

UNIVERSIDAD POLITÉCNICA DE MADRID
FACULTAD DE INFORMÁTICA



**Transporte del Sistema Operativo de Tiempo
Real ORK a la Plataforma i386-PC y
Aplicación de Demostración**

Autor: Pedro Palomo Pérez
Tutor: Juan Zamorano Flores

**Trabajo Fin de Carrera
Marzo 2002**

Título:

Transporte del Sistema Operativo de Tiempo Real ORK a la Plataforma i386-PC y Aplicación de Demostración

Resumen:

El proyecto consiste en portar el entorno de desarrollo de sistemas operativo de tiempo real ORK (*Open Ravenscar Real Time Kernel*) para aplicaciones espaciales, a la plataforma i386-PC, para su uso en entornos educativos e industriales. Así como el diseño de una librería en Ada95 para el control de un brazo robótico con cuatro grados de libertad sobre ORK-i386 y bajo el perfil de Ravenscar.

Agradecimientos

En este apartado quisiera agradecer el apoyo prestado por todo el mundo que me ha rodeado durante aproximadamente el año que ha llevado desarrollar este proyecto, el cual se ha alargado más de lo esperado debido a diversos motivos.

Especialmente me gustaría agradecer el apoyo prestado por mi familia durante este último año, sin el cual no hubiera sido posible su finalización.

También me gustaría agradecer al Departamento de Arquitectura y Tecnología de Sistemas Informáticos de la Facultad de Informática los recursos que me han sido facilitados para poder llevar a cabo el proyecto.

Sobre todo me gustaría agradecer el apoyo de Juan Zamorano que siempre a estado dispuesto a echarme una mano con cualquier problema que surgiera en el proyecto, y muy especialmente por la ayuda prestada a la hora de incorporarme al mundo laboral proporcionándome los contactos necesarios en el apasionante mundo de los Sistemas de Tiempo Real.

Índice General

1. Contexto del proyecto	1
1.1. El sistema operativo de tiempo real ORK	3
1.2. Objetivos	3
1.3. Estructura del documento	4
2. Sistemas de tiempo real crítico	7
2.1. Sistemas de tiempo real	7
2.2. Sistemas de tiempo real críticos	8
2.3. Ada95 como lenguaje de desarrollo de sistemas críticos	8
2.4. El perfil de Ravenscar	9
2.4.1. Definiciones	11
2.4.1.1. Aspectos prohibidos	11
2.4.1.2. Aspectos soportados	12
2.4.1.3. Semántica dinámica	13
2.4.1.4. Identificadores para las restricciones	13
2.4.1.5. Restricciones adicionales	14
3. Las restricciones del perfil de Ravenscar en GNAT	15
3.1. El pragma Ravenscar	15
3.2. Otros pragmas	18
3.2.1. Pragma Restrictions	18
3.2.2. Pragmas relativos a la planificación	18

3.3. El fichero gant.adc para el perfil Ravenscar.....	19
4. Arquitectura de GNAT/ORK	21
4.1. Introducción.....	21
4.2. GNARL (GNU Ada Runtime Library)	22
4.3. Implementación del perfil Ravenscar en GNAT	22
4.3.1. Adaptación del GNARL a ORK.....	23
4.3.2. Uso del kernel con programas C.....	24
4.4. Diseño del Open Ravenscar Real-Time Kernel	25
4.4.1. Arquitectura	25
4.4.2. Gestión de threads	26
4.4.3. Gestión de tiempo.....	27
4.4.4. Gestión de almacenamiento.....	27
4.4.5. Manejo de interrupciones	28
4.4.6. Otros características.....	28
5. Introducción a la familia ix86 y la arquitectura del PC	29
5.1. Introducción.....	29
5.2. Orígenes.....	29
5.3. La familia básica	30
5.4. Modos de funcionamiento	31
5.4.1. El Modo Real.....	31
5.4.1.1. El modelo de programación.....	32
5.4.1.2. El direccionamiento de la memoria	33
5.4.1.3. Los puertos de entrada/salida (E/S)	35
5.4.2. El Modo Protegido	35
5.4.2.1. Soporte hardware del modo protegido	36
5.4.2.2. El acceso a memoria	38
5.4.2.3. Las interrupciones y excepciones	41
5.4.2.4. Otros aspectos.....	42
5.5. La arquitectura del PC	43

5.5.1.	Compatibilidad	44
5.5.2.	Diagrama general de bloques del PC.....	44
6.	Adaptación de ORK a la plataforma i386-PC	49
6.1.	Introducción.....	49
6.2.	Estructura de directorios de la distribución de ORK-i386	50
6.3.	Modificación de los paquetes de ORK.....	51
6.3.1.	Ada.Interrupts.Names.....	52
6.3.2.	System.OS_Interface	54
6.3.3.	System.Interrupts.....	55
6.3.4.	Kernel.Interrupts.....	55
6.3.5.	Kernel.CPU_Primitives	56
6.3.6.	Kernel.Time	59
6.3.7.	Kernel.Memory	59
6.3.8.	Kernel.Peripherals	60
6.3.9.	Kernel.Peripherals.Registers	66
6.3.10.	Kernel.Serial_Output.....	69
6.3.11.	Kernel.Parameters.....	69
6.4.	Funcionamiento e implementación de ORK-i386	72
6.4.1.	Inicialización del entorno	72
6.4.2.	Gestión de interrupciones y traps	75
6.4.2.1.	El sistema de interrupciones de la familia ix86	75
6.4.2.2.	Hardware de interrupciones.....	79
6.4.2.3.	Rutina de tratamiento de interrupciones	80
6.4.2.4.	Tratamiento de las interrupciones en ORK-i386	85
6.4.2.5.	Rutina de tratamiento de excepciones	86
6.4.2.6.	Tratamiento de las excepciones en ORK-i386	88
6.4.3.	Soporte de entrada/salida.....	89
6.4.4.	Gestión de tiempos	91
6.4.4.1.	Implementación del soporte de tiempo real.....	91

6.4.4.2. Implementación del soporte de tiempo real en para el procesador Pentium	93
6.4.5. Soporte para las tarea.....	96
6.4.5.1. Gestión de prioridades	98
6.4.6. Soporte de depuración con GDB	99
6.4.7. Montaje y arranque del sistema	101
6.4.8. Pruebas del sistema.....	105
6.4.8.1.Requisitos temporales de las tareas	106
6.4.9. Otros aspectos.....	109
7. Desarrollo de una aplicación sobre ORK-i386	111
7.1. Introducción.....	111
7.2. Entorno hardware	111
7.2.1. Sistema físico: Brazo robótico.....	113
7.2.2. Entorno de ejecución: Target.....	116
7.2.3. Adecuación de señales: Tarjeta de interface.....	116
7.2.4. Entorno de desarrollo: Host.....	126
7.3. Diseño del software	126
7.3.1. Gráfico de dispositivos externos.....	126
7.3.2. Diseño de la arquitectura lógica	127
7.3.2.1. Brazo.....	129
7.3.2.2. Monitor_Posicion	132
7.3.2.3. Interfaz_Robot.....	134
7.3.2.4. Parametros_Brazo.....	135
7.3.2.5. Motor	137
7.3.2.6. Estado	139
7.3.2.7. Velocidad.....	141
7.3.2.8. Gestion_Movimiento	141
7.3.2.9. Cola.....	145
7.3.2.10. Posicion	146

7.3.3. Diagrama de procesos.....	149
7.3.4. Diagrama de módulos.....	150
7.4. Conclusiones	151
8. Conclusiones y posibles mejoras	153
8.1. Conclusiones	153
8.2. Mejoras a ORK-i386	154
A. Creación de un disquete de arranque con GRUB	157
B. Instalación y uso del entorno ORK-i386	161
B.1. Instalación y estructura de directorios.....	161
B.1.1. Obtener ORK-i386	161
B.1.2. Instalación de ORK-i386	161
B.1.2.1. Instalación de los binarios de ORK-i386.....	161
B.1.2.2. Instalación de los fuentes de ORK-i386	162
B.1.3. Compilación del kernel.....	162
B.2. Desarrollo de software en el entorno ORK-i386.....	163
B.2.1. Escribir y compilar programas Ada95	163
B.2.2. Ejecución y depuración de programas Ada95	164
C. Hojas de características del hardware.	167
C.1. PIC-16F84	167
C.2. Circuito L993b	172
Bibliografía	179

Capítulo 1

Contexto del proyecto

En esta introducción se intentará situar la posición del presente trabajo dentro del contexto de los procesos de desarrollo de software embarcado en la ESA (*European Space Agency*), y la evolución que han empezado a sufrir estos recientemente.

Dentro de la ESA hace unos años se empezó a ver la necesidad de mejorar el proceso de desarrollo de software embarcado, en su mayor parte dedicado al control de satélites artificiales, la nueva generación de sistemas embarcados debería desarrollarse bajo el paradigma de lo que se conoce como “*más barato, más rápido y mejor*”, este objetivo es necesario debido a las siguientes demandas:

- Permitir la reducción de los actuales tiempos de desarrollo de un satélite de 3-4 años a 18-24 meses.
- Llevar a cabo el incremento de la autonomía, responsabilidad y productos de misión de las capacidades embarcadas.
- Dar soporte a la tendencia actual de incrementar los sistemas controlados por software lo cual se refleja en el incremento de 15000-20000 líneas de código a las 50000-60000 líneas actuales.

Como resultado de esta demanda la productividad del proceso deberá ser al menos el cuádruple para que el doble de software sea desarrollado en la mitad de tiempo. A la vista de los anteriores requisitos se encontraron dos puntos especialmente débiles que provocaban el cuello de botella en los actuales desarrollos de software, como son.

- Actualmente y siguiendo los estándares de desarrollo de la ESA, el esfuerzo de verificación es excesivo ocupando en algunos casos el 60% del tiempo de desarrollo del sistema, donde la mayor parte del tiempo la ocupa la ejecución de test dinámicos.
- La arquitectura del software en los sistemas convencionales embarcados es típicamente centralizada alrededor de un rígido e inflexible ejecutivo cíclico. La

naturaleza monolítica de esta arquitectura la hace poco tolerante a modificaciones.

Como solución a esta problemática y poder satisfacer los puntos anteriormente expuestos se propuso desarrollar nuevo soporte para optimizar el desarrollo de software embarcado, centrado en procesos de verificación, análisis de tiempo real estático como sustituto de los test dinámicos y utilización de métodos de planificación expansiva con prioridades fijas. Estos objetivos se llevan a cabo de dos maneras básicamente.

- Mejorar el modelo computacional es, decir lograr mayor expresividad en la fase de definición del diseño para capturar mejor los aspectos relativos a tiempo real tales como atributos de tiempo real y características de ejecución de los componentes sobre los cuales el sistema es construido.
- Crear un marco de trabajo que englobe todas las fases del diseño de sistemas de tiempo real (requisitos, análisis, diseño e implementación), dentro del cual se pueda iterar.

Para lograr todo esto es necesario una base tecnológica, partiendo de Ada83 como lenguaje de implementación y de HOOD 3.1 como metodología de diseño se ha construido un conjunto de herramientas de trabajo, compuestas por los siguientes componentes:

- Se crea el nuevo método de diseño llamado HRT-HOOD basado en HOOD pero que añade aspectos de tiempo real.
- Se crea HOORA como método de análisis de requisitos, basado en el lenguaje UML y cuyo producto de salida se convierte en la entrada que recibe la fase de diseño.
- Métodos de análisis estático de la planificabilidad, basados en tareas con prioridades fijas, planificación expansiva y metodología de asignación de prioridades DMS (*Deadline Monotonic Scheduling*).
- Empleo de Ada95 como lenguaje de implementación.
- Se define un microprocesador estándar de la ESA como es ERC-32 (32-bit Embedded Real-Time Computing Core), basado en una arquitectura SPARC V7. Actualmente ya existe una versión más moderna del microprocesador denominado LEON.
- Utilización del Perfil de Ravenscar que define el subconjunto seguro de los aspectos relacionados con las tareas de Ada.
- GNAT/ORK como entorno de desarrollo cruzado que de soporte a las tareas de Ada95.

Como se ha podido ver en el punto anterior el proyecto ORK se sitúa dentro las herramientas que dan soporte tecnológico al nuevo proceso de desarrollo de software, concretamente siendo el sistema operativo que de soporte a las tareas de Ada95 y al perfil Ravenscar y funcionando sobre el microprocesador estándar de la agencia ERC-32. Con esto ya se tiene un marco de trabajo completo y de libre distribución accesible para que profesionales e investigadores puedan desarrollar y adquirir experiencia en las

nuevas prácticas, quizás el único escollo que quede por salvar es el acceso al hardware ya que el microprocesador ERC-32 tiene un coste elevado dado que está preparado especialmente para soportar las condiciones adversas del espacio. Es aquí donde surge este proyecto de la necesidad de portar el entorno de desarrollo a una arquitectura más estándar, barata y de fácil acceso que el microprocesador ERC-32, como es el caso de la arquitectura i386

1.1 El sistema operativo de tiempo real ORK.

El sistema operativo ORK (*Open Ravenscar Real-Time Kernel*) está diseñado para realizar sistemas de tiempo real críticos para sistemas embarcados en naves espaciales. Una de las características importantes de ORK es que está diseñado de forma que los sistemas desarrollados con este núcleo puedan someterse a los complejos procesos de certificación necesarios para las aplicaciones espaciales embarcadas.

Los sistemas informáticos embarcados en satélites y otras naves espaciales tienen unos requisitos de seguridad muy críticos. La creciente dependencia de los computadores para realizar funciones de navegación, adquisición de datos, comunicaciones, control de dispositivos y otras en sistemas espaciales hace que un fallo de hardware o software comprometa seriamente el éxito de la misión. Fallos como los del cohete Ariane 5 o las últimas misiones de la NASA a Marte ponen de manifiesto la necesidad de asegurar la calidad del software embarcado en naves espaciales.

Esto hace que el diseño de los sistemas informáticos para el espacio sea muy conservador. Normalmente se utilizan componentes hardware resistentes a la radiación, que además de ser mucho más costosos que los convencionales, tienen un ciclo de desarrollo mucho más prolongado. La cantidad de memoria principal disponible suele ser limitada, debido a la necesidad de limitar el consumo de energía eléctrica, el peso y el volumen de los sistemas embarcados, y normalmente no se dispone de ningún tipo de memoria secundaria.

En cuanto al software, normalmente se trata de asegurar la ausencia de fallos de fallos mediante técnicas preventivas, basadas sobre todo en el empleo de procesos de desarrollo adecuados con métodos de garantía de calidad bien definidos. En la mayoría de los casos se somete el sistema –hardware y software- a un proceso de certificación, realizado por un equipo independiente de los desarrolladores, que suele ser muy costoso y complejo. El objetivo del proceso de certificación es obtener una evidencia razonable de que el sistema está libre de fallos, para lo cual se utilizan técnicas como inspecciones de software, verificación formal, pruebas y análisis estáticos, según el grado de criticidad de los subsistemas que se analicen.

Estos requisitos han conducido, en general, al empleo de herramientas de software cerradas y costosas, con poco espacio para el software libre. Sin embargo, el

panorama está empezando a cambiar. La calidad y robustez de herramientas como gcc, GNAT y GDB les han abierto las puertas del software crítico.

1.2 Objetivos.

El objetivo fundamental de este trabajo consiste en portar el sistema ORK, y todo su entorno de desarrollo a una arquitectura hardware más estandarizada y general como es la arquitectura i386-PC, y así poder utilizar el sistema en otros dominios de aplicación más heterogéneos, tanto industriales como educativos.

Con el objetivo de demostrar la validez del sistema de compilación cruzada desarrollado, se ha desarrollado una aplicación sobre ORK-i386, consistente en realizar un sistema de control de un brazo robótico, de esta forma se prueba de manera exhaustiva la funcionalidad del sistema ORK-i386, así como la adecuación del perfil de Ravenscar para realizar aplicaciones empotradas y de tiempo real.

ORK junto a todo el entorno de desarrollo compuesto por el sistema de compilación GNAT/ORK, y las versiones adaptadas de GDB Y DDD para ORK han sido desarrolladas por miembros de la Universidad Politécnica de Madrid en colaboración con la Universidad Rey Juan Carlos la empresa CASA y la propia Agencia Espacial Europea. La adaptación del dicho entorno a la arquitectura i386-PC ha sido llevada a cabo por el autor de este trabajo.

1.3 Estructura del documento.

El capítulo 1 del documento describe el entorno en que se utiliza ORK, el cual ha sido construido para el desarrollo de sistemas de tiempo real críticos embarcados en vehículos espaciales.

En capítulo 2 se hace una introducción a los sistemas de tiempo real críticos, así como una breve introducción al lenguaje Ada95, en este capítulo también se describe el perfil de Ravenscar, bajo el cual se deben diseñar las aplicaciones que utilicen el entorno ORK, que se puede considerar como el aspecto más novedoso del sistema.

En el capítulo 3 se describe cómo se utilizan las restricciones del perfil Ravenscar en el entorno de compilación GNAT.

El capítulo 4 muestra de manera simplificada cual es la estructura interna del sistema ORK.

En el capítulo 5 se hace una introducción a la arquitectura ix86-PC, donde se explicarán tanto aspectos históricos como técnicos de esta arquitectura.

En el capítulo 6 se desarrollan los aspectos más significativos de la adaptación del sistema ORK a la plataforma i386-PC.

Por último en el capítulo 7 se describe la aplicación que se ha realizado sobre ORK-i386, para el control de un brazo robótico, siguiendo el Perfil Ravenscar.

Existe además un apéndice A en el que se explica el proceso de construcción de un disquete de arranque a partir de la herramienta GRUB, un apéndice B en el que explica la instalación y el modo de uso del entorno ORK-i386 y en el apéndice C se muestran las hojas de especificación de alguno de los componentes hardware que se han empleado en la construcción del brazo róbóton.

Capítulo 2

Sistemas de tiempo real crítico

2.1 Sistemas de tiempo real.

Los sistemas de tiempo real constituyen un campo de la ingeniería del software en gran expansión, y con una enorme proyección en el futuro.

Una de las características fundamentales de estos sistemas es que tienen una elevada interacción con su entorno, debiendo responder a los estímulos externos produciendo una respuesta sobre el sistema que controlan.

Como en cualquier sistema, que el resultado sea correcto o no depende de la corrección lógica del resultado obtenido. En el caso de los sistemas de tiempo real, además de esta corrección lógica, es fundamental responder dentro de unos márgenes de tiempo determinados previamente establecidos. Alcanzar un resultado lógico pero fuera de tiempo se considera un fallo del sistema.

En la práctica, a veces no es tan estricto el cumplimiento de los plazos, distinguiéndose entre dos tipos de sistemas:

- Críticos. En estos sistemas no se puede permitir ningún fallo en el cumplimiento de los plazos. Superar el plazo una vez, aunque solo sea ligeramente, es un error grave, que puede acabar en una catástrofe. Ejemplos de estos sistemas pueden ser centrales nucleares, sistemas de aviación, guiado de misiles, etc.
- Acríticos. En estos sistemas el cumplimiento de los plazos es importante y deseable, pero nos podemos permitir perder algún plazo de vez en cuando (si el sistema se sobrecarga), llegando a una pequeña, aunque soportable degradación del sistema. En un sistema de adquisición de datos, estará muestreado a una determinada frecuencia, pero es muy posible que podamos permitir que el muestreo se retarde ligeramente en alguna ocasión, sin que el sistema se degrade de forma considerable.

Un subconjunto muy importante de los sistemas de tiempo real lo constituyen los sistemas empotrados. Como ya se ha comentado, el equipo informático encargado del control del sistema interacciona con el entorno para controlar el funcionamiento del sistema. En el caso de los sistemas empotrados, el computador encargado del control del sistema es un componente más del sistema, integrado en el mismo espacio físico.

2.2 Sistemas de tiempo real críticos.

Como se ha explicado en la introducción, los sistemas de tiempo real críticos se suelen someter a un proceso de certificación riguroso, que tiene como objetivo llegar a un grado de confianza razonable en el comportamiento del sistema. Para ello se utilizan distintos métodos, según el grado de criticidad del sistema y los requisitos específicos del tipo de aplicación de que se trate (los sistemas embarcados en naves espaciales se suelen considerar críticos con respecto al cumplimiento de su misión (*mission-critical*)). En general, se considera que sólo se puede conseguir una confianza adecuada en la corrección del software utilizando técnicas de análisis estático del software, además de las pruebas dinámicas habituales.

2.3 Ada95 como lenguaje de desarrollo de sistemas críticos.

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*. Más tarde, el documento se propuso a ISO, que lo adoptó como estándar número 8652 en 1987. 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.



El lenguaje de programación empleado es uno de los elementos clave para la utilización de técnicas de análisis estático. Aunque ningún lenguaje puede garantizar por sí solo la corrección de los programas escritos en él, Ada 95 es uno de los que facilitan en mayor medida la comprobación de los programas. Ada tiene una semántica bien definida y una sintaxis con un grado de redundancia elevado, que ayuda a detectar muchos errores durante la compilación. Los compiladores se suelen someter a un proceso de validación que asegura su conformidad con el estándar, y el lenguaje se ha

empleado con éxito en muchos sistemas críticos. Además, hay un grupo de trabajo ISO, que se ocupa de definir y revisar periódicamente un conjunto de directrices para el uso de Ada en sistemas críticos [37]. Por todos estos motivos, la Agencia Europea del Espacio recomienda el uso de Ada para el software embarcado.

Ada es un lenguaje muy extenso, y no es posible analizar programas escritos en ese lenguaje si se utilizan todas sus posibilidades. Por ello, para programar sistemas críticos se suelen definir subconjuntos seguros del lenguaje, de los cuales el más conocido es probablemente el que forma parte de SPARK, un lenguaje basado en Ada



con anotaciones que facilitan el análisis de los programas. El informe sobre sistemas críticos de la ISO [37] proporciona directrices para seleccionar las partes del lenguaje que se deben utilizar en proyectos concretos, según el grado de integridad deseado y las técnicas de análisis que se necesite utilizar. Algunos de los elementos del lenguaje que se suelen restringir en

sistemas críticos son los tipos acceso, y la memoria dinámica, la herencia, los manejadores de excepciones, y las tareas. Algunos de estos componentes del lenguaje se empiezan a utilizar en sistemas críticos, sin embargo, a medida que se conoce mejor su funcionamiento y se perfeccionan los métodos de análisis.

2.4 El perfil de Ravenscar.

Las tareas concurrentes son uno de los elementos de Ada que se han excluido habitualmente de los sistemas de tiempo real críticos. Tradicionalmente se ha considerado que el uso de tareas o *threads*, en cualquiera de sus formas, hace que la ejecución del programa no sea determinista, y que esto impide analizar de forma adecuada los aspectos temporales de la ejecución de los programas, que son clave en los sistemas de tiempo real. En consecuencia, los sistemas de tiempo real críticos se han diseñado hasta hace poco con un esquema denominado de *planificación cíclica*. En este esquema el núcleo del sistema operativo se sustituye por un ejecutivo cíclico, que es un procedimiento activado por una señal periódica proveniente de un reloj de hardware, que invoca secuencialmente otros procedimientos que contienen el código correspondiente a cada una de las actividades del sistema, en función de un plan de ejecución elaborado de antemano.

En los últimos años, sin embargo, se ha producido un avance considerable en los métodos de análisis temporal de sistemas multiprogramados, basados sobre todo en un modelo de multiprogramación en el que las tarea concurrentes se reparten el uso del procesador mediante un método de planificación basado en prioridades fijas. Este modelo se ha tenido en cuenta al definir las tareas concurrentes Ada y su uso en sistemas de tiempo real, por lo que los métodos de análisis de tiempo de respuesta se pueden aplicar de manera muy directa a muchos de los programas de tiempo real escritos en Ada. Sin embargo, el uso de tareas concurrentes sin ninguna restricción lleva

fácilmente a realizar programas difíciles o imposibles de analizar, y hace necesario, por tanto, definir un subconjunto seguro de la parte concurrente de Ada, igual que se suele hacer con la parte secuencial.

El *8th International Real-Time Ada Workshop* (IRTAW'8) se celebró en 1997 en Ravenscar (Yorkshire, Inglaterra). En esta reunión se definió un subconjunto de la parte concurrente de Ada 95, denominado *perfil de Ravenscar* (posteriormente se definió “RAVENSCAR” como acrónimo de *Reliable Ada Verifiable Executive Needed for Scheduling Critical Applications in Real-Time*), con intención de facilitar el uso de tareas en sistemas de tiempo real críticos. En el perfil se permite utilizar tareas y objetos protegidos en el nivel de biblioteca (es decir, estáticos), una entrada como máximo en objetos protegidos, con una barrera formada por una variable booleana local, y se permite únicamente el método de planificación con prioridades fijas y techo de prioridad para el acceso a los objetos protegidos. Se prohíben las tareas y objetos protegidos dinámicos, y otros elementos del lenguaje que pueden impedir el análisis del comportamiento temporal de los programas. El perfil se revisó en la siguiente reunión (IRTAW'9), y posteriormente se elaboró una versión refundida, que actualmente forma parte de la guía para sistemas críticos de ISO[37].

El perfil de Ravenscar está pensado de tal forma que se puede comprobar si un programa cumple sus restricciones al compilarlo, mediante un uso adecuado del “pragma Restriction” de Ada. De esta forma es relativamente sencillo sustituir el núcleo de ejecución (*run-time system*) de Ada por otro más sencillo y robusto que permita certificar la seguridad de las aplicaciones construidas sobre él. Hay ya una implementación comercial del mismo, denominada *Raven*, que está basada en el compilador *Object Ada* de Aonix. La experiencia inicial de los desarrolladores de *Raven* ha permitido confirmar la utilidad del perfil para la realización de sistemas críticos, y ha aumentado el interés hacia el mismo.

El perfil está basado en un modelo de computación con las siguientes características:

- Un único procesador.
- Un número fijo de tareas.
- Un único suceso de activación por cada tarea. La activación puede ser generada por el paso del tiempo (tareas activadas por tiempo) o por un suceso de otra tarea o del entorno (tareas esporádicas).
- La interacción entre tareas se realiza únicamente a través de datos compartidos con acceso en exclusión mutua.

Este conjunto de características permite la construcción de sistemas con los siguientes tipos de componentes:

- Tareas periódicas.

- Tareas esporádicas dirigidas por programa.
- Tareas esporádicas dirigidas por interrupciones.
- Datos compartidos a través de objetos protegidos (normalmente sin entradas).
- Objetos protegidos para la sincronización de sucesos (con una entrada como máximo por objeto protegido).

Se considera que estos componentes son suficientemente expresivos para implementar sistemas críticos para aplicaciones espaciales sobre un único procesador.

2.4.1 Definiciones.

2.4.1.1 Aspectos prohibidos.

El perfil de Ravenscar define las siguientes restricciones y aspectos prohibidos:

- RP1** Declaración de tipos tarea y objetos a un nivel que no sea nivel de biblioteca. Por lo tanto no hay jerarquías de tareas.
- RP2** Creación dinámica y eliminación sin verificación de objetos protegidos y tareas
- RP3** Reencolamiento.
- RP4** ATC (transferencia asíncrona de control mediante la sentencia `asynchronous_select`).
- RP5** Sentencias abortantes, incluida `Abort_Task` del paquete `Ada.Task_Identification`.
- RP6** Tareas con puntos de entrada.
- RP7** Prioridades dinámicas.
- RP8** Paquete `Calendar`.
- RP9** Retardos relativos.
- RP10** Declaración de tipos y objetos protegidos a un nivel que no sea el nivel de librería.

- RP11** Tipos protegidos con más de una entrada.
- RP12** Entradas protegidas con barreras de otro tipo que no sea una variable booleana declarada dentro del mismo tipo protegido.
- RP13** Llamadas a entradas protegidas desde una llamada ya encolada.
- RP14** Control de tareas asíncrono.
- RP15** Todas la formas de la sentencia **select**.
- RP16** Atributos de tareas definidos por el usuario.

En suma a estas restricciones, las implementaciones pueden hacer la siguiente suposición:

- RP17** Las tareas no terminan.

