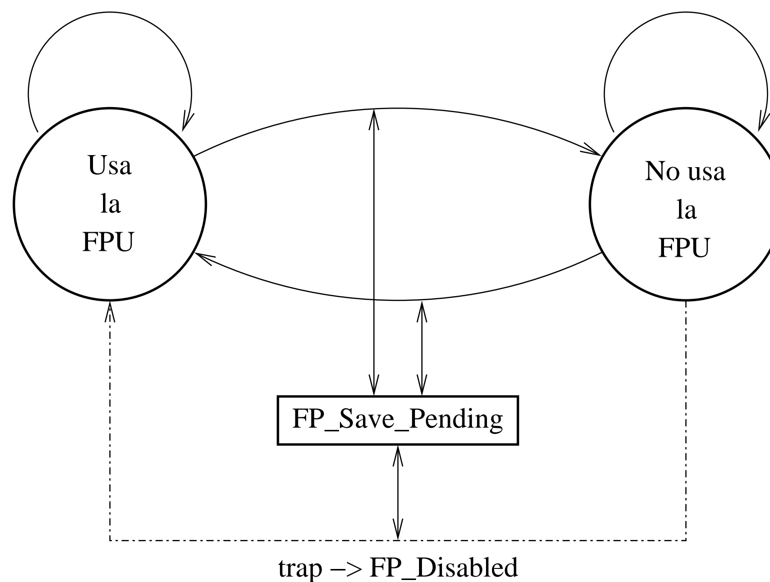


Figura 3.2: Optimización del Cambio de Contexto por uso de la FPU

3.2 La Rutina del Cambio de Contexto en ORK

A lo largo de la vida de ORK, se han utilizado dos estrategias diferentes para realizar el cambio de contexto. La primera de ellas consistió en escribirlo como una subrutina "hoja". La segunda estrategia, que es la que se utiliza actualmente, se basa en llamar al cambio de contexto mediante un trap software. Esta última aproximación tiene la ventaja de que integra el cambio de contexto con el manejo de interrupciones, que se verá más adelante.

Para entender la primera estrategia, es necesario saber qué se quiere decir cuando se habla de subrutina hoja. Una subrutina hoja tiene la particularidad de que, desde su código, no se llama a ninguna otra subrutina. Por esta razón, las subrutinas hoja no necesitan reservar una nueva ventana de registros cuando son llamadas. Utilizan la misma ventana de registros que la subrutina llamante. Por esta razón, han de tener cuidado de no modificar el estado local de la subrutina que las llamó. Una subrutina hoja sólo puede usar los registros globales y los registros *out* de la ventana en la que se encuentra, exceptuando el registro %o6 y el %o7, que se utilizan como puntero de pila y como dirección de retorno respectivamente.

Cuando llegue el momento de devolver el control a la rutina llamante, habrá que saltar a la dirección de retorno almacenada. Pero, como acabamos de ver, la dirección de retorno no se encuentra en %i7, como pasa con las subrutinas normales, sino en %o7, por lo que no se deberá emplear la instrucción *ret* para volver. En su lugar, habrá que utilizar la instrucción *retl* (la "l" viene de *leaf*, hoja), que el programa ensamblador traduce a:

```
jmpl    %o7, %g0
```

3.2.1 ORK versión 2.1 y anteriores

Anteriormente a la aparición de la versión 2.2 de ORK, el cambio de contexto de bajo nivel se efectuaba siguiendo la primera estrategia de las dos comentadas. Esto es, la rutina de cambio de contexto se implementaba como una subrutina hoja. Esta subrutina recibía dos parámetros: las dos direcciones de memoria donde cada una de las tareas implicadas en el cambio (la entrante y la saliente) tienen un espacio reservado para guardar su contexto de ejecución.

En este espacio reservado de cada tarea, la rutina de cambio de contexto guarda la última de las ventanas de registros usada por la tarea saliente, junto con los registros globales, los registros de la FPU, el registro de estado, el registro %y, las dos direcciones límite de su pila y los dos contadores de programa. El resto de las ventanas de registros usadas por la tarea se guardan en su pila (de igual manera a como se guardan cuando se produce un desbordamiento de ventana). Así no hay

que sobredimensionar el espacio que destina cada tearea a guardar su contexto y se utiliza la pila en su lugar.

Seguidamente, se recupera la última ventana de registros que usó la tarea entrante, junto con el resto de registros que forman el contexto de dicha tarea. El resto de las ventanas de registros usadas se recuperarán de la pila mediante el mecanismo de desbordamiento hacia abajo (*underflow*) habitual de las subrutinas. Es decir, no se recuperan todas las ventanas de golpe, sino que se espera a que el programa las necesite. No hay que olvidar que estas ventanas se reservaron mediante llamadas a subrutina, por lo que será a la vuelta de cada subrutina de la tarea cuando se recupere la ventana correspondiente. Esta forma de restaurar las ventanas de registros tiene sus ventajas frente a una estrategia en la que se recuperasen todas las ventanas de golpe durante el cambio de contexto. En primer lugar, esta segunda estrategia va contra la filosofía con la que se ideó la pila de registros: posponer el acceso a memoria lo más tarde posible. Recuperar todas las ventanas supone dedicar un tiempo importante del cambio de contexto a esta labor. Labor que, por otra parte, resultaría inútil en el caso de que se produjera rápidamente otro cambio de contexto. De nuevo habría que guardar todas las ventanas que tanto tiempo se había tardado en recuperar.

Además, una tarea puede usar más ventanas de las disponibles en la pila de registros. En este caso sólo se podrían recuperar las siete ventanas correspondientes a las siete subrutinas más anidadas. También puede darse la situación de que la tarea no haya usado todas las ventanas de la pila de registros. De alguna forma habría que saber que no se deben recuperar todas las ventanas (las siete disponibles), sino sólo aquéllas que lo necesiten. Esto se podría saber si, al restaurar una ventana, el puntero de pila alcanza el límite de la pila de la tarea, es decir, su valor inicial para dicha tarea. Esta comprobación no es demasiado difícil pero ya empieza a introducir cierta complejidad innecesaria al cambio de contexto. Este problema, junto con las otras desventajas de la segunda estrategia, ha hecho que nos decantemos por no recuperar todas las ventanas de registros a la vez sino por dejar que se restauren a medida que se vayan necesitando.

3.2.2 ORK versión 2.2 y posteriores

Para poder unificar el cambio de contexto con el tratamiento de interrupciones de ORK, se utilizó la segunda de las estrategias comentadas más arriba. Recordemos que dicha estrategia consistía en usar una instrucción de trap para llamar a la rutina de cambio de contexto.

La utilización de un trap software tiene la ventaja de que se pueden guardar los registros de una tarea de igual forma a como se guardarían en el caso de que se hubiera producido una interrupción. Así, al volver de un manejador de interrupción, se puede recuperar el contexto de cualquier tarea de la misma manera que se

utiliza normalmente para devolver el control a la tarea interrumpida. Es decir, a la salida de los manejadores de interrupción se puede realizar un cambio de contexto directamente, cosa que no sucedía en las versiones anteriores de ORK, donde había que llamar a la rutina de cambio de contexto separadamente. Esta diferencia no es simplemente estética. Su importancia se verá en el capítulo dedicado a las interrupciones.

Por otra parte, el uso de una instrucción de trap para efectuar el cambio de contexto modifica ligeramente la forma de realizarlo. Para empezar, la instrucción de trap, como cualquier otro trap, decrementa el CWP, situándose en la ventana siguiente a la actual. También ocurre que los contadores de programa se guardan en los registros %l1 y %l2, indicando la dirección de retorno. Esto contrasta con el método anterior, en el que la ventana a guardar era la ventana actual y la dirección de retorno se almacenaba en el registro %o7.

La situación que se plantea justo después del trap se muestra en la figura 3.3. De la ventana apuntada por el CWP, ahora sólo se guardan los registros *in* (correspondientes a los *out* de la ventana que estaba en uso) y los registros locales %l1 y %l2 (por tener la dirección de retorno). El espacio que cada tarea reserva para guardar su contexto se ve, por lo tanto, reducido. A cambio, los registros *in* y *local* de la ventana que estaba en uso se guardarán en la pila junto con los de las demás ventanas de registros usadas.

Para recuperar el contexto de la tarea, habrá que recorrer el camino inverso. Se cargan los registros *out* de la tarea desde el espacio de memoria donde se guardaron, así como los registros %l1 y %l2 (que se cargarán más tarde como contadores de programa). Se incrementa el CWP en una unidad y se rescatan los registros *in* y *local* de la ventana en uso. Se decrementa el CWP para volver a la ventana de procesado del trap y se ejecuta una instrucción de retorno de trap. Esta instrucción vuelve a incrementar el CWP y permite que la tarea continúe su ejecución normal. Al igual que en el caso anterior, el resto de las ventanas de registros que se guardaron en la pila se recuperan mediante *underflow*, el mecanismo habitual de las subrutinas.

En la figura 3.4 observamos el diagrama de flujo correspondiente al cambio de contexto de la versión 2.2 de ORK. El cambio de contexto en la versión 2.1 es algo más simple. La lógica empleada para comprobar si es necesario guardar o restaurar el contexto de coma flotante de las tareas entrante y saliente no existe en dicha versión. Simplemente, el estado de la FPU se guarda y se restaura siempre, aunque algunas veces esto no sea lo más eficiente.

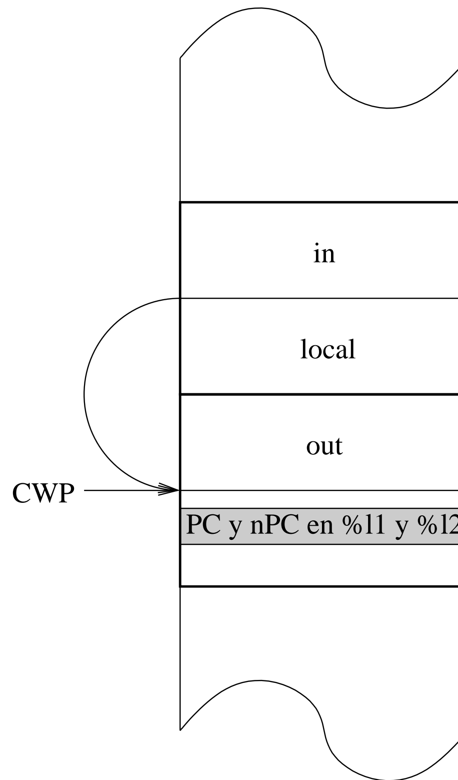


Figura 3.3: Trap del cambio de contexto

3.3 Medición del tiempo de cambio de contexto

Como ya se dijo en la introducción, el cambio de contexto es una de las partes fundamentales de un sistema multitarea. El tiempo de cómputo del cambio de contexto se suele utilizar como cifra de mérito en este tipo de sistemas. En este apartado veremos algunas medidas realizadas sobre la rutina del cambio de contexto de bajo nivel de ORK, tanto para la versión 2.1 como para la 2.2.

Estas medidas fueron tomadas usando el emulador de ERC32 con el que se hecho todas las pruebas de ejecución de aplicaciones sobre ORK. Una característica del emulador permite la visualización del tiempo transcurrido desde que se ejecuta una instrucción cualquiera del programa hasta que se alcanza otro punto determinado del mismo. El emulador muestra entonces el número de instrucciones ejecutadas, la cantidad de ciclos de reloj que consumen y el tiempo en milisegundos al que equivalen. Por ejemplo:

```
Cycles       :       444
Instructions :       161
```

```
Overall CPI  :      2.76

CPU performance (14.0 MHz)  :  5.08 MOPS ( 5.08 MIPS )
Simulated time              :   0.03 ms
Processor utilisation       : 100.00 %
Real-time performance       :   5.98 %
Simulator performance       : 303.77 KIPS
Used time (sys + user)      :   0.00 s
```

Una de las cifras que llama la atención es el alto número de ciclos por instrucción (CPI) que se consumen durante un cambio de contexto. Más aún cuando dijimos que gracias al *pipeline* esta cifra se debía aproximar a la unidad. Sin embargo, no hay que olvidar que la rutina de cambio de contexto emplea en su mayor parte instrucciones de acceso a la memoria para cargar y guardar registros. Este tipo de instrucciones son todas multiciclo (ver apartado 2.2.4) por lo que no es tan extraño que el número de ciclos por instrucción supere ampliamente la unidad. A pesar de todo, esta cifra podría mejorarse si se sigue una estrategia de no utilizar en una instrucción ningún registro que haya sido modificado por la instrucción anterior, algo que suele ser complicado cuando se programa manualmente en lenguaje ensamblador.

El tiempo de ejecución del cambio de contexto en el ERC32 depende de varios factores, los cuales se han ido comentando a lo largo de este capítulo. Depende, recordemos, del número de ventanas de registros usadas por la tarea en ejecución y de la utilización o no de la unidad de coma flotante por parte de las tareas que intervienen en el cambio de contexto.

En las versiones de ORK 2.1 y anteriores, el cambio de contexto incluía siempre el almacenamiento y recuperación del estado de la FPU. Esto se debía a que la detección automática de tareas que usan la FPU todavía no estaba implementada. Todas las tareas eran marcadas como usuarias de la unidad de coma flotante, desde el inicio del sistema, poniendo a uno el bit EF del registro de estado. A partir de ORK 2.2, la detección de operaciones de coma flotante se hizo operativa y sólo se salva el contexto de la FPU en los casos mencionados en el apartado 3.1.2.

En cualquiera de los casos, siempre se salvan, como mínimo, dos ventanas además de la actual, debido a las llamadas a procedimiento del kernel que preceden al cambio de contexto. El número máximo de ventanas que se pueden guardar durante un cambio de contexto es siete (de las ocho que tiene el ERC32, la inválida no necesita guardarse nunca).

Los tiempos de cambio de contexto para las distintas versiones de ORK se muestra en las tablas 3.1, 3.2 y 3.3. En cada tabla se muestran el número de ventanas que se guardan, los ciclos de reloj consumidos, el número de instrucciones

ejecutadas y el tiempo total de ejecución para un ERC32 a 10 MHz. Volvemos a insistir que estos tiempos son los que tarda en ejecutarse la rutina de cambio de contexto de bajo nivel. No está incluido, por lo tanto, el tiempo que ORK dedica al manejo de colas, cambio de prioridades y demás operaciones de alto nivel previas.

Ventanas	Ciclos	Instrucciones	Tiempo (μs)
3	428	151	42,8
4	474	165	47,4
5	520	179	52,0
6	566	193	56,6
7	612	207	61,2

Tabla 3.1: ORK v2.1, incluye contexto de FPU

Ventanas	Ciclos	Instrucciones	Tiempo (μs)
3	314	130	31,4
4	360	144	36,0
5	406	158	40,6
6	452	172	45,2
7	498	186	49,8

Tabla 3.2: ORK v2.2, sin guardar contexto de FPU

Vemos que, en ambas versiones de ORK, se necesitan 46 ciclos (14 instrucciones) por cada ventana adicional que se requiera guardar. No es de extrañar que sea la misma cifra para las dos versiones, ya que el bucle encargado de guardar las ventanas de registros es siempre el mismo.