2.4.1.2 Aspectos soportados.

A pesar de las restricciones anteriores, todavía se soporta un amplio rango de las características de las tareas, tales como:

- RP18** Objetos tarea, con las restricciones anteriores.
- RP19** Objetos protegidos, con las restricciones anteriores.
- RP20** Los pragmas **Atomic** y **Volatile**.
- RP21** Sentencia **Delay until**.
- RP22** Política **Ceiling Locking** y despacho **FIFO_within_priorities**.
- RP23** Atributo **count** para entradas protegidas (pero no dentro de la barrera de una entrada).
- RP24** Identificadores de tareas. Ej: **T'Identity**, **E'Caller**.
- RP25** Control de tareas síncrono.
- RP26** Tipos tareas con discriminante.
- RP27** Paquete **Real_Time**.
- RP28** Procedimientos protegidos como manejadores de interrupciones.

2.4.1.3 Semántica dinámica.

Dos aspectos del perfil requieren de una semántica dinámica para poder ser definidos:

- RP29** Si una llamada a una entrada es hecha desde una entrada que ya ha sido aceptada, es decir que ya tiene una llamada encolada. (el tamaño de la cola sería 2), se activaría la excepción `Program_Error`.

Esto es consistente con el uso de `Program_Error` en la definición de la suspensión síncrona de objetos.

- RP30** Si una tarea intenta terminar, esto se clasifica como un error de límite (hay requisitos de documentación sobre la implementación, que deben definir sus efectos), y la tarea como resultado quedará permanentemente suspendida.

2.4.1.4 Identificadores para las restricciones.

La mayoría de las restricciones del perfil Ravenscar se pueden comprobar en tiempo de compilación de Ada95 mediante identificadores estándar usados en “pragma Restrictions”. Los identificadores aplicables son los siguientes:

RP31 `No_Task_Hierarchy`.

RP32 `No_Abort_Statement`.

RP33 `No_Task_Allocators`.

RP34 `No_Dynamic_Priorities`.

RP35 `No_Asynchronous_Control`.

RP36 `Max_Task_Entries => 0`.

RP37 `Max_Protected_Entries => 1`.

RP38 `Max_Asynchronous_Select_Nesting => 0`.

RP39 `Max_Tasks => N` – definido por la aplicación.

De todas formas estas restricciones no son suficientes para asegurar todas las restricciones del perfil. Los siguientes identificadores de restricciones adicionales han sido propuestos para tal fin:

RP40 Simple_Barrier_Variables.

RP41 Max_Entry_Queue_Depth => 1 – o, en general, N.

RP42 No_Calendar.

RP43 No_Relative_Delay.

RP44 No_Protected_Type_Allocators.

RP45 No_Local_Protected_Objects.

RP46 No_Requeue.

RP47 No_Select_Statements.

RP48 No_Task_Attributes.

RP49 No_Task_Termination.

2.4.1.5 Restricciones adicionales.

Las siguientes restricciones relativas a tareas fueron también propuesta en la IRTAW9.

RP50 Static_Storage_Size.

Otras restricciones que no son parte del perfil mismo, pero que se consideran útiles para mantener la parte secuencial de Ada de forma determinista son:

RP51 No_Exception_Handlers

RP52 No_Standard_Storage_Pools.

RP53 No_IO.

RP52 No_Nested_Finalization.

Hay que tener en cuenta que actualmente solo No_IO y No_Nested_Finalization forman parte del estándar de Ada 95.

Capítulo 3

Las restricciones del perfil de Ravenscar en GNAT

3.1 El pragma Ravenscar.

La mayoría de las restricciones del perfil Ravenscar pueden ser comprobadas en tiempo de compilación usando el conjunto apropiado de identificadores con las “pragma Restrictions”. Sea como sea, no todas las restricciones del perfil Ravenscar pueden ser comprobadas con los identificadores estándar de restricción, y por lo tanto han sido propuestas una serie de identificadores de restricciones adicionales en las reuniones IRTAW8 y IRTAW9 para este propósito. El conjunto completo de restricciones Ravenscar puede verse en el capítulo 2.

Sin embargo la aproximación adoptada en GNAT es diferente: la implementación define un pragma (**pragma Ravenscar**) que se usa para establecer el conjunto completo de restricciones. Las restricciones establecidas por el pragma **Ravenscar** no son exactamente las mismas que las definidas en el perfil. Esto significa que hay algunas pequeñas diferencias entre el pragma **Ravenscar** de GNAT y el perfil de Ravenscar.

La tabla 3.1 enumera las restricciones comprobadas por este pragma y su relación con las restricciones del perfil.

De la tabla se pueden sacar las siguientes conclusiones:

1. Una restricción que forma parte del perfil Ravenscar no esta incluida en el pragma **Ravenscar**:
 - RP39 Max_Tasks => N

Hay que tener en cuenta que esta restricción no se puede incluir en el pragma ya que el valor de N tiene que ser definido por la aplicación. En todo caso

Max_Tasks es un parámetro de restricción estándar, esta restricción puede ser comprobada en tiempo de compilación mediante el pragma **Restriction**.

Tabla 3.1: los pragmas Ravenscar (PR) y las restricciones del perfil Ravenscar (RP).

Código	Restricción	Restricción RP	Comentario
PR1	No_Task_Hierarchy	RP31	
PR2	No_Abort_Statement	RP32	
PR3	No_Task_Allocators	RP33	
PR4	No_Dynamic_Priorities	RP34	
PR5	No_Aynchronous_Control	RP35	
PR6	Max_Task_Entries => 0	RP36	
PR7	Max_Protected_Entries => 1	RP37	
PR8	Max_Async_Select_Nesting => 0	RP38	
—	Max_Tasks => N	RP39	No incluidas en PR
PR10	Boolean_Entry_Barrings	RP40	Identificador diferente
PR11	No_Entry_Queue	RP41	Semántica diferente (Aceptable para N=1)
PR12	No_Calendar	RP42	
PR13	No_Relative_Delay	RP43	
PR14	No_Protected_Type_Allocators	RP44	
PR15	No_Local_Protected_Objects	RP45	
PR16	No_Requeue	RP46	
PR17	No_Select_Statements	RP47	
PR18	No_Task_Attributes	RP48	
PR19	No_Task_Termination	RP49	
PR20	Static_Storage_Size	RP50	
PR21	No_Dynamic_Interrupt	—	No requerido por RP
PR22	No_Terminate_Alternatives	—	Definido por RP47
PR23	Max_Select_Alternatives => 0	—	Definido por RP47

2. Cuatro restricciones las cuales no forman parte del perfil Ravenscar propiamente, pero que se considera útil tomar, no son forzadas por el pragma **Ravenscar**:

- RP51 No_Exception_Handlers
- RP52 No_Standard_Storage_Pools
- RP53 No_IO
- RP54 No_Nested_Finalization

Sin embargo, todas ellas están disponibles en GNAT como identificadores de restricción, y pueden ser forzadas usando el pragma **Restriction**.

3. La restricción,

- PR10 Boolean_Entry_Barrings

tiene un nombre diferente, pero la misma semántica que el identificador propuesto en la definición del perfil Ravenscar:

- RP40 Simple_BARRIER_Variables

Esta diferencia puede causar problemas de portabilidad, pero no impide forzar al compilador a usar la restricción correcta.

4. La restricción,

- PR11 No_Entry_Queue

tiene un nombre diferente, y un semántica ligeramente diferente de la restricción propuesta en la definición del perfil Ravenscar:

- Max_Entry_Queue_Depth => N

Desde N=1 en el perfil de Ravenscar, la restricción en GNAT del pragma **Ravenscar** es aceptable. Además, **Max_Entry_Queue_Depth => N** esta implementado en GNAT como un parámetro de restricción, y por lo tanto puede ser explícitamente seleccionado con el pragma **Restrictions**.

5. Dos de las restricciones forzadas por el pragma **Ravenscar** son redundantes:

- PR22 No_Terminate_Alternatives
- PR23 Max_Select_Alternatives => 0

Ambas restricciones están definidas por RP47. Aunque no provoca ningún perjuicio el tenerlas incluidas en el pragma **Ravenscar**.

6. Una de las restricciones es forzada por el pragma **Ravenscar**, pero no se requiere por la definición del perfil **Ravenscar**:

- PR21 No_Dynamic_Interrups

La definición del perfil de **Ravenscar** no dice nada acerca de los manejadores de interrupciones dinámicas, pero este es un aspecto que no se utiliza en sistemas críticos.

La conclusión es que, aunque hay algunas diferencias entre las restricciones impuestas por el pragma **Ravenscar** y la definición por parte de IRTAW del perfil **Ravenscar**, el pragma puede ser usado en su actual forma, junto con otros pragmas de configuración (ver más adelante), para forzar el perfil **Ravenscar** a los programas Ada compilados con GNAT.

3.2 Otros pragmas.

3.2.1 Pragma Restrictions.

Algunas de las restricciones Ravenscar pueden ser forzadas con identificadores de restricción. Los identificadores de las restricciones Ravenscar que acepta GNAT 3.13 se muestran en la tabla 3.2.

Tabla 3.2: identificadores de restricciones Ravenscar en GNAT.

Código	Restricción GNAT	Restricción RP	Comentario
RI1	No_Task_Hierarchy	RP31	
RI2	No_Abort_Statement	RP32	
RI3	No_Task_Allocators	RP33	
RI4	No_Dynamic_Priorities	RP34	
RI5	No_Asynchronous_Control	RP35	
RI6	Max_Task_Entries => 0	RP36	
RI7	Max_Protected_Entries => 1	RP37	
RI8	Max_Asynchronous_Select_Nesting => 0	RP38	
RI9	Max_Tasks => N	RP39	N depende de la aplicación
RI10	Boolean_Entry_Barriers	RP40	Identificador diferente
RI11	Max_Entry_Queue_Depth => 1 No_Entry_Queue	RP41	Semántica diferente
RI12	No_Calendar	RP42	
RI13	No_Relative_Delay	RP43	
RI14	No_Protected_Type_Allocators	RP44	
RI15	No_Local_Protected_Objects	RP45	
RI16	No_Requeue	RP46	
RI17	No_Select_Statements	RP47	
RI18	No_Task_Attributes	RP48	
RI19	No_Task_Termination	RP49	
RI20	Static_Storage_Size	RP50	
RI21	No_Exception_Handler	RP51	
RI22	No_Standar_Storage_Pools	RP52	
RI23	No_IO	RP53	
RI24	No_Nested_Finalization	RP54	

El conjunto de identificadores de restricción cubre completamente la especificación del perfil Ravenscar y por lo tanto puede ser usado para forzar el perfil en tiempo de compilación.

3.2.2 Pragmas relativo a la planificación.

Los siguientes pragmas estándar deberían ser usados en los programas con perfil Ravenscar:

- pragma Task_Dispatching_Policy (FIFO_Within_Priorities)
- pragma Locking_Policy (Ceiling_Locking)

El pragma `Queuing_Policy` no es necesario ya que el tamaño máximo de una entrada protegida esta fijado a 1 por (RP41).

3.3 El fichero gnat.adc para el perfil Ravenscar.

GNAT permite que los pragmas de configuración se pongan en un fichero aparte, llamador `gnat.adc`. Una plantilla `gnat.adc` para un programa que cumpla el perfil Ravenscar puede ser la siguiente.

```
-- gnat.adc - configuration file template for the Ravenscar profile
pragma Ravenscar;
pragma Restrictions (Max_Tasks => 2);
-- N must be equal to the number of tasks of the application
pragma Restrictions(No_Allocators);

pragma Restrictions(No_IO);
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy          (Ceiling Locking);
```

El máximo número de tareas es dependiente de la aplicación. Hay un límite máximo para este número, el cual depende del kernel que de soporte a la aplicación. En el caso que no ocupa dicho kernel será ORK.

Capítulo 4

Arquitectura de GNAT/ORK

4.1 Introducción.

GNAT es un acrónimo de *GNU NYU Ada Translator*. Se trata de un compilador creado a partir del compilador de GNU para C (GCC) existente. Es un compilador que ha conseguido una elevada difusión, y que ofrece unas altas garantías en cuanto a posibles errores, ya que ha sido exhaustivamente probado en sus diferentes aspectos. Las partes que constituyen el GNAT propiamente dicho, y que son específicas para Ada 95 son el front-end y el sistema de ejecución (*runtime*). Del runtime se hablará posteriormente más en profundidad.

Este compilador utiliza el *back-end* de GCC como generador de código, aprovechando así una de las principales ventajas del GCC: es capaz de producir código de gran calidad para una gran cantidad de máquinas distintas usando distintos sistemas operativos.

El *front-end* proporciona los procedimientos de análisis y síntesis de las construcciones de Ada 95, para obtener código interpretable por el compilador GCC. Para realizar el tránsito de Ada a C se utiliza un Árbol Sintáctico Abstracto (*Abstract Sintactic Tree, AST*) que se va construyendo y expandiendo hasta conseguir el código C equivalente a las construcciones en Ada 95. El programa escrito en Ada se pasa a su código C equivalente, y a partir de aquí se obtiene el código objeto compilando compilando este código ya en C con el *back-end* de GCC.

El *runtime* de Ada, llamado GNARL (*GNU Ada Runtime Library*), se encarga de todo lo relacionado, con la gestión de tareas de Ada (*task*). El *runtime* mantiene las estructuras de datos necesarias para manejar, planificar y sincronizar las actividades relacionadas con las tareas. Como ya se ha dicho antes el *runtime* es exclusivo y específico de GNAT. El desarrollo de GNARL ha sido el fruto del esfuerzo y cooperación de la *Florida State University* a través de su proyecto *POSIX/Ada Real-Time (PART)* y el equipo de desarrollo de GNAT en la *New York University*.

4.2 GNARL (GNU Ada Runtime Library).

El soporte de tareas Ada está implementado en GNAT mediante una “Runtime Library” o entorno de ejecución llamado GNARL (GNU Ada Runtime Library). La parte del GNARL que es dependiente de una determinada máquina y sistema operativo se conoce como GNULL (GNU Low-level Library), y su interfaz con la parte independiente de la plataforma se llama GNULLI (GNULL Interface). La mayoría de las implementaciones de GNARL están construidas sobre la capa de threads POSIX, la cual a su vez puede estar implementada sobre un sistema operativo (figura 4.1).

Los programas compilados con el pragma **Ravenscar** usan un conjunto limitado de los aspectos de GNARL que toma las ventajas del modelo de tareas simplificado del perfil Ravenscar para reducir la sobrecarga en tiempo de ejecución y tamaño del runtime. Este reducido GNARL todavía usa el mismo interfaz de bajo nivel (GNULLI) con el sistema operativo de la capa inferior.

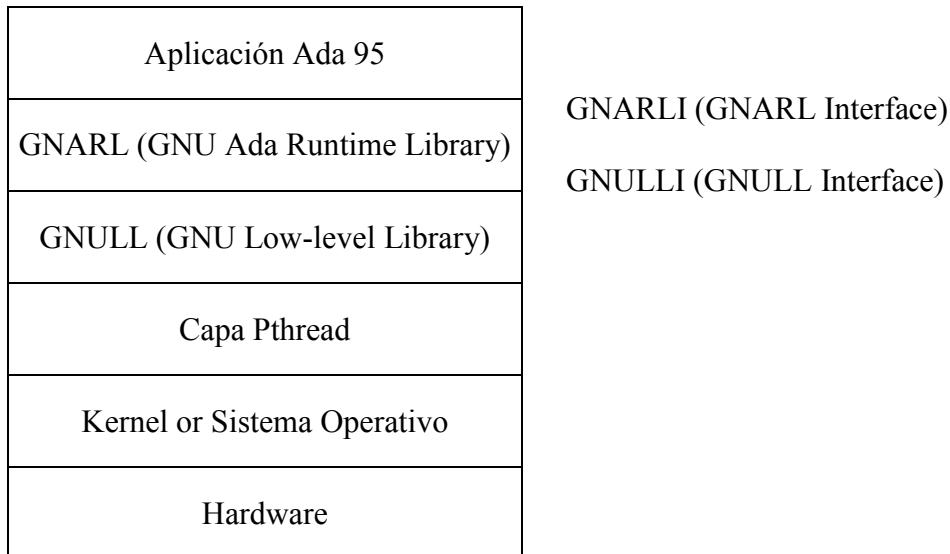


Figura 4.1. Arquitectura del run-time system de GNAT

4.3 Implementación del perfil **Ravenscar** en GNAT.

El actual GNAT aprovecha el construir GNARL sobre un interfaz **pthread** para tomar todas las ventajas que se obtienen desde el punto de vista de la portabilidad, pero a costa de imponer una excesiva sobrecarga en el sistema, dado que GNARL ya provee mucha de la funcionalidad relacionada con la gestión de tareas, y en consecuencia no usa muchas de las capacidades de los **pthread** o el sistema operativo. Esto eleva la complejidad, y por tanto va contra el propósito del perfil **Ravenscar** de dar soporte a las tareas Ada con un pequeño y seguro entorno de ejecución.

El enfoque de ORK consiste en implementar la gestión de tareas, con un pequeño y especializado kernel, el cual no tiene la sobrecarga innecesaria de la implementación completa de `pthread`. ORK proporciona una implementación casi directa de GNULLI, con los actuales paquetes de GNULL actuando como una fina capa para GNARL (figura 4.2).

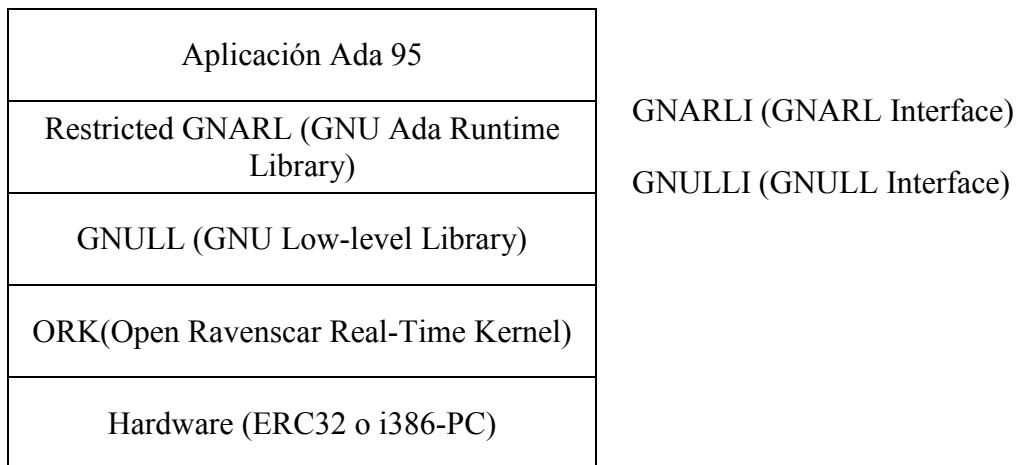


Figura 4.2. Arquitectura del run-time system de GNAT sobre ORK.

4.3.1 Adaptación de GNARL a ORK.

El siguiente paso es construir una versión de GNARL para ORK. Para llevar a cabo esto, los paquetes de GNULL han sido reescritos de nuevo para adaptarlos al interfaz de ORK. La capa de interfaz GNULLI prácticamente no ha sido modificada, por lo que las modificaciones a la capa superior de GNARL son mínimas. Los paquetes de GNULL modificados son los siguientes:

- `System.Task_Primitives`
- `System.Task_Primitives.Operations`
- `System.OS_Primitives`
- `System.OS_Interface`

ORK proporciona la mayoría de la funcionalidad de la gestión de tareas, tanto la especificación como los cuerpos de estos paquetes contienen renombramientos y llamadas a funciones del interfaz de ORK.

Se asume que todos los programas que usan ORK se compilan con el pragma `Ravenscar`, y por lo tanto solo se usará el GNARL restringido con el kernel. Por supuesto esto significa que no todos los programa Ada pueden funcionar sobre esta

versión especial de GNARL, pero esta limitación es necesaria ya que ORK está pensado para soportar los aspectos relativos a tareas del perfil Ravenscar y por lo tanto se toman las ventajas que supone tener solo un kernel pequeño y seguro.

Bajo esta condición, solo algunas partes de los paquetes de GNARL correspondientes a la capa superior han sido adaptadas a ORK, los paquetes modificados son:

- **System**: Parámetros específicos como el rango de prioridades.
- **System.Interrupts**: La implementación anterior no era conforme con el perfil de Ravenscar.
- **Ada.Interrupts.Names**: Los nombres de las interrupciones son coherentes con las interrupciones hardware.

4.3.2 Uso del kernel con programas C.

El kernel también puede ser usado con programas escritos en lenguaje C. C no tiene soporte para concurrencia, por lo que las funciones para crear y manejar threads tienen que ser explícitamente llamadas desde el código C. Se provee un API C para este propósito. El interfaz para lenguaje C se llama (CIL), el cual está integrado con el sistema de compilación GCC. (figura 4.3).

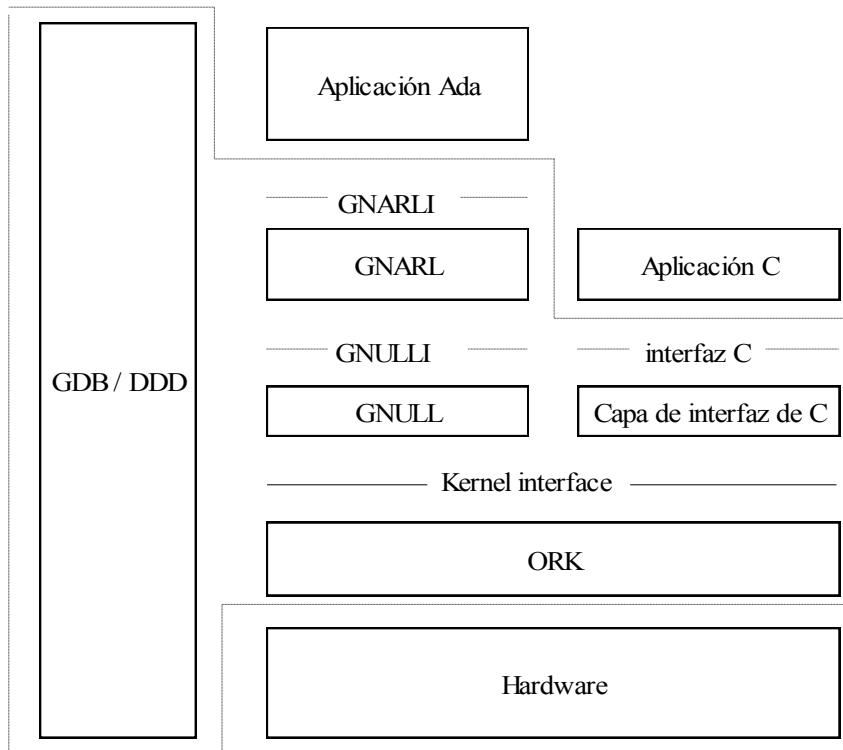


Figura 4.3. Arquitectura del run-time system GNAT/GCC basado en ORK.

4.4 Diseño del Open Ravenscar Real-Time Kernel.

4.4.1 Arquitectura.

La funcionalidad provista por ORK puede ser dividida en los siguiente servicios:

- Gestión de bajo nivel de tareas (threads).
- Sincronización.
- Planificación.
- Gestión de almacenamiento.
- Retardos y gestión de tiempos.
- Gestión de interrupciones.

El kernel está dividido en una serie de paquetes Ada, todos ellos hijos de un paquete superior llamado **Kernel**. Esta estructura es similar a la de otros kernels de similares características.

Los principales paquetes de ORK son:

- **Kernel.Threads**. Este paquete proporciona todo el soporte que requiere la creación, planificación y sincronización de threads.
- **Kernel.Time**. Este paquete proporciona el soporte para el reloj de tiempo real y los retardos absolutos.
- **Kernel.Interrupts**. Este paquete proporciona el soporte para la identificación de las interrupciones, y para asignar manejadores de interrupciones a las interrupciones.
- **Kernel.Memory**. Este paquete proporciona una gestión de almacenamiento dinámico muy limitada.
- **Kernel.Parameters**. Este paquete define los parámetros dependientes de la implementación.
- **Kernel.CPU_Primitives**. Este paquete contiene todos los elementos dependientes del procesador.
- **Kernel.Peripherals**. Este paquete proporciona el soporte para los periféricos disponibles en la plataforma de ejecución.

Sólo los primeros cuatro paquetes son visibles para GNULL. Los otros tres, **Kernel.Parameters**, **Kernel.CPU_Primitives**, **Kernel.Peripherals**, son usados por el resto de paquetes, y encapsulan elementos dependientes de la implementación, para así hacer más fácil portar el kernel a otra plataforma hardware.

Algunos de estos paquetes (ej: **Kernel.Threads**), están compuestos por más hijos que extienden su interfaz de tal manera que parte de su funcionalidad interna se vuelve visible para otros paquetes.

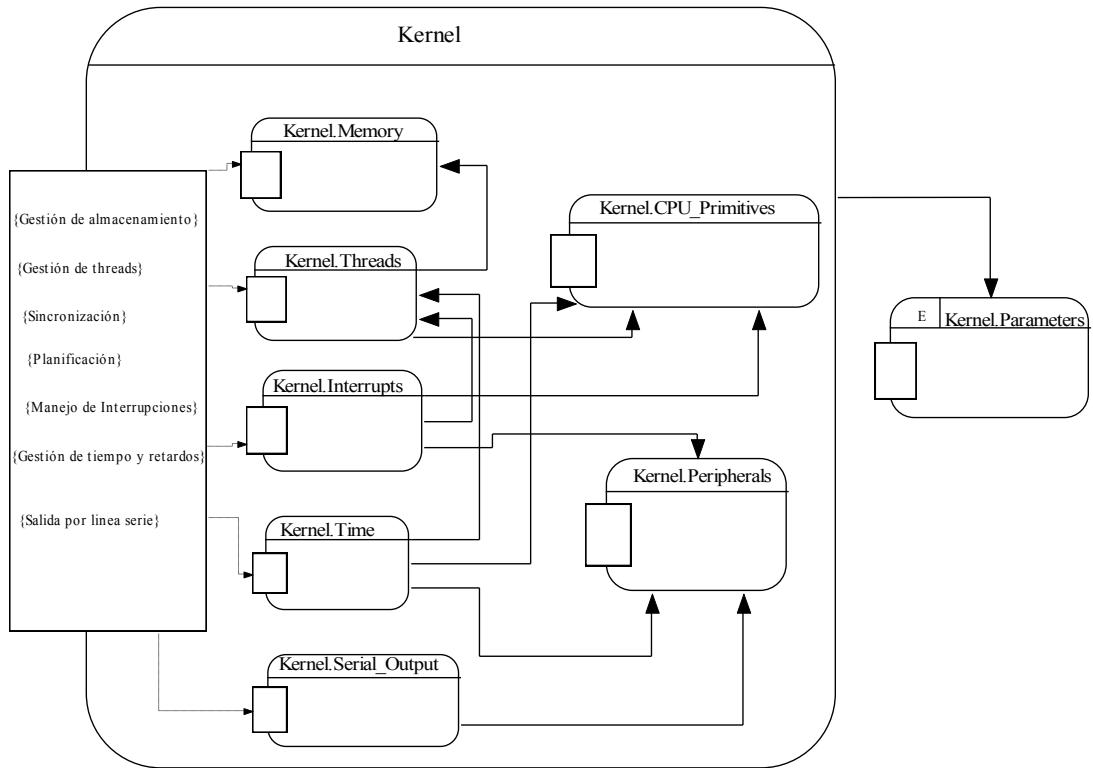


Figura 4.4. Arquitectura de ORK.

4.4.2 Gestión de threads.

Las tareas Ada están implementadas a bajo nivel por threads del kernel. Los threads son creados en la inicialización del programa, así que no es necesario reservar o liberar dinámicamente recursos tales como el TCB (Task Control Block) o el espacio de pila.

Los threads son planificados mediante una política *FIFO con prioridades*. Hay una cola de listos en la cual se ordenan por prioridades y orden de llegada. Las primitivas de sincronización insertan o eliminan threads directamente de la cola de listos.

El kernel proporciona dos tipos de primitivas de sincronización: cerros y variables de condición. Estos elementos son usados por GNARL para implementar los objetos protegidos. A pesar de la similitud de nombres con POSIX, las primitivas de sincronización de ORK son definidas de tal forma que los elementos relacionados con GNULLI pueden ser directamente implementados, por lo tanto obtenemos una implementación más eficiente de los objetos protegidos.

Han sido cogidas las ventajas que proporciona el perfil Ravenscar para implementar solo un tipo de cerros de exclusión mutua.

Las variables condición han sido también simplificadas con respecto a POSIX, dado que no hay tiempos relativos de espera ni colas de tareas suspendidas, y por tanto el número de tareas que pueden estar esperando una condición es igual a una. Además el perfil Ravenscar no permite la sentencia `select`, operaciones de esperas relativas, o ATCs (transferencias asíncronas de control), lo que da como resultado unas variables condición muy simplificadas.

La implementación de los cerrojos sigue el protocolo de herencia inmediata del techo de prioridad, que se corresponde con la política `Ceiling Locking` del estándar de Ada.

4.4.3 Gestión de tiempo.

El paquete `Kernel.Time` proporciona soporte de tiempo de una manera muy simplificada. Se define un tipo tiempo que representa tiempo absoluto y intervalos de tiempo como un número entero de nanosegundos. Hay un reloj que da los tiempos absolutos medidos desde que el sistema se inició.

De acuerdo con la especificación del perfil Ravenscar, sólo se permiten los retardos absolutos (`delay until`). Los threads retrasados o demorados (`delay`) son mantenidos en una cola de espera la cual está ordenada por tiempo de cumplimiento del plazo. Desde una espera no puede haber cancelaciones, ya que no se permiten las sentencias abortivas o ATCs, por lo tanto no hay necesidad de mantener los threads que están esperando un plazo en una variable de condición, como ocurre en la actual implementación de GNARL.

4.4.4 Gestión de almacenamiento.

A pesar de que el perfil Ravenscar no prohíbe la gestión dinámica de almacenamiento, ya que el perfil no trata con aspectos que no se refieran a las tareas Ada, parece razonable esperar que los programas bajo el perfil Ravenscar no utilicen memoria dinámicos. Consecuentemente, solo se proporciona una muy limitada gestión de memoria, destinada a reservar espacio para TCB y espacio de pila para nuevos threads. Dado que un thread se inicia al comienzo de la ejecución del programa y no puede terminar, el algoritmo de asignación de memoria es lineal y muy sencillo.

En la versión de ORK para ERC32 (la original) la pila de tareas está protegida para evitar superar los límites. Para este propósito se utilizan los mecanismos de protección de segmentos.

4.4.5 Manejo de interrupciones.

Una interrupción representa una clase de evento que es detectado por el propio hardware. Cuando una interrupción ocurre una *Rutina de Servicio de Interrupción (ISR)* es invocada para poner a disposición del kernel dicha interrupción. En Ada 95, un manejador de alto nivel puede ser asociado a la interrupción, para que así este sea invocado directamente por el entorno de ejecución cuando ocurra la interrupción. El manejador que se pasa como parámetro puede ser o un procedimiento protegido o una entrada de una tarea, se hace referencia a esta última para mantener la compatibilidad con Ada 83.

La actual implementación de GNARL utiliza tareas de servicio de interrupción que se activan cuando ocurre una interrupción, y llaman al procedimiento protegido asociado a la interrupción. De esta manera, tanto las prioridades como la exclusión mutua son manejadas de la misma forma para tareas y objetos protegidos. Los manejadores de interrupciones se ejecutan en el contexto de las tareas de servicio de interrupciones, las cuales facilitan un modelo de ejecución limpio comparado con otras aproximaciones en las que el manejador se ejecuta en el contexto de la tarea interrumpida.

ORK sigue una filosofía diferente. Bajo el perfil Ravenscar solo se pueden definir procedimientos protegidos como manejadores. Además, la única política de bloqueos es **Ceiling Locking**. Esto significa que un manejador de interrupción nunca puede ser bloqueado esperando que un procedimiento protegido se quede libre. Actualmente, los manejadores de interrupciones se ejecutan como si hubieran sido invocados por la tarea interrumpida, con la diferencia que se usa la pila de interrupción en vez de la pila de la tarea interrumpida.

El paquete `Kernel.Interrupts` proporciona nombres simbólicos para las interrupciones y operaciones para asignar manejadores a interrupciones. Los manejadores son directamente asignados a interrupciones sin la sobrecarga de usar las señales POSIX como hace el actual GNARL.

4.4.6 Otras características de diseño.

El interfaz del kernel es totalmente procedural, y no necesita modos de funcionamiento de supervisor y usuario. Todo el programa ejecuta en modo supervisor, como es típico en los sistemas empotrados. La exclusión mutua en el kernel se consigue haciendo que este se comporte como un monitor monolítico, protegiendo los acceso al kernel desactivando las interrupciones, así las interrupciones son pospuestas mientras se esté ejecutando un función del kernel.

Las operaciones *lock* de los cerrojos se implementan elevando la prioridad activa del thread al techo de prioridad del cerrojo.

Capítulo 5

Introducción a la familia ix86 y la arquitectura del PC

5.1 Introducción.

La familia de microprocesadores que llamaremos ix86 nace en 1978 cuando Intel presenta su 8086, uno de los primeros de 16 bits y, desde luego, el primero que alcanzó un éxito notable. En años posteriores fueron apareciendo el 80186, el 80286, el 80386, el 80486, y el *Pentium* con todos sus sucesores. No podemos olvidar en esta introducción la importancia que ha tenido para esta familia el haber sido elegida por IBM para construir el órgano central de su “PC”. El hecho es que quizás fue un cúmulo de casualidades lo que ha llevado a esta familia a ocupar el puesto que hoy ocupa. Es un claro ejemplo de cómo a menudo no son los mejores los que ganan la competición sino los que han sabido anticiparse y colocarse en mejor puesto en línea de salida.

Pero, por otro lado, esta elección tuvo sus inconvenientes. Si, como comentaremos, el 8086 pretendía compatibilidad con microprocesadores anteriores –el 8085-, su necesaria evolución se ha visto obligada en todo momento a mantener compatibilidad con todo lo anterior. Ha sido necesaria mucha imaginación para lograr que el 286 y siguientes sean capaces de ejecutar programas creados para sus predecesores, a la vez que esto ha supuesto una clara limitación en su diseño. La familia ix86 se verá atada para siempre a estos requisitos de compatibilidad hacia atrás que le imponen el elevado valor de los programas desarrollados para sus primeros miembros.

5.2 Orígenes.

Cuando Intel se propuso lanzar al mercado un nuevo microprocesador, ya con registros internos y bus de datos de 16 bits, y con una capacidad de direccionamiento de memoria de 1 Mbyte, su microprocesador de 8 bits estaba en pleno apogeo, luego es lógico que desease mantener su importante cuota de mercado y no se arriesgase a

perder clientela dando un brusco salto hacia un microprocesador completamente nuevo. Así es que propuso que el nuevo diseño mantuviese toda la compatibilidad hacia atrás que fuese posible.

Esta pretendida compatibilidad fue la que marcó para siempre a la familia ix86, con su peculiar arquitectura y con su “endiablado invento” de los SEGMENTOS de memoria. Esta complicada arquitectura y el manejo de la memoria se explica más adelante.

5.3 La familia básica.

Llamaremos familia básica al conjunto de microprocesadores, todos ellos derivados de 8086, que constituyen el tronco principal, del cual surgen otros como ramificaciones. Con el objeto de que el lector tenga una visión global y comparativa de esta familia, se presenta en la tabla 5.1 que resume las características diferenciales más importantes.

Tabla 5.1:Características esenciales de los procesadores básicos de la familia ix86.

CARACTERÍSTICAS	8086	80186	80286	80386	80486	Pentium
Bus de datos	16 bits	16 bits	16 bits	32 bits	32 bits	64 Bits
Bus de direcciones	20 bits	20 bits	24 bits	32 bits	32 Bits	32 bits
Memoria direccionable	1 Mbyte	1 Mbyte	16 Mbyte	4 Gbyte	4 Gbyte	4 Gbyte
Registros internos	16 bits	16 bits	16 bits	32 bits	32 bits	32 Bits
Coprocesador interno	NO	NO	NO	NO	SI	SI
Modo REAL	SI	SI	SI	SI	SI	SI
Modo PROTEGIDO	NO	NO	SI	SI	SI	SI
Modo VIRTUAL-86	NO	NO	NO	SI	SI	SI
Paginación	NO	NO	NO	SI	SI	SI
Cache interna	NO	NO	NO	NO	SI	SI

El 80186 conserva la arquitectura básica del 8086. Su principal característica es la integración en la misma pastilla de otros elementos HW como generador de reloj, dos canales de DMA, 3 temporizadores, controlador de interrupciones, generador de estados de espera programables y lógica de selección programable. Su arquitectura mejorada proporciona una velocidad doble de la del 8086. Este fue un microprocesador que apenas se empleo en la construcción de PCs.

Un salto más importante fue el paso al 80286, en el que se ampliaba el bus de direcciones hasta 24 bits (direcccionamiento de 16 Mbytes). Puede funcionar en dos modos: el **MODO REAL** es en esencia un 8086 mejorado, con capacidad de ejecutar instrucciones de forma más eficaz y por lo tanto más rápida, y el **MODO PROTEGIDO** en el cual se logra el acceso a toda la memoria direccionable, y se

establecen mecanismos hardware capaces de implementar un sistema de protecciones muy completo, apto para ejecutar sistemas operativos multiusuario y/o multitarea. Incorpora también una unidad de manejo de memoria virtual.

El 80386 perfecciona el camino emprendido por el 286. Es un microprocesador de 32 bits, es decir, con bus de datos y registros internos de 32 bits. En su modo protegido maneja direcciones de 32 bits, lo que le da una capacidad de direccionamiento de memoria física de hasta 4 Gigabytes. Añade un tercer modo de funcionamiento llamado **MODO VIRTUAL-86** que es capaz de simular la ejecución simultánea de varias tareas, cada una de ellas como si estuviese en el entorno de un 8086. Añade también un nuevo modo de manejo de memoria basado en páginas de 4 Kbytes. Por último introduce un nuevo concepto en el tamaño de los segmentos, siendo capaz de manejar toda su memoria como si estuviese contenida en un mismo segmento (modelo de memoria plana).

El 486 apareció en 1.989 sin introducir grandes cambios respecto al 386. Conserva su arquitectura básica y dedica la mayor parte de sus mejoras a integrar en el mismo chip el coprocesador matemático, que hasta entonces se encontraba en otra pastilla, y una pequeña memoria cache de 8 KB, con su controlador correspondiente. Su más eficiente diseño permite que las instrucciones se ejecuten en menos ciclos de reloj.

Por último Intel ha sacado la familia de los *Pentium*, en el que se confirman las conjeturas anteriores, ya que no añaden ninguna característica nueva a la arquitectura del 386. Integra también un coprocesador matemático muy mejorado y añade una segunda memoria cache, también de 8 KB. De esta forma se dedica una cache para instrucciones y otra para datos. Pero sus virtudes más relevantes se encuentran en un nuevo diseño que le permite introducir grandes mejoras en la velocidad. Este nuevo diseño se basa en utilizar en todo lo posible los conceptos de las arquitectura RISC, y la ejecución superescalar basada en dos unidades de enteros que pueden trabajar en paralelo. Añade también una unidad de predicción de salto.

5.4 Modos de funcionamiento.

Como puede verse en la tabla 5.1, los procesadores de la familia ix86 pueden funcionar en varios modos, según la CPU de que se trate. Estos modos se denominan:

- Modo Real.
- Modo Protegido.
- Modo Virtual-86.

5.4.1 El Modo Real.

El Modo Real constituye la base de funcionamiento de los PCs. Todos los programas escritos para este modo tienen la garantía de que podrán ser ejecutados en cualquier

procesador de la familia ix86. En este apartado se hará una breve introducción al modo Real original es decir el que mantiene absoluta compatibilidad con el 8086 original.

5.4.1.1 El modelo de programación.

El modelo de programación recoge todos los registros de la CPU que son accesibles mediante instrucciones. Los registros se pueden dividir en los siguientes grupos:

- **Registros generales:** Son registros de propósito general, es decir que pueden utilizarse para cualquier fin. A este grupo pertenecen AX, BX, CX y DX, todos ellos de 16 bits. Sin embargo algunas instrucciones los usan de forma específica, por lo que reciben los nombres que aparecen en la Figura 5.1, aunque no se entrara aquí en profundidad a explicar su uso.

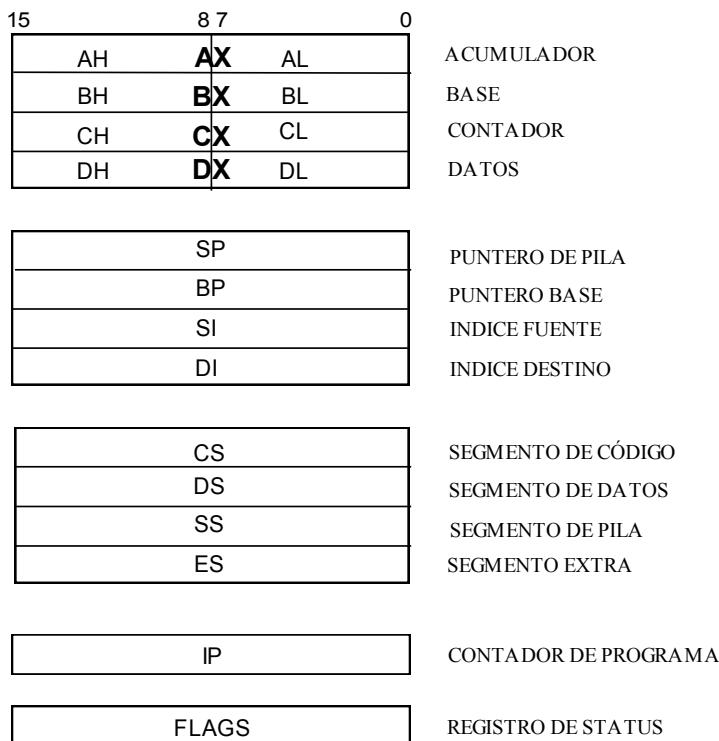


Figura 5.1: Modelo básico de programación del MODO REAL.

- **Registros punteros e índices:** Son también registros de 16 bits, a diferencia de los anteriores no se puede acceder a sus bytes alto y bajo independientemente.
 - SP(Stack Pointer): Es el clásico puntero de pila.
 - BP(Base Pointer): Se puede usar para cualquier finalidad pero se suele usar como marco de pila.

- SI(Source Index): Se puede usar para cualquier fin, pero algunas instrucciones lo usan como registro índice.
 - DI(Destination Index): Se puede usar para cualquier fin, pero algunas instrucciones lo usan como registro índice.
- **Registro de estado:** El registro de estado se denomina generalmente como F (Flags) su estructura ha ido modificándose en los diferentes microprocesadores de la familia. En él se encuentran los clásicos bits de estado como bit de cero, bit de signo, bit de acarreo, bit de paridad, que muestran los resultados de distintas instrucciones. Bits de control entre los que cabe destacar el bit que controla la inhibición de las interrupciones, además de otros bits que no se entra a detallar aquí.
 - **Puntero de instrucciones:** Es el contador de programa de todo microprocesador llamado en este caso (EIP). Su función es la de apuntar sucesivamente a las instrucciones que componen un programa.
 - **Registros de segmentos:** En relación con el mecanismo de direccionamiento de memoria que se explica en el siguiente apartado, existen 4 registros, también de 16 bits, denominados CS, DS, SS y ES, a partir del 386 se añaden nuevos registros de segmento. Cada uno de ellos contiene la dirección de un segmento.
 - CS(Code Segment): Registro de Segmento de Código. Contiene la dirección del segmento de memoria que contiene el código.
 - DS(Data Segment): Registro de Segmento de Datos. Contiene la dirección del segmento de memoria que contiene los datos que el programa utiliza.
 - SS(Stack Segment): Registro de Segmento de Pila. Contiene la dirección del segmento de memoria donde está situada la pila.
 - ES(Extra Segment): Registro de Segmento Extra. Contiene la dirección de un segmento adicional de memoria utilizado para manejar datos.

5.4.1.2 El direccionamiento de la memoria.

El direccionamiento de la memoria es uno de los temas más espinosos de la familia ix86. En este sentido, las CPUs de esta familia tienen un modo de funcionamiento que las diferencia de cualquier otro microprocesador. El origen de esta complicación parte del interés de Intel de presentar la microprocesador 8086 como sucesor del 8085. En el Modo Real permite direccionar hasta 1 MB de memoria, lo cual es claramente muy poco actualmente, por lo que el fabricante ha tenido que recurrir a complicados e inteligentes métodos para poder acceder a un mapa de memoria más grande (16 MB en el 286 y 4000 MB en los siguientes). Un segmento es un bloque de memoria de 64 KB, definido por su dirección base, expresada con 16 bits y contenida en alguno de los registros de segmento descritos en el párrafo anterior. Puesto que la dirección de memoria física perteneciente a un mapa de 1 MB debe ser expresada mediante 20 bits,

la dirección contenida en el registro de segmento debe completarse con 4 bits de valor 0 añadidos a su derecha. Esto es equivalente a multiplicar por 16 el valor contenido en el registro de segmento. En definitiva, un segmento de memoria física debe comenzar siempre en una dirección múltiplo de 16.

Una posición concreta de memoria se define mediante un par de números de 16 bits que representa:

- El segmento al que pertenece.
- El desplazamiento (offset) de dicha posición respecto al origen del segmento.

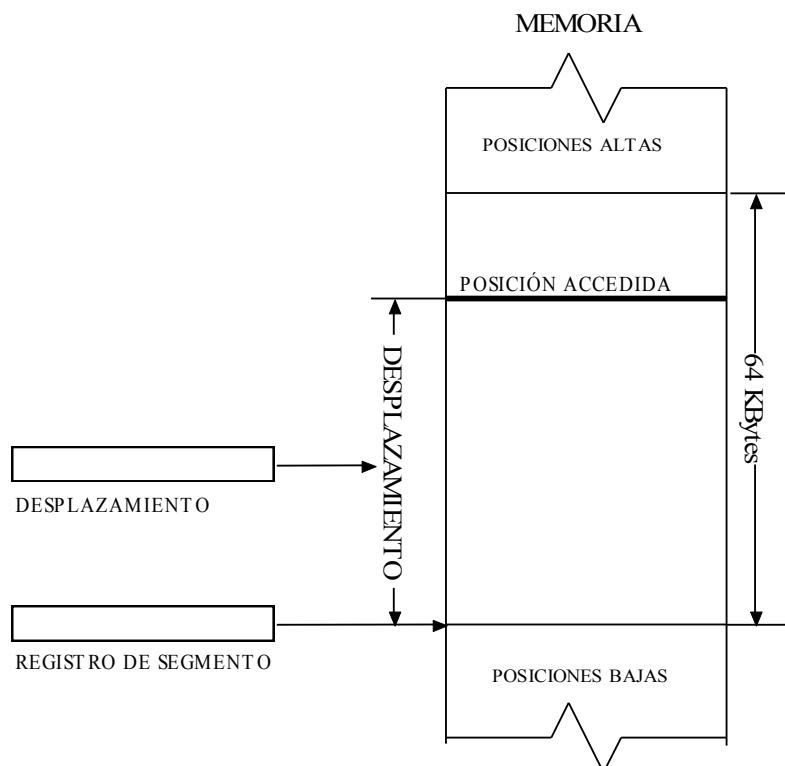


Figura 5.2: Mecanismo de acceso a la memoria en el MODO REAL.

Puesto que en la CPU existen cuatro registros de segmento, están disponibles simultáneamente cuatro segmentos. Todas sus posiciones se pueden recorrer nada mas variar su desplazamiento. Cuando se requiere el acceso a otras zonas de memoria, es necesario alterar el contenido de algún registro de segmento. En función del ciclo que este llevando a cabo la CPU en un determinado momento usará un registro de segmento u otro, según lo indicado en la tabla 5.2.

Tabla 5.2:Registros de segmento utilizado por los diversos punteros.

Tipo de acceso a memoria	Registro de segmento utilizado	Procedencia del desplazamiento
Ciclo de búsqueda de instrucción (<i>fetch</i>)	CS	IP
Ciclo de acceso a la PILA	SS	SP
Ciclo de acceso a DATOS	Por defecto DS. Con prefijo de segmento cualquiera: CS, DS, SS, ES	Según el modo de direccionamiento utilizado

Los modos de direccionamiento son muy variados, incluyendo direccionamiento a registro, direccionamiento inmediato, direccionamiento directo, direccionamiento indirecto por registro, direccionamiento relativo a base, y algunos más que no entraremos a explicar, para más información se recomienda acudir a la bibliografía.

5.4.1.3 Los puertos de entrada/salida (E/S).

En todos los procesadores de la familia ix86 los puerto de entrada/salida se manejan en un mapa independiente del de memoria, que comprende las direcciones de 0 hasta FFFFh. Son 65536 los puertos (64KB), aunque en los PC generalmente este limitado a 1000. Desde el punto de vista hardware se utiliza el mismo bus de direcciones, pero existen señales de control diferentes para los accesos a memorias o al mapa de E/S. De este modo puede existir un puerto con la dirección 300h, que no tiene nada que ver con la dirección de memoria 300h.

Las operaciones de E/S permiten la escritura y lectura de los puertos instalados en el mapa de entradas/salidas. Los instrucciones que se emplean para este fin son **IN** y **OUT**.

5.4.2 El Modo Protegido.

Cuando los microprocesadores 286 y siguientes entran en el modo protegido, sus posibilidades se expanden para dar cabida a las siguientes características:

- Posibilidad de acceder a toda la memoria que el hardware es capaz de manejar.
- Posibilidad de utilizar la MMU(*Memory Management Unit*) incluida en el chip que permite el manejo de memoria virtual.
- Múltiples mecanismos de protección que permiten implementar sistemas operativos multitarea y multiusuario.

- Mecanismos de conmutación de tareas muy ágil que permite salvaguardar un contexto y sustituirlo por el de una tarea nueva en poco tiempo.
Además a partir del 386 se incluyen nuevas prestaciones:
 - Mecanismos de paginación.
 - Introducción del modo Virtual86.
 - Los segmentos pueden tener un tamaño mayor de 64 KB.

5.4.2.1 Soporte hardware del modo protegido.

La implementación del modo protegido se realiza por medio de registros especiales, que se añaden al modelo de programación básico del modo real. Existen variaciones en la configuración y tamaño de los registros asociados al modo real, en función del microprocesador de la familia al que nos refiramos. De ahora en adelante nos referiremos siempre al microprocesador i386, ya que es a partir del cual se empieza a utilizar realmente el modo protegido. Los principales registros del Modo Protegido son los siguientes:

- GDTR (*Global Descriptor Table Register*): Apunta a la posición de comienzo de la Tabla de Descriptores Globales, que contiene todos los descriptores de los segmentos manejados por el sistema operativo. Consta de dos campos:
 - Dirección base de la tabla, 32 bits.
 - Límite de la tabla, 16 bits.
- LDTR (*Local Descriptor Table Register*): Apunta a la posición de comienzo de la Tabla de Descriptores locales, que contiene todos los descriptores de los segmentos manejados por la tarea que está activa en un determinado momento. Este registro tiene asociados determinados registros cache ocultos, con la siguientes estructura:
 - Dir. Lineal que define la base de LDT, 32 bits.
 - Límite de LDT, 32 bits.
 - Un byte de derechos de acceso.
- IDTR (*Interrupt Descriptor Table Register*): Apunta a la posición de comienzo de la Tabla de Descriptores de Interrupciones, que contiene todos los descriptores de los segmentos que contienen las rutinas de atención a interrupción. Consta de dos campos:
 - Dirección base de la tabla, 32 bits.
 - Límite de la tabla, 16 bits.
- TR (*Task Register*): Apunta al Segmento de Estado de la Tarea (TSS, *Task State Segment*), que contiene el contexto de la tarea activa en un determinado momento. Este registro tiene asociados determinados registros ocultos.
 - Dir. Lineal que define la base del TSS, 32 bits.
 - Límite del TSS, 32 bits.

- Un byte de derechos de acceso.
- MSW (*Machine Status Word*): Que en el 386 y siguientes forma parte del registro CR0(*Control Register 0*). Es un conjunto de flags con diversas funciones. La activación de uno de ellos, llamado PE(*Protected Mode Enable*) obliga a la CPU a funcionar en modo protegido.

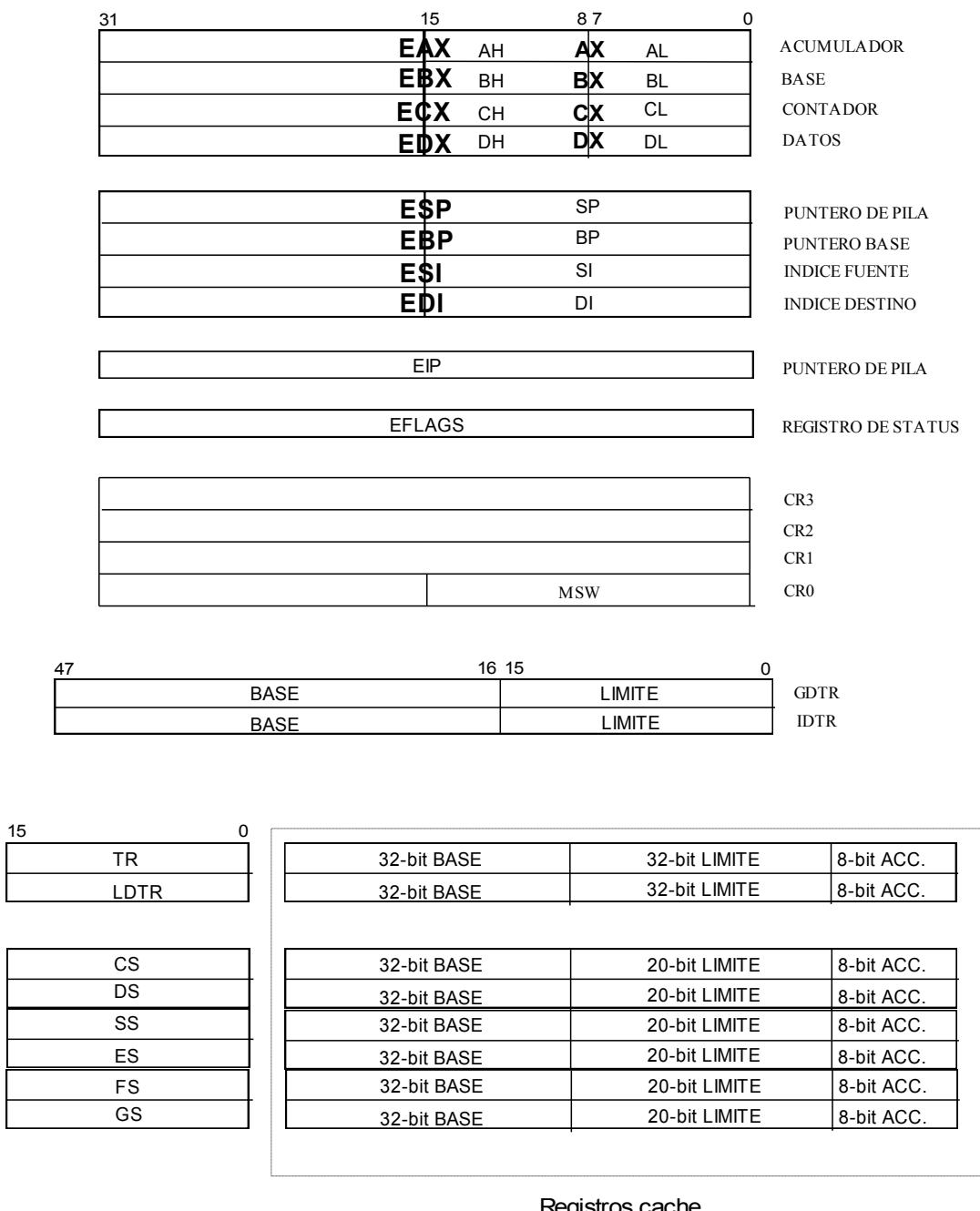


Figura 5.3: Modelo de programación en el MODO REAL.