Ventanas	Ciclos	Instrucciones	Tiempo (μs)
3	444	161	44,4
4	490	175	49,0
5	536	189	53,6
6	582	203	58,2
7	628	217	62,8

Tabla 3.3: ORK v2.2, incluyendo contexto de FPU

Observamos también que, en caso de que se tenga que guardar el contexto de coma flotante, la versión antigua es ligeramente más rápida que la moderna. Concretamente, existe una diferencia de 16 ciclos de reloj entre una y otra. Esta diferencia se debe a que la llamada a una subrutina hoja es más eficiente que disparar un trap. En cualquier caso, esta diferencia se compensa ampliamente con la que se produce en caso de no tener que guardar el estado de la FPU (114 ciclos de reloj, favorables a la última versión) y a las ventajas que tiene la integración del cambio de contexto con las interrupciones.

3.4 Las interrupciones y los cambios de contexto

En este apartado hemos supuesto que el cambio de contexto se puede realizar siempre en el mismo momento de su invocación. Sin embargo, en la práctica no es así. Las interrupciones interfieren de forma significativa en los cambios de contexto. Un manejador de interrupción puede requerir, como consecuencia de su ejecución, que se produzca un cambio de contexto. A pesar de ello, el cambio en sí no se deberá producir hasta que no se haya finalizado completamente el procesamiento de la interrupción.

Sucede también el caso contrario, en un cambio de contexto se realizan operaciones muy delicadas, durante las cuales no debería admitirse la llegada de interrupciones. Las interrupciones deben estar absolutamente prohibidas, por ejemplo, mientras se guardan las ventanas de registros de la tarea en ejecución.

El porqué de estos comportamientos se tratará en detalle en la sección dedicada al procesamiento de las interrupciones.

3.5 La protección de pilas y los cambios de contexto

Cada tarea dispone de un espacio de pila propio en memoria. Ninguna otra tarea debería acceder a él. Si, por causa de un desbordamiento de pila, una tarea invadiera el espacio de pila de otra, ésta última encontraría datos erróneos que producirían un efecto imprevisible sobre el sistema.

Para solucionar este problema, ORK implementa lo que se conoce como protección de pilas. Durante la ejecución de una tarea, se protege su espacio de pila aprovechando una característica del controlador de memoria (el MEC) que permite prohibir la escritura de bloques enteros de memoria. El desbordamiento de la pila provocaría una escritura en un bloque de los protegidos por el MEC. El sistema detecta de esta manera el desbordamiento, suspendiendo la tarea para siempre.

El MEC sólo puede gestionar a la vez dos bloques de los arriba mencionados. Por este motivo, durante los cambios de contexto se debe cambiar la zona de la

memoria protegida por estos bloques, escribiendo en los registros del MEC. La forma en la que se protegen las pilas de las tareas se describe detalladamente en el capítulo 5, que trata sobre este tema.

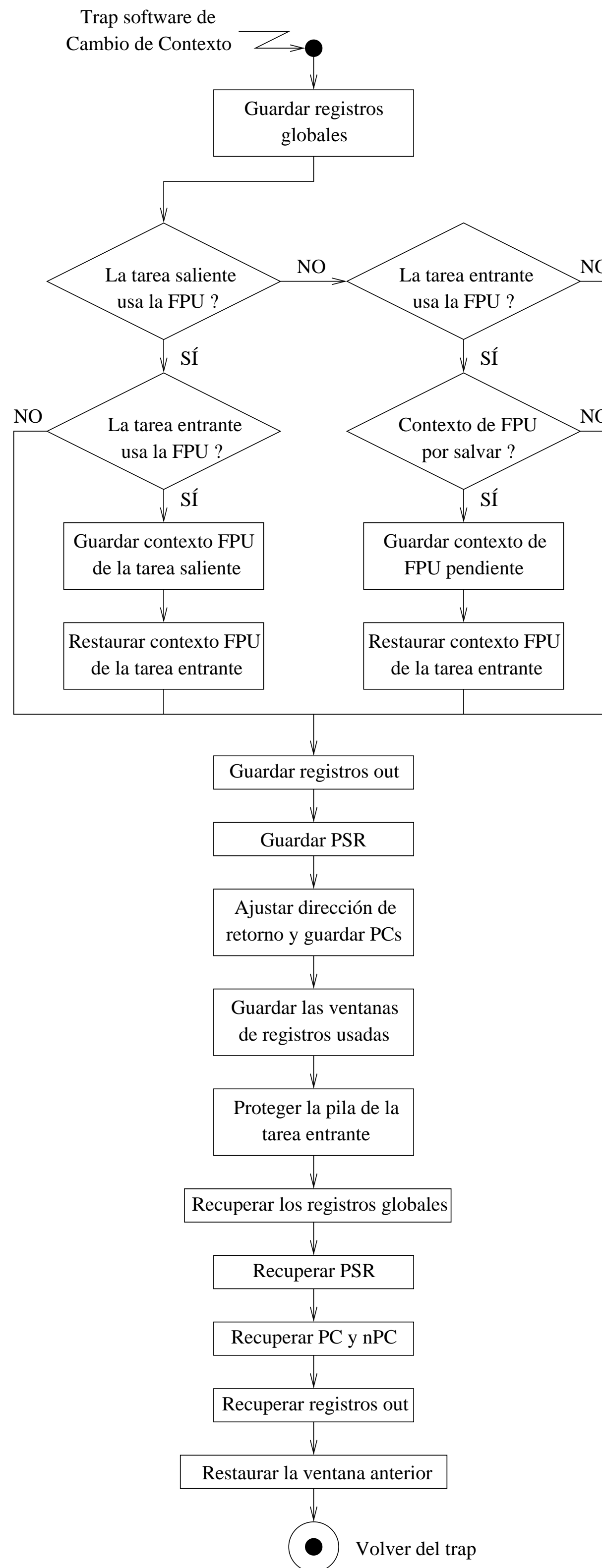


Figura 3.4: Cambio de contexto en ORK v2.2

Capítulo 4

Las Interrupciones

Los sistemas de tiempo real que manejan dispositivos físicos deben interactuar de alguna forma con su entorno. Si el sistema ha de dar una respuesta a un suceso del mundo exterior, tiene que saber de alguna manera que dicho evento se ha producido. Las interrupciones son una de las formas que tienen los dispositivos externos de comunicarse con el procesador. Se llaman interrupciones porque el procesador interrumpe su procesamiento normal para atender la petición del periférico que lo requiera.

Una de las interrupciones habituales de cualquier sistema es la de reloj. El reloj del sistema interrumpe al procesador cada cierto tiempo para que éste actualice su cuenta interna del tiempo transcurrido y actúe en consecuencia.

En un sistema dado, las causas de interrupción no tienen todas la misma importancia. La señal de un detector de colisión, por ejemplo, será prioritaria sobre la de un detector de proximidad en un aparato móvil. El sistema atenderá a la señal más prioritaria en caso de que se produzcan varias a la vez.

4.1 Las Interrupciones en el ERC32

En la arquitectura SPARC, las interrupciones reciben el nombre de traps asíncronos (ver apartado 2.2.5), aunque en algunos casos también se emplea indistintamente la denominación de interrupción. Un procesador SPARC v7 admite hasta quince niveles de interrupción diferentes. El campo PIL (*Processor Interrupt Level*, Nivel de Interrupción del procesador) del registro de estado indica el nivel a partir del cual se admiten interrupciones. Esto es, si llega una señal de interrupción cuyo nivel es menor que el indicado por el PIL, esta interrupción no es atendida y queda pendiente hasta que dicho nivel baje.

En el ERC32, es el controlador de memoria (el MEC) quien gestiona las interrupciones. De los quince niveles de interrupción que admite el procesador, los

periféricos que acompañan al MEC ocupan diez, dejando los cinco restantes a los dispositivos externos que se conecten al controlador de memoria. La correspondencia entre dispositivos y nivel de interrupción se muestra en la tabla 4.1.

Interrupción	Tipo de Trap	Nivel de Interrupción
Watch Dog (reloj de actividad)	0x1F	15
Interrupción externa 4	0x1E	14
Reloj de tiempo real	0x1D	13
Reloj de propósito general	0x1C	12
Interrupción externa 3	0x1B	11
Interrupción externa 2	0x1A	10
Expiración DMA	0x19	9
Error de acceso DMA	0x18	8
Error de UART	0x17	7
Error corregible en memoria	0x16	6
UART B, Datos o Tx preparados	0x15	5
UART A, Datos o Tx preparados	0x14	4
Interrupción externa 1	0x13	3
Interrupción externa 0	0x12	2
Errores hardware enmascarados	0x11	1

Tabla 4.1: Interrupciones y su nivel correspondiente

Las interrupciones que quedan pendientes de ser atendidas se guardan en un registro del MEC (el *Interrupt Pending Register*). Si, por cualquier motivo, no se quisieran atender, éstas pueden borrarse usando otro registro del MEC (el *Interrupt Clear Register*). También puede enmascararse cualquier interrupción (excepto la quince, que no es enmascarable) poniendo a uno el bit correspondiente en el registro *Interrrupt Mask Register.* Cualquier señal de interrupción que esté enmascarada se ignorará.

Las interrupciones de la tabla 4.1 marcadas como externas son aquéllas a las que se pueden conectar los periféricos adicionales. Programando el registro *Interrupt Shape Register* es posible definir si estas interrupciones son activas a nivel alto, a nivel bajo o si son dirigidas por flanco o por nivel. El MEC se adapta así a la señal que ofrezca el dispositivo físico que produzca la interrupción.

Cuando recibe una interrupción, el MEC manda una petición al procesador a través de sus salidas IRL (*Interrupt Request Level*), indicando el nivel correspondiente. Si dos o más interrupciones se producen simultáneamente, se manda la de mayor nivel. Si el MEC recibe una interrupción de mayor nivel antes de que el procesador haya reconocido una de menor nivel, la de mayor nivel reemplaza a la

que estaba en las salidas IRL del MEC.

4.2 El procesamiento de las interrupciones en ORK

ORK permite que los programas en Ada escritos para el ERC32 puedan definir sus propios manejadores de interrupción. El perfil de Ravenscar admite el uso de procedimientos protegidos como manejadores de interrupción. La correspondencia entre un procedimiento protegido y una interrupción ha de ser estática, es decir, el enlace se establece en el momento de la creación del objeto protegido y se mantiene hasta la finalización del mismo. Como los objetos protegidos no pueden crearse dinámicamente según el perfil de Ravenscar, el enlace entre una interrupción y un procedimiento protegido se mantendrá hasta la muerte del sistema.

Una vez el procesador reconoce una interrupción, deja lo que estaba haciendo en ese momento y atiende la interrupción (siempre que no estuviera atendiendo otra interrupción de nivel superior). Para atenderla, deberá ejecutar su manejador asociado. Sin embargo, si el manejador está escrito como un procedimiento protegido de Ada, no se deberá ejecutar inmediatamente. La razón es muy simple: el estado en el que se encontraba el procesador antes de que llegara la interrupción hay que recuperarlo una vez se haya ejecutado el manejador de dicha interrupción. Por lo tanto, antes de empezar a procesar un manejador escrito en un lenguaje de alto nivel (Ada en nuestro caso) hay que guardar el estado del procesador y modificarlo adecuadamente para que pueda tratar el resto de la interrupción sin problemas.

De nuevo en esta ocasión, al igual que sucedía con los cambios de contexto, habrá que prestar especial atención al contenido de las ventanas de registros. Sin embargo, al contrario a lo que sucedía en un cambio de contexto, cuando se produce una interrupción no se guardan las ventanas usadas de la tarea que se estaba ejecutando hasta entonces. Es decir, no hay que guardar todo el contexto de la tarea en ejecución, sino que vale con que se guarden los registros globales y otros de uso general, pero no los de las ventanas de registros. Esto tiene su razón de ser. Durante el procesamiento de una interrupción sabemos que se pueden producir llamadas a procedimiento y, por lo tanto, reserva de ventanas. Pero se pueden mantener las ventanas usadas por la tarea porque el mecanismo de desbordamiento de ventanas funcionará bien en este caso. El orden de llamadas se mantiene: al procesar una interrupción, nunca se podrá volver de un procedimiento antes de que se haya producido una llamada. Además, a la hora de volver de la interrupción, el balance total de llamadas a procedimiento y retornos de procedimiento debe ser cero, encontrándonos en la misma ventana de partida.

Esto no sucede en los cambios de contexto. Si la tarea entrante se quedó a mitad de un procedimiento, puede ejecutar un retorno de subrutina antes que

una llamada. Si la tarea saliente no hubiera guardado sus ventanas de registros, la entrante podría ocuparlas creyéndolas suyas, ya que el mecanismo de *overflow/underflow* ha dejado de ser válido, pues supone que siempre se producirá una llamada a procedimiento antes que un retorno.

Así pues, para llamar a los manejadores de interrupción escritos en Ada, se implementó en lenguaje ensamblador una rutina común a todos ellos. Esta rutina se encarga de llevar a cabo todas las operaciones de preparación que permitan la ejecución normal del manejador. Las interrupciones y traps que están directamente implementados en lenguaje ensamblador no necesitan de esta rutina, ya se ocupan ellos de que el estado original no sea modificado. Los pasos que sigue el "manejador común" son los siguientes:

1. **Ajuste de la dirección de retorno** La rutina se puede usar para manejar traps síncronos además de interrupciones. Si el trap era síncrono, hay que añadir 4 a la dirección de retorno para volver a la instrucción que sigue a la que provocó el trap.

2. **Comprobación de desbordamiento** Al producirse un trap, siempre se decrementa el CWP. Hay que comprobar si, como resultado de esta operación, el CWP señala a la ventana inválida. En tal caso, habrá que proceder igual que cuando se produce un desbordamiento (ver figura 2.14).

3. **Comprobación de anidamiento** Si la interrupción ocurre mientras se está ejecutando una tarea, hay que modificar el puntero de pila para que apunte a la pila de las interrupciones. Si la interrupción está anidada, es decir, ha ocurrido durante la ejecución de otra interrupción, el puntero de pila ya estará apuntando donde debe. En cualquiera de los casos, se incrementa el nivel de anidamiento.

4. **Guardar el estado de la tarea/interrupción** Dependiendo de si hubo anidamiento o no, habrá que guardar el estado de la interrupción inacabada o de la tarea interrumpida. El estado de una interrupción se guarda en la pila de las interrupciones, el de una tarea se guarda en el espacio que cada tarea tiene reservado para salvar su contexto de ejecución.

5. **Establecer el nivel de interrupción** Se ajusta el valor del campo PIL del registro de estado para reflejar el nuevo nivel de interrupción. De esta manera, cuando se habiliten los traps, las interrupciones de menor nivel al marcado se ignorarán. Obsérvese que el nivel de interrupción se obtiene fácilmente a través del número que indica el tipo de trap (tt). Basta con restar 16 (0x10 en hexadecimal) a este número para calcular el nivel de interrupción correspondiente (ver tabla 4.1).

6. **Llamar al manejador de usuario** Una vez guardado el estado del procesador, se pueden habilitar los traps y llamar a la rutina de servicio de interrupción programada por el usuario (en lenguaje de alto nivel).

7. **Vuelta del manejador de usuario** Durante la ejecución del manejador pueden producirse interrupciones de mayor nivel que deban ser atendidas. En tal caso, para procesar la interrupción se siguen los mismos pasos descritos hasta el momento. Habiéndose terminado de ejecutar el manejador de usuario, se deshabilitan los traps para recuperar el estado guardado y finalizar la rutina de servicio de interrupción.

8. **Recuperación del contexto y vuelta de la interrupción** En la versión 2.2 de ORK, esta parte del tratamiento de interrupciones difiere en gran medida de lo implementado en la versión 2.1 y anteriores. Esto se debe, principalmente, a que el manejador de interrupción ha podido desencadenar un cambio de contexto. Como ya dijimos en el capítulo 3, el cambio de contexto es distinto en las versiones 2.1 y 2.2 de ORK. Estas diferencias se reflejan también en el manejador común de interrupciones. En ambas versiones, sin embargo, el cambio de contexto no se realiza hasta que el nivel de anidamiento de interrupciones llega a cero. Es decir, aunque un manejador haya pedido un cambio de contexto, éste no se llevará a cabo hasta que todos los demás manejadores hayan terminado de ejecutarse también.

   - En la versión 2.1, siempre que se requiera, se llama directamente a la rutina de cambio de contexto a la salida del manejador (recuérdese que ésta estaba implementada como subrutina hoja). Si el cambio de contexto se produce, una pequeña parte del manejador común de las interrupciones (la que recupera el estado de la tarea interrumpida) se deja sin ejecutar. Esta parte se ejecuta cuando ocurra otro cambio de contexto hacia la tarea que fue interrumpida. Cuando dicha tarea recibe el control, termina el procesamiento de la interrupción y continúa con su ejecución normal.
   - En la versión 2.2 de ORK, por el contrario, el cambio de contexto se realiza directamente. Esto significa que una buena parte del código de la rutina de cambio de contexto está prácticamente replicado (quizás se podría compartir de algún modo). A cambio, se gana más control sobre el procesamiento de las interrupciones. Recordemos que en la versión 2.2 de ORK, el cambio de contexto se implementaba como un trap, siendo ahora posible que sea a la salida del manejador común de interrupciones cuando se haga efectivo el cambio de contexto, ya que el contexto de ejecución se guarda de igual forma en ambos casos (interrupción o trap de cambio de contexto). Este control adicional sobre

el cambio de contexto y las interrupciones ha permitido implementar una optimización relativa a la utilización de operaciones de coma flotante en los manejadores de interrupción (ver apartado 4.2.1).

4.2.1 Uso de la FPU en manejadores de interrupción

Al igual que sucede con las tareas, los manejadores de interrupción pueden realizar operaciones de coma flotante. Esto implica tener que guardar el estado de la FPU cada vez que se produce una interrupción y recuperarlo al término del procesado de la misma. En la versión 2.1 de ORK, era así como efectivamente se actuaba. Recordemos que en dicha versión, todas las tareas estaban marcadas como usuarias de la FPU, puesto que el mecanismo de detección del uso de la coma flotante todavía no estaba operativo.

Sin embargo, a partir de la versión 2.2 de ORK, la detección automática de tareas que usan la FPU está activada, permitiendo la optimización del cambio de contexto cuando no se necesitan guardar los registros de coma flotante. La pregunta es si se podría hacer algo similar con los manejadores de interrupción, es decir, si se podría detectar automáticamente el uso de la FPU durante la ejecución de un manejador para no tener que guardar siempre el contexto de coma flotante. La respuesta es que, gracias a la integración entre el cambio de contexto y el tratamiento de interrupciones, sí es posible.

Los pasos a seguir para que el contexto de coma flotante sólo se guarde cuando sea absolutamente necesario son los que se muestran a continuación. Estos pasos están codificados dentro del manejador común de las interrupciones y se sitúan entre los descritos más arriba:

1. *Dejar pendiente el almacenamiento del estado de la FPU* Una vez se ha guardado el estado de la tarea interrumpida, si ésta era usuaria de la FPU, la dirección donde almacena su contexto de ejecución se guarda en la variable FP_Save_Pending (igual que se hace cuando se cambia el contexto desde una tarea que usa la FPU a otra que no). Si es un manejador de interrupción el que, a su vez, es interrumpido, es la dirección de la pila donde el manejador guardó el estado del procesador la que se almacena en la variable FP_Save_Pending (siempre que el manejador haya hecho uso de la FPU). En ambos casos, el orden en el que se guardan los registros de coma flotante es el mismo, siendo compatibles entre sí. Si bien, en el caso del manejador, es necesario realizar un pequeño ajuste a la dirección de la pila para señalar el inicio del espacio donde hay que guardar el estado de la FPU.

2. *Ejecución del manejador* Antes de empezar a ejecutar el manejador de interrupción del usuario, se pone a cero el bit EF del registro de estado. De esta

forma, si el manejador intenta realizar una operación de coma flotante, se disparará el trap `fp_disabled`. El manejador de este trap guarda el estado de la FPU en la dirección indicada por la variable FP_Save_Pending y pone a uno el bit EF, permitiendo que el manejador de usuario pueda realizar sus operaciones de coma flotante en lo sucesivo.

3. *Vuelta del manejador* Al volver del manejador de usuario, primero se comprueba si la interrupción estaba anidada. Dependiendo de si la interrupción era la más externa o no, se efectúan distintas operaciones:

   - *La interrupción estaba anidada.* Eso significa que se va a volver a otro manejador de interrupción. Si el manejador al que se va a volver no usó la FPU, no es necesario hacer nada. En cambio, si hizo uso de la unidad de coma flotante, hay que comprobar si el valor de la variable FP_Save_Pending coincide con la dirección de su pila destinada a almacenar el estado de la FPU. Si no coincide, significa que un manejador de mayor nivel cambió el valor de la variable porque hizo uso de la FPU, disparando el trap de `fp_disabled` y, consecuentemente, guardándose los registros de coma flotante en el espacio de pila mencionado. Ahora es cuando se recuperan estos registros para que el manejador interrumpido pueda continuar con sus cálculos.
   - *La interrupción no estaba anidada.* En este caso, se devuelve el control a la tarea que fue interrumpida o a otra más prioritaria que haya sido despertada por el manejador de la interrupción. Antes se comprueba si la tarea a la que se va a volver usa la FPU. Si la respuesta es no, no es necesario hacer nada. Pero si la respuesta es positiva, seguidamente se observa el valor de la variable FP_Save_Pending. En caso de que el valor de dicha variable coincida con la dirección donde la tarea guarda su contexto, no es necesario hacer nada, ya que esto quiere decir que ninguna otra tarea ni manejador de interrupción han efectuado ninguna operación de coma flotante desde que la tarea en cuestión dejó el procesador. Por lo tanto, el contenido de los registros de la FPU seguirá siendo igualmente válido. Si los valores son distintos, sí que habrá que guardar el contexto de coma flotante, excepto en el caso de que la variable FP_Save_Pending tenga valor nulo, es decir, no haya ningún contexto de FPU pendiente de ser guardado. Si se llega a este punto de la ejecución es porque la tarea a la que se va a volver sí utiliza la FPU, por lo que habrá que recuperar su estado a partir del espacio de memoria donde la tarea lo tiene guardado. Después de esto, se recupera el contexto de ejecución completo de la tarea y se continúa con su procesamiento.

Veamos un ejemplo de una tarea que utiliza la FPU y que es interumpida. El manejador de la interrupción también usa la FPU y ocurre que, a mitad de su procesamiento, se produce una segunda interrupción de mayor nivel que la primera. El manejador de esta segunda interrupción, al igual que el primero, también realiza operaciones de coma flotante. El escenario resultante se muestra en la figura 4.2.

Para cada uno de los puntos marcados en el diagrama, la figura 4.3 nos indica si la FPU está habilitada (bit EF del PSR a uno), dónde se guarda el contexto de coma flotante (espacio de la tarea o pila de interrupciones) y el valor de la variable FP_Save_Pending.