- Registros ocultos de los segmentos: Asociado a cada registro de segmento CS, DS, SS y ES (también FS y GS a partir del 386), se encuentran un registro oculto destinado a contener la información del descriptor de segmento seleccionado. Los registros de cache funcionan automáticamente y son transparentes al programador.
 - Dirección base del segmento, 32 bits.
 - Límite del segmento, 20 bits.
 - Un byte de derechos de acceso al segmentos.
- Registros de control: Además del registro de estado del Modo Real, el modo protegido incorpora una serie de registros de control (CR0, CR1, CR2 y CR3), que contienen en su mayor parte información asociada con la paginación de memoria.

5.4.2.2 El acceso a memoria.

En el modo protegido es posible el acceso a toda la memoria que la CPU es capaz de direccionar, es decir 4GB a partir del 386. Para ello se requiere manejar 32 bits de direcciones. Pero para poder mantener la compatibilidad con el Modo Real que únicamente podía direccionar 1 MB de memoria, fue necesario introducir una complicación adicional: los DESCRIPTORES DE SEGMENTOS. En el modo protegido, todo segmento de memoria que va a utilizar un programa debe estar previamente definido mediante una estructura de datos residente en memoria, denominada DESCRIPTOR. Más adelante se estudiará la estructura de un segmento, fundamentalmente el contenido más importante de un segmento es la dirección física de la base del segmento al que se accede. La dirección se expresa de la forma ya conocida en el Modo Real: SEGMENTO:DESPLAZAMIENTO, en donde ambos elementos son de 16 bits. La dirección de la base del segmento contenida en un descriptor es de 32 bits en el 386 y siguientes. El mecanismo básico de acceso a una dirección concreta, especificada mediante el esquema SEGMENTO:DESPLAZAMIENTO es el siguiente. El SEGMENTO localiza un DESCRIPTOR que contiene la dirección base de comienzo del segmento, expresada en 32 bits, dentro de este segmento, una posición de memoria concreta se localiza por el DESPLAZAMIENTO, lo mismo que en el modo real.

Pero además se aprovechó la necesidad de los descriptores para especificar otras propiedades del segmento como son su LIMITE y sus DERECHOS DE ACCESO.

Todos los descriptores se agrupan en una TABLA DE DESCRIPTORES residente en memoria, que está apuntada por un registro hardware de la CPU (GDTR, LDTR o IDTR, según la tabla de que se trate). En estas circunstancias, cuando un programa desea acceder a un posición de memoria concreta utiliza el esquema SEGMENTO:DESPLAZAMIENTO pero su efecto será distinto al del modo real.

El SEGMENTO selecciona un elemento de la tabla de descriptores, por lo que se le va a denominar SELECTOR. Lo mismo que ocurre en el modo real es un valor de 16 bits. El proceso se representa en la figura 5.4.

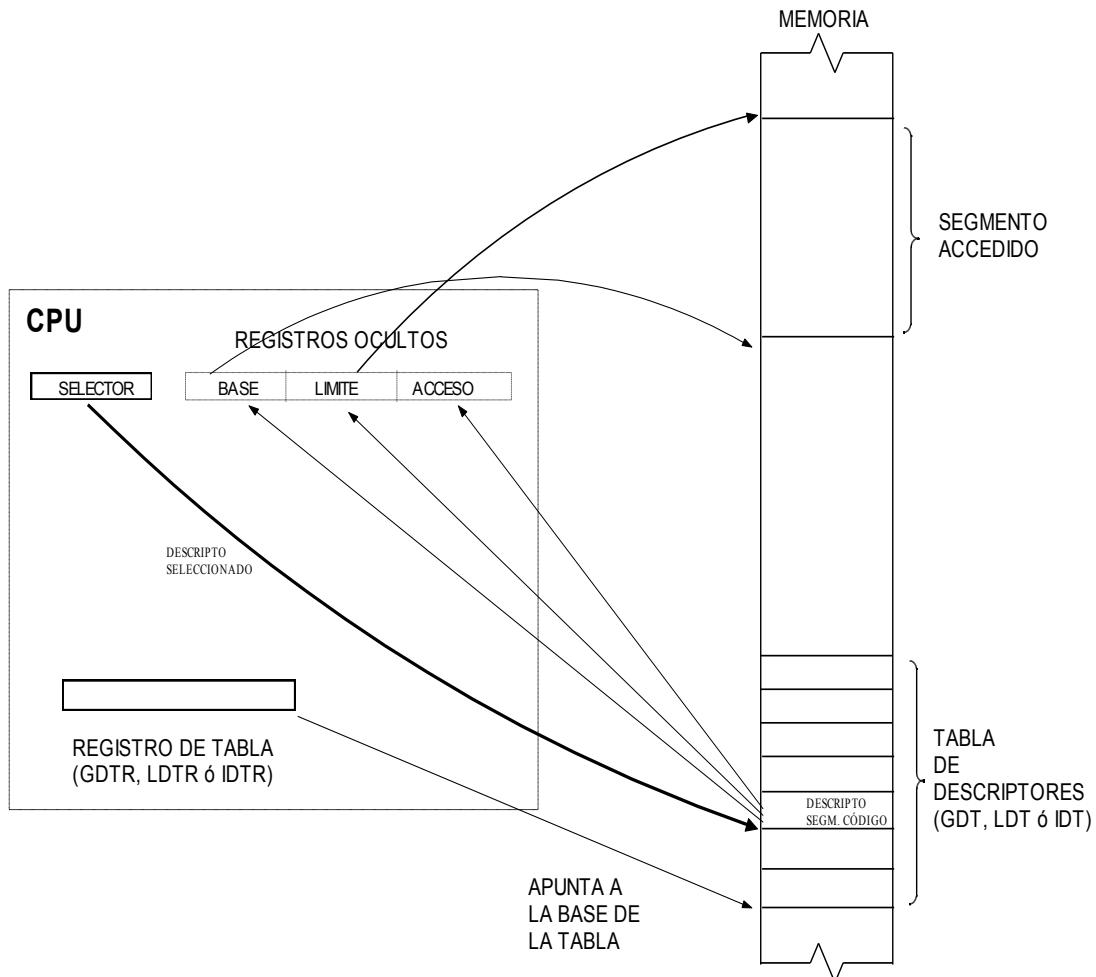


Figura 5.4: Localización de un segmento a través de la tabla de descriptores.

Los pasos a seguir son los siguientes:

- Existe previamente una TABLA DE DESCRIPTORES que define todos los segmentos que se van a usar.
- Un REGISTRO DE TABLA en la CPU apunta a la base de dicha tabla.
- Se escribe un SELECTOR en alguno de los registros de segmento (CS, DS,...).
- Este selector permite localizar un elemento de la tabla, es decir un descriptor concreto de la misma.
- Se procede a la lectura del descriptor.
- El contenido del descriptor (BASE, LIMITE y ACCESO) son guardados automáticamente en los registros cache asociados.
- El valor BASE sirve para determinar la posición de memoria en donde comienza el segmento.

- El valor LIMITE determina el tamaño del segmento. Este atributo puede ser empleado como uno de los mecanismos de protección disponibles en el modo protegido.
- Una vez localizado el segmento, el desplazamiento se usará de la forma habitual para la localización de la posición de memoria concreta a la que se accede.

Este proceso se desencadena cada vez que se accede a un nuevo segmento y supone la realización de varios ciclos de memoria, por lo que es relativamente lento. Sin embargo los accesos sucesivos a un segmento ya seleccionado se realizan a velocidad normal.

En la tabla 5.3 se resumen los diferentes tipos de descriptores que se manejan en el modo protegido.

Tabla 5.3. Tipos de descriptores

TIPO DE DESCRIPTOR	
NORMALES	de SEGMENTO DE DATOS
	de SEGMENTO DE CODIGO
ESPECIALES	de TABLA LTD
	de ESTADO DE TAREA
	de CALL GATE
	de TASK GATE
	de INTERRUPT GATE
	de TRAP GATE

Como puede verse, existen dos tipos de descriptores normales que se refieren a segmentos de código y datos (los segmentos de pila utilizan el formato de los descriptores de datos). Los restantes son especiales y realmente no todos ellos describen segmentos propiamente dichos sino otras estructuras de datos que no serán explicadas en este trabajo.

Todos los descriptores de segmentos que se utilicen en la ejecución de un programa se agrupan en TABLAS DE DESCRIPTORES. Cada uno de los descriptores ocupa 8 bytes, mientras que el tamaño de cada tabla queda limitado a un segmento, es decir 64 Kbytes. En consecuencia, cada tabla va a contener como máximo 8192 descriptores. En un sistema dado se distinguen tres tablas diferentes:

- **GDT (Global Descriptor Table):** Contiene todos los descriptores propios del sistema operativo, lo que incluye descriptores de segmento de código, de datos y descriptores especiales. La base de esta tabla estará apuntada por el registro GDTR.
- **LDT (Local Descriptor Table):** Contiene todos los descriptores de los segmentos asociados a una tarea. Incluye descriptores de segmentos de código, de datos y

de algunos tipos de puertas de acceso. La base de esta tabla estará apuntada por el registro LDTR.

- IDT (*Interrupt Descriptor Table*): Contiene todos los descriptors de los segmentos que contienen las rutinas de atención de interrupción. Está apuntado por el registro IDTR.

Las tablas GDT e IDT son propias del sistema operativo. En el proceso de arranque del ordenador se inicializan los registro GDTR e IDTR de forma que apunten al comienzo de las mismas. En un funcionamiento normal el contenido de estos registros no se altera, trabajando siempre sobre las mismas tablas. Son comunes para todas las tareas.

Por el contrario, la tabla LDT está asociada a una tarea determinada y contiene todos los descriptores de los segmentos utilizados por la misma. En un sistema multitarea, al producirse una commutación de tareas, mecanismo mediante el cual se suspende la ejecución de una tarea y se pasa a ejecutar una distinta, el registro LDTR toma un nuevo valor, apuntando a la LDT propia de la nueva tarea. Cada tabla LDT está apuntada por un descriptor especial contenido en la tabla GDT.

5.4.2.3 Las interrupciones y excepciones.

En este capítulo no se estudiará en profundidad el funcionamiento de las interrupciones en la arquitectura i386-PC, ya que al tratarse de un punto crítico de los sistemas de tiempo real, su explicación exhaustiva se dejará para el capítulo siguiente donde se explique a la vez su funcionamiento e implementación.

Realmente interrupciones y excepciones representan el mismo concepto. Funcionan de forma idéntica, diferenciándose entre sí porque las interrupciones pueden producirse de forma asíncrona con la ejecución del programa, mientras que las excepciones son síncronas con éste.

Entendemos como interrupción a las que tienen un origen hardware por la activación de los terminales INTR o NMI.

Las excepciones pueden ser de dos tipos:

- Detectadas por el procesador en la ejecución de alguna instrucción.

FALLOS: Un fallo es una excepción que generalmente puede ser corregida, y una vez corregido el fallo permitir al programa continuar. Son los detectados antes de ejecutarse una instrucción. La instrucción puede ser reemprendida. Los valores de CS e IP guardados en la pila corresponden a la instrucción que ha causado la excepción.

TRAPS: Son las detectadas inmediatamente después de ejecutarse una instrucción. Los valores de CS e IP guardados en la pila corresponden a la instrucción siguiente a la que ha causado la excepción. En el caso de instrucciones de transferencia de control, debe interpretarse que se trata de la siguiente instrucción a ejecutar.

ABORTOS: Son errores que no admiten corrección. Pueden deberse a errores graves en el hardware o en la inconsistencia de determinados valores.

- Interrupciones software, producidas por las instrucciones **INT**, **INTO**, **INT n** y **BOUND**.

5.4.2.4 Otros aspectos.

El modo protegido ofrece multitud de recursos y características que no se han entrado a explicar en el presente capítulo, dado que no serán de vital importancia en la implementación del sistema que nos ocupa ORK, pues o no se utilizarán o se utilizarán de manera “testimonial”. Algunos de estos aspectos presentes en la arquitectura i386 son los siguientes.

LA PAGINACIÓN.

El mecanismo de paginación aparece por primera vez en el 386, pero su uso es opcional. Constituye una indirección adicional que se interpone en la salida del bus de direcciones de la CPU para generar las direcciones físicas que realmente acceden a la memoria. El mecanismo de la paginación hace una traslación de las *direcciones lineales* en *direcciones físicas* de forma transparente al usuario. Las ventajas de la paginación son varias, como son el uso de esta para la implementación de la memoria virtual, o implementar mecanismos de protección de memoria efectivos. Las ventajas anteriores no lo son tal en un sistema de tiempo real, luego la paginación y la memoria virtual no serán características de nuestro sistema.

TSS.

Un elemento clave en el modo protegido asociado a las tareas es el TSS (*Task State Segment*), Segmento de Estado de Tarea , que define el contexto inicial de una tarea y salvaguardar su contexto en el momento en que ha sido suspendida. El TSS no es sino una estructura de datos, residente en memoria, que ocupa un segmento, y que como tal, está definido por un descriptor. Esta estructura se la puede considerar como de grano grueso y está pensada para el manejo de *tareas pesada*, dado que nuestro sistema funcionará sobre el concepto de *procesos ligeros*, su uso será mínimo.

MECANISMOS DE PROTECCIÓN.

En el modo protegido se manejan 4 niveles de privilegio. Cada recurso del sistema está señalado por un cierto nivel de privilegio, así como también el código que se ejecuta en cada momento, la regla general dice que para acceder a un recurso es necesario que el código tenga un nivel de privilegio igual o superior al recurso accedido. Aunque dado que no nos encontramos ante un sistema multiusuario, todos el sistema funcionará bajo

el mismo nivel de privilegio desde un principio, luego no será un tema que esté presente el de los mecanismos de protección a lo largo de la implementación del sistema.

5.5 La arquitectura del PC.

En este punto se hará una revisión histórica de lo que se entiende por un PC y sus diversas actualizaciones a lo largo de los años. Claro está, que se harán referencias a aspectos del PC que no serán utilizados por nuestro sistema final, pero se considera interesante tener una referencia aproximada de todos estos aspectos.

Dentro de su corta vida, desde 1981 hasta la actualidad , el PC ha sufrido una constante evolución, casi siempre marcada por la compatibilidad software y hardware con todo lo anterior. Veamos seguidamente las etapas más importantes de esta evolución:

- El **PC**, así se denominó el IBM-PC original. Sus características básicas son:
 - CPU8088, con bus de datos de 8 bits y 1 Mbyte de direccionamiento de memoria.
 - Bus de expansión con bus datos de 8 bits y 16 bits de direcciones.
 - Un solo controlador de interrupciones (PIC 8259), con 8 canales, de los cuales 6 están disponibles en el bus.
 - Un solo controlador de DMA (8237A) con 4 canales, de los cuales 3 están disponibles en el bus de expansión.
 - Unidades de disquete para el almacenamiento masivo.
- El **XT**, es una arquitectura idéntica a la del PC original pero con la incorporación de un disco duro para el almacenamiento masivo. Los programas BIOS para el control del disco duro se encuentran en ROM situada físicamente en la tarjeta controladora de disco que se sitúa en el mapa de memoria entre las posiciones C8000 Y CFFF.
- El **AT**, fue la primera gran innovación en la familia. Convierte al PC/XT original en un ordenador de 16 bits. Sus características más destacadas son las siguientes:
 - CPU 80286, con bus de datos de 16 bits y 16 Mbytes de direccionamiento de memoria.
 - Bus de expansión ampliado con bus de datos de 16 bits y 24 bits de direcciones.
 - Dos controladores de interrupciones (PIC 8259) con 16 canales, de los que 11 están disponibles en el bus de expansión.
 - Dos controladores de DMA(8237A) con 8 canales, de los cuales 7 están disponibles en el bus de expansión.
 - Incorpora RAM CMOS alimentada por batería, lo que le permite almacenar la configuración cuando el equipo está apagado.

- Incorpora RELOJ DE TIEMPO REAL, también alimentado por baterías, lo que permite que el equipo conserve la fecha y hora durante los intervalos en que está apagado.
- El **AT-386**, básicamente conserva las características de su predecesor, el AT, pero su CPU es un 386. Obsérvese que el 386 es un micro de 32 bits, sus buses de datos y direcciones son ahora de 32 bits, sin embargo conserva el mismo bus de expansión que el AT.
- De aquí en adelante las modificaciones fundamentales han consistido en la variación del tipo de los buses y el aumento de tamaño del ancho de los mismos.

5.5.1 Compatibilidad.

Toda la gama de ordenadores surgidos a partir del PC de IBM se ha caracterizado siempre por su compatibilidad hacia atrás. Esto quiere decir que son capaces de ejecutar todos los programas escritos para ordenadores más antiguos. Resumiendo el punto anterior para que un ordenador sea considerado como un IBM-PC-Compatible este debe tener al menos los siguientes elementos hardware:

- Una CPU compatible con el 8086/88.
- Dos controladores de interrupciones tipo 8259 para la compatibilidad con AT.
- Dos controladores de DMA.
- Un subsistema de temporización realizado entorno a la unidad de timer i8253.
- Memoria RAM instalada desde la posición 00000 hacia arriba.
- Memoria ROM conteniendo el BIOS instalada desde la posición FFFF hacia abajo. La BIOS debe contener los servicios básicos de acceso a los elementos hardware a través de interrupciones software.
- Un elemento que en principio era opcional es el coprocesador matemático, que originalmente tenía un zócalo reservado en la placa madre, pero que a partir del 486, este elemento está incluido en la misma pastilla que la CPU.

Todos estos elementos deben estar interconectados de una determinada forma, como por ejemplo, el temporizador 0 debe estar conectado a la IRQ_0.

5.5.2 Diagrama general de bloques del PC.

Se intentará en este punto explicar todos los bloques más significativos que comprenden un PC, seguidamente se hace una breve descripción de cada uno de ellos con el fin de lograr una visión de conjunto.

CPU.

Este es el lugar ocupado por el microprocesador que constituye el corazón del PC. Originalmente se han venido usando microprocesadores Intel, pero también se usan CPUs de otros fabricantes como NEC, AMD y CIRYX.

El coprocesador numérico.

En principio nació como un elemento opcional. Casi todos los PCs tenían la posibilidad de instalarlo, para lo que disponían de un zócalo en la placa madre del ordenador. A partir de del procesador 486, este elemento está incluido en la misma pastilla que la CPU.

La presencia del coprocesador acelera notablemente la velocidad de ejecución de determinados programas preparados para utilizar este potente recurso. El coprocesador numérico trabaja en paralelo con la CPU, encargándose de ejecutar determinadas instrucciones de cálculo matemático, manejando datos de hasta 80 bits, conformes al estándar IEEE-754.

La memoria RAM.

La memoria constituye el espacio de trabajo de la CPU. Todos los programas que se están ejecutando y los datos que se van a manejar deben estar cargados en memoria. En la época en que se diseño el PC la memoria era un componente caro, y no se pensó que se pudieran llegar a las cantidades de memoria actuales, por lo que se previó una memoria de hasta 640 KB. Con el paso de los años la memoria se ha abaratado y los 640 KB originales se han quedado desfasados, ello ha dado lugar a incorporar a la arquitectura del PC antigua dos métodos para la superación de esta barrera de los 640 KB. El primero denominado MEMORIA EXPANDIDA, que comprende la memoria entre los 640 KB y 1 MB. El segundo se denomina MEMORIA EXTENDIDA, que comprende la memoria por encima de 1 MB, para que la CPU pueda utilizarla debe entrar en el llamado Modo Protegido.

La memoria ROM.

Está constituida por una o varias pastillas ROM o EPROM, de contenido permanente, que contiene datos y programas indispensables para que el hardware funcione. La pastilla principal el BIOS (*Basic Input Output System*), o conjunto de rutinas elementales que manejan los elementos hardware del PC. La ROM-BIOS es la encargada del arranque del PC, de las rutinas de AUTOTEST, o pruebas del buen funcionamiento del hardware, y contiene un conjunto de servicios para el manejo del teclado, de la pantalla, discos, etc. El BIOS normaliza el PC asegurando su compatibilidad entre diferentes PCs, que pueden estar construidos con diversas alteraciones sobre el diseño original.

La ROM-BIOS debe ser suministrada por el propio fabricante del ordenador, único conocedor a fondo del funcionamiento del diseño electrónico de los circuitos.

La memoria cache.

Es una memoria de baja capacidad, típicamente desde 32 hasta 256 KB, construida con chips rápidos. No constituye un espacio de almacenamiento adicional. Se utiliza para acelerar los accesos a memoria, por lo que su presencia contribuye a mejorar la velocidad del ordenador. Este sistema se ha incorporado a los PCs a partir del 386.

El controlador de DMA.

El DMA (*Direct Memory Access*), es un procedimiento de intercambio de datos entre la memoria y periféricos más eficaz que el que se puede ejecutar por un programa ejecutado por la CPU. En los PCs se utiliza fundamentalmente para la transferencia de datos entre la memoria y los discos. Además puede usarse para dar servicio a tarjetas especiales de comunicaciones, de adquisición de datos y otras varias.

El PC original incorporó a su arquitectura un pastilla 8237, capaz de controlar 4 canales de transferencia DMA. Uno de ellos lo dedicó precisamente al REFRESCO DE RAM DINAMICA, dejando otros 3 libres, que se usaron para los discos. A partir del AT la arquitectura se mejoró con un 8237 adicional, dando un total de 7 canales posible. Además el refresco de memoria se transfirió a otros circuitos específicos, con lo que se dejó libre un canal.

El controlador de interrupciones.

El mecanismo de las interrupciones es básico en el funcionamiento del PC. Todos los servicios del BIOS y los que añade el sistema operativo se manejan mediante llamadas a interrupciones software o hardware. El controlador de interrupciones controla todas las interrupciones hardware. En la versión original estaba constituido por una pastilla 8259(PIC, *Peripheral Interrupt Controller*). A partir del AT se amplía a dos patillas de este tipo conectadas en cascada, lo que permite hasta 15 entradas de interrupción. De ellas algunas quedan reservadas para funciones básicas del PC. Las restantes entradas están destinadas a las tarjeta enchufables que se puedan instalar. Algunas tienen una función predefinida (disco duro, disquete, comunicaciones,...), pero siempre quedan algunas libres.

Los temporizadores.

La arquitectura original del PC incorporó tres temporizadores de 16 bits, todos ellos con la misma frecuencia 4.77/4 MHz, implementados mediante una pastilla 8253, para realizar tres funciones específicas. La primera de ellas es la generación de interrupciones periódicas (18.2 int/seg) para el mantenimiento de la hora del sistema. La segunda se utiliza para el mantenimiento del refresco de memoria dinámico, ya mencionado. Por último, la tercera se destinó a la generación de sonidos a través del pequeño altavoz que siempre incorpora el PC.

El teclado.

El teclado es un elemento totalmente independiente del PC, que se comunica a través de una línea específica. El teclado, propiamente dicho, a parte de sus elementos visibles, contiene un microcontrolador específico, generalmente de la familia 8051 de Intel, que se encarga de la exploración de la matriz de contactos en la que están constituidas las teclas. Sus circuitos internos se alimentan de una tensión de 5V, procedentes del PC, que se suministra a través del cable que conecta el teclado con el PC.

El teclado envía al PC información sobre qué tecla se ha pulsado o liberado. Es lo que se llaman SCAN CODES, que no tiene nada que ver con el CODIGO ASCII de la tecla pulsada. Esto permite que una misma implementación física del teclado pueda ser utilizada, simplemente con una alteración de la serigrafía de las teclas, para

diferentes idiomas. En el otro extremo del cable del teclado, es decir en el PC, es necesario un circuito que reciba la señal y que genere una interrupción por cada tecla pulsada.

El reloj de tiempo real y RAM-CMOS.

Se trata de otra de las mejoras que introdujo el AT. En el AT se incorporó un pastilla de reloj alimentada por batería, es decir, que seguía funcionando aunque el ordenador permaneciese apagado. Esta pastilla, además, suele tener una pequeña memoria RAM, capaz de almacenar los datos básicos sobre la configuración del ordenador.

Durante el proceso de arranque del PC, se ejecuta el BIOS, y se da la opción al usuario de entrar en el programa de configuración o SETUP. Durante este programa, que depende del BIOS, es posible informar al PC de que tipo de accesorios está dotado. Todos los datos que el usuario introduzca serán almacenados en la RAM-CMOS, memoria de bajo consumo.

El bus de expansión.

Se trata de un conjunto de conectores idénticos, o casi idénticos, en los que se pueden conectar diversas tarjetas enchufables opcionales. A través de este bus se maneja un conjunto normalizado de señales: bus de datos, bus de direcciones, interrupciones, protocolo de DMA, relojes, etc...

A lo largo de la historia del PC, este bus ha sufrido diversas variaciones para irse adaptando a la evolución de la tecnología. El primitivo bus de expansión ISA (*Industry Standard Adapter*). Este bus recibió una ampliación cuando se diseño el AT, que podemos llamar ISA-AT, posteriormente han ido saliendo multitud de buses nuevos como MCA, EISA, VL, PCI, USB, etc., que no entraremos a estudiar en el presente trabajo.

Las tarjetas enchufables.

El sistema de tarjetas enchufables dota de una gran versatilidad al PC y es una de las claves de que, pasados 20 desde su diseño, haya podido soportar los cambios tecnológicos habidos en este periodo. Una de las claves del éxito de la gama PC está basada en que el ordenador podía ser ampliado mediante tarjetas de expansión. Así se pueden encontrar tarjetas muy diferentes para aplicaciones muy diferentes.

- Tarjetas de ampliación de memoria.
- Tarjetas controladoras de video.
- Tarjetas con puertos serie y paralelos.
- Tarjetas controladoras de discos y disquetes.
- Tarjetas de comunicaciones.
- Tarjetas específicas de diversos periféricos.
- Tarjetas para la construcción de prototipos.
- Tarjetas para la realización de medidas y adquisición de datos.
- Tarjetas para el procesado digital de señales.
- Tarjetas para aplicaciones multimedia.

Capítulo 6

Adaptación de ORK a la plataforma i386-PC

6.1 Introducción.

En este capítulo se hace mención a la adaptación del entorno ORK a la plataforma i386-PC. Originalmente ORK se ejecuta sobre un microprocesador ERC-32 el cual es un SPARC v7 resistente a la radiación, cuya versión convencional hace ya mucho tiempo que no se utiliza. La finalidad de portar el sistema a una plataforma convencional y bien conocida como es i386-PC es la de poder utilizar el entorno tanto en ambientes educativos como en ambientes industriales que utilicen la plataforma i386-PC en sus aplicaciones.

Como principal objetivo se plantea el hecho de intentar modificar únicamente las partes de ORK dependientes de la máquina, intentando alterar lo menos posible la estructura del sistema original. Teniendo en cuenta lo anterior, las partes imprescindibles que ha habido que modificar son las siguientes.

- Arranque del sistema.
- Inicialización del sistema.
- Gestión de interrupciones externas y traps.
- Soporte de Entrada/Salida.
- Gestión de tiempos mediante temporizadores (TIC).
- Soporte para cambios de contexto.
- Gestión de prioridades.
- Soporte de depuración con GDB.
- Ejecución del sistema

Para la adaptación de entorno se ha tomado como principal referencia el sistema OSKit: The Flux Operating System Toolkit, que como su nombre indica es una caja de

herramientas consistente en componentes modulares y código de librerías necesario para la construcción de núcleos de sistemas operativos, servidores y todo tipo de funcionalidades que pueda incorporar un sistema operativo. También se ha tomado como referencia el sistema MaRTe OS de la Universidad de Cantabria y el sistema RTEMS para aspectos puntuales relacionados con las interrupciones.

Este proyecto, tanto la versión de ORK original como la versión i386-PC, se desarrolla bajo licencia GNU, es decir software de libre distribución, todo el entorno cruzado de desarrollo esta basado en herramientas GNU, como son el compilador `gcc` o el depurador `gdb`.