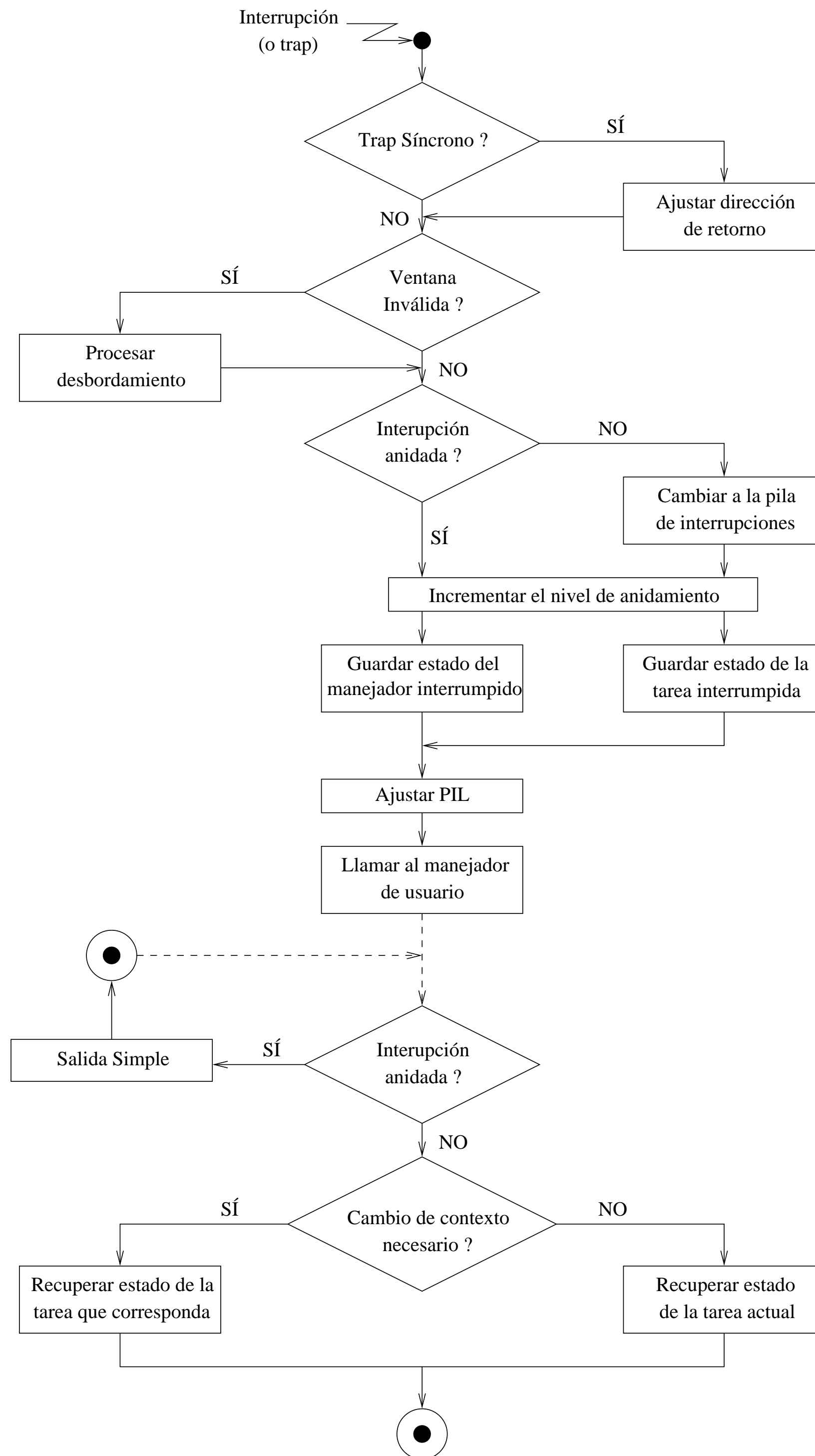


Figura 4.1: Manejador común de las interrupciones

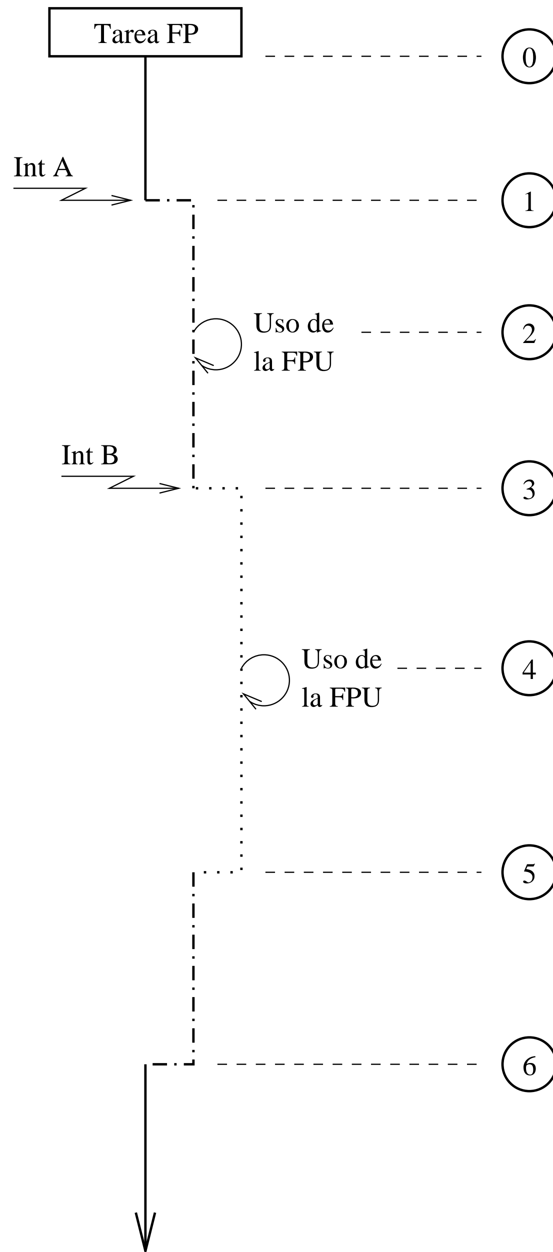


Figura 4.2: Ejemplo de tarea interrumpida

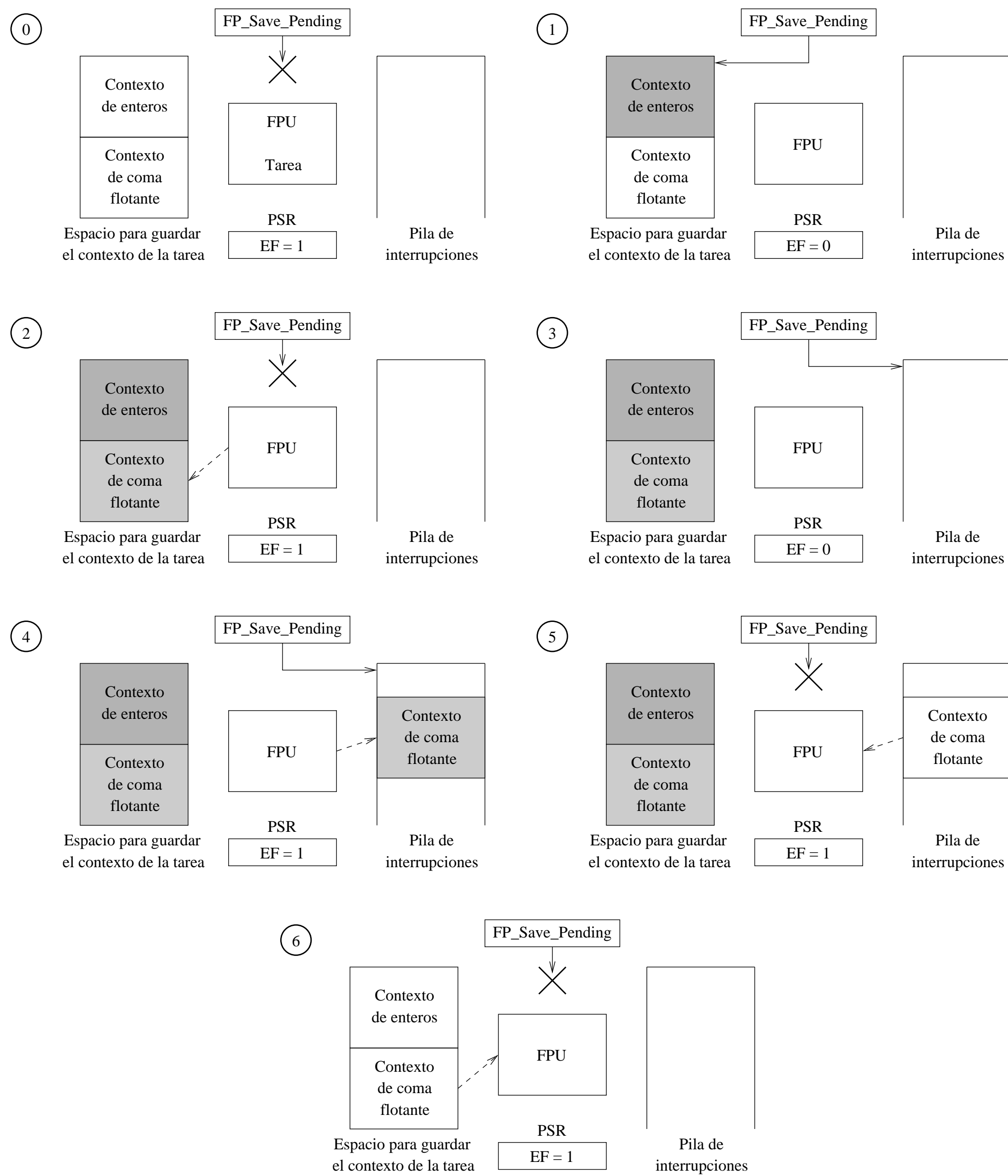


Figura 4.3: Gestión de la FPU en interrupciones

Capítulo 5

La Protección de Pilas

En el apartado sobre el cambio de contexto ya se introdujo someramente el tema de la protección de pilas. En este apartado veremos específicamente en qué consiste la protección de pilas y cuáles son las limitaciones del modelo de protección que se ha usado finalmente en ORK.

5.1 Introducción

En un sistema multitarea, cada tarea debe guardar cierta información referente a su estado en la memoria. El espacio de memoria que se suele usar a este efecto es la pila. Así pues, cada tarea deberá disponer de su propio espacio de pila donde poder guardar su información local. Ninguna tarea debería poder escribir sobre el espacio de pila de cualquier otra. En caso contrario, se corromperían los datos de la tarea cuya pila ha sido invadida, siendo imprevisibles las consecuencias que esto tendría sobre el funcionamiento del sistema.

La pila es una estructura de datos que funciona habitualmente como un contenedor de elementos en el que el último elemento en entrar es el primero en salir (estructura *LIFO, Last In First Out*). Habitualmente, las pilas comienzan en las direcciones altas de la memoria y crecen hacia las bajas, a medida que se les van añadiendo elmentos. Los procesadores suelen tener instrucciones de tipo *push* (para añadir un elemento a la pila) y *pop* (para retirar el último elemento que se añadió). Sin embargo, la arquitectura SPARC no define ninguna de dichas instrucciones (en parte porque es una arquitectura RISC) y se accede a la pila mediante las instrucciones de acceso a memoria habituales. De cualquier manera, las pilas crecen igualmente hacia abajo, debido al uso que las subrutinas hacen de la instrucción *save* para reservar marcos de pila.

El caso más habitual de escritura en el espacio de pila de otra tarea es el desbordamiento. En los sistemas con memoria virtual, las pilas se colocan de tal forma

que haya grandes espacios de memoria entre ellas y cualquier otro segmento del programa. Así, si se necesita más espacio de pila debido a una adición excesiva de elementos sobre la misma (desbordamiento), bastará con pedir más espacio al sistema, que éste ya se encargará de proporcionarlo de algún modo (puede que tenga que utilizar el disco). En los sistemas sin memoria virtual y sin dispositivos de almacenamiento secundario como el que nos ocupa, no se dispone de tanta memoria o al menos no de tantas direcciones lógicas como para aislar una pila del resto del programa. En este tipo de sistemas, se suele optar por una configuración de pilas en serie: se colocan una detrás de otra, ocupando un espacio en la memoria que dependerá del número de tareas y del tamaño de sus pilas. El problema con esta configuración es que el desbordamiento de una de las pilas afecta a la que está situada en el espacio de memoria inmediatamente inferior. Esta situación, reflejada en la figura 5.1, provoca una pérdida irreversible de información, puesto que parte de los datos de la pila invadida son destruídos.

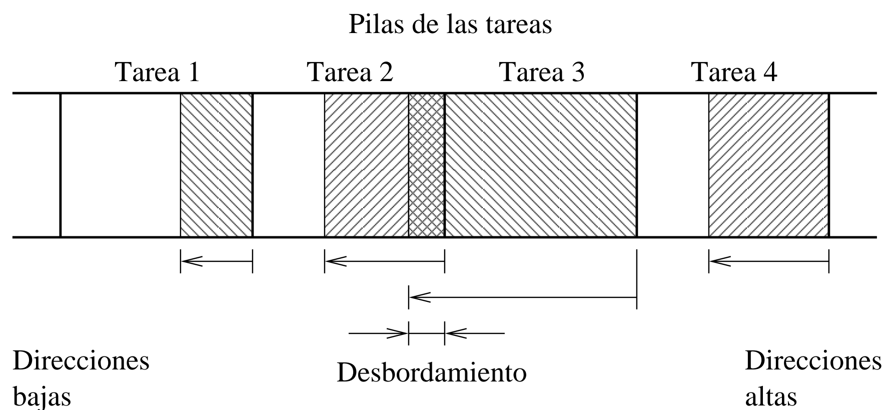


Figura 5.1: Desbordamiento de una pila

En un sistema de tiempo real crítico se debe evitar a toda costa el desbordamiento de cualquiera de sus pilas. Durante el análisis estático del sistema, se determina el espacio máximo de pila que puede usar cada tarea. Esta cifra se calcula teniendo en cuenta el número de llamadas a procedimiento que efectúa la tarea y el espacio que ocupan sus variables locales. Si los cálculos se hacen correctamente, ninguna de las tareas deberá sobrepasar su espacio de pila asignado y el sistema funcionará bien en este aspecto.

A pesar de todo, conviene que el sistema disponga de algún mecanismo alternativo que detecte el desbordamiento de la pila de una tarea. Hay dos razones principales para ello:

1. El desarrollo de aplicaciones prototipo o de prueba en las que no se calcu-

le el tamaño máximo de las pilas por razones de tiempo. En este caso, la detección es el único método que permite saber si se ha producido desbordamiento.

2. Comprobar la validez de los cálculos referentes al tamaño de las pilas, realizados durante el análisis estático del sistema.

5.2 Implementación

El controlador de memoria del ERC32 (MEC) posee un mecanismo para proteger contra escritura ciertas zonas de memoria RAM. La protección contra escritura está basada en segmentos. Un segmento define un área de la memoria donde están permitidos los ciclos de escritura. El MEC está diseñado para manejar dos segmentos. Cada segmento está implementado mediante dos registros: el registro **Base de segmento** y el registro **Fin de segmento**. El registro base de segmento indica la dirección de memoria en la cual comienza el segmento. Además, contiene los bits **SE/UE** (*Supervisor Enabled/User Enabled*), que habilitan el funcionamiento de la protección de memoria según el modo en el que se encuentre el procesador, supervisor o usuario. El registro fin de segmento indica la primera dirección de memoria que deja de pertenecer al segmento.

En realidad, los valores que se almacenan en los registros de segmento no son directamente las direcciones de memoria que protegen. Para obtener la dirección de memoria a la que apuntan, hay que multiplicar dicho contenido por cuatro y sumarle `0x2000000` (la dirección donde comienza la RAM). Las direcciones que definen un segmento, por lo tanto, están alineadas a un tamaño de palabra (*word*) del procesador.

Existe un bit en el registro de control del MEC que define el modo de funcionamiento de la protección mediante segmentos. Es el bit **BP** (*Block Protection*, Protección de Bloque). Si el bit está a cero, cualquier escritura fuera de los segmentos definidos es detectada, generándose un trap. En el caso de que el bit esté a uno, el comportamiento se invierte: los segmentos pasan a ser bloques protegidos de memoria. La escritura sobre cualquiera de estos bloques será la que dispare el trap.

ORK implementa la protección de pilas aprovechando las capacidades de protección de memoria del MEC. Durante el desarrollo de ORK, se probaron dos posibles configuraciones de protección de pilas. Las dos se basan en usar el modo de protección de bloque que se acaba de describir.

5.2.1 Primera Opción de Configuración

En ORK, las pilas de las tareas están situadas en memoria una a continuación de la otra. Además, se necesita una pila adicional que puedan utilizar los manejadores de interrupción. La primera configuración en la que se pensó para implementar el mecanismo de protección de pilas se recoge en la figura 5.2.

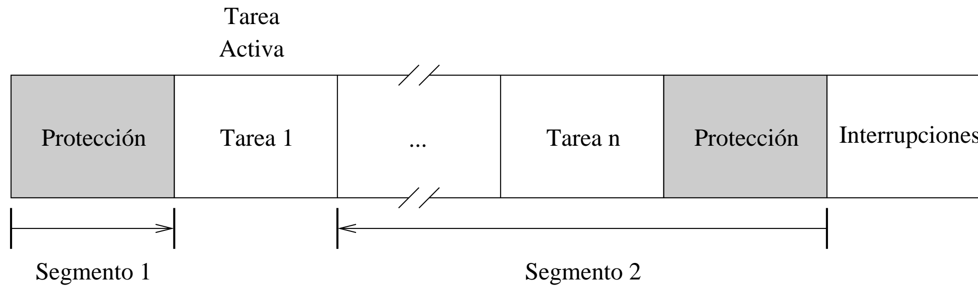


Figura 5.2: Protección de pilas: Opción 1

En la figura podemos distinguir las pilas de las tareas, la pila de interrupciones y dos áreas de memoria denominadas de protección. Una de estas áreas está destinada a separar el bloque de pilas del resto de los datos del programa, que se encuentran por debajo. La otra área de protección separa la pila de las interrupciones de las pilas de las tareas. La pila de las interrupciones no se protege nunca. Esto es necesario porque da igual la tarea que se esté ejecutando en un instante dado, en cualquier momento puede llegar una interrupción y el manejador de la misma debe disponer de un espacio de pila adecuado.

La longitud de los segmentos de protección se modifica durante los cambios de contexto. Los segmentos se alargan o se acortan de tal manera que la pila de la tarea activa sea la única sobre la que se pueda escribir, protegiendo el resto de pilas.

Este modelo de protección de pilas es bastante seguro, pero existe un problema que impide adoptarlo como solución definitiva. La causa del problema reside en una optimización de los objetos protegidos de Ada conocida por el nombre de modelo *Proxy*. El modelo *Proxy* puede hacer que una tarea deba acceder a la pila de otra en ciertos casos.

Supongamos, por ejemplo, que una tarea realiza una llamada a un punto de entrada de un objeto protegido y que la condición impuesta por la barrera de la entrada no se cumple, quedándose la tarea bloqueada en la llamada. Supongamos ahora que otra tarea ejecuta una operación sobre el mismo objeto protegido. Una vez finalizada la operación, se vuelven a evaluar las barreras de cada uno de los puntos de entrada del objeto protegido. Puede suceder que, como consecuencia de

la ejecución de la operación protegida, la barrera que bloqueaba nuestra primera tarea se haya abierto, permitiendo que continúe su ejecución. Pero recordemos que todavía estamos dentro del contexto de ejecución de la segunda tarea. Habría que efectuar pues, un cambio de contexto desde la tarea en ejecución a la que quedó bloqueada.

Como el ejemplo expuesto es un caso habitual en el funcionamiento de los objetos protegidos, se ideó una forma de optimizarlo para máquinas monoprocesador. Para ahorrarse el último cambio de contexto, la tarea que ejecuta la operación protegida y que evalúa las barreras también ejecutará el código de la entrada en la que se quedó bloqueada la otra tarea. Esta suplantación de una tarea por otra hace que la implementación de los objetos protegidos sea más eficiente, pero también es responsable de que no se pueda emplear el modelo de protección de pilas descrito. Durante la ejecución del código de la entrada al objeto protegido, puede suceder que se necesite escribir un valor en la pila de la tarea que se quedó bloqueada. Sin embargo, la tarea activa en ese momento es la otra, estando prohibida la escritura en cualquier pila que no sea la suya o la de interrupciones. El modelo falla en este caso, pues el supuesto original de que una tarea no debe interferir en el estado local de otra no se cumple.

5.2.2 Segunda opción de configuración

Para solucionar este problema se ideó otro sistema de protección de pilas, el cual se expone a continuación (ver figura 5.3). En esta ocasión, se reserva un espacio de protección entre la pila de una tarea y la siguiente. De esta forma, la pila de cualquier tarea es accesible independientemente de cuál sea la tarea activa. Sin embargo, la tarea que esté activa seguirá estando protegida contra desbordamiento gracias a su espacio de protección.

Figura 5.3: Protección de pilas: Opción 2

En este caso, el segundo segmento permanece fijo para proteger la pila de

interrupciones y es el primer segmento el que va cambiando de sitio según la tarea que esté activa.