Los paquetes incluidos en la distribución de ORK-i386 son los siguientes:

- Los fuentes del sistema ORK (Open Ravenscar Real Time Kernel).
- La versión de GNAT 3.13 adaptada a ORK para el sistema de compilación cruzado sobre i386. Esta versión esta compuesta de los siguientes paquetes:
 - Los fuentes de GNAT 3.13 con los parches para ORK, y versiones especiales de los paquetes que componen GNARL(GNU Ada Runtime Library) y todo el GNULL(GNU Lower Level).
 - Los fuentes de binutils-2.9.1 con los parches para ORK-i386.
 - Los fuentes de newlib-1.8.2 con los parches para ORK-i386.
 - Los fuentes de gcc-2.8.1 con los parches para ORK-i386.
- La versión de GDB 4.17.

6.2 Estructura de directorios de la distribución de ORK-i386.

Se recomienda ver el apéndice B, en donde se explica como instalar el entorno ORK-i386. Suponemos que tanto los fuentes como los binarios se encuentran en el directorio **/usr/local/openravenscar**. Aunque como se explica en el apéndice B para los binarios se puede elegir el lugar donde se instalan.

Contenido de **/usr/local/openravenscar**

bin: ejecutables.

demo: aplicación de demostración.

include: ficheros de cabecera.

lib: librerías de gcc las cuales incluyen la ORK-adalib para i386.

info: documentación de gcc en formato info.

man: páginas man.

i386-ork-elf: librería newlib (libc) para la familia i386.

src: fuentes de ORK y las herramientas relacionadas.

Contenido de **/usr/local/openravenscar/src**

binutils-2.9.1: fuentes adaptadas de binutils para ORK.

newlib-1.8.2: fuentes adaptadas de newlib para ORK.

gcc-2.8.1: fuentes adaptadas gcc para ORK.

gcc-2.8.1/ada: fuentes adaptadas de gnat-3.13 para ORK, incluido el propio ORK.

ORK: código de inicio y diversas rutinas de soporte para ORK.

Contenido de **/usr/local/openravenscar/src/ORK**

include: ficheros de cabecera para el código de inicio y soporte.

boot: código de inicio, soporte para depuración cruzada, interrupciones, traps, inicialización, y soporte de entrada/salida.

libmc: librería de que da soporte al código de inicio.

ork_init: código previo a la ejecución del runtime de Ada, incluido el manejo de excepciones.

A lo largo de la explicación de cómo se ha llevado a cabo el *porting* se hará referencia explícita a los fichero que implementa la funcionalidad de cada aspecto concreto.

6.3 Modificación de los paquetes de ORK.

Como se vio en el capítulo 5, ORK se compone de una serie de paquetes propios que constituyen la implementación del sistema, además de estos también se compone de paquetes del runtime de Ada95 que han sido modificados. De estos paquetes no ha sido necesario modificar todos, los siguientes paquetes no han sufrido modificaciones.

- System.
- System.OS_Primitives.
- System.Task_Primitives.
- System.Task_Primitives.Operations.
- System.Tasking.Protected_Objects.Single_Entry.
- Kernel.Threads.
- Kernel.Threads.ATCB.
- Kernel.Threads.Queues.
- Kernel.Threads.Protection.

El resto de paquetes han sufrido modificaciones en mayor o menor medida, en los puntos posteriores se expondrá en detalle los paquetes que han sufrido alguna modificación:

- Ada.Interrupts.Names.
- System.Interrupts.

- System.OS_Interface.
- System.Task_Primitives.Operations.
- Kernel.Parameters.
- Kernel.Interrupts.
- Kernel.Time.
- Kernel.CPU_Primitives.
- Kernel.Memory.
- Kernel.Peripherals.
- Kernel.Peripherals.Registers.
- Kernel.Serial_Output.

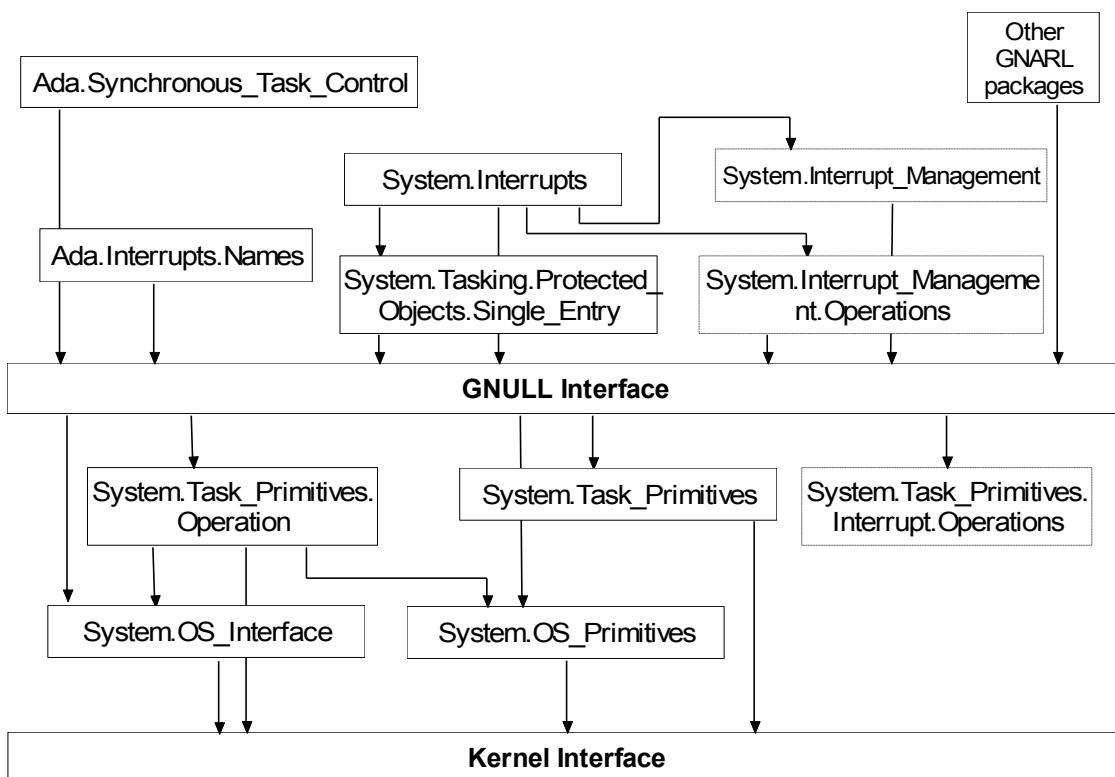


Figura 6.0 : Diagrama de los paquetes de GNARL modificados en ORK.

6.3.1 Ada.Interrupts.Names.

El paquete Ada.Interrupts.Names contiene los nombres de las interrupciones que reconoce el sistema. Evidentemente ahora los nombres de las interrupciones se corresponden con los de la arquitectura i386-PC:

```

with Ada.Interrupts;
-- used for Interrupt_ID

with System.OS_Interface;
  
```

```
-- used for names of interrupts

package Ada.Interrupts.Names is

-----
-- Reserved Interrupts --
-----

RESERVED1_IRQ : constant Interrupt_ID := System.OS_Interface.RESERVED1_IRQ;
RESERVED1_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 10;

RESERVED2_IRQ : constant Interrupt_ID := System.OS_Interface.RESERVED2_IRQ;
RESERVED2_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 9;

RESERVED3_IRQ : constant Interrupt_ID := System.OS_Interface.RESERVED3_IRQ;
RESERVED3_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 8;

RESERVED4_IRQ : constant Interrupt_ID := System.OS_Interface.RESERVED4_IRQ;
RESERVED4_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 5;

-----
-- Timers Interrupts --
-----

TIMER_IRQ : constant Interrupt_ID := System.OS_Interface.TIMER_IRQ;
TIMER_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 15;

RTC_IRQ : constant Interrupt_ID := System.OS_Interface.RTC_IRQ;
RTC_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 12;

-----
-- Serial and Parallel Interrupts --
-----

SERIAL1_IRQ : constant Interrupt_ID := System.OS_Interface.SERIAL1_IRQ;
SERIAL1_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 4;

SERIAL2_IRQ : constant Interrupt_ID := System.OS_Interface.SERIAL2_IRQ;
SERIAL2_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 3;

PARALLEL1_IRQ : constant Interrupt_ID := System.OS_Interface.PARALLEL1_IRQ;
PARALLEL1_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First;

PARALLEL2_IRQ : constant Interrupt_ID := System.OS_Interface.PARALLEL2_IRQ;
PARALLEL2_IRQ_Priority : constant System Interrupt_Priority :=
  System Interrupt_Priority'First + 2;

-----
-- Miscelaneous Interrupts --
-----
```

```

KEYBOARD_IRQ : constant Interrupt_ID := System.OS_Interface.KEYBOARD_IRQ;
KEYBOARD_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 14;

CTLR2_IRQ : constant Interrupt_ID := System.OS_Interface.CTLR2_IRQ;
CTLR2_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 13;

DISKETTE_IRQ : constant Interrupt_ID := System.OS_Interface.DISKETTE_IRQ;
DISKETTE_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 1;

SOFT_IRQ : constant Interrupt_ID := System.OS_Interface.SOFT_IRQ;
SOFT_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 11;

COPROCESSOR_IRQ : constant Interrupt_ID := System.OS_Interface.COPROCESSOR_IRQ;
COPROCESSOR_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 7;

FIXED_DISK_IRQ : constant Interrupt_ID := System.OS_Interface.FIXED_DISK_IRQ;
FIXED_DISK_IRQ_Priority : constant System Interrupt_Priority :=
    System Interrupt_Priority'First + 6;

end Ada.Interrupts.Names;

```

6.3.2 System.OS_Interface.

El paquete System.OS_Interface hace de interfaz entre ORK y el runtime de Ada, lo que quiere decir que forma parte de GNULL, básicamente lo que hace es invocar directamente procedimientos de ORK. Los nombres de las interrupciones han sido modificados totalmente así como las prioridades asociadas a las interrupciones.

```

package System.OS_Interface is

    -- Identifiers for the underlying threads
    subtype Thread_Id      is Kernel.Threads.Thread_Id;

    -- Interrupt identifiers
    type Interrupt_ID is new Kernel.Interrupts Interrupt_ID;

    -- Names and priorities for each interrupt
    Interrupt_Name : constant := ...;
    Interrupt_Name _Priority : constant System Interrupt_Priority := ...;

    -- Obtain the software priority of any hardware interrupt. This makes
    -- easier the selection of the priority of the protected handler
    -- attached to interrupts.
    function Priority_Of_Interrupt (Level : Interrupt_ID) return

```

```

System.Any_Priority;

-- Attach a handler to a hardware interrupt.
procedure Attach_Handler (Handler : System.Address;
                           Id : Interrupt_ID);
pragma Inline (Attach_Handler);

-- Detach the handlers previously attached to the interrupt.
procedure Detach_Handler (Id : Interrupt_ID);
pragma Inline (Detach_Handler);

-- Return the address of the handler currently used for the specified
-- interrupt.
function Current_Handler (Id : Interrupt_ID) return System.Address;
pragma Inline (Current_Handler);

end System.OS_Interface;

```

6.3.3 System.Interrupts.

Este paquete contiene las definiciones dependientes de la implementación para el soporte de interrupciones. Dentro del cuerpo de este paquete se reservan las interrupciones que están asignadas de antemano, concretamente para la arquitectura i386-PC, las únicas interrupciones que hay que han sido reservadas son las correspondientes al timer y al reloj de tiempo real. Únicamente ha sido necesario esta pequeña modificación.

6.3.4 Kernel.Interrupts.

Este paquete corresponde a ORK, la única modificación que ha sufrido tiene que ver con el parámetro que recibe la función que envuelve “Wrapper” al manejador de interrupción, que antes recibía como parámetro el número de interrupción y ahora recibe una estructura más compleja llamada “Trap_State”, que se encuentra especificada en el paquete Kernel.CPU_Primitives. El paquete conserva el mismo interfaz, solo se han hecho cambios a nivel del cuerpo.

```

package Kernel.Interrupts is

  -- The interrupts are distinguished by its interrupt level.
  subtype Interrupt_ID is Natural range
    0 .. Kernel.Parameters.Interrupt_Levels;

  -- Hardware interrupts (PCs have 2 PICs with 8 IRQ each)

  -- Attach the procedure Handler as handler of the interrupt Id.
  procedure Attach_Handler (Handler : System.Address;
                            Id : Interrupt_ID);

  -- The interrupt handler is detached from the interrupt.
  procedure Detach_Handler (Id : Interrupt_ID);

  -- Return the address of the handler currently attached to the
  -- interrupt specified as argument of the function.
  function Current_Handler (Id : Interrupt_ID) return System.Address;

end Kernel.Interrupts;

```

El paquete define el tipo (**Interrupt_ID**), que identifica a cada interrupción en un rango de 0 a Kernel.Parameters.Interrupt_Levels.

La operaciones que provee este paquete son **Attach_Handler**, **Detach_Handler**, **Current_Handler**. Su semántica y funcionamiento es el mismo que la versión original de ORK.

6.3.5 Kernel.CPU_Primitives.

Es paquete contiene rutinas que aíslan aspectos concretos del procesador, con lo cual se ha reescrito de nuevo.

```

package Kernel.CPU_Primitives is

    package KPa renames Kernel.Parameters;
    package KPe renames Kernel.Peripherals;

    type Context_Buffer is private;
    -- This type contains the saved register set for each thread.

    -----
    -- TRAP --
    -----

    type Trap_State is record -- PC
        MASCARA : KPe.Unsigned_32;
        GS : KPe.Unsigned_32;
        FS : KPe.Unsigned_32;
        ES : KPe.Unsigned_32;
        DS : KPe.Unsigned_32;

        EDI : KPe.Unsigned_32;
        ESI : KPe.Unsigned_32;
        EBP : KPe.Unsigned_32;
        CR2 : KPe.Unsigned_32;
        EBX : KPe.Unsigned_32;
        EDX : KPe.Unsigned_32;
        ECX : KPe.Unsigned_32;
        EAX : KPe.Unsigned_32;

        TRAPNO : KPe.Unsigned_32;

        ERR : KPe.Unsigned_32;

        EIP : KPe.Unsigned_32;
        CS : KPe.Unsigned_32;
        EFlags : KPe.Unsigned_32;
        ESP : KPe.Unsigned_32;
        SS : KPe.Unsigned_32;

        V86_ES : KPe.Unsigned_32;
        V86_DS : KPe.Unsigned_32;
        V86_FS : KPe.Unsigned_32;
        V86_GS : KPe.Unsigned_32;
    end record;
    pragma Convention (C, Trap_State);
    type Trap_State_Ac is access Trap_State;

    type Trap_Handler_Type is access
        function (State : in Trap_State_Ac) return Integer;

```

```

-----
-- IRQ --
-----

type IRQ is new KPe.Integer_32 range 0 .. 15;
for IRQ'Size use 32;

type IRQ_Handler_Type is access
    procedure (state : in Trap_State_Ac);

procedure Context_Switch (Current : System.Address;
                           Next     : System.Address;
                           Running_Thread_Id : System.Address);

procedure Initialize_Context (Buffer : access Context_Buffer;
                               Program_Counter : System.Address;
                               Priority : System.Any_Priority;
                               Stack_Pointer : System.Address;
                               Stack_Size : Integer);

procedure Change_Priority (Buffer : access Context_Buffer;
                           Priority : System.Any_Priority);

procedure Install_Trap_Handler
    (Vector   : in KPa.Range_Of_Vector;
     Service_Routine : in Trap_Handler_Type);

procedure Install_Interrupt_Handler
    (Vector : in IRQ;
     Service_Routine : in IRQ_Handler_Type);

function Current_Handler (Vector : IRQ)
    return System.Address;

procedure Disable_Interrupts renames KPe.CLI;

procedure Enable_Interrupts (Level : in KPa.Interrupt_Level);

-----
-- Save and restore the "eflags" register --
-----
function Save_Flags return Integer renames KPe.Save_Flags;
pragma Inline (Save_Flags);

procedure Restore_Flags (EFlags : in Integer) renames KPe.Restore_Flags;
pragma Inline (Restore_Flags);

private
    subtype Range_Of_Context is Natural range 1 .. 40;

    type Context_Buffer is array (Range_Of_Context) of System.Address;
    for Context_Buffer'Alignment use Standard'Maximum_Alignment;

end Kernel.CPU_Primitives;

```

Los tipos definidos en este paquete son los siguientes:

Trap_State: Esta estructura define como se organiza la información que se almacena en la pila cada vez que se produce un trap o una interrupción. Máscara de interrupción en el momento de producirse la interrupción, descriptores de segmentos, registros generales, número de interrupción o trap, flags y contador de programa fundamentalmente.

Context_Buffer: En el buffer de contexto se almacena el contexto de cada tarea (registros generales, registros de la unidad de coma flotante, puntero de pila, dir. de retorno, prioridad, flags y límites de la pila):

Trap_Handler_Type: Define el tipo que deben tener los manejadores de traps.

IRQ_Handler_Type: Define el tipo que deben tener los manejadores de interrupciones.

IRQ: El tipo IRQ, define un entero en el rango de las interrupciones (0..15).

Las operaciones que provee este paquete son las siguientes:

Context_Switch: La rutina de cambio de contexto, salva en el contexto de la tarea actual en su buffer de contexto, y restaura el contexto de la tarea siguiente, además actualiza el valor de la variable Running_Thread, con el valor del parámetro Running_Thread_Id.

Initialize_Context: Este procedimiento inicializa los valores fundamentales del contexto de un thread, como son el contador de programa, la prioridad y los parámetros de la pila.

Change_Priority: Este nuevo procedimiento sirve para cambiar el valor de la prioridad de una tarea, en su buffer de contexto. Dadas las características del perfil Ravenscar, que no permite la modificación dinámica de las prioridades de las tareas. Este procedimiento solo se ejecuta cuando se le asigna una prioridad a una tarea en su declaración a través de un pragma.

Install_Trap_Handler: Este procedimiento instala la rutina pasada como parámetro como manejador del trap especificado en la llamada.

Install Interrupt Handler: Al igual que la llamada anterior, instala un manejador de interrupciones.

Current_Handler: Devuelve la dirección del manejador de una interrupción.

Disable_Interrupts: Este procedimiento inhabilita todas las interrupciones externas enmascarables.

Enable_Interrupts: Este procedimiento habilita las interrupciones externas al nivel que se indique en la llamada.

Save_Flags: Esta función devuelve los flags actuales.

Restore_Flags: Este procedimiento restaura los flags que se pasan como parametro.

6.3.6 Kernel.Time.

El paquete Kernel.Time contiene las funcionalidades del reloj y los retardos. Desde el punto de vista del interfaz no ha sufrido modificaciones, sus modificaciones más importantes se encuentran en la implementación debido a las diferencias en el número y tipo de temporizadores entre ERC-32 y el PC.

```
package Kernel.Time is

  -- Time is represented at this level as a 64-bit integer number of
  -- nanoseconds. The interval of time values that can be represented in
  -- this way is approximately -292 .. +292 years or -106_752 .. +106_752
  -- days.
  type Time is range -(2**63) .. +(2**63 - 1);

  -----
  -- Clock      --
  -----
  function Clock  return Time;

  Tick : constant Time := Time (1_000.0 / Kernel.Parameters.Clock_Frequency);

  -----
  -- Delay until --
  -----
  procedure Delay_Until (T : Time);

end Kernel.Time;
```

Los tipos y constantes definidos en este paquete son:

Time: El tipo time define el número de nanosegundos transcurridos desde el inicio del sistema.

Tick: La constante tick define el número de milisegundos que dura un tick del temporizador.

Las operaciones que provee este paquete son:

Clock: Devuelve el valor actual del reloj del sistema, dicho valor está expresado en nanosegundos, desde que se arranco el sistema.

delay_Until: Este procedimiento es la primitiva básica para realizar esperas temporales.

6.3.7 Kernel.Memory.

Este paquete proporciona los limitados mecanismos de gestión de memoria. Dado que el perfil Ravenscar no permite la memoria dinámica, los recursos de este paquete solo se usan en la inicialización. Las modificaciones han sido mínimas, solo se ha modificado el

ancho de palabra de memoria de 64 bits a 32 bits, aunque el interfaz permanece inalterado.

```
package Kernel.Memory is

    function New_Memory_Region (Size : System.Parameters.Size_Type) return
        System.Address;

    function New_Stack (Size : System.Parameters.Size_Type) return
        System.Address;

end Kernel.Memory;package Kernel.Memory is
```

Las operaciones que provee este paquete son:

New_memory_Region: Actualmente no tiene ninguna funcionalidad.

New_Stack: Reserva espacio de pila para las tareas.

6.3.8 Kernel.Peripherals.

Este paquete contiene definiciones de tipos y rutinas de manejo de los periféricos y dispositivos de la arquitectura i386-PC, realmente su contenido va más allá del significado que tenía en la versión original, pero dado que no se han querido crear más paquetes de los que tenía el ORK original, en este paquete se han incluido otras operaciones de bajo nivel del i386.

```
package Kernel.Peripherals is

    package KPa renames Kernel.Parameters;
    -----
    -- IO_Port --
    -----
    type IO_Port is range 0 .. 16#FFFF#; -- I/O directions
    for IO_Port'Size use 16;

    -----
    -- TIPO ENTEROS --
    -----
    type Integer_8 is range -2 ** 7 .. 2 ** 7 - 1;
    for Integer_8'Size use 8;

    type Integer_16 is range -2 ** 15 .. 2 ** 15 - 1;
    for Integer_16'Size use 16;

    type Integer_32 is range -2 ** 31 .. 2 ** 31 - 1;
    for Integer_32'Size use 32;

    type Integer_64 is range -2 ** 63 .. 2 ** 63 - 1;
    for Integer_64'Size use 64;

    type Unsigned_8 is mod 2 ** 8;
    for Unsigned_8'Size use 8;

    type Unsigned_16 is mod 2 ** 16;
    for Unsigned_16'Size use 16;

    type Unsigned_32 is mod 2 ** 32;
```

```
for Unsigned_32'Size use 32;

type Unsigned_64 is mod 2 ** 64;
for Unsigned_64'Size use 64;

-- C types
subtype Int    is Integer;
subtype Short is Short_Integer;
subtype Long   is Long_Integer;

type Signed_Char is range -2 ** 7 .. 2 ** 7 - 1;
for Signed_Char'Size use 8;

type Unsigned      is mod 2 ** Int'Size;
for Unsigned'Size use Int'Size;
type Unsigned_Short is mod 2 ** Short'Size;
for Unsigned_Short'Size use Short'Size;
type Unsigned_Long is mod 2 ** Long'Size;
for Unsigned_Long'Size use Long'Size;

type Unsigned_Char is mod 2 ** Signed_Char'Size;
for Unsigned_Char'Size use Signed_Char'Size;

subtype Plain_Char is Unsigned_Char;

subtype Size_T is Int;

-- Operations
function Shift_Left
  (Value : Unsigned_8;
   Amount : Natural)
  return Unsigned_8;

function Shift_Right
  (Value : Unsigned_8;
   Amount : Natural)
  return Unsigned_8;

function Shift_Right_Arithmetic
  (Value : Unsigned_8;
   Amount : Natural)
  return Unsigned_8;

function Rotate_Left
  (Value : Unsigned_8;
   Amount : Natural)
  return Unsigned_8;

function Rotate_Right
  (Value : Unsigned_8;
   Amount : Natural)
  return Unsigned_8;

function Shift_Left
  (Value : Unsigned_16;
   Amount : Natural)
  return Unsigned_16;

function Shift_Right
  (Value : Unsigned_16;
   Amount : Natural)
  return Unsigned_16;

function Shift_Right_Arithmetic
  (Value : Unsigned_16;
   Amount : Natural)
  return Unsigned_16;

function Rotate_Left
  (Value : Unsigned_16;
```

```

        Amount : Natural)
      return Unsigned_16;

function Rotate_Right
  (Value  : Unsigned_16;
   Amount : Natural)
  return Unsigned_16;

function Shift_Left
  (Value  : Unsigned_32;
   Amount : Natural)
  return Unsigned_32;

function Shift_Right
  (Value  : Unsigned_32;
   Amount : Natural)
  return Unsigned_32;

function Shift_Right_Arithmetic
  (Value  : Unsigned_32;
   Amount : Natural)
  return Unsigned_32;

function Rotate_Left
  (Value  : Unsigned_32;
   Amount : Natural)
  return Unsigned_32;

function Rotate_Right
  (Value  : Unsigned_32;
   Amount : Natural)
  return Unsigned_32;

function Shift_Left
  (Value  : Unsigned_64;
   Amount : Natural)
  return Unsigned_64;

function Shift_Right
  (Value  : Unsigned_64;
   Amount : Natural)
  return Unsigned_64;

function Shift_Right_Arithmetic
  (Value  : Unsigned_64;
   Amount : Natural)
  return Unsigned_64;

function Rotate_Left
  (Value  : Unsigned_64;
   Amount : Natural)
  return Unsigned_64;

function Rotate_Right
  (Value  : Unsigned_64;
   Amount : Natural)
  return Unsigned_64;

pragma Import (Intrinsic, Shift_Left);
pragma Import (Intrinsic, Shift_Right);
pragma Import (Intrinsic, Shift_Right_Arithmetic);
pragma Import (Intrinsic, Rotate_Left);
pragma Import (Intrinsic, Rotate_Right);

type Timer_Interval is mod KPa.Clock_Interrupt_Period - 1;
for Timer_Interval'Size use 64;

```

```

procedure Set_Alarm (nanoseconds : in Timer_Interval;
                     Next_Activation : out Integer_64;
                     Total_Time       : in out Integer_64);

function Read_Clock return Timer_Interval;

procedure Cancel_And_Set_Alarm (nanoseconds      : in Timer_Interval;
                                 Next_Activation : out Integer_64;
                                 Total_Time      : in out Integer_64);

procedure Clear_Alarm_Interrupt;

-- Function to obtain the priority of an interrupt.
function Priority_Of_Interrupt (Level : KPa.Interrupt_Level) return
    System.Any_Priority;

procedure Init_Board (Total_Time : out Integer_64);

-----
-- FUNCIONES DE ENTRADA SALIDA --
-----
-- Read byte from port
function Inb (Port : in IO_Port) return Unsigned_8;

-- Read byte from port with delay
function Inb_P (Port : in IO_Port) return Unsigned_8;

-- Read word (2 bytes) from port
function Inw (Port : in IO_Port) return Unsigned_16;

-- Read word (2 bytes) from port with delay
function Inw_P (Port : in IO_Port) return Unsigned_16;

-- Read long word (4 bytes) from port
function Inl (Port : in IO_Port) return Unsigned_32;

-- Read long word (4 bytes) from port with delay
function Inl_P (Port : in IO_Port) return Unsigned_32;

-- Write in port
procedure Outb (Port : in IO_Port; Val : in Unsigned_8);

-- Write in port with delay
procedure Outb_P (Port : in IO_Port; Val : in Unsigned_8);

-- Write word (2 bytes) in port
procedure Outw (Port : in IO_Port; Val : in Unsigned_16);

-- Write word (2 bytes) in port with delay
procedure Outw_P (Port : in IO_Port; Val : in Unsigned_16);

-- Write long word (4 bytes) in port
procedure Outl (Port : in IO_Port; Val : in Unsigned_32);

-- Write long word (4 bytes) in port with delay
procedure Outl_P (Port : in IO_Port; Val : in Unsigned_32);

-----
-- IRQ-INTERRUPCIONES --
-----
TIMER_IRQ      : constant KPa.Interrupt_Level := 0;
KEYBOARD_IRQ   : constant KPa.Interrupt_Level := 1;
CTLR2_IRQ     : constant KPa.Interrupt_Level := 2;
SERIAL2_IRQ   : constant KPa.Interrupt_Level := 3;
SERIAL1_IRQ   : constant KPa.Interrupt_Level := 4;
PARALLEL2_IRQ : constant KPa.Interrupt_Level := 5;
DISKETTE_IRQ  : constant KPa.Interrupt_Level := 6;
PARALLEL1_IRQ : constant KPa.Interrupt_Level := 7;
RTC_IRQ        : constant KPa.Interrupt_Level := 8;

```

```

SOFT_IRQ      : constant KPa.Interrupt_Level := 9;
RESERVED1_IRQ : constant KPa.Interrupt_Level := 10;
RESERVED2_IRQ : constant KPa.Interrupt_Level := 11;
RESERVED3_IRQ : constant KPa.Interrupt_Level := 12;
COPROCESSOR_IRQ : constant KPa.Interrupt_Level := 13;
FIXED_DISK_IRQ : constant KPa.Interrupt_Level := 14;
RESERVED4_IRQ : constant KPa.Interrupt_Level := 15;

-----
-- PIC --
-----

-- Initialize both PICs
procedure PICs_Init (Master_Base : in Unsigned_8;
                      Slave_Base : in Unsigned_8);

-- Enable PICs
procedure PIC_Master_Enable_Irq;

procedure PIC_Slave_Enable_Irq;

-- Unmask IRQs
procedure PIC_Master_Unmask IRQs;
procedure PIC_Slave_Unmask IRQs;

-----
-- Interruption_Level --
-----

procedure Interruption_Level (Priority : in KPa.Interrupt_Level);

-----
-- Mostrar_Level --
-----

procedure Mostrar_Level;

-----
-- Are Interrupts_Enable --
-----

function Are Interrupts Enabled return Boolean;

-----
-- Disable-Enable interrupts --
-----

procedure CLI;

procedure STI;

-----
-- Save and restore the "eflags" register --
-----

function Save_Flags return Integer;

procedure Restore_Flags (EFlags : in Integer);

-----
-- Salida a traves de consola --
-----

function Puts (Str_Addr : in System.Address) return Integer;
procedure Put (Str : in String);
procedure Put (Addr : in System.Address);
procedure Put (Int : in Integer; Base : in Integer := 10);
procedure Put (Int : in Integer_32; Base : in Integer := 10);
procedure Put (Int : in Integer_8; Base : in Integer := 10);
procedure Put (Int : in Unsigned_64; Base : in Integer := 10);
procedure Put (Int : in Unsigned_32; Base : in Integer := 10);
procedure Put (Int : in Unsigned_8; Base : in Integer := 10);
procedure Put (C : in Character);
procedure New_Line;

```

```

-- Init_Serial_Communication_With_Gdb --
-----
Serial_Port_1 : constant Integer := 1;
Serial_Port_2 : constant Integer := 2;
procedure Init_Serial_Communication_With_Gdb (Com_Port : in Integer);

-----
-- Set_Break_Point_Here --
-----
procedure Set_Break_Point_Here;

-----
-- SALIDA POR LINEA SERIE --
-----

procedure Serial_Cons_Init (Port : in Integer_32);
procedure Com_Putchar (Port : in Integer_32;
                      C    : in Character);

end Kernel.Peripherals;

```

Dado el tamaño y complejidad que ha adquirido este paquete se resume seguidamente el contenido de dicho paquete.