Los dos métodos vistos detectan el desbordamiento de las pilas gracias a un mecanismo hardware. Existen también otros métodos para comprobar *a posteriori* si ha habido desbordamiento de pila. Se basan en inicializar los elementos de la pila con un valor determinado y luego van comprobando hasta qué punto se ha rellenado la pila con otros valores. Estos métodos tienen la ventaja de que son siempre aplicables, incluso cuando no existe un mecanismo especial en el hardware. Nosotros, sin embargo, hemos preferido utilizar la detección en lugar de la comprobación, debido a que presenta una serie de ventajas importantes:

- La detección no permite la destrucción de datos. En cuanto se intenta escribir en una dirección protegida, el sistema lo impide y se genera una excepción. La comprobación, en cambio, detecta el desbordamiento una vez que este ya se ha producido.

- La detección es inmediata y no introduce ninguna sobrecarga al sistema, exceptuando la necesidad de cambiar los segmentos de protección durante los cambios de contexto. La comprobación, por su parte, debe recorrer la pila que esté examinando para verificar que el valor de inicialización se mantiene en la parte de la pila que no ha sido utilizada todavía.

- La detección está siempre activa. La comprobación ha de realizarse cada cierto tiempo o en momentos específicos de la ejecución del programa. Cuando se hace la comprobación, puede que sea ya demasiado tarde

Por todo ello, concluímos que para ORK se debe mantener, en la medida de lo posible un esquema de detección de desbordamiento. De cualquier manera, el sistema que se ha adoptado no está exento de problemas, entre los cuales podemos destacar los siguientes:

- El tamaño de las áreas de protección

  En la primera de las configuraciones vistas sólo hay dos espacios de protección, por lo que se les puede dar un tamaño razonablemente grande sin que esto influya demasiado en el espacio de memoria total dedicado a las pilas.

  En la segunda configuración, sin embargo, se requiere un espacio de protección por cada tarea del sistema. Si este espacio es muy grande, el mecanismo será más seguro, pero se desperdiciará una cantidad importante de la memoria del sistema y recordemos que los circuitos de memoria destinados a aplicaciones espaciales son muy caros. Si, por el contrario, las áreas de protección fueran muy pequeñas, se corre el riesgo de que una tarea reserve

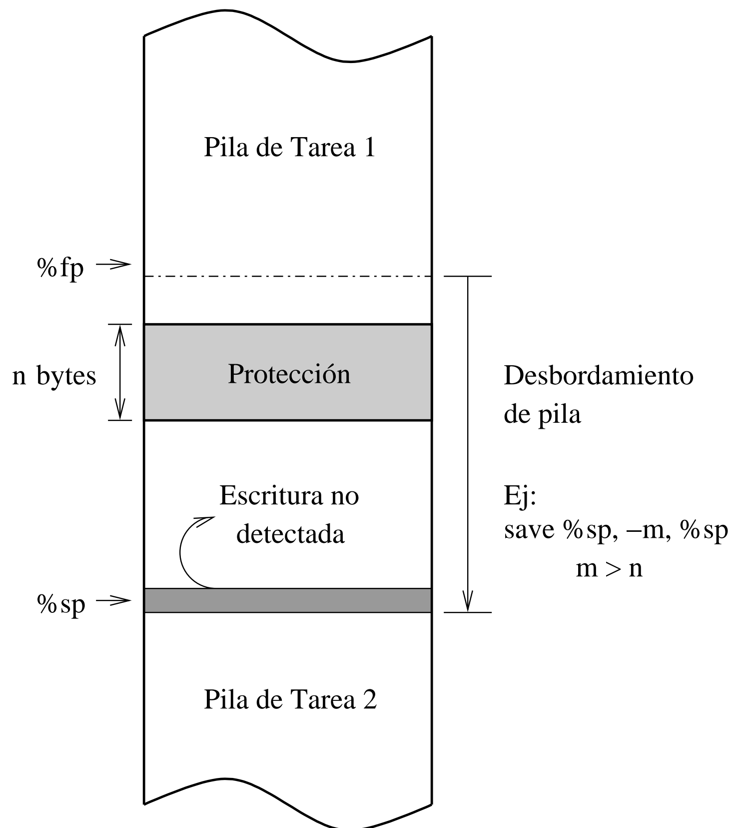


Figura 5.4: Sobrepasar el espacio de protección

mucho espacio de pila de golpe y haga inútil el mecanismo de protección (ver figura 5.4).

- Desbordamiento en objetos protegidos

  Como hemos visto más arriba, durante la ejecución de los objetos protegidos y debido al modelo *Proxy*, una tarea puede ejecutar el código correspondiente a otra para evitar un cambio de contexto.

  Sin embargo, el translado de los segmentos de protección de pilas se lleva a cabo, precisamente, durante los cambios de contexto. Por lo tanto, cuando una tarea ejecuta el código de una entrada por otra, la pila de esta última podrá ser modificada, pero estará desprotegida.

  Si se diera el caso que durante la ejecución de la entrada al objeto protegido se desbordara la pila de la tarea que fue bloqueada, no habría ninguna forma de detectar dicho desbordamiento. La única solución a este problema

sería la de mover los segmentos de protección en el momento en el que la tarea que cambia el estado de la barrera vaya a substituir a una de las tareas bloqueadas. La desventaja de esta solución es la repercusión que tendría sobre la eficiencia de los objetos protegidos. El modelo *Proxy* se basa en la supresión del cambio de contexto y la solución propuesta hace justamente lo contrario: ejecutar una parte del mismo.

- El procesado del trap de desbordamiento

  Una vez se ha producido el desbordamiento de una pila, el sistema debe procesar este caso de error y tomar las medidas oportunas. Es normal que las rutinas del sistema operativo encargadas de esta labor necesiten cierto espacio de pila para efectuar su trabajo. Pero la pila de la tarea en ejecución estará agotada debido al desbordamiento, por lo que no conviene ejecutar inmediatamente las rutinas del sistema operativo.

  Primero habrá que cambiar a la pila de las interrupciones, que es lo que se hace normalmente durante el procesamiento de un trap. A diferencia de un trap normal, no se deberá guardar el estado de la tarea interrumpida en su propia pila porque, insistimos, está desbordada.

  El caso más delicado de desbordamiento es justamente el que se produce cuando es la pila de las interrupciones la que se desborda. En esta ocasión no tiene mucho sentido seguir usando la misma pila para procesar el trap de desbordamiento y, a pesar de todo, no hay otra pila en el sistema preparada para esta eventualidad.

  Una particularidad de la pila de interrupciones es su unicidad en el sistema. Si la pila de una tarea se desborda, se puede suspender la tarea para siempre y continuar con las demás. Pero si es la pila de interrupciones la que agota su espacio, es todo el sistema el que debe suspenderse. Por ello, recomendamos que para procesar el desbordamiento de la pila de interrupciones, se siga utilizando dicha pila, pero con el mecanismo de protección ya deshabilitado. La sobreescritura de datos sobre la pila de alguna tarea no tendrá ya ninguna importancia porque el objetivo de procesar el trap es provocar la parada total del sistema. Podemos concluir que, si bien todas las pilas deben estar bien dimensionadas, la pila de las interrupciones debe estarlo especialmente porque las consecuencias de su desbordamiento son muy graves.

  En la implementación actual de ORK, una vez se produce el trap que indica el desbordamiento de una pila, se llama directamente al tratamiento de excepciones de Ada. Como hemos señalado más arriba, sería mejor situarse en la pila de interrupciones antes de llamar a ningún otro procedimiento.

También convendría detectar si el desbordamiento se produjo durante el procesamiento de una interrupción y, por lo tanto, si hay que parar todo el sistema.

5.3 Posibles mejoras

Como acabamos de ver, los problemas que presenta la protección de pilas son muchos y variados. Estos problemas se deben, en parte, a que el momento en el que se reserva espacio de pila se considera desconocido. Sin embargo, en la arquitectura SPARC, la reserva de pila se produce en momentos muy determinados de la ejecución de un programa: las entradas a subrutina (ver apartado 2.3).

El código que GCC utiliza como prólogo de cada subrutina se encuentra en un fichero que depende de la máquina para la cual se configuró el compilador. Estos ficheros depedientes del hardware se encuentran dentro de la distribución de los fuentes de GCC en el directorio `config/cpu/`, donde texttcpu es el nombre del procesador para el que se desea generar código. En nuestro caso, el nombre del procesador es “sparc” y el fichero que se encuentra en el directorio `config/sparc/` y contiene el código prólogo de las subrutinas recibe el nombre de “sparc.c”. La función `output_function_prologue` de este fichero es la encargada de generar las instrucciones que conforman el prólogo de todas las funciones C compiladas con GCC y, por extensión, de todos aquellos procedimientos y funciones Ada que se compilen con el frontal de GNAT.

Es aquí, justo en el momento en el que se reclama memoria para la pila, donde se podría incorporar algún trozo de código que comprobara si el puntero de pila rebasa los límites impuestos.

El problema que encontramos con esta aproximación es que los límites del espacio de pila de cada tarea no son fácilmente accesibles desde esta función del compilador. Insertar una gran cantidad de código en esta función para poder determinarlos tendría un efecto negativo sobre el rendimiento de las aplicaciones. No hay que olvidar que dicha función produce código para todas y cada una de las subrutinas del programa a compilar.

Aun así, esta opción no ha sido investigada muy a fondo. Si se encontrase un conjunto de instrucciones pequeño que pudiera hacer el trabajo, sin duda éste sería el lugar idóneo donde situarlo. Para no tener que comparar el puntero de pila con los límites se podría pensar en realizar una combinación de este método con el de la protección de bloques de memoria.

En cualquier caso, el acceso por parte de una tarea a la pila de otras seguirá siendo el escollo más importante a la hora de encontrar soluciones al problema de la protección de pilas.

Capítulo 6

Los periféricos del MEC

El controlador de memoria MEC, a pesar de su nombre, no limita sus funciones al manejo de los circuitos de memoria del sistema. El MEC contiene también un gran número de periféricos que dotan al ERC32 de capacidades propias de un microcontrolador en cuanto a comunicación con el exterior y temporización se refiere. En este capítulo veremos algunas de las capacidades del MEC y cómo programar este dispositivo usando el lenguaje Ada 95.

6.1 Programación de bajo nivel con Ada 95

El lenguaje Ada posee unos mecanismos que le permiten especificar cómo ciertas variables han de ser representadas en el hardware. Por ejemplo, permiten definir su tamaño, su alineamiento o su dirección en la memoria e incluso su organización a nivel de bit. Esto es muy útil para programar los registros del MEC usando Ada. Un registro del MEC se puede implementar como un **record** indicando cómo han de representarse sus componentes y qué dirección de memoria debe usar para dicho registro. Estas indicaciones a un **record** y a sus componentes se denominan atributos y cláusulas de representación, respectivamente, según la terminología de Ada 95.

Usando estos mecanismos, ORK define en el paquete Kernel.Peripherals.Registers aquellos registros del MEC que son utilizados por el sistema operativo. Veremos algunos de ellos como ejemplo de las capacidades de representación de Ada 95 en los siguientes apartados.

6.2 El puerto serie

El MEC está dotado internamente de dos transmisores/receptores universales asíncronos (**UART**). Este tipo de dispositivos permiten la comunicación con el exte-

rior a través de un puerto serie. En las pruebas de ORK, se han utilizado para simular la salida por pantalla. El emulador de ERC32 usado (el TSIM) tiene la capacidad de emular gran parte de las funciones y periféricos del MEC, entre los cuales se encuentran los UART. El TSIM conecta el canal A del UART emulado a la entrada y salida estándar del sistema operativo sobre el que se ejecute (GNU/Linux en nuestro caso). De esta forma, se visualizan por pantalla los datos que transmite el puerto serie y un programa puede leer datos que le lleguen del teclado. Las aplicaciones sobre hardware real pueden usar el puerto serie para su comunicación con el exterior y los programas monitor pueden utilizarlo igualmente para ofrecer información de depuración.

Para configurar la velocidad, la paridad y el número de bits de parada del puerto serie hay que programar el registro de control del MEC. Una vez está configurado, ya se pueden a transmitir o recibir datos a través de los dos canales UART. Existe también un registro que permite conocer el estado de la transmisión. Consultando los bits de este registro se puede saber, por ejemplo, si se ha recibido un octeto y está esperando a ser leído o si se ha transmitido un octeto y ya se puede enviar el siguiente. Usaremos este registro de ejemplo para mostrar cómo se representa un registro de dispositivo en Ada 95.

La definición de este registro se encuentra en la tabla 6.1. Para implementarlo en Ada hay que utilizar un **record** con claúsulas de representación. Cada bit del registro se representa con una variable booleana: si el valor de la variable es "verdadero" el bit contiene un uno y si el valor es "falso" contiene un cero. Los grupos de bits reservados se representan mediante **arrays** de booleanos empaquetados.

```
1   type UART_Status_Register is
       record
          DRA : Boolean;
          TSEA : Boolean;
5         THEA : Boolean;
          Reserved1A : Boolean;
          FEA : Boolean;
          PEA : Boolean;
          OEA : Boolean;
10        CUA : Boolean;
          Reserved8A : Reserved_8;
          DRB : Boolean;
          TSEB : Boolean;
          THEB : Boolean;
15        Reserved1B : Boolean;
          FEB : Boolean;
          PEB : Boolean;
```

Bits	Nombre	VI	Función	E/L
0	DRA	0	Datos preparados en canal A	L
1	TSEA	1	Transmisor A, registro de envío vacío	L
2	THEA	1	Transmisor A, registro de espera vacío	L
3	Reservado	0	No usado	L
4	FEA	0	Error de trama en receptor A	L
5	PEA	0	Error de paridad en receptor A	L
6	OEA	0	Error de sobrecarga en receptor A	L
7	CUA	0	Borrar UART A	L/E
8-15	Reservados	0	No usados	L
16	DRB	0	Datos preparados en canal B	L
17	TSEB	1	Transmisor B, registro de envío vacío	L
18	THEB	1	Transmisor B, registro de espera vacío	L
19	Reservado	0	No usado	L
20	FEB	0	Error de trama en receptor B	L
21	PEB	0	Error de paridad en receptor B	L
22	OEB	0	Error de sobrecarga en receptor B	L
23	CUB	0	Borrar UART B	L/E
24-31	Reservados	0	No usados	L

Tabla 6.1: Registro de estado del puerto serie

```
            OEB : Boolean;
            CUB : Boolean;
20          Reserved8B : Reserved_8;
        end record;

    for UART_Status_Register use
        record
25          DRA        at 0 range 31 .. 31;
            TSEA       at 0 range 30 .. 30;
            THEA       at 0 range 29 .. 29;
            Reserved1A at 0 range 28 .. 28;
            FEA        at 0 range 27 .. 27;
30          PEA        at 0 range 26 .. 26;
            OEA        at 0 range 25 .. 25;
            CUA        at 0 range 24 .. 24;
            Reserved8A at 0 range 16 .. 23;
            DRB        at 0 range 15 .. 15;
```

```
35            TSEB       at 0 range 14 .. 14;
              THEB       at 0 range 13 .. 13;
              Reserved1B at 0 range 12 .. 12;
              FEB        at 0 range 11 .. 11;
              PEB        at 0 range 10 .. 10;
40            OEB        at 0 range 9 .. 9;
              CUB        at 0 range 8 .. 8;
              Reserved8B at 0 range 0 .. 7;
         end record;

45    for UART_Status_Register'Size use 32;

      pragma Pack (UART_Status_Register);

      pragma Atomic (UART_Status_Register);
50
      pragma Suppress_Initialization (UART_Status_Register);
```

En las claúsulas de representación, podemos observar que los campos del registro se colocan de forma inversa a como estaban definidos en la tabla 6.1. Sin embargo, esta es la forma correcta de ponerlos, debido a que la arquitectura SPARC guarda las variables en memoria empezando con el bit más significativo en la dirección más baja (*big endian*).

El atributo de representación `Size` le dice al compilador que debe usar 32 bits para almacenar el valor del registro en la memoria. Luego se empaqueta el registro con un `pragma Pack` para que el compilador optimice el espacio de memoria del registro y los booleanos declarados ocupen realmente un solo bit.

El MEC es un dispositivo cuyos registros han de leerse y escribirse enteramente en una sola operación. Es decir, para acceder a los registros del MEC hay que usar instrucciones de carga o almacenamiento cuyo tamaño de operando sea una palabra (32 bits). De lo contrario, el MEC da un error de paridad interno que paraliza el procesador y lo deja en estado erróneo. El problema es que no existe ninguna cláusula ni atributo de representación en Ada que informe al compilador sobre el tamaño de operando que debe usar para modificar o leer ciertas variables. Por ejemplo, para escribir sobre el registro de estado del UART nada impediría al compilador usar dos instrucciones que usen un tamaño de operando de 16 bits, escribiendo primero el los 16 bits más significativos y luego los menos significativos. Sin embargo, como acabamos de decir, esto provocaría un error de paridad en el MEC. La solución que se ha encontrado es utilizar el `pragma Atomic` sobre la variable que representa el registro. Esta orden le dice al compilador que el acceso al tipo que modifica ha de ser indivisible desde el punto de vista de la

concurrencia. Es decir, si una tarea está escribiendo sobre una variable *atómica* y otra está leyendo de ella, ésta última podrá leer un valor antiguo o uno nuevo de la variable, pero nunca podrá leer un valor intermedio de la variable mientras la otra tarea la está modificando: la escritura y la lectura son indivisibles. Esta característica se implementa en monoprocesadores usando una sola instrucción para acceder a la variable. De esta forma no hay interferencia posible. De hecho, el `pragma Atomic` no se puede utilizar con cualquier tipo de variable. Para acceder atómicamente a variables de gran tamaño es necesario usar otros mecanismos de sincronización como los objetos protegidos. Lo interesante es el hecho de que sólo se utilice una instrucción para acceder a las variables *atómicas*, porque esto es precisamente lo que necesitábamos para leer y escribir sobre los registros del MEC sin generar errores.

Finalmente, el *pragma Suppress_Initialization* sirve para que el compilador no intente inicializar los bloques de bits reservados dentro del registro. De este modo, el código generado es más eficiente y evitamos problemas de acceso parcial a la variable.

Para poder utilizar el registro recién creado, hay que declarar una variable del tipo `UART_Status_Register`. El registro hardware real se encuentra mapeado en memoria (como todos los registros accesibles del MEC) en la posición 0x01F800E. Para hacer corresponder la variable declarada con el registro real se utiliza un atributo de representación. Este atributo indica la dirección de la memoria donde se debe guardar el contenido de la variable. La dirección de memoria que se usa es, obviamente, la misma donde se encuentra el registro hardware.

```
1   UART_Status_Register_Address : constant System.Address :=
2                                  To_Address (16#01F800E8#);

4   UART_Status : UART_Status_Register;

6   for UART_Status'Address use UART_Status_Register_Address;
```

Los dos registros del puerto serie que se utilizan para enviar y recibir datos por cada canal se han implementado de forma similar. El módulo de ORK que utiliza estos registros para simular la salida por pantalla es el Kernel.Serial.Output. Los subprogramas de este paquete han tomado el nombre de los que se encuentran en la biblioteca estándar de Ada para las funciones entrada y salida (paquete Text_IO). Esto no quiere decir que no se pueda utilizar el paquete Text_IO en los programas de ORK. De hecho, los ficheros de apoyo en C (ver apartadořefsec:apoyo) permiten el uso de las funciones estándar de entrada/salida. Sin embargo, la inclusión del paquete Text_IO aumenta de manera sensible la huella en memoria de cualquier aplicación. El paquete Kernel.Serial.Output, en cambio, contiene lo estrictamente necesario para implementar las funciones de salida

por pantalla, ocupando un espacio en memoria muchísimo menor. Este paquete configura el puerto serie a una velocidad de 19.200 baudios. Para ello hay que programar el campo **Scaler** y el bit **UBR** del registro de control del MEC. El valor a introducir se calcula con la siguiente fórmula:

$$Scaler = \frac{FrecdeReloj}{32 \times Baudrate \times (2 - UBR)} - 1$$

Donde la frecuencia del reloj de referencia puede escogerse entre el reloj del sistema o el de *Watchdog*.

6.3 Temporización

El controlador de memoria lleva incorporados dos relojes independientes: el reloj de propósito general y el reloj de tiempo real (aparte del reloj especial de detección de actividad *Watchdog*, que no se ha usado en ORK).

El tiempo se representa en ORK en nanosegundos mediante un número entero de 64 bits. Con esta longitud, se puede representar un intervalo de tiempo de casi 585 años. El tiempo del sistema se actualiza con una interrupción periódica producida por el reloj de tiempo real. El periodo de la interrupción es configurable, siendo por defecto de un segundo.

Para tener una mayor resolución se utiliza una combinación del reloj de tiempo real con el de propósito general. El primero se utiliza para generar interrupciones periódicas cada segundo. El de propósito general se programa para funcionar como reloj de alarmas: sólo se dispara una vez cuando alcanza el valor cero, no empieza a contar de nuevo. Combinando los dos relojes, se puede tener una precisión de un microsegundo en la generación de alarmas sin que por ello se tenga que producir una interrupción cada microsegundo (algo que, por otra parte, sobrecargaría el sistema de tal forma que no se podría procesar otra cosa más que interrupciones). La idea consiste en contar el tiempo con dos resoluciones distintas. Con el reloj periódico se van descontando los segundos que quedan para que se dispare la alarma. Cuando queda menos de un segundo para que el tiempo expire, se programa el reloj de propósito general, que es el que nos da una mejor resolución. Aunque este reloj cuente muy rápido, no existe el problema de que se desborde, puesto que queda menos de un segundo para que se alcance el tiempo deseado.

Cada reloj se compone de un escalador y un contador. El escalador es un contador suplementario que ajusta el paso al que ha de ir el contador principal. El escalador va descontando al ritmo que le marca el reloj del sistema. Cuando llega a cero, indica al contador principal que descienda en una unidad. Usando valores grandes en el escalador se consigue aumentar el intervalo temporal abarcable por

el reloj, si bien se disminuye su resolución. Los valores iniciales del escalador y el contador de cada reloj se cargan en los registros del MEC dispuestos para tal función. La única diferencia entre el reloj de tiempo real y el de propósito general es que este último dispone de un escalador de 16 bits, mientras que el escalador del primero es de 8 bits. Los contadores de ambos relojes son de 32 bits. El tiempo de expiración de los relojes se calcula así:

$$Tiempo = \frac{Contador(Scaler + 1)}{FrecdeReloj}$$

El máximo intervalo de tiempo representable por el reloj de propósito general es (para una frecuencia de reloj del sistema de 10 MHz):

$$Tiempo = \frac{(2^{32} - 1)((2^{16} - 1) + 1)}{10Mhz} = 28.147.497s$$

Con la misma frecuencia de reloj de sistema, el reloj de tiempo real alcanza a representar un valor de tiempo de:

$$Tiempo = \frac{(2^{32} - 1)((2^{8} - 1) + 1)}{10Mhz} = 109.951s$$

Siendo el valor de tiempo mínimo el mismo en ambos casos y limitado por la frecuencia del reloj del sistema. De nuevo, para una frecuencia de reloj de 10 MHz, el mínimo intervalo de tiempo programable sería:

$$Tiempo = \frac{1}{10Mhz} = 0.1\mu s$$

6.4 La protección de memoria

Los registros que rigen la protección de la memoria RAM del ERC32 se modifican habitualmente durante los cambios de contexto. Sin embargo, en la inicialización del sistema, estos registros se modifican gracias a un procedimiento codificado en Ada que usa las cláusulas de representación sobre records que se han visto en la discusión dedicada al puerto serie.

Los registros del MEC que se modifican están definidos, al igual que los del UART, en el paquete Kernel.Peripherals.Registers. El procedimiento que modifica estos registros se define en el paquete padre Kernel.Peripherals y se utiliza en el código de inicialización de la memoria destinada a las pilas. Este código se encuentra en el bloque de sentencias del paquete Kernel.Memory.

El procedimiento `Protect_Segment` es el que permite especificar un bloque de memoria RAM protegido contra escritura. Las direcciones de comienzo y final del segmento protegido deben estar alineadas a palabra (32 bits) y estar situadas a

partir de la dirección 0x2000000 (el inicio de la memoria RAM en una configuración habitual del ERC32).

Otra característica del MEC es que detecta el acceso a zonas no implementadas de la memoria o a zonas en las cuales la escritura es ilegal (ROM). En cualquiera de los casos, incluído el de escritura en un segmento protegido, se produce un trap que deberá ser tratado en consecuencia. Para ayudar al correcto tratamiento del trap, el MEC dispone de dos registros:

Registro de Estado de Fallo en el Sistema (*System Fault Status Register*) Identifica, mediante un código, la causa de la excepción.

Registro de Dirección Fallida (*Failing Address Register*) Contiene la dirección de la memoria a la que se accedió en el momento del fallo.

Estos datos pueden ser usados por el manejador del trap o por la persona encargada de depurar la aplicación, en el caso de que el manejador no pueda tratar la excepción correctamente.

6.5 Generación de interrupciones

El MEC cuenta con una herramienta muy útil para probar los manejadores de interrupción de una aplicación: la generación artificial de interrupciones. Escribiendo en uno de los registros del controlador de memoria (el **Interrupt Force Register**) se puede simular la llegada de una interrupción cualquiera. Basta con poner a uno el bit correspondiente a dicha interrupción en el citado registro.

Esta característica se puede emplear tanto en las pruebas que se hacen con el simulador TSIM, en las que no hay periféricos reales, como en las pruebas con la placa de ERC32, si no se dispone de ningún periférico que conectar.

Para las pruebas conviene usar aquellas interrupciones que estén asociadas a periféricos externos (ver tabla 4.1). Las interrupciones que ya tienen un determinado dispositivo del MEC asociado han de usarse con cuidado, puesto que compartirán el mismo manejador de interrupción y no habrá una forma inmediata de distinguir si la interrupción fue producida por el dispositivo real o si fue simulada.

Gracias a la conexión del canal A del puerto serie a la entrada estándar se pueden simular interrupciones esporádicas con una simple pulsación del teclado. Una forma práctica de probar un manejador continuadamente consiste en crear una tarea que genere interrupciones periódicamente. Si el periodo de la interrupción se elige adecuadamente, el programa puede dar lugar a una simulación cercana a la realidad estadísticamente hablando (y siempre dependiendo del tipo de dispositivo que se pretenda simular).

Capítulo 7

Herramientas GNU en Sistemas Embarcados

En este capítulo se recogen algunas de las modificaciones y añadidos que se han efectuado sobre las herramientas de desarrollo de GNU para adaptarlas al ERC32 y a las necesidades de ORK.

7.1 El fichero de órdenes al enlazador

El enlazador es el programa que recoge los ficheros objeto generados por el compilador y los transforma en un fichero ejecutable. El fichero de órdenes del enlazador le dice a éste dónde debe colocar cada sección del programa (código, datos inicializados, datos sin inicializar, etc...), definiendo el mapa de memoria de la aplicación.

En ORK, este fichero recibe el nombre de "commands.ld". Los símbolos que se declaran en este fichero son públicos y pueden ser usados por las rutinas de inicialización y de apoyo de ORK. De hecho, algunos de estos símbolos se usan para las funciones de reserva de memoria dinámica (ver apartado 7.4) y en el código de inicio de ORK (ver sección 7.3).

Gracias a este fichero, las distintas secciones que componen un programa pueden situarse en el espacio de memoria deseado. También se puede informar al enlazador sobre qué partes de la memoria se pueden leer, sobre cuáles está permitido escribir y qué otras se pueden ejecutar. Por ejemplo, en el fichero usado por ORK para producir ejecutables que probar sobre el simulador, la memoria se divide en dos zonas: ROM y RAM. La ROM ocupa 2 Mbytes de memoria y la RAM ocupa 4 Mbytes. La ROM comienza en la dirección cero y su contenido se puede leer y ser ejecutado. El espacio de memoria de la RAM, por su parte, empieza en la dirección 0x2000000 y su contenido puede leerse, ser reescrito o

ejecutarse. Todas las secciones de los programas de ORK destinados al simulador se cargan en la zona RAM de la memoria.

Las secciones que se definen en el fichero de órdenes son las siguientes:

Texto (text) En esta sección se almacena el código ejecutable del programa y los datos de solo lectura. Aunque en el simulador se carga en RAM, esta sección podría cargarse en ROM [1].

Datos (data) En esta sección se encuentran las variables globales del programa inicializas a algún valor. Las variables locales de los procedimientos no se guardan en esta sección porque utilizan la pila.

Datos no inicializados (bss) Esta sección contiene los datos globales no inicializados de la aplicación. Por ejemplo, el espacio dedicado a las pilas de las tareas está englobado dentro de esta sección.

Depuración (stab y stabstring) Si el programa se compila con opciones de depuración, se hacen necesarias estas dos secciones adicionales para almacenar los símbolos del programa. En la práctica, no se cargan en la memoria del simulador.

Al cargar cualquier sección sobre la zona de memoria deseada, también se puede especificar el alineamiento de dicha sección. O sea, es posible decirle al enlazador que use la primera dirección disponible para la sección o que utilice la primera dirección que sea divisible entre dos, entre cuatro o entre ocho. Esto es útil porque las instrucciones deben estar alineadas a palabra (direcciones divisibles por cuatro) y conviene alinear la sección de datos no inicializados a doble palabra [2].

Aparte de definir el mapa de memoria de la aplicación, el fichero de órdenes del enlazador también permite determinar la arquitectura para la cual ha de generarse el ejecutable (SPARC en nuestro caso) y el formato del mismo. A lo largo del proyecto ORK se cambió de formato *a.out* a formato *elf*, debido a que este último es más nuevo y flexible y a que las nuevas versiones del simulador TSIM son compatibles con el formato *elf*. También se usa el formato *srecord* para cargar la aplicación sobre hardware real. Este formato, ideado por Motorola, es una representación ASCII de una imagen binaria. La mayoría de monitores (programas que se instalan en ROM para cargar y depurar las aplicaciones) reconocen el formato *srecord*.