- Definiciones de tipos para puertos (**IO_Port**), enteros con signo (**Integer**) y sin signo (**Unsigned**), funciones de manejo de bits (**Shift_Left**, **Shift_Right**).
- Funciones de entrada/salida a los puertos(**In**, **Out**).
- Funciones de salida por consola(**Put**, **New_Line**).
- Funciones de soporte para la salida por línea serie:
 - **Serial_Cons_Init**: Inicialización de línea serie.
 - **Com_Putchar**: Envío de un carácter por la línea serie.
- Funciones para la depuración por línea serie.
 - **Init_Serial_Communication_With_Gdb**: Este procedimiento inicializa una línea serie para depuración.
 - **Set_Break_Point_Here**: Este procedimiento provoca un break-point que posteriormente recogerá GDB.
- Definición de los nombres de las interrupciones y de las principales funciones relacionadas con las interrupciones y las prioridades de interrupción:
 - **PICs_Init**: Este procedimiento inicializa los controladores de interrupciones con los valores adecuados.
 - **PIC_Master_Enable_Irq** y **PIC_Slave_Enable_Irq**: Estos procedimientos activan los PICs para la recepción de interrupciones.
 - **PIC_Master_Unmask IRQs** y **PIC_Slave_Unmasks IRQs**: Estos procedimientos ponen las máscaras de los PICs a 0.
 - **Are Interrupts Enable**: Esta función comprueba si están habilitadas las interrupciones.

- **Interruption_Level:** Este procedimiento coloca la máscara adecuada en los PICs de acuerdo al nivel de interrupción que se pasa como parámetro.
 - **Mostrar_Level:** Este procedimiento muestra los valores actuales de las máscaras de los PICs.
 - **CLI** y **STI:** Estos procedimientos son equivalentes a las mismas instrucciones en ensamblador. Habilitan y deshabilitan las interrupciones a través del bit IF en los flags.
 - **Priority_Of_Interrupt:** Esta función devuelve la prioridad de la interrupción que se pasa como parámetro.
- Procedimiento para la inicialización de valores del sistema (**Init_Board**), como el tiempo, y los controladores de interrupciones.
 - Funciones y definiciones relacionadas con el reloj y los timers, entre las fundamentales están:
 - **Read_Clock:** Esta función devuelve el valor actual del timer.
 - **Set_Alarm:** Este procedimiento arma el temporizador con el nuevo valor y actualiza la hora global del sistema.

6.3.9 Kernel.Peripherals.Registers.

El paquete Kernel.Peripherals.Registers originalmente contenía las definiciones de los registros para los periféricos y dispositivos del SPARC, dado que la arquitectura SPARC tiene la entrada/salida mapeada en memoria. Este no es el caso de la arquitectura Intel que tiene la entrada/salida en un espacio de direcciones distinto, con lo cual no se pueden definir registros para los periféricos de la forma que permite Ada. Los datos que se han definido en este paquete son las constantes referentes a los puertos de entrada/salida, y los valores de configuración de los PICs (*Programmable Interrupt Controller*), y el PIT(*Programmable Interval Timer*).

```
package Kernel.Peripherals.Registers is

-----
-- PIC --
-----
MASK_0 : constant Unsigned_8 := 16#00#;
MASK_1 : constant Unsigned_8 := 16#ff#;

-- The following are definitions used to locate the PICs in the system
Master_Pic_IObase : constant IO_Port := 16#20#;
Slave_Pic_IObase : constant IO_Port := 16#A0#;
Off_Icw           : constant IO_Port := 0;
Off_Ocw           : constant IO_Port := 1;

Master_Icw : constant IO_Port := Master_Pic_IObase + Off_Icw;
Master_Ocw : constant IO_Port := Master_Pic_IObase + Off_Ocw;
Slave_Icw   : constant IO_Port := Slave_Pic_IObase + Off_Icw;
Slave_Ocw   : constant IO_Port := Slave_Pic_IObase + Off_Ocw;

-- The following banks of definitions ICW1, ICW2, ICW3, and ICW4 are used
-- to define the fields of the various ICWs for initialisation of the PICs

-- ICW1
```

```
ICW_TEMPLATE : constant Unsigned_8 := 16#10#;

LEVEL_TRIGGER : constant Unsigned_8 := 16#08#;
EDGE_TRIGGER : constant Unsigned_8 := 16#00#;
ADDR_INTRVL4 : constant Unsigned_8 := 16#04#;
ADDR_INTRVL8 : constant Unsigned_8 := 16#00#;
SINGLE_MODE : constant Unsigned_8 := 16#02#;
CASCADE_MODE : constant Unsigned_8 := 16#00#;
ICW4_NEEDED : constant Unsigned_8 := 16#01#;
NO_ICW4_NEED : constant Unsigned_8 := 16#00#;

--      ICW3
--
SLAVE_ON_IR0 : constant Unsigned_8 := 16#01#;
SLAVE_ON_IR1 : constant Unsigned_8 := 16#02#;
SLAVE_ON_IR2 : constant Unsigned_8 := 16#04#;
SLAVE_ON_IR3 : constant Unsigned_8 := 16#08#;
SLAVE_ON_IR4 : constant Unsigned_8 := 16#10#;
SLAVE_ON_IR5 : constant Unsigned_8 := 16#20#;
SLAVE_ON_IR6 : constant Unsigned_8 := 16#40#;
SLAVE_ON_IR7 : constant Unsigned_8 := 16#80#;

I_AM_SLAVE_0 : constant Unsigned_8 := 16#00#;
I_AM_SLAVE_1 : constant Unsigned_8 := 16#01#;
I_AM_SLAVE_2 : constant Unsigned_8 := 16#02#;
I_AM_SLAVE_3 : constant Unsigned_8 := 16#03#;
I_AM_SLAVE_4 : constant Unsigned_8 := 16#04#;
I_AM_SLAVE_5 : constant Unsigned_8 := 16#05#;
I_AM_SLAVE_6 : constant Unsigned_8 := 16#06#;
I_AM_SLAVE_7 : constant Unsigned_8 := 16#07#;

--      ICW4
--
SNF_MODE_ENA : constant Unsigned_8 := 16#10#;
SNF_MODE_DIS : constant Unsigned_8 := 16#00#;
BUFFERD_MODE : constant Unsigned_8 := 16#08#;
NONBUFD_MODE : constant Unsigned_8 := 16#00#;
AUTO_EOI_MOD : constant Unsigned_8 := 16#02#;
NRML_EOI_MOD : constant Unsigned_8 := 16#00#;
I8086_EMM_MOD : constant Unsigned_8 := 16#01#;
SET_MCS_MODE : constant Unsigned_8 := 16#00#;

--      OCW1
--
PICM_MASK : constant Unsigned_8 := 16#FF#;
PICS_MASK : constant Unsigned_8 := 16#FF#;

--      OCW2
--
NON_SPEC_EOI : constant Unsigned_8 := 16#20#;
SPECIFIC_EOI : constant Unsigned_8 := 16#60#;
ROT_NON_SPEC : constant Unsigned_8 := 16#a0#;
SET_ROT_AEOI : constant Unsigned_8 := 16#80#;
RSET_ROTAEAOI : constant Unsigned_8 := 16#00#;
ROT_SPEC_EOI : constant Unsigned_8 := 16#e0#;
SET_PRIORITY : constant Unsigned_8 := 16#c0#;
NO_OPERATION : constant Unsigned_8 := 16#40#;

SEND_EOI_IR0 : constant Unsigned_8 := 16#00#;
SEND_EOI_IR1 : constant Unsigned_8 := 16#01#;
SEND_EOI_IR2 : constant Unsigned_8 := 16#02#;
SEND_EOI_IR3 : constant Unsigned_8 := 16#03#;
SEND_EOI_IR4 : constant Unsigned_8 := 16#04#;
SEND_EOI_IR5 : constant Unsigned_8 := 16#05#;
SEND_EOI_IR6 : constant Unsigned_8 := 16#06#;
SEND_EOI_IR7 : constant Unsigned_8 := 16#07#;
```

```

--      OCW3
--
OCW_TEMPLATE : constant Unsigned_8 := 16#08#;
SPECIAL_MASK : constant Unsigned_8 := 16#40#;
MASK_MDE_SET : constant Unsigned_8 := 16#20#;
MASK_MDE_RST : constant Unsigned_8 := 16#00#;
POLL_COMMAND : constant Unsigned_8 := 16#04#;
NO_POLL_CMND : constant Unsigned_8 := 16#00#;
READ_NEXT_RD : constant Unsigned_8 := 16#02#;
READ_IR_ONRD : constant Unsigned_8 := 16#00#;
READ_IS_ONRD : constant Unsigned_8 := 16#01#;

--      Standard PIC initialization values for PCs.
--
PICM_ICW1 : constant Unsigned_8 :=
    ICW_TEMPLATE or EDGE_TRIGGER or ADDR_INTRVL8 or CASCADE_MODE or
    ICW4_NEEDED;
PICM_ICW3 : constant Unsigned_8 := SLAVE_ON_IR2;
PICM_ICW4 : constant Unsigned_8 := SNF_MODE_ENA or NONBUFD_MODE or
    NRML_EOI_MOD or I8086_EMM_MOD;

PICS_ICW1 : constant Unsigned_8 := ICW_TEMPLATE or EDGE_TRIGGER or
    ADDR_INTRVL8 or CASCADE_MODE or ICW4_NEEDED;
PICS_ICW3 : constant Unsigned_8 := I_AM_SLAVE_2;
PICS_ICW4 : constant Unsigned_8 := SNF_MODE_ENA or NONBUFD_MODE or
    NRML_EOI_MOD or I8086_EMM_MOD;

-----
-- PIT --
-----

--
-- I/O port addresses of the PIT registers.

PIT_CNT0      : constant IO_Port := 16#40#; -- timer 0 counter port
PIT_CNT1      : constant IO_Port := 16#41#; -- timer 1 counter port
PIT_CNT2      : constant IO_Port := 16#42#; -- timer 2 counter port
PIT_CONTROL   : constant IO_Port := 16#43#; -- timer control port

--
-- Control register bit definitions

PIT_SEL0      : constant Unsigned_8 := 16#00#; -- select counter 0
PIT_SEL1      : constant Unsigned_8 := 16#40#; -- select counter 1
PIT_SEL2      : constant Unsigned_8 := 16#80#; -- select counter 2
PIT_INTC      : constant Unsigned_8 := 16#00#; -- mode 0, intr on
--                                terminal cnt
PIT_ONESHOT   : constant Unsigned_8 := 16#02#; -- mode 1, one shot
PIT_RATEGEN   : constant Unsigned_8 := 16#04#; -- mode 2, rate generator
PIT_SQWAVE    : constant Unsigned_8 := 16#06#; -- mode 3, square wave
PIT_SWSTROBE  : constant Unsigned_8 := 16#08#; -- mode 4, s/w triggered
--                                strobe
PIT_HWSTROBE  : constant Unsigned_8 := 16#0a#; -- mode 5, h/w triggered
--                                strobe
PIT_LATCH     : constant Unsigned_8 := 16#00#; -- latch counter for reading
PIT_LSB       : constant Unsigned_8 := 16#10#; -- r/w counter LSB
PIT_MSB       : constant Unsigned_8 := 16#20#; -- r/w counter MSB
PIT_16BIT     : constant Unsigned_8 := 16#30#; -- r/w counter 16 bits, LSB
--                                first
PIT_BCD       : constant Unsigned_8 := 16#01#; -- count in BCD

--
-- Constants for delayed reading ('rtlinux/rt_time.c')
PIT_LATCH_CNT0 : constant Unsigned_8 := 16#D2#; -- 1101 0010
PIT_LATCH_CNT2 : constant Unsigned_8 := 16#D8#; -- 1101 1000
PIT_LATCH_CNT0_2: constant Unsigned_8 := 16#DA#; -- 1101 1010

```

```

PIT_LATCH_CTR_CNT0_2 : constant Unsigned_8 := 2#1100_1010#;

--
-- Clock speed at which the standard interval timers in the PC are driven,
-- in hertz (ticks per second) and nanoseconds per tick, respectively.

PIT_HZ : constant Unsigned_64 := 1193182; -- ticks/sec
PIT_NS : constant Unsigned_64 := 1000000000 / PIT_HZ; -- ~838.1 nsec/tick

end Kernel.Peripherals.Registers;

```

6.3.10 Kernel.Serial_Output.

Este paquete contiene el interfaz para hacer la salida de texto por la línea serie en vez de por la pantalla. El mecanismo de la salida por línea serie varía un poco con respecto al original, ya que ahora es necesario iniciar la línea explícitamente antes de hacer cualquier Put.

```

package Kernel.Serial_Output is

    procedure Init_Serial_Line (Port : Integer);
    procedure Put (item : Character);
    procedure Put (item : String);
    procedure New_Line;
    procedure Put_Line (item : Character);
    procedure Put_Line (item : String);

end Kernel.Serial_Output;

```

Los procedimientos que contiene el paquete son los siguientes:

Init_Serial_Line: Este procedimiento inicializa una de las líneas serie (Serial_Port_1 o Serial_Port_2) hay que ejecutarlos antes de hacer cualquier salida por la línea serie.

Put: Manda por la línea serie tanto un carácter individual, como tiras de caracteres.

New_Line: Manda por la línea serie un salto de línea.

Put_Line: Manda por la línea serie caracteres y tiras de caracteres, seguido de un salto de línea.

6.3.11 Kernel.Parameters.

El paquete contiene definiciones generales del kernel y parámetros de configuración. Los principales cambios tienen que ver con el número de interrupciones que pasan de 15 a 16, y los parámetros del reloj.

```

package Kernel.Parameters is

    Max_Tasks      : constant := 64;

    Memory_Size : constant := 1*(2**20) - 1; -- 1 MB
    -- Actually, both Memory spaces RAM and ROM are defined in
    -- the linker script commands.ld which is located at
    -- /usr/local/openravenscar-i386/i386-ork-elf/lib/

    Interrupt_Stack_Size : constant := 4_096; -- bytes

    -- Maximum space to allocate stacks for threads. It is equal to
    -- (64+2)*(5K + 256 bytes) + Interrupt_Stack_Size
    Stack_Area_Size : constant := 358_912; -- bytes

    Default_Stack_Size : constant Integer := 5_120; -- bytes

    -- Priority range supported by the kernel is 0 .. Max_Priority
    Max_Priority : constant Positive := 48;

    -- Integer number of nanoseconds
    -- Clock_Interrupt_Period < ~54.9 ms
    Clock_Interrupt_Period : constant := 50_000_000; -- 50 ms

    Clock_Frequency : constant := 1.193182; -- real (Megahertz)

    -- Interrupt levels in the x86 architecture
    Interrupt_Levels : constant Positive := 16;

    -- Any_Priority range supported by the kernel is
    -- 0 .. Max_Interrupt_Priority
    Max_Interrupt_Priority : constant Positive := Max_Priority +
                                Interrupt_Levels;

    subtype Interrupt_Level is Natural range 0 .. Interrupt_Levels - 1;
    subtype Range_Of_Vector is Natural range 0 .. 255;

    -- Size of the memory pages
    Page_Size : constant := 1024;

    -- Size of the blocks that protects from stack overflow
    Protection_Stack_Size : constant := 256; -- bytes

End Kernel.Parameters;

```

Los parámetros configurables son:

Max_Tasks: Número de tareas que pueden ser declaradas en un programa.

Max_Memory_Size: Espacio de memoria disponible. Realmente este parámetro se controla desde el fichero /usr/local/openravenscar-i386/i386-ork-elf/lib/commands.ld, que es el script donde el enlazador toma los datos acerca de la cantidad de memoria.

Stack_Area_Size: Área de memoria reservada para la pila de las tareas.

Protection_Stack_Size: Área de memoria que se usa para la protección de la pila, actualmente no tiene uso, pues no hay implementados mecanismos de protección de memoria.

Interrupt_Stack_Size: Tamaño de la pila que usan los manejadores de interrupciones.

Existen dos parámetros que antes eran configurables y ahora son constantes:

Clock_Frequency: La frecuencia del temporizador(timer), es 1.193182 Megahertz, y no se puede modificar, dado que es la misma en todos los PC-compatibles.

Clock_Interrupt_Period: El periodo máximo de interrupción que puede mantener el PIT estándar de la arquitectura i386-PC es de aproximadamente 54.9 ms, pero para redondear el periodo se dejará en 50 ms. Esto es debido a que tenemos un contador de 16 bit con una frecuencia de 1.193182 Megahertz.

El periodo se calcula mediante la formula: $T = N \times C$

Con N valor máximo del contador ($2^{16}-1$), y C periodo de cada *tick* de reloj, $C = 1 / F$ (F es la frecuencia del reloj).

6.4 Funcionamiento e implementación de ORK-i386.

6.4.1 Inicialización del entorno.

El *bootloader* (ver sección 6.4.7) pasa el control a la etiqueta “`_start`” que será la primera instrucción que se ejecute dentro del sistema.

El control de la inicialización y activación del entorno lo lleva a cabo la función `multiboot_main`, que se encuentra en el fichero ‘`base_multiboot_main.c`’, se encargará de realizar todas las llamadas que dejarán preparadas todas las estructuras del entorno para que la aplicación pueda ejecutarse correctamente. Los pasos fundamentales que se llevan a cabo son los siguientes:

1. Se crea una pila de entorno, que quedará definitivamente como la pila de entorno del sistema (*Environment_Stack*) durante toda la vida del sistema, no utilizándose el espacio de memoria reservado para tal fin en el fichero “`kernel-memory.adb`”.
2. Se procede a la identificación de la CPU mediante la instrucción “`CPUID`” que ofrece información del tipo de procesador, memoria instalada, etc. Seguidamente se inicializan las estructuras necesarias para poder trabajar en el MODO PROTEGIDO, que será el que nos permita acceder a toda la memoria instalada. Primero se inicializa la Tabla de Descriptores Globales GDT con los descriptores globales de los segmentos de memoria que serán utilizados durante la vida del sistema (ver figura 6.1), concretamente la tabla GDT tendrá 5 descriptores útiles.
 - Uno para el Segmento de Estado de la Tarea (TSS).
 - Dos descriptores de segmento que se utilizan en el paso del MODO REAL al MODO PROTEGIDO, uno para el segmento del código y otro para el segmento de datos.
 - Dos descriptores de segmento que se utilizan en MODO PROTEGIDO, uno para el segmento del código y otro para el segmento de datos. Que serán los que se utilicen durante todo el resto de la vida del sistema, ambos descriptores abarcan toda la memoria direccionable del sistema, lo que da un aspecto final de memoria plana aunque exista la segmentación.

Los registros de segmento como son CS, DS, ES, SS, FS y GS se inicializan con los valores adecuados, concretamente CS contiene la dirección del descriptor de del segmento de código. DS, SS y ES comparten el mismo descriptor de datos, FS y GS no apuntan a ningún descriptor por lo que permanecerán inoperativos. Una vez realizados estos pasos el sistema está en condiciones de empezar a trabajar en el MODO PROTEGIDO. El modelo de memoria elegido es el de

memoria plana sin paginación, solo se utiliza la segmentación ya que este es un aspecto que no puede evitarse en esta arquitectura.

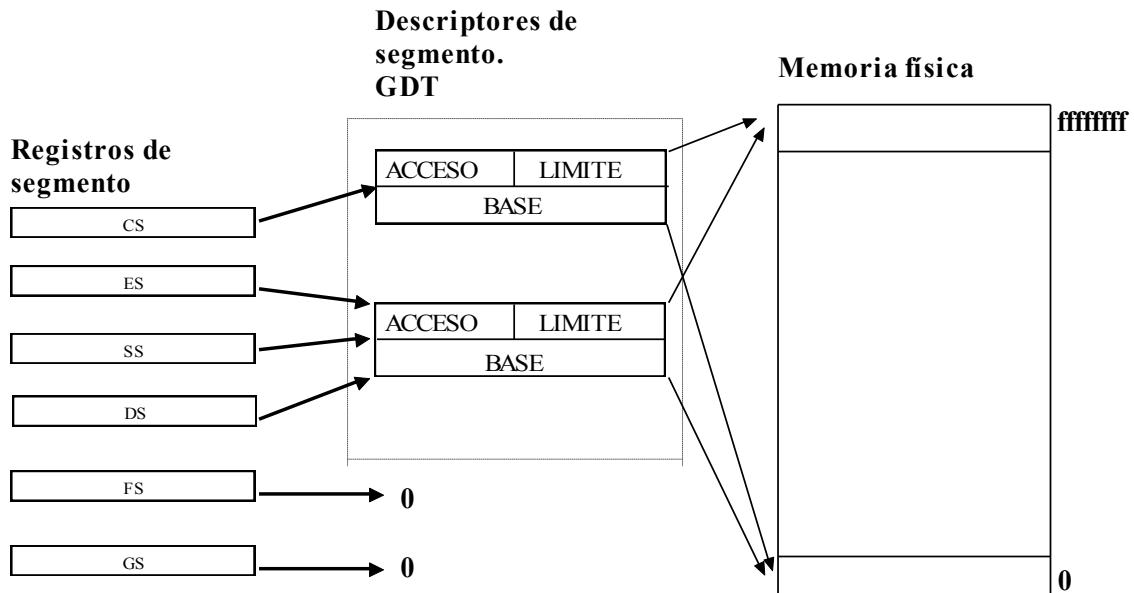


Figura 6.1: Modelo de memoria

3. Se inicializa de la Tabla de Descriptores de Interrupción (IDT), con los valores por defecto para el tratamiento de las excepciones y las interrupciones externas (ver sección 6.4.2), esta tabla tendrá un tamaño máximo de 256 entradas, y su dirección base se almacena en el registro IDTR.
4. Se crea un Segmento de Estado de la Tarea (TSS) global para todo el sistema apuntado por el registro TR, y se rellena con un estado mínimo, los únicos campos que se utilizan son el que apunta al segmento de pila de la tarea(ss0) y el valor del puntero de pila (esp0). Toda la aplicación se ejecuta en modo privilegiado sobre las estructuras que se crean al inicializar el sistema. Después de la inicialización del sistema se pasa al modo protegido de la arquitectura i386.
5. Seguidamente se inicializa la salida por defecto a la pantalla.
6. Se rellena la tabla de los manejadores de los *trap* del sistema, con los siguientes valores para cada posición (Ver sección 6.4.2.6):
 0. trap_divide_error
 1. trap_debug
 2. trap_nmi
 3. trap_int3
 4. trap_overflow

- 5. trap_out_of_bounds
 - 6. trap_invalid_opcode
 - 7. trap_no_fpu
 - 8. trap_double_fault
 - 9. trap_fpu_fault
 - 10. trap_invalid_tss
 - 11. trap_segment_not_present
 - 12. trap_stack_fault
 - 13. trap_general_protection
 - 14. trap_page_fault
 - 15. trap_floating_point_error
 - 16. trap_alignment_check
 - 17. trap_machine_check
7. Antes de ejecutar el código de la aplicación, se inicializan algunos aspectos del runtime, de ORK y del hardware. Para el correcto funcionamiento de las interrupciones los PICs han sido programados en el MODO ESPECIAL PLENAMENTE ANIDADO (*Special Fully Nested Mode*), de esta manera cuando se estén atendiendo interrupciones en el esclavo estas no enmascaren otras interrupciones del esclavo aunque estas últimas tengan mayor. Se configuran los PIT para la gestión del tiempo y se instala el manejador de las alarmas.
8. Por último se crean dos thread, *Dummy_Thread*, que será el thread que se ejecute en caso de que no haya ningún otro para ejecutar, y el thread de entorno *Environment_Thread*, que será el que se asigne a la aplicación del usuario.

ARCHIVOS:

No se comentan todos los archivos que contienen aspectos relacionados con inicialización del sistema, solo aquellos que contiene los aspectos más importantes a la hora de la implementación.

Ficheros	Descripción
/boot/multiboot.S	Entrada del sistema.
/boot/base_stack.S	Definición de la pila de entorno.
/ork_ker/base_multiboot_main.c	Contiene la función <i>multiboot_main</i> que dirige la inicialización del sistema.
/boot/base_gdt.c	Tabla de descriptores globales GDT.
/boot/base_gdt_init.c	Inicialización de la tabla de descriptores globales GDT.
/boot/base_gdt_load.c	Carga de la tabla de descriptores globales GDT en el registro GDTR y los registros de segmento.
/boot/base_idt.c	Tabla de descriptores de interrupción IDT

/boot/base_idt_load.c	Inicialización de la tabla de descriptores de interrupción IDT.
/boot/base_irq_init.c	Carga de la tabla de descriptores de interrupción IDT en el registro IDTR e inicialización de los PIC.
/boot/base_trap_init.c	Inicialización de la tabla de descriptores de interrupción IDT en la parte que respecta a los <i>traps</i> del sistema.
/boot/base_tss.c	TSS (<i>Task State Segment</i>), Segmento de Estado de Tarea único para todo el sistema
/boot/base_tss_init.c	Inicialización de TSS.
/boot/base_tss_load.c	Carga de TSS en el registro TR.
/boot/base_multiboot_init_mem.c	Inicialización de la memoria.
/boot/base_console_init.c	Inicialización de la consola.

6.4.2 Gestión de interrupciones y traps.

6.4.2.1 El sistema de interrupciones de la familia ix86.

La arquitectura i386 considera tres tipos de interrupciones:

Interrupciones Internas:

También llamadas **Traps** o **EXCEPCIONES**, son interrupciones producidas por acontecimientos internos que ocurren dentro de la CPU. Estas interrupciones se producen automáticamente y transfieren la ejecución a vectores de interrupción prefijados. Las interrupciones internas alertan sobre acontecimientos internos que pudieran no estar previstos por el programador. En general señalan situaciones de error.

Interrupciones Externas:

- **Enmascarables:** Son las producidas por la activación del terminal INTR. El que sean atendidas o no depende del estado del flag IF del registro de estado. Si este bit está a 1 las interrupciones son atendidas, , el programador puede controlar su estado mediante las instrucciones CLI y STI. En el caso contrario son ignoradas.
- **No enmascarables:** Son las producidas por la activación del terminal NMI de la CPU. Estas interrupciones son siempre atendidas, independientemente del estado del flag IF.

Las interrupciones externas son asíncronas, con la ejecución del programa.

Interrupciones Software:

Son las que se producen por software, es decir mediante la instrucción INT en los programas. Todas las interrupciones pueden producirse mediante la instrucción INT **n**, en donde n es un número comprendido entre 0 y 255. Son interrupciones síncronas.

De aquí en adelante entenderemos como **interrupciones** a las que tienen un origen hardware por la activación de los terminales INTR o NMI (interrupciones externas), el resto serán consideradas como **excepciones** (traps e interrupciones software).

En el modo protegido las excepciones pueden ser de dos tipos (ver tabla 6.1):

- Detectadas por el procesador en la ejecución de alguna instrucción.

FALLOS: Un fallo es una excepción que generalmente puede ser corregida, y una vez corregido el fallo permitir al programa continuar. Son los detectados antes de ejecutarse una instrucción. La instrucción puede ser reemprendida. Los valores de CS e IP guardados en la pila corresponden a la instrucción que ha causado la excepción.

TRAPS: Son las detectadas inmediatamente después de ejecutarse una instrucción. Los valores de CS e IP guardados en la pila corresponden a la instrucción siguiente a la que ha causado la excepción. En el caso de instrucciones de transferencia de control, debe interpretarse que se trata de la siguiente instrucción a ejecutar.

ABORTOS: Son errores que no admiten corrección. Pueden deberse a errores graves en el hardware o en la inconsistencia de determinados valores.

- Interrupciones software, producidas por las instrucciones **INT**, **INTO**, **INT n** y **BOUND**.

Tabla 6.1: Excepciones e interrupciones en el modo protegido a partir del 486.

Nº Vector Nº Int	Descripción	Tipo	Fuente
0	Error de división	FALLO	Error de división al ejecutar las instrucciones DIV o IDIV .
1	Debug, int paso a paso	FALLO / TRAP	Ejecución paso a paso (<i>flag TF</i> activado). Produce INT 1 al final de cada interrupción.
2	NMI	INTERRUP.	Int. externa aplicada al terminal NMI .
3	Breakpoint	TRAP	Instrucción INT 3 . se usa en depuración
4	Overflow	TRAP	Instrucción INTO .
5	BOUND	FALLO	Instrucción BOUND .

6	Código de operación no valido	FALLO	Al intentar ejecutar un código de operación no definido en el juego de instrucciones. Instrucción UD2 . ¹
7	Coprocesador no disponible	FALLO	Unidad de coma flotante o instrucciones WAIT/FWAIT .
8	Doble fallo	ABORTO	Una instrucción que provoca dos excepciones que no pueden ser tratadas secuencialmente.
9	Violación segmento coprocesador	FALLO	Instrucción de coma flotante. ²
10	TSS no válido	FALLO	Una conmutación de tareas, cuando el TSS no es válido o contiene datos inválidos.
11	Segmento NO PRESENTE	FALLO	Carga de los registros de segmentos y acceso al sistema de registros.
12	Excepción de pila	FALLO	Operaciones de pila y carga del registro SS.
13	Violación de protección	FALLO	Cualquier referencia a memoria y chequeos de protección.
14	Fallo de pagina	FALLO	Cualquier referencia a memoria. Se accede a pagina no presente o sin el suficiente privilegio.
15	RESERVADA (no usada por Intel)		
16	Error en el coprocesador	FALLO	Unidad de coma flotante e instrucciones WAIT/FWAIT .
17	Error de alineación	FALLO	Cualquier referencia a datos en memoria. Cuando se accede a un operando no alineado y esta activado el <i>flag AM</i> . ³
18	Chequeo de la máquina	ABORT	Los códigos de error y las fuentes dependen del modelo. ⁴
19	Reservadas por Intel. No usadas		
32 - 255	Definidas por el usuario. Interrupciones	INTERRUP.	Interrupciones externas o instrucción INT n.

NOTAS:

- (1) La instrucción **UD2** fue introducida en el procesador Pentium Pro.
- (2) Los procesadores después del intel386 no generan esta excepción.
- (3) Esta excepción fue introducida en el procesador intel486.
- (4) Esta excepción fue introducida en el procesador Pentium y modificada en procesadores procesadores posteriores.

En el modo protegido las rutinas se localizan a través de la tabla IDT (*Interrupt Descriptor Table*) que puede estar situada en cualquier lugar del mapa de memoria.

El registro IDTR (*IDT register*) contiene la dirección base de la tabla IDT así como su límite. Cada elemento de esta tabla es un descriptor, por lo que ocupa 8 bytes. Puesto que hay 256 interrupciones, la IDT, como máximo ocupará 2 KB, aunque puede ser menor si no se utilizan todas las interrupciones. Cada elemento de la tabla IDT puede ser un descriptor de *INTERRUPT GATE*, *TRAP GATE* o *TASK GATE*, en nuestro caso solo tendremos descriptores de los dos primeros tipos. En la figura 6.2 se muestran las sucesivas indirecciones para alcanzar la rutina de atención a la interrupción.

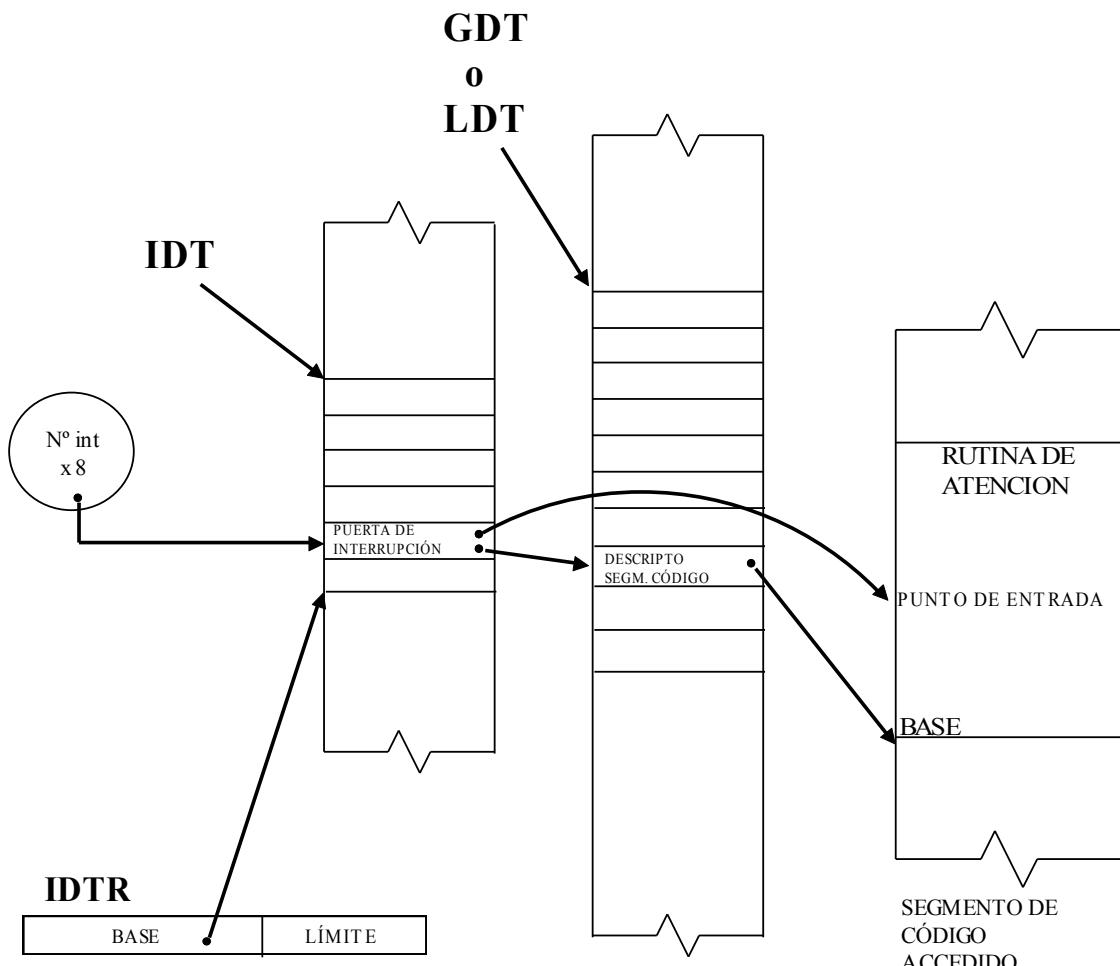


Figura 6.2: Atención de una interrupción a través de la tabla IDT.

La interrupción, sea cual sea su origen, está identificada por su número de interrupción o número de vector como se indica en la tabla 6.1 para las excepciones y en tabla 6.2 para las interrupciones hardware. Multiplicando este número por 8 se localiza la puerta de acceso a la interrupción o al trap, que apunta a un descriptor de segmento de

código, que puede pertenecer tanto a la LDT actual como a la GDT, y contiene además un *desplazamiento*. Con esto se localiza el segmento que contiene la rutina de atención, mientras que el *desplazamiento* señala, sin lugar a dudas, el punto de entrada, es decir, la primera instrucción de dicha rutina.

6.4.2.2 Hardware de interrupciones.

El control de las interrupciones se hace a través de la pastilla 8259, PIC (*Programmable Interrupt Controller*) es un controlador de interrupciones diseñado específicamente para complementar a los microprocesadores de la familia 8085. Permite el manejo de hasta 8 dispositivos capaces de solicitar interrupción hardware, pudiendo ser expandido este número hasta 64 con la utilización de pastillas 8259 adicionales. Cuando apareció el 8086, Intel hizo una nueva versión de esta pastilla, que denominó 8259-A para la familia ix86.

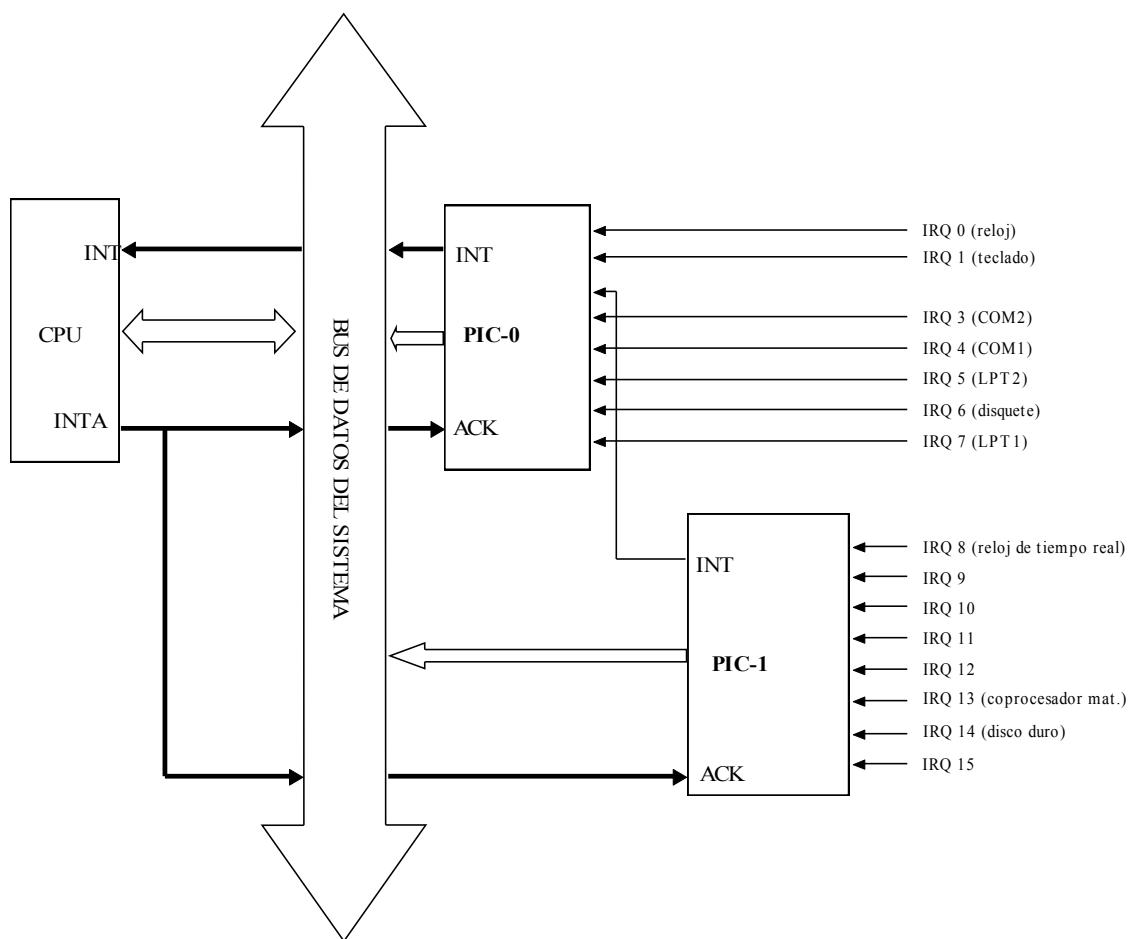


Figura 6.3: Conexión de los dos PIC al sistema

El PIC se encarga de enviar a la CPU el número de interrupción que va a servir para la identificación del periférico solicitante, lo que facilita el diseño de éste.

Además ejerce de árbitro de prioridades para solucionar aquellas situaciones en que dos o más periféricos solicitan interrupciones simultáneamente, permite el enmascaramiento individual de cada una de las ocho entradas y muchas otras características del funcionamiento de las interrupciones. La arquitectura PC/AT contiene dos PIC conectados en cascada como se indica en la figura 6.3. El PIC-0 se corresponde con el único PIC del antiguo PC/XT.

En la tabla 6.2 se muestran la función estándar que se ha dado a algunas de las entradas de los PIC, otras permanecen libres en el bus ISA, también se muestra la prioridad que tienen las interrupciones en ORK-i386, que viene dada por el hecho de que los PIC han sido programados en el modo completamente anidados.

6.4.2.3 Rutina de tratamiento de interrupciones.

La rutina de tratamiento de interrupciones es común para todas las interrupciones, una vez que se produce la interrupción y se desencadena el proceso visto en el punto 6.4.2.1, llegamos a la rutina propiamente dicha. Notese que se permite el anidamiento de la R.T.I según el modelo habitual, es decir una petición más prioritaria puede interrumpir la R.T.I de una petición menos prioritaria, pero no al revés. La rutina como el resto del código de bajo nivel, en su implementación contempla situaciones que no se producirán en nuestro sistema, pero que no han sido eliminadas de la versión original de *OS_Toolkit*, para mantener la máxima compatibilidad con el resto del código. Los pasos que sigue la rutina de tratamiento de interrupciones son los siguientes:

1. Cuando se llega a la rutina ya esta almacenado en la pila: el registro de estado EFLAGS, el registro de segmento CS y el contador de programa EIP
2. Se comprueba si estamos ante una interrupción anidada, es decir si la interrupción se ha producido mientras se estaba tratando otra interrupción, en caso no ser así se cambia la pila de la tarea interrumpida por la pila propia del tratamiento de interrupciones, tanto en el caso de tratarse de una interrupción anidada como si no, se incrementa el nivel de anidamiento.
3. Se almacena en la pila el número de *irq* producida (la entrada del PIC que ha producido la interrupción) y el número de vector (la posición de la interrupción en la tabla de descriptores de interrupciones IDT).
4. Se salvan los registros generales en la pila: EAX, ECX, EDX, CR2, EBP, ESI, EDI. Un caso especial es el de CR2, ya que en función de si se trata de una excepción, concretamente un fallo de página (no será nuestro caso pues no tenemos paginación) se salvará el registro de control CR2, en el resto de los traps o interrupciones lo que se salva es el valor de ESP en el momento previo a salvarlo.

5. Se almacena temporalmente la actual máscara de interrupción de los PIC, y se ajusta la máscara de los controladores de interrupción de acuerdo a la interrupción producida.
6. Se salvan los registros de segmento: DS, ES, FS, GS. Y la máscara de los PIC que había sido guardada temporalmente en el paso anterior.
7. Se desinhiben las interrupciones.
8. Una vez almacenado todo el estado del sistema en la pila, se llama al manejador de interrupciones, pasándole como parámetro la estructura almacenada en la pila, que recibe el nombre de *Trap_State*, con la siguiente forma definitiva.

Tabla 6.3: Estructura *Trap_State* que recibe un manejador de interrupciones.

mascara	Máscara de interrupción
gs	Registro de segmento
fs	Registro de segmento
es	Registro de segmento
ds	Registro de segmento
edi	Registro
ei	Registro
ep	Registro
cr2	En ORK-i386 se almacena esp
ebx	Registro
edx	Registro
ecx	Registro
eax	Registro
trapno	Número de vector
err	Número de irq
eip	Contador de programa
cs	Registro de segmento
eflags	Flags del sistema

La estructura *Trap_State* contiene campos que solo serán usados si el sistema estuviera funcionando en modo Virtual-86 (V86).

9. Cuando se vuelve del manejador de interrupciones se vuelven a inhibir las interrupciones, durante la ejecución del manejador se han podido producir interrupciones de mayor prioridad para las cuales se procederá de igual forma.
10. Se restauran la máscara de los PIC, los registros generales y los registros de segmento.

Tabla 6.2: Interrupciones Hardware.

Nombre en ORK	Entrada del PIC	Nº int Nº Vector	Prioridad en ORK	Función estándar
TIMER_IRQ	PIC-0.IRQ0	32	System.Interrupt_Priority'First + 15	Reloj del Timer a 18.2 ints./seg
KEYBOARD_IRQ	PIC-0.IRQ1	33	System.Interrupt_Priority'First + 14	Interrupción del teclado
CTRL2_IRQ	PIC-0.IRQ2	34	System.Interrupt_Priority'First + 13	Expansión al esclavo PIC-1
SERIAL2_IRQ	PIC-0.IRQ3	35	System.Interrupt_Priority'First + 4	Puerto serie COM2
SERIAL1_IRQ	PIC-0.IRQ4	36	System.Interrupt_Priority'First + 3	Puerto serie COM1
PARALLEL2_IRQ	PIC-0.IRQ5	37	System.Interrupt_Priority'First + 2	Puerto paralelo LPT2
DISKETTE_IRQ	PIC-0.IRQ6	38	System.Interrupt_Priority'First + 1	Disquete
PARALLEL1_IRQ	PIC-0.IRQ7	39	System.Interrupt_Priority'First + 0	Puerto paralelo LPT1
RTC_IRQ	PIC-1.IRQ8	40	System.Interrupt_Priority'First + 12	Reloj de Tiempo Real RTC
SOFT_IRQ	PIC-1.IRQ9	41	System.Interrupt_Priority'First + 11	Libre
RESERVED1_IRQ	PIC-1.IRQ10	42	System.Interrupt_Priority'First + 10	Libre
RESERVED2_IRQ	PIC-1.IRQ11	43	System.Interrupt_Priority'First + 9	Libre
RESERVED3_IRQ	PIC-1.IRQ12	44	System.Interrupt_Priority'First + 8	Libre
COPROCESSOR_IRQ	PIC-1.IRQ13	45	System.Interrupt_Priority'First + 7	Coprocesador matemático
FIXED_DISK_IRQ	PIC-1.IRQ14	46	System.Interrupt_Priority'First + 6	Disco duro
RESERVED5_IRQ	PIC-1.IRQ15	47	System.Interrupt_Priority'First + 5	Libre

11. Se comprueba si estamos tratando una interrupción anidada, en caso de ser así se sale inmediatamente de la rutina de tratamiento de interrupciones, recuperando plenamente el estado anterior a producirse dicha interrupción, y se decrementa el nivel de anidamiento.
12. En caso de estar en una interrupción no anidada, se restaura la pila de la tarea que fue interrumpida y se comprueba si hay algún cambio de contexto pendiente, si no fuera así, se sale de la rutina de igual forma que en el punto anterior, recuperando plenamente el estado de la tarea interrumpida.
13. Si hubiera un cambio de contexto pendiente, se llama a la rutina de cambio de contexto, en este caso la tarea que fue interrumpida dejará una parte de la rutina de interrupciones sin ejecutar, concretamente sin restaurar el estado, y esta será terminada cuando se produzca otro cambio de contexto que le de el control a la tarea interrumpida, por último antes de salir esta tarea de la R.T.I, calculará de nuevo la máscara de los PIC de acuerdo a su prioridad.

ARCHIVOS:

No se enumeran todos los archivos que contienen aspectos relacionados con el tratamiento de las interrupciones, solo aquellos que contiene los aspectos más importantes a la hora de la implementación

Ficheros	Descripción
boot/base_irq_inittab.S	Rutina de tratamiento de interrupciones.
boot/base_irq.c	Tabla de los manejadores de interruptores “base_irq_handlers”.
boot/base_irq_default_handler.c	Manejador de interrupciones por defecto.
boot/interrupt_tables.c	Funciones de manejo de la tabla de interrupciones y traps.
boot/pc_asm.h	Contiene las direcciones de las <i>IRQs</i> en la tabla IDT
boot/trap_dump	Actualmente en este archivo se encuentra la tabla de las máscaras de prioridad de los PIC.

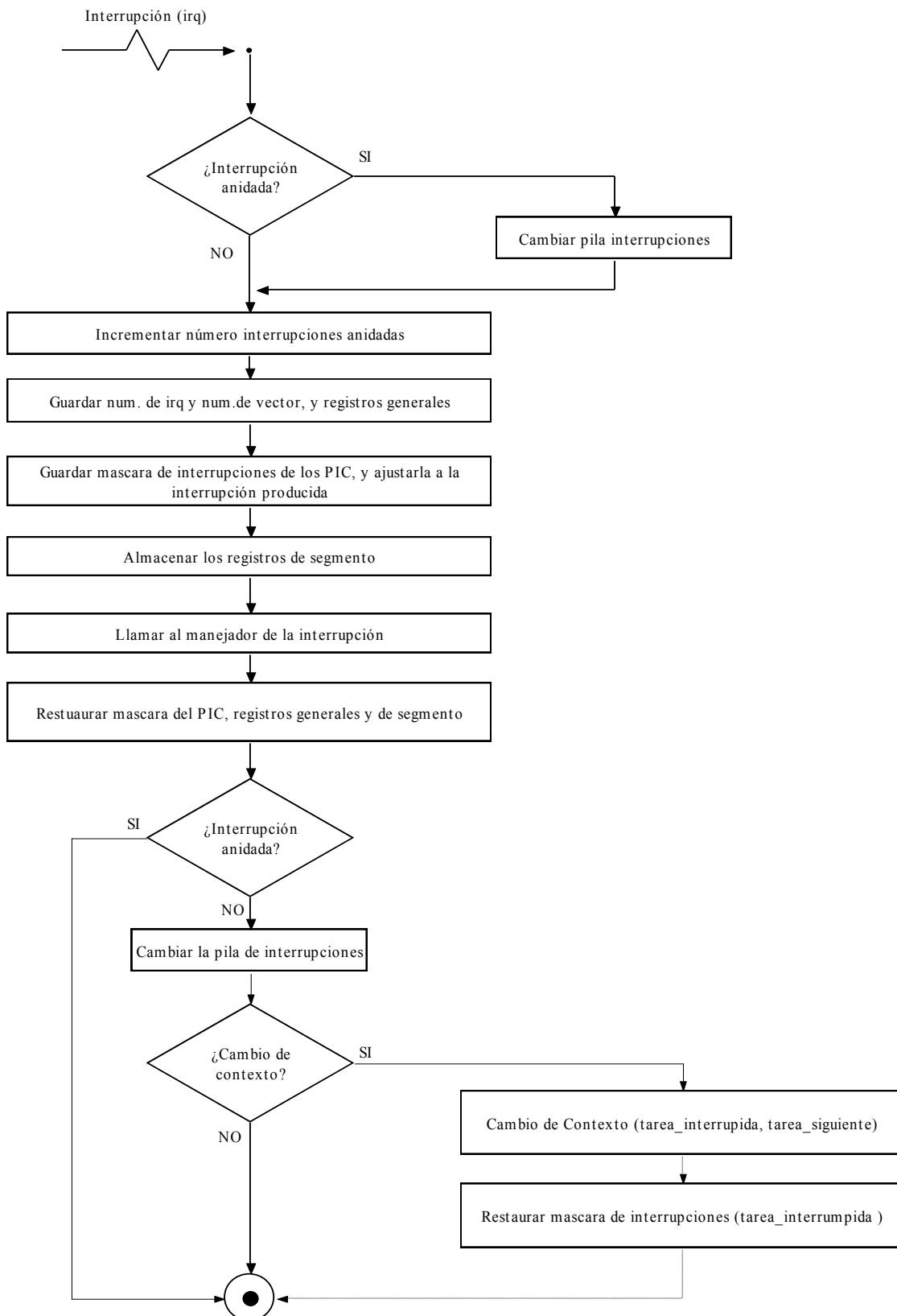


Figura 6.4: Rutina de tratamiento de interrupciones en ORK-i386

6.4.2.4 Tratamiento de las interrupciones en ORK-i386.

El tratamiento de las interrupciones no varía mucho en ORK-i386 con respecto al ORK original, y la implementación es prácticamente la misma, con las variaciones que ya se explicaron el apartado 6.3.5. La diferencia fundamental consiste en que los manejadores de interrupciones, concretamente la función *Wrapper* (es el manejador general que envuelve al manejador de interrupciones real) reciben como parámetro la estructura *Trap_State*. El proceso global del procesamiento de una interrupción queda reflejado en la figura 6.5, los pasos que se siguen son los siguientes.

1. Se produce la interrupción, y se accede a través de la tabla de descriptores de interrupciones IDT, a la rutina de tratamiento de interrupciones, como ya se vio en los apartados anteriores, la R.T.I llama al manejador de interrupciones instalado en la tabla “base_irq_handler”, que contiene los manejadores por defecto *base_irq_default_handler* o el manejador colocado por el Runtime de GNAT, en el caso que se haya ejecutado la función *Attach_Handler*.
2. Si ha sido colocado el manejador por la función *Attach_Handler*, se ejecuta el procedimiento *Wrapper*, este es un procedimiento genérico para todos los manejadores, que conoce la interrupción que se ha producido a través del campo *ERR* de la estructura *Trap_State* que recibe como parámetro.

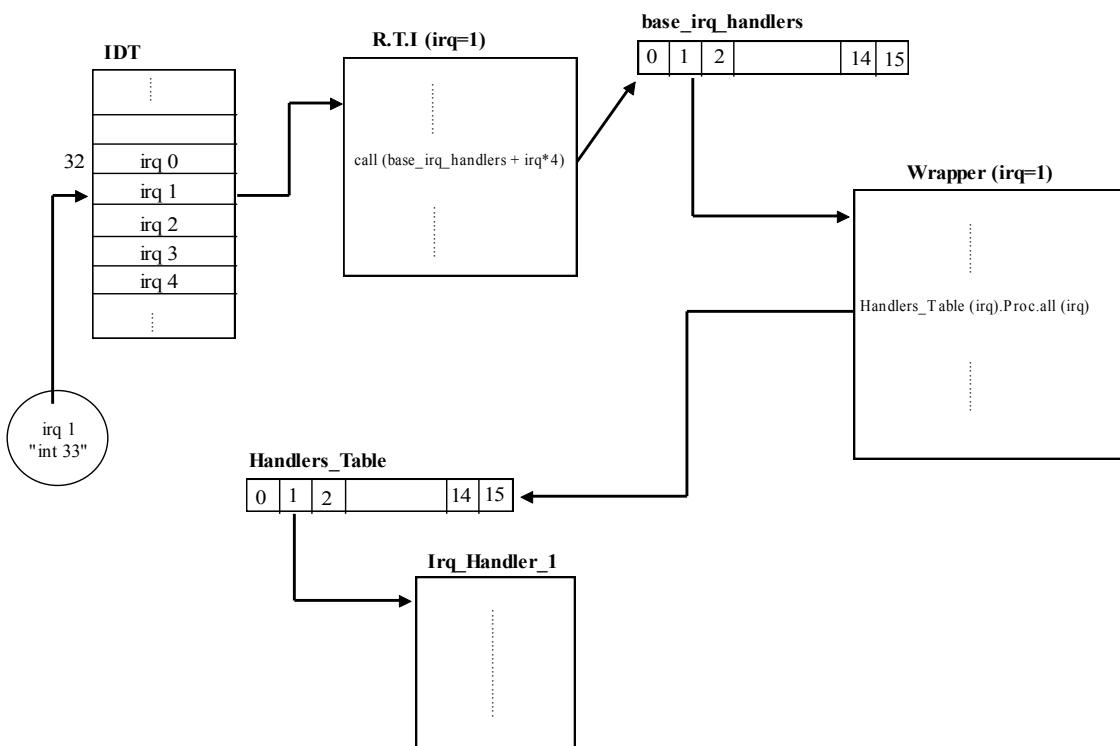


Figura 6.5: Estructura global de la indirección de una interrupciones en ORK-i386

3. *Wrapper* ajusta el nivel de prioridad actual de acuerdo a la interrupción producida, y llama al manejador real de la interrupción que se encuentra en la tabla “Handler_Table”.
4. Por último una vez ejecutado el manejador de interrupciones devuelve el control a la función *Wrapper* que restaura el nivel de interrupción anterior, y acaba pasando el control al epílogo de la R.T.I.

ARCHIVOS:

No se enumeran todos los archivos que contienen aspectos relacionados con el tratamiento de las interrupciones, solo aquellos que contiene los aspectos más importantes a la hora de la implementación.

Ficheros	Descripción
gcc-2.8.1/ada/kernel-interrupts.adb	Cuerpo de las funciones <i>Attach_Handler</i> , <i>Detach_Handler</i> , <i>Current_Handler</i> que gestionan la tabla “Handler_Table”, y cuerpo del procedimiento <i>Wrapper</i> .
gcc-2.8.1/ada/kernel-interrupts.ads	Definiciones de las funciones <i>Attach_Handler</i> , <i>Detach_Handler</i> y <i>Current_Handler</i> .
gcc-2.8.1/ada/kernel-cpu_primitives.ads	Definición de la estructura <i>Trap_State</i> y las funciones <i>Install Interrupt_Handler</i> y <i>Current_Handler</i> que gestionan la tabla “base_irq_handlers”.
gcc-2.8.1/ada/kernel-peripherals.ads	Definiciones de las interrupciones del sistema, manejadores de las máscaras de los PIC, y funciones de gestión de la prioridad de las interrupciones en el sistema.
gcc-2.8.1/ada/kernel-peripherals.adb	Cuerpo de las funciones manejadoras de las máscaras de los PIC, y funciones de gestión de la prioridad de las interrupciones en el sistema.
gcc-2.8.1/ada/kernel-peripherals-registers.ads	Definiciones para la configuración de los PIC.

6.4.2.5 Rutina de tratamiento de excepciones.

La rutina de tratamiento de excepciones es común para todas las excepciones internas del sistema, una vez que se produce la excepción o *trap* y se desencadena el proceso visto en el punto 6.4.2.1, llegamos a la rutina propiamente dicha. La rutina como el resto del código de bajo nivel, en su implementación contempla situaciones que no se producirán en nuestro sistema, pero que no han sido eliminadas de la versión original de

OS_Toolkit, para mantener la máxima compatibilidad con el resto del código. Los pasos que sigue la rutina de tratamiento de excepciones son los siguientes:

1. Cuando se llega a la rutina ya están almacenados en la pila los siguientes registros: el registro de estado EFLAGS, el registro de segmento CS y el contador de programa EIP
2. Se almacena en la pila el número de vector (la posición del trap en la tabla de descriptores de interrupciones IDT).
3. Se salvan los registros generales en la pila: EAX, ECX, EDX, CR2, EBP, ESI, EDI. Un caso especial es el de CR2, ya que en función de si se trata de la excepción de fallo de página (no será nuestro caso pues no tenemos paginación), se salva el registro de control CR2, en el resto de los traps lo que se salva es el registro ESP.
4. Se salvan los registros de segmento: DS, ES, FS, GS.
5. Una vez almacenado todo el estado del sistema en la pila, se llama al manejador de excepciones, pasándole como parámetro la estructura almacenada en la pila, que recibe el nombre de *Trap_State*.
6. Cuando se vuelve del manejador de interrupciones se restauran los registros generales y los registros de segmento, y por último se retorna al punto en que se produjo la excepción.

ARCHIVOS:

No se enumeran todos los archivos que contienen aspectos relacionados con el tratamiento de las excepciones, solo aquellos que contiene los aspectos más importantes a la hora de la implementación.

Ficheros	Descripción
boot/base_trap_inittab.S	Rutina de tratamiento de excepciones
boot/base_trap.c	Contiene la tabla “base_trap_handlers”.
boot/base_trap_default_handler.c	Manejador por defecto de las excepciones.
boot/interrupt_tables.c	Función para instalar manejadores de excepciones en la tabla “base_trap_handlers”.
boot/trap_asm.h	Posiciones de los campos de la estructura <i>Trap_State</i> en la pila.
boot/trap_dump.c	Función que hace un volcado de la estructura <i>Trap_State</i> por la salida estandar.
boot/trap_dump_panic.c	Función que llama a <i>trap_dump</i> y provoca la salida total del sistema.
ork_ker/ada_excepcion.c	Manejadores de las excepciones del sistema para ORK-i386

6.4.2.6 Tratamiento de las excepciones en ORK-i386.

El tratamiento de las excepciones no ha variado en ORK-i386, siguiendo las pautas fundamentales del perfil Ravenscar:

- Los *traps* síncronos o *traps* del sistema son redireccionados a excepciones Ada, como indica la tabla 6.4. Los manejadores de los traps se encuentran en la tabla “*base_trap_handler*”.
- Excepciones software. Son activadas por el software del usuario a través de la sentencia “*raise*”.
- Excepciones activadas por el kernel. El núcleo de ORK-i386 no usa ninguna excepción definida dentro del kernel. Todas las excepciones activadas son o bien pertenecientes a la capa del runtime de GNAT o excepciones del sistema.
- Excepciones en manejadores de interrupciones. Cualquier excepción activada dentro de un manejador de interrupciones, solo se podrá manejar dentro espacio del manejador de interrupciones. Cualquier excepción propagada desde un manejador de interrupciones no tendrá efecto.
- Sentencias bloqueantes en procedimientos protegidos y funciones, levantan la excepción *Program_Error*.
- La violaciones del techo de prioridad y las llamadas a una entrada que ya tiene encolada otra llamada activan la excepción *Program_Error*.

Tabla 6.4: Excepciones Ada activadas por los *traps* del sistema.

Nº	Trap	Excepción Ada activada en ORK-i386
0	trap_divide_error	constraint_error
1	trap_debug	constraint_error
2	trap_nmi	constraint_error
3	trap_int3	constraint_error
4	trap_overflow	storage_error
5	trap_out_of_bounds	storage_error
6	trap_invalid_opcode	storage_error
7	trap_no_fpu	constraint_error
8	trap_double_fault	constraint_error
9	trap_fpu_fault	constraint_error
10	trap_invalid_tss	constraint_error
11	trap_segment_not_present	storage_error
12	trap_stack_fault	storage_error
13	trap_general_protection	constraint_error
14	trap_page_fault	storage_error
16	trap_floating_point_error	constraint_error
17	trap_alignment_check	storage_error
18	trap_machine_check	constraint_error

- Excepciones en la elaboración de las tareas. Se debería producir la excepción `Tasking_Error` que debería ser manejada por la tarea padre. Pero el perfil Ravenscar no permite el manejo de esta excepción ya que no se permiten las jerarquías de tareas, por lo tanto, cualquier aplicación con una tarea que active una excepción durante su inicialización, se silenciará, intentando seguir su ejecución normal.
- Terminación de tareas. Debería ser tratado de acuerdo con el perfil Ravenscar como un error de límite, activando por ejemplo la excepción `Program_Error`, pero como ocurría en el caso anterior, esta excepción no puede ser manejada, por la tarea padre, por lo que será silenciada, a no ser que se defina otra acción específica para tratar la finalización de una tarea a través del procedimiento interno `Set_Exit_Task_Procedure`.

ARCHIVOS:

No se enumean todos los archivos que contienen aspectos relacionados con el tratamiento de las excepciones, solo aquellos que contiene los aspectos más importantes a la hora de la implementación.

Ficheros	Descripción
<code>ork_ker/ada_excepcion.c</code>	Contiene los manejadores de excepciones para ORK-i386
<code>ork_ker/base_multiboot_main.c</code>	Inicialización del sistema y concretamente de la tabla de los <i>traps</i> del sistema.

6.4.3 Soporte de Entrada/Salida.

ORK-i386 ofrece una serie de servicios para realizar entrada/salida (realmente solo salida) de datos por pantalla o por puerto serie, y así poder observar el comportamiento de las aplicaciones, de una manera visual.

La salida de datos por consola se realiza a través de las llamadas a las primitivas `Put` y `New_Line` del paquete “Kernel.Peripherals”, estas primitivas utilizan a su vez la llamada a la función `printk` (homónima de `printf`) de la librería C del *OS_Toolkit*, para realizar la salida definitiva por pantalla o por puerto serie. La implementación del soporte para consola que ofrece el *OS_Toolkit* es complejo, y se escapa del objetivo del trabajo, por lo que no se va a explicar.

La primitiva `Put` permite mostrar diferentes tipos de datos, como son:

- Character.
- String.
- Integer.
- Integer_8

- Integer_32.
- Unsigned_8.
- Unsigned_32.
- Unsigned_64.
- System.Address.

Como ya se ha mencionado anteriormente, también se permite realizar salida de datos por uno de los puertos que tiene la arquitectura PC (COM1, COM2). Para esto hay que utilizar las primitivas del paquete “Kernel.Serial_Output”: *Put*, *Put_Line* y *New_Line*. Los datos de entrada de la primitiva *Put* son caracteres y tiras de caracteres, por lo que para poder enviar por la línea serie cualquier otro tipo de datos será necesario pasarlo previamente a una tira de caracteres. Para poder realizar la salida de datos por una de las líneas serie, es preciso iniciarla primero mediante la llamada a la primitiva *Init_Serial_Line* (ver código de ejemplo) que recibe como parámetro *Serial_Port_1* o *Serial_Port_2*.

```
with Kernel.Serial_Output; use Kernel. Serial_Output;
procedure My_Proc is
begin

    -- Código de la aplicación;

    Init_Serial_Line (Serial_Port_1);
    Put_Line ("Hola mundo");

    -- Código de la aplicación;

end My_Proc;
```

Para la implementación se usa una vez más el soporte de *OS_Toolkit*, concretamente las funciones *com_cons_init* y *com_cons_putchar* para la inicialización de la línea serie y el envío de caracteres respectivamente. En la inicialización de la línea esta se configura a una velocidad de 115200 baudios.

En el caso de tener activa la depuración por línea serie es importante tener cuidado con no utilizar el mismo puerto para depuración y salida de datos.

ARCHIVOS:

Ficheros	Descripción
gcc-2.8.1/ada/kernel-peripherals.adb	Cuerpo de los procedimientos y funciones para realizar salida por línea serie o por pantalla.
gcc-2.8.1/ada/kernel-peripherals.ads	Especificación de los procedimientos y funciones para realizar salida por línea serie o por pantalla.
gcc-2.8.1/ada/kernel-serial_output.adb	Cuerpo de los procedimientos y funciones para realizar salida por línea serie.

gcc-2.8.1/ada/kernel-serial_output.ads	Especificación de los procedimientos y funciones para realizar salida por línea serie.
boot/basic_console_io_c.c	Funciones que envuelven las llamadas a <i>printk</i> , usadas por los procedimientos <i>Put</i> de Kernel.Peripherals.
boot/com_cons.c	Contiene las rutinas <i>com_cons_init</i> y <i>com_cons_putchar</i> , para la inicialización de la línea serie y el envío de caracteres respectivamente.

6.4.4 Gestión del tiempo.

ORK provee dos servicios básicos para la gestión del tiempo, que son: *Clock* el cual proporciona el tiempo que ha pasado desde que se inicio el sistema y *Delay_Until* que permite bloquear una tarea hasta un tiempo determinado, los cuales a su vez dan soporte a los servicios estándar de Ada: Ada.Real_Time.Clock y retardos absolutos (delay until).

6.4.4.1 Implementación del soporte de tiempo real.

A nivel hardware, en la arquitectura hay un temporizador programable PIT (*Programmable Interval Timer*). Este recibe unos pulsos externos generados por un oscilador estable de frecuencia conocida de forma que, contando los pulsos que le llegan, tiene una medida del tiempo que ha pasado. El PIT es un dispositivo que necesita una inicialización y una rutina de servicio encargada de rearmarlo cuando este llegue al final de su cuenta. Cuando este contador llega a cero envía una interrupción (por lo tanto esta interrupción es periódica de periodo conocido). El temporizador está basado en una pastilla 8253 de Intel que contiene 3 temporizadores independientes de 16 bits (ver tabla 6.5):

Tabla 6.5: Frecuencias entregadas por al 8253 en el PC.

Temporizador	Función	Frecuencia CLK
0	Reloj del sistema	1,19318 MHz
1	Refresco de la memoria dinámica	1,19318 MHz
2	Generación de tonos en el altavoz	1,19318 MHz

En ORK-i386 la configuración de los temporizadores difiere un poco de su uso en un PC normal. El temporizador 0 se inicializa en modo 0 (Comutación al fin de cuenta '*Interrupt on Terminal Count*'), que será el temporizador que lleve la cuenta y el cual provoca la interrupción del reloj cuando llegué a cero, es importante resenar que tan solo uno de los temporizadores es capaz de generar interrupciones, en nuestro caso es el temporizador 0. Dadas las características de los temporizadores, son capaces de producir 1193182 ticks/sec los que equivale a aproximadamente un tick cada 838.1 ns. Por otra parte el temporizador 2 que originalmente está pensado para el servir al altavoz

interno del PC, en ORK-i386 se programa en modo 3 (Astable, generador de onda cuadrada ‘*Square Wave Rate Generator*’), perdiendo su antigua función. Su nueva misión es medir el tiempo que se tarda en programar y armar el temporizador 0 cada vez que espira la cuenta de este, para añadir ese tiempo a la cuenta del tiempo transcurrido total y que se produzca el mínimo error. El periodo máximo de interrupción que puede mantener el PIT estándar de la arquitectura i386-PC es de aproximadamente 54.9 ms, pero para redondear el periodo se dejará en 50 ms. Esto es debido a que tenemos un contador de 16 bit, y una frecuencia de 1.193182 Megahertz.

El periodo se calcula mediante la formula: $T = N \times C$. Con N valor máximo del contador ($2^{16}-1$), y C periodo de cada *tick* de reloj, $C = 1 / F$ (F es la frecuencia del reloj).

La arquitectura del PC también cuenta con un reloj de tiempo real RTC (*Real Time Clock*), capaz de mantener la fecha cuando el sistema se encuentre apagado. A pesar de su nombre, no es de gran utilidad en un sistema como ORK-i386. Ya que su granularidad es excesiva, como mucho podría valer para poner en hora el sistema cuando este arranque.

Para el manejo a bajo nivel de los temporizadores, se dispone de dos primitivas.

- ***Set_Alarm***: Se encarga de actualizar la hora del sistema, y de configurar el temporizador 0 con el nuevo intervalo.
- ***Read_Clock***: Lee la cuenta del temporizador 0.

A más alto nivel se dispone de las dos primitivas que implementan los servicios básicos del paquete Ada.Real_Time (Ada.Real_Time.Clock y retardos absolutos (delay until)) y que usan a su vez las primitivas anteriores:

- ***Clock***: Esta primitiva facilita la cantidad de nanosegundos que han pasado desde que se inició el sistema. Dado que los temporizadores que facilita la arquitectura estándar del PC, son claramente insuficientes para mantener la cuenta del tiempo que transcurre, se usa una variable para mantener la parte más significativa de la cuenta del tiempo. Esta variable se irá actualizando, periódicamente, cada vez que vaya espirando la cuenta del temporizador.
- ***Delay_Until***: Este procedimiento provoca que la tarea que lo invoca se quede bloqueado hasta que llegue el tiempo que se pasa como parámetro. En caso de que el tiempo en que debe despertarse, sea menor que el actual, la tarea será colocada en la cola de las tareas listas de su misma prioridad.

La implementación de los procedimientos anteriores es prácticamente la misma que en el ORK original, la mayor modificación corresponde al manejador de la alarma, ya que al contrario que en la versión para ERC-32, aquí solo se tiene un temporizador con el que tiene que mantener tanto la hora del sistema, como gestionar las alarmas. El

manejador de la alarma corresponde con la IRQ_0, y su lógica se corresponde con el diagrama de flujo de la figura 6.6.

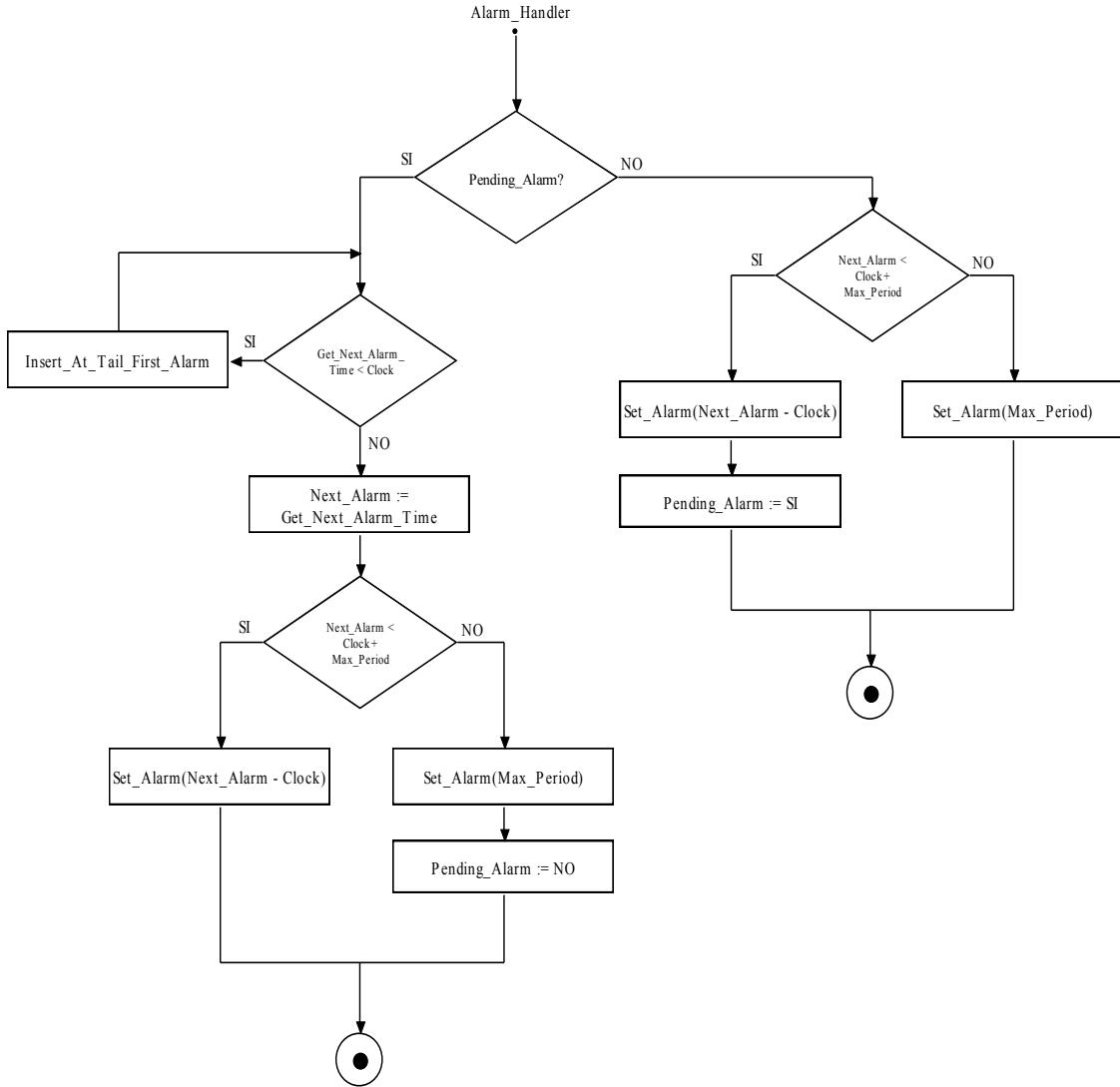


Figura 6.6 : Alarm_Handler.

6.4.4.2 Implementación del soporte de tiempo real para procesadores Pentium.

En una arquitectura *Pentium* se puede optimizar mucho la gestión del tiempo, mejorando sobre todo la precisión en la medición del tiempo. Usando el nuevo hardware que incorpora esta arquitectura, concretamente un contador de tiempo (TSC Time Stamp Counter) con un ancho de 64 bits. Este contador nos permite utilizar un registro del

hardware para contar el número de ticks, sin necesidad, de mantener la parte más significativa en memoria. Para el manejo a bajo nivel de los temporizadores, se dispone de dos primitivas.

- ***Set_Alarm***: Se encarga de configurar el temporizador 0 con el nuevo intervalo.
- ***Read_Clock***: Lee la cuenta del TSC (*Time Stamp Counter*), la transforma a nanosegundos y la devuelve.

A más alto nivel se dispone de las dos primitivas que implementan los servicios básicos del paquete Ada.Real_Time (Ada.Real_Time.Clock y retardos absolutos (*delay until*)) y que usan a su vez las primitivas anteriores:

- ***Clock***: Esta primitiva facilita el número de nanosegundos que han pasado desde que se inició el sistema. Su implementación es mucho más optima ya que no necesita leer el tiempo en dos veces como ocurre con la implementación sobre los temporizadores de la pastilla i8253, y por lo tanto se evitan problemas de carrera [11], y se gana en precisión.
- ***Delay_Until***: Este procedimiento provoca que la tarea que lo invoca se quede bloqueado hasta que llegue el tiempo que se pasa como parámetro. En caso de que el tiempo en que debe despertarse, sea menor que el actual, la tarea será colocada en la cola de las tareas listas de su misma prioridad. Además en el caso que la cola de las alarmas este previamente vacía y que la tarea pase a estar dormida se debe activar el manejador de la alarma, rearmando de nuevo el temporizador que permanecía parado hasta ese momento.

Con el uso del TSC, ya no es necesario que el manejador de la alarma interrumpa periódicamente para actualizar el tiempo que se mantiene en una variable del sistema (ya que el tiempo se mantiene en un registro del TSC). Tan solo será necesario que la interrupción se produzca cuando alguna tarea este esperando en la cola de alarmas. En este caso, el manejador interrumpirá periódicamente para comprobar si se cumple el tiempo de la alarma, si se cumple el tiempo de la alarma, y esta fuera la única tarea que está esperando, se saca la tarea de la cola de alarmas, y se paran los temporizadores hasta que otra tarea ejecuta la orden (*delay until*). Esta optimización puede mejorar sensiblemente la planificabilidad del sistema, siempre y cuando existan períodos de tiempo en los que las tareas no ejecuten retardos absolutos. El manejador de la alarma se comporta como muestra el diagrama de flujo de la página siguiente.

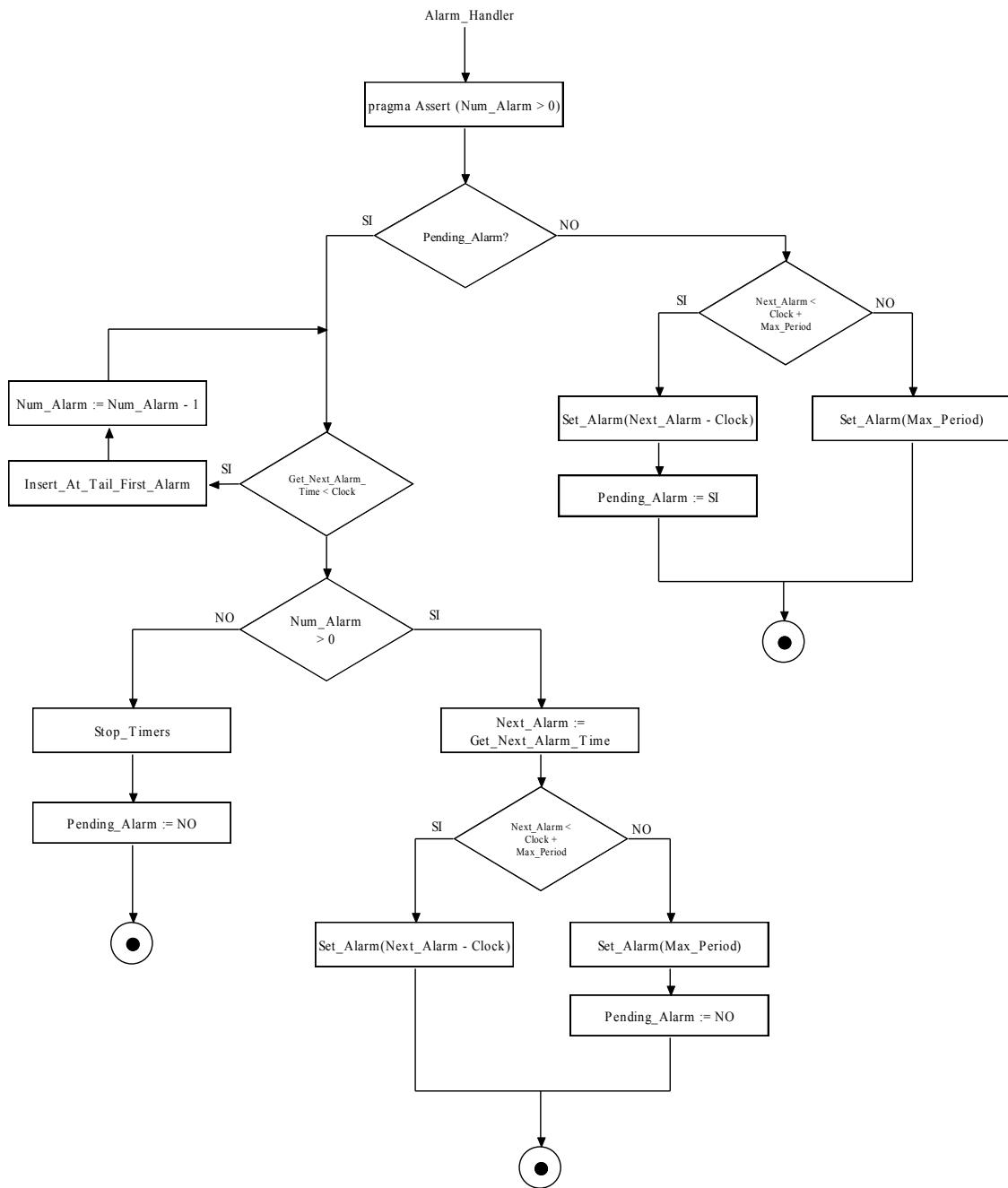


Figura 6.7 : Alarm_Handler usando TSC.

ARCHIVOS:

Archivos con la implementación del soporte de tiempo real en la arquitectura i386.

Ficheros	Descripción
gcc-2.8.1/ada/kernel-peripherals.adb	Cuerpo de las funciones <i>Set_Alarm</i> y <i>Read_Clock</i> .

gcc-2.8.1/ada/kernel-peripherals.ads	Definición de las funciones <i>Set_Alarm</i> y <i>Read_Clock</i> .
gcc-2.8.1/ada/kernel-peripherals