[1] A excepción de la tabla de interrupciones, que se modifica para cambiar los manejadores por defecto (ver 7.3)

[2] La sección de datos no inicializados se borra durante el código de inicio del sistema (ver apartado 7.3) y esto se hace más rápido usando instrucciones de carga doble. Para poder usar estas instrucciones, el espacio de memoria debe estar alineado correctamente).

También en este archivo se puede determinar el símbolo de entrada de la aplicación, es decir, el símbolo desde el cual deberá empezar la ejecución del programa. El símbolo en cuestión deberá estar definido en el fichero que contiene el código de inicialización del sistema. También puede declararse directamente en la línea de órdenes, cuando se llama al enlazador.

7.2 El fichero de especificaciones de GCC

Durante este documento hemos estado hablando del programa `gcc` como compilador de C. En realidad, `gcc` es un programa que va llamando a otros que son los que realmente efectúan la compilación en varias pasadas:

El preprocesador de C (`cpp`) Expande macros y directivas de preprocesamiento.

El compilador de C (`cc1`) Traduce el código en C a lenguaje ensamblador de la máquina para la que se configuró. Es el compilador de C propiamente dicho.

El ensamblador (`as`) Traduce un fichero fuente en ensamblador a lenguaje de máquina, dando como resultado un fichero objeto.

El enlazador (`ld`) Recoge los distintos ficheros objeto de los que se compone la aplicación y crea una imagen binaria a partir de ellos.

Cada uno de estos programas necesita que se le pasen opciones y argumentos en la línea de órdenes. Como el programa `gcc` controla el desarrollo de todas las fases de la compilación, ha de saber de algún modo qué opciones tiene que pasar a cada uno de los programas. Esta información se encuentra en el fichero de especificaciones de `gcc`. Para ORK se creó un nuevo fichero de especificaciones, llamado "ork_specs", en el que se detallan las opciones y argumentos concretos que se han de pasar a los programas que realizan las distintas fases de la compilación.

Aunque originalmente se diseñó como un compilador de C, GCC ha pasado a ser una colección de compiladores. De hecho, las siglas de GCC han dejado de significar *GNU C Compiler* (Compilador de C de GNU) para querer decir *GNU Compiler Collection* (Colección de Compiladores de GNU) [3]. Gracias al diseño original de GCC, dividido en partes dependientes de la máquina y partes que no, se le han podido ir añadiendo diferentes compiladores correspondientes a otros tantos lenguajes de programación que utilizan el generador de código de GCC

[3] http://gcc.gnu.org

para producir sus ejecutables. Actualmente existen compiladores de Ada, C, C++, Chill, Fortran, Java, Objective-C y Pascal para GCC. Todos ellos se aprovechan de la inmensa variedad de máquinas a las que se ha portado GCC, pudiendo generar ejecutables para cualquiera de ellas. GNAT es el compilador de Ada de GNU y, por lo tanto, también utiliza el generador de código de GCC. El *frontal* de GNAT `gnat1` es el programa que traduce el código en Ada al formato intermedio que entiende GCC para producir código ensamblador de la máquina para la que se configuró. Al compilar un programa en Ada, la ejecución de `gnat1` equivale a las pasadas que realizan el preprocesador de C y el frontal `cc1` cuando se compila un programa en C. Las pasadas que efectúan el ensamblador y el enlazador siguen siendo necesarias para los programas en Ada. Incluso es necesaria una pasada adicional para controlar el orden de elaboración (muy importante en el lenguaje Ada). Esta labor la efectúa el *binder* de GNAT antes de enviar los ficheros objeto al enlazador.

Como la compilación de un programa Ada conlleva el uso de varias de las utilidades comunes de GCC, las opciones que se pasen a estas utilidades afectarán a la forma de generar los ejecutables. Como ya hemos dicho, estas opciones se controlan a través del fichero de especificaciones de GCC.

A pesar de que las opciones necesarias para compilar programas para ORK no son las que vienen por defecto, el fichero original de especificaciones no se ha borrado. Este fichero contiene muchas opciones que han de conservarse. Para que además puedan usarse las opciones que necesita ORK, hay que decirle al compilador que lea las especificaciones del nuevo fichero creado. Esto se hace desde la línea de órdenes al utilizar `gnatmake`:

```
$ sparc-ork-gnatmake <aplicacion.adb> -largs -k -specs=ork_specs
```

Entre las opciones que se pasan al enlazador (las precedidas por `-largs`) están la que indica el fichero de especificaciones a usar y otra opción: `-k`. Esta opción selecciona los argumentos que se deben pasar al compilar aplicaciones para ORK. Estos argumentos se extraen, precisamente, del fichero de especificaciones seleccionado.

Veamos un extracto del fichero de especificaciones “ork_specs”:

```
*lib:
%{!k: %(old_lib)}
%{k: --start-group -lc -lgcc -lork -lgnarl --end-group}

*startfile:
%{!k: %(old_startfile)} %{k: ada_crt0%O%s}
```

```
*link:
%{!k: %(old_link)} %{k: -dc -dp -N -e start -T commands.ld%s}
```

En este extracto se distinguen tres apartados:

- Bibliotecas (lib)
- Fichero de inicio (startfile)
- Enlazador (link)

Cada uno de los apartados define los argumentos que se utilizarán en la compilación con `gcc`. En cualquiera de los tres apartados, vemos que se repite un mismo patrón: siempre aparece, dos veces por apartado, un signo de porcentaje seguido una expresión entre llaves. La primera expresión de cada apartado comienza por un signo de exclamación seguido de la letra k y dos puntos. La segunda sólo tiene la letra k seguida de los dos puntos. Esto significa que la primera expresión será la que se utilice si no se incluye la opción `-k` en la línea de órdenes y que se utilizará la segunda expresión en el caso de que sí se incluya dicha opción.

Así, la primera expresión de cada apartado obliga a que se utilicen las opciones definidas en el fichero de especificaciones original. Por el contrario, la segunda expresión indica los argumentos que se utilizarán en el caso de que se compile con la opción `-k` en la línea de órdenes y que se utilizará la segunda expresión en el caso de que sí se incluya dicha opción.

Así, la primera expresión de cada apartado obliga a que se utilicen las opciones definidas en el fichero de especificaciones original. Por el contrario, la segunda expresión indica los argumentos que se utilizarán en el caso de que se compile con la opción `-k`, que son los necesarios para construir aplicaciones para ORK. Estos argumentos provocan las siguientes acciones:

1. Incluir los ficheros de apoyo mediante el uso de la biblioteca "libork.a".
2. Señalar al archivo "ada_crt0.o" como el fichero de inicio que debe situarse primero en memoria.
3. Especificar el símbolo de comienzo (`-e start`) y el fichero de órdenes que usará el enlazador (`-T commands.ld`).

Compilando y enlazando con estas opciones, obtenemos un ejecutable perfectamente adaptado a las necesidades de ORK y el simulador de ERC32.

7.3 El fichero de inicio

El código de inicialización de ORK se encuentra en un fichero llamado “ada_crt0.o”. Este archivo, programado en lenguaje ensamblador, es el que primero se ejecuta al cargar una aplicación. Es por eso que en él encontramos la etiqueta `start`, un símbolo que se declara en el fichero de órdenes del enlazador [4] como el punto de arranque del programa (ver apartado 7.1).

El fichero hereda este nombre del que se utiliza en los sistemas UNIX para llamar al código de inicialización de un programa: “crt0.o”, que son las siglas de *C Run Time 0* (C Entorno de Ejecución 0). Como ORK está pensado especialmente para escribir programas en Ada (aunque también se pueda programar en C sobre ORK), se decidió cambiar el nombre a “ada_crt0.o”.

Este fichero contiene, aparte del código de inicialización, la tabla de los traps junto con algún manejador de bajo nivel. Precisamente, el símbolo `start` está colocado al comienzo de la tabla de traps. Esto significa que el trap colocado en la base de la tabla es lo primero que ejecuta cualquier aplicación de ORK. Este trap es el trap de *hard reset*, que se ocupa de reinicializar el sistema por completo cada vez que la máquina se enciende de nuevo. Para ello, sigue los pasos que se detallan a continuación:

1. Carga el registro TBR con la dirección inicial de la tabla de traps.

2. Borra el contenido de los registros globales y limpia todas las ventanas de registros.

3. Inicializa el registro de estado (PSR), el de ventana inválida (WIM) y el registro Y.

4. Informa al MEC sobre la cantidad y tipo de memoria disponibles. Los valores utilizados son los declarados en el fichero de órdenes al enlazador (ver apartado 7.1).

5. Borra el contenido de todos los registros de la unidad de coma flotante y, seguidamente, deshabilita la FPU escribiendo un cero sobre el bit EF del registro de estado. Esta última operación

6. Borra toda la sección de memoria de datos sin inicializar. Para ello utiliza los símbolos definidos en el fichero de órdenes del enlazador `bss_start` y `bss_end`.

[4] También se encuentra entre las opciones para el enlazador descritas en el fichero de especificaciones de `gcc`.

7. Hace que el puntero de pila apunte a la dirección de memoria en la que se encuentra la pila de la tarea de entorno. La tarea de entorno en ORK es aquélla que se ejecuta cuando no hay ninguna otra que deba procesarse en ese momento.

8. Llama al procedimiento principal del programa. A partir de este momento, la aplicación toma el control. En caso de que se vuelva de la aplicación, algo que normalmente no debería suceder, el fichero de inicio para el procesador.

Los manejadores de trap de bajo nivel que se implementan en el fichero de inicio del sistema son:

- Los traps de desbordamiento de ventanas: *overflow* y *underflow*.
- El trap que guarda todas las ventanas de registros (*flush windows*). Es usado por el compilador durante el tratamiento de excepciones.
- El trap de FPU deshabilitada. Este trap es el que se utiliza para detectar qué tareas usan la FPU y así poder optimizar los cambios de contexto y el tratamiento de interrupciones.

La tabla de traps, colocada al principio del fichero de inicio y de la memoria del sistema, contiene las cuatro primeras instrucciones de todos los manejadores de trap. En total son 512 entradas, de las cuales la mitad se reservan para traps hardware y la otra mitad para traps software (aquéllos que se producen con una instrucción de trap).

Tantas entradas dan como resultado un tamaño de la tabla de traps no despreciable:

$$512 entradas \times 4 \frac{inst}{entrada} \times 4 \frac{bytes}{inst} = 4096 bytes = 4Kb$$

Sin embargo, una tabla de traps tan grande aumenta de forma considerable la huella mínima del sistema en memoria. Para sistemas embarcados con restricciones fuertes de memoria (como es el ERC32), esto supone un derroche importante. Más aún cuando la gran mayoría de las entradas de la tabla no se utilizan nunca. A partir de la versión 2.2 de ORK, el fichero de inicio ha sido ligeramente modificado para aprovechar en lo posible el espacio no útil de la tabla de traps.

El problema es que cada una de las entradas de la tabla de traps ocupa una dirección de memoria determinada por el *tipo de trap (tt)* al que corresponde la entrada (ver apartado 2.2.5). Esto provoca que el espacio de la tabla esté fragmentado en traps que sí se utilizan y traps que no, haciendo más difícil su aprovechamiento.

A pesar de lo dicho, la parte final de la tabla de traps está ocupada enteramente por entradas correspondientes a traps software que no se utilizan en ORK. Mediante una directiva condicional de compilación, todas estas entradas han sido eliminadas de la tabla. El espacio que ocupaban pasa a estar disponible para el código de la aplicación. En caso de que sea necesario, las entradas pueden recuperarse fácilmente cambiando el valor de la expresión empleada en la directiva condicional.

Además de la parte final de la tabla, se ha aprovechado otro gran hueco que se situaba entre la última de las entradas correspondientes a un trap hardware y la primera de los traps software. Este espacio era lo suficientemente grande como para albergar el código de inicio y los manejadores de bajo nivel que se implementan en este mismo fichero. Así pues, el código de dichos manejadores se ha colocado en este espacio interior de la tabla de traps. El resultado se puede ver en la figura 7.1.

Esta diposición permite un mejor aprovechamiento de la memoria disponible. Sin embargo, los programas de monitorización existentes (uno de los cuales ha sido adaptado a ORK) esperan que los cuatro primeros *kilobytes* de la aplicación estén ocupados por la tabla de traps. Este programa *monitor* sirve para cargar y depurar las aplicaciones sobre el hardware real (la placa de ERC32). Para ello, necesita ciertos espacios de la tabla de traps donde instalar algunos manejadores propios. Antes de adoptar definitivamente la segunda configuración de memoria, será necesario modificar el programa monitor para que éste también la pueda aceptar sin problemas.

7.4 Las Rutinas de Apoyo en C

El compilador GCC está dividido en partes que no dependen de la máquina sobre la que se ejecute y otras que sí. Cuando se porta el compilador GCC a una arquitectura embarcada, hay que proporcionarle una serie de rutinas estándar de bajo nivel [17]. Estas rutinas permiten a los programas interaccionar con el sistema. La biblioteca C que se ha utilizado para el proyecto ORK es la implementación conocida como **newlib**. El soporte de bajo nivel para **newlib** lo proporciona **libgloss**, una biblioteca dependiente del hardware que se ocupa básicamente del código de inicio y de las funciones de entrada/salida. Desafortunadamente, no existe una implementación de **libgloss** para ERC32.

Tal inconveniente no ha supuesto un problema demasiado grave. Es más, el hecho de tener que escribir las propias rutinas de bajo nivel para el ERC32, nos ha permitido tener un mayor control y conocimiento de las mismas (ver también 7.3). La mayoría de estas rutinas, por ejemplo, no son necesarias para un sistema embarcado o tienen una funcionalidad limitada al estar relacionadas con el sistema

de ficheros o el control de procesos. Normalmente, estos subsistemas no se implementan en sistemas embarcados de pequeño tamaño (como es ORK).

Las rutinas de apoyo se han escrito en lenguaje C. El uso de Ada habría sido excesivo en este caso, puesto que la mayoría de las funciones, como ya hemos dicho, son simplemente esqueletos que no implementan ninguna funcioalidad. Además, GNAT espera que estas rutinas estén implementadas en lenguaje C, que es lo habitual, ya que las importa a través del interfaz C de Ada.

El compilador reclama 21 rutinas de apoyo para funciones de bajo nivel (si los símbolos de las funciones no se declaran, la biblioteca **newlib** no puede compilarse). De estas 21 funciones, en ORK sólo tres han sido implementadas para . Las otras 18 simplemente devuelven un código de error (distinto según cada caso) y terminan. Por ejemplo, al ser llamada, la función *open* declara un error de entrada/salida y devuelve -1 (valor típico de las funciones C para indicar una situación errónea).

Las tres funciones que sí se han implementado son *read*, *write* y *sbrk*. Las funciones de lectura y escritura (*read* y *write*) han sido diseñadas para utilizar el puerto serie del MEC (el controlador de memoria y periféricos del ERC32) como simulación de la entrada y salida por pantalla. Se hizo así porque el emulador de ERC32 empleado para las pruebas de ORK (el TSIM) conecta el canal A del UART simulado a la salida y entrada estándar del sistema en el que corre, permitiendo el uso del teclado para introducir datos y la visualización por pantalla de los caracteres que se envían por el puerto serie.

La función *sbrk* se encarga, por su parte, de dar soporte a las rutinas de reserva de memoria dinámica de C (malloc, calloc, realloc). Esta rutina utiliza los símbolos `heap_start` y `heap_end` que se encuentran definidos en el fichero de órdenes del enlazador (ver 7.1). Estos símbolos marcan, respectivamente, el inicio y el final del espacio de direcciones destinado a la reserva de memoria dinámica en ORK. Lo único que hace la rutina *sbrk* es llevar la cuenta del espacio de memoria consumido, basándose en el número de bytes que se van pidiendo en cada reserva de memoria. Si la cantidad demandada supera la disponible, devuelve un error indicando que no hay memoria suficiente.

Esta función se usa actualmente en ORK pero tenderá a desaparecer. La razón es que las últimas modificaciones al perfil de Ravenscar preveen una restricción en el uso de la memoria dinámica. El sistema no podrá gestionar implícitamente el espacio de direcciones destinado a la reserva de memoria dinámica. Esto quiere decir que un sistema conforme con el perfil de Ravenscar no puede hacer reservas de memoria dinámica por sí mismo (ni siquiera durante la creación de objetos), sino que debe haber una petición explícita del programador. En caso de que los necesite, el usuario deberá proporcionar él mismo los mecanismos de gestión de memoria dinámica. Como normalmente estos sistemas estarán dedicados a aplicaciones de tiempo real crítico, los tiempos de ejecución de las funciones de reserva

de memoria dinámica deberán ser predecibles. Con los mecanismos convencionales, la fragmentación del espacio de memoria dinámica puede llevar a tiempos de reserva no predecibles.

Todos los ficheros de apoyo han sido empaquetados en una biblioteca mediante el programa `ar`. La utilidad *ar* permite almacenar varios ficheros objetos en uno solo, que recibe el nombre de biblioteca (*library*). Además, *ar* incluye opciones para gestionar los ficheros objetos que componen la biblioteca: borrar un fichero, actualizar un fichero, añadir un fichero, etc. La biblioteca creada para ORK recibe el nombre de "libork.a". Gracias a ella, las rutinas de apoyo se añaden fácilmente al proceso de compilación de cualquier programa. Basta con añadir la biblioteca "libork.a" a la línea de órdenes para que entren a formar parte de los ficheros objeto a enlazar (ver apartado 7.2).

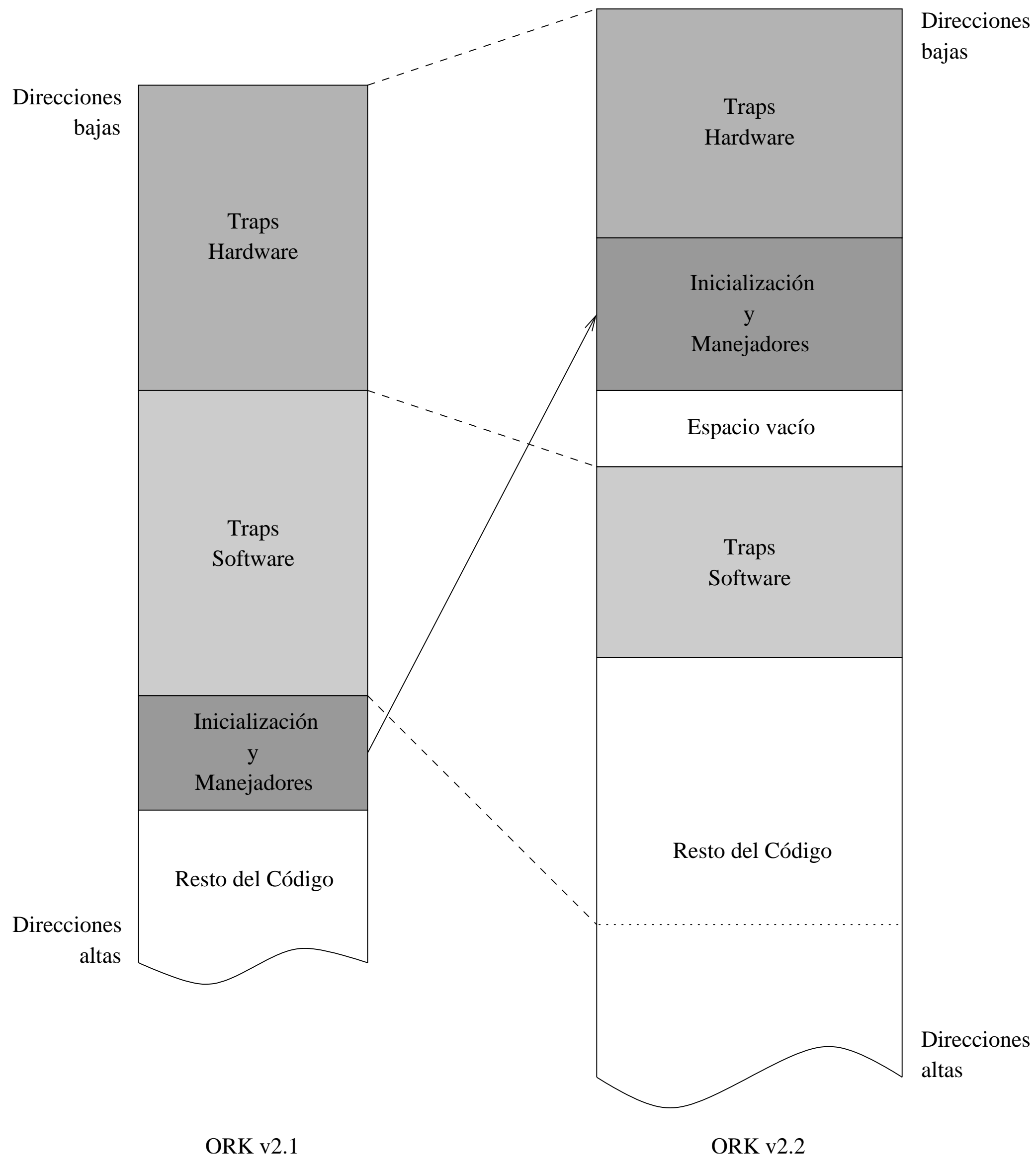


Figura 7.1: Aprovechamiento de la memoria no útil de la tabla de traps.

Apéndice A

Presupuesto

Como ya se dijo en la introducción, este proyecto forma parte de otro mayor que lo engloba y completa. El presupuesto que aquí se expone refleja exclusivamente los costes imputables a la parte del proyecto conjunto sobre la que trata este documento. Habiendo dejado claro este punto, el coste total del proyecto se desglosa en los siguientes conceptos:

- Coste de personal
- Coste de material
- Costes generales y beneficio industrial
- Honorarios

A.1 Presupuesto de Ejecución Material

El presupuesto de ejecución material comprende los costes que se derivan de la mano de obra y de los recursos materiales empleados.

A.1.1 Coste de Personal

La duración del proyecto se extendió desde Noviembre de 1999 hasta Junio de 2001, ambos meses incluídos. Se necesitaron dos meses suplementarios para la redacción y mecanografiado de la presente memoria, por lo que el tiempo total consumido en la ejecución del proyecto fue de 10 meses.

Responsable	Tiempo	Coste por hora	Importe
Ingeniero de Telecomunicación	1.600 horas	4.000	6.400.000
Auxiliar Administrativo	300 horas	1.500	450.000
Total costes de personal			6.850.000

A.1.2 Coste de Material

El coste del material se obtiene sumando el valor de los recursos materiales enumerados en el apartado 1.6. Sólo incluye el valor de los objetos recogidos en la lista de recursos físicos. Los recursos software empleados en el proyecto se han obtenido gratuitamente gracias a que todos los programas usados son libres o de libre distribución.

Material	Importe
Ordenador personal tipo PC	250.000
Tarjeta de red Ethernet	5.500
Libros de consulta	28.000
Material de oficina	5.000
Software y licencias	0
Total costes de material	288.500

A.1.3 Coste Total de Ejecución Material

Se obtiene sumando los costes de personal y de material.

Concepto	Importe
Coste total de personal	6.850.000
Coste total del material	288.500
Total coste de ejecución material	7.138.500

A.2 Gastos Generales y Beneficio Industrial

Incluye los gastos derivados del uso de infraestructura e instalaciones donde llevar a cabo los trabajos, amortizaciones, gastos financieros, etc. Para el calculo de los gastos generales se establece un recargo del 16% sobre el presupuesto de ejecución material, mientras que el porcentaje establecido para el beneficio industrial es del 6%.

Concepto	Importe
Gastos Generales	1.142.160
Beneficio Industrial	428.310
Total gastos generales más beneficio industrial	1.570.470

A.3 Presupuesto de Ejecución por Contrata

Es el resultado de sumar los gastos generales y el beneficio industrial al presupuesto de ejecución material.

Concepto	Importe
Presupuesto de ejecucion material	7.138.500
Gastos Generales y Beneficio Industrial	1.570.470
Presupuesto de ejecución por contrata	8.708.970

A.4 Honorarios

Los honorarios se calculan en base a un recargo del 7% sobre el presupuesto de ejecución material. Según el coste del proyecto, se aplican unos coeficientes de reducción a los honorarios siguiendo el baremo especificado por el Colegio Oficial de Ingenieros de Telecomunicación.

Baremo	Cantidad	Coeficiente	Importe
Hasta 1 millon	1.000.000	1	70.000
De 1 a 5 millones	5.000.000	0.8	280.000
De 5 a 15 millones	1.138.500	0.6	47.817
Total honorarios			397.817

A.5 Presupuesto Total

El presupuesto total comprende todos los costes del proyecto. Se calcula sumando los honorarios al presupuesto de ejecución por contrata.

Concepto	Importe
Presupuesto de ejecución por contrata	8.708.970
Honorarios	397.817
Total	9.106.787

El importe total del proyecto asciende a la cantidad de **NUEVE MILLONES CIENTO SEIS MIL SETECIENTAS OCHENTA Y SIETE** pesetas.

Madrid, 24 de Julio de 2001,

Rodrigo García García,
Ingeniero de Telecomunicación.

Bibliografía

[1] *Ada 95 Reference Manual: Language and Standard Libraries. International Standard ANSI/ISO/IEC-8652:1995*, 1995. Available from Springer-Verlag, LNCS no. 1246.

[2] Alejandro Alonso and Juan Antonio de la Puente. Implementation of mode changes with the ravenscar profile. In *10th International Real-Time Ada Workshop*, volume XXI. Ada Letters, Marzo 2001.

[3] American National Standard Institute, Inc. *Military Standard Ada Programming Language*, January 1983. ANSI/MIL-STD-1815A.

[4] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. Deadline monotonic scheduling theory. In Luc Boullart and Juan A. de la Puente, editors, *Real-Time Programming 1992. Proceedings of the IFAC/IFIP Workshop*. Pergamon Press, 1992.

[5] T.P. Baker and Offer Pazy. A unified priority-based kernel for Ada. Technical report, ACM SIGAda, Ada Run-Time Environment Working Group, March 1995.

[6] Alan Burns. The Ravenscar profile. *Ada Letters*, XIX(4):49–52, 1999.

[7] Alan Burns, Brian Dobbing, and George Romanski. The Ravenscar profile for high integrity real-time programs. In Lars Asplund, editor, *Reliable Software Technologies — Ada-Europe'98*, number 1411 in LNCS. Springer-Verlag, 1998.

[8] Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, 2 edition, 1996.

[9] Juan A. de la Puente, José F. Ruiz, and Jesús M. González-Barahona. Real-time programming with GNAT: Specialised kernels versus POSIX threads. *Ada Letters*, XIX(2):73–77, 1999. Proceedings of the 9th International Real-Time Ada Workshop.

[10] Juan A. de la Puente, José F. Ruiz, and Juan Zamorano. An open Ravenscar real-time kernel for GNAT. In Hubert B. Keller and Erhard Ploedereder, editors, *Reliable Software Technologies — Ada-Europe 2000*, number 1845 in LNCS, pages 5–15. Springer-Verlag, 2000.

[11] Juan A. de la Puente, José F. Ruiz, Juan Zamorano, Rodrigo García, and Ramón Fernández-Marina. ORK: An open source real-time kernel for on-board software systems. In *DASIA 2000 - Data Systems in Aerospace*, Montreal, Canada, May 2000.

[12] Juan A. de la Puente, Juan Zamorano, José F. Ruiz, Ramón Fernández, and Rodrigo García. The design and implementation of the Open Ravenscar Kernel. In Michael González-Harbour, editor, *10th International Real-Time Ada Workshop*, volume XXI, Las Navas del Marqués, Ávila, Spain, Marzo 2001. Ada Letters.

[13] E.W. Giering and T.P. Baker. The GNU Ada Runtime Library (GNARL): Design and implementation. In *Proceedings of the Washington Ada Symposium*, 1994.

[14] ISO. *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD-1815A-1983; ISO/8652-1987.

[15] C. Douglass Locke. Software architectures for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[16] Richard P. Paul. *Sparc Architecture, Assembly Language Programming, and C*. Prentice Hall, first edition, 1993.

[17] Rob Savoye. *Embed with GNU*. Cygnus, 1995. Rough Draft.

[18] Edmond Schonberg and Bernard Banner. The GNAT project: A GNU Ada-9x compiler. In *Tri-Ada 94 Proceedings*. ACM Press, 1994.

[19] TEMIC. *SPARC V7 Instruction Set Manual*, 1996.

[20] TEMIC. *TSC691E Integer Unit User s Manual for Embedded Real Time 32 bit Computer (ERC32)*, 1996.

[21] TEMIC. *TSC692E Floating Point Unit User s Manual for Embedded Real Time 32 bit Computer (ERC32)*, 1996.

[22] Temic/Matra Marconi Space. *SPARC RT Memory Controller (MEC) User's Manual*, April 1997.

[23] Tullio Vardanega and Gert Caspersen. Using the Ravenscar Profile for space applications: The OBOSS case. In Michael González-Harbour, editor, *Proceedings of the 10th International Real-Time Ada Workshop*, 2001. To appear in Ada Letters.

[24] Tullio Vardanega, Rodrigo García, and Juan A. de la Puente. An application case for ravenscar technology: Porting oboss to gnat/ork. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies. Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[25] Juan Zamorano, Alejandro Alonso, and Juan Antonio de la Puente. Building safety critical real-time systems with reusable cyclic executives. *Control Engineering Practice*, 5(7), July 1997.

[26] Juan Zamorano, José F. Ruiz, and Juan A. de la Puente. Implementing ada.real_time.clock and absolute delays in real-time kernels. In D. Craeynest and A. Strohmeier, editors, *Reliable Software Technologies. Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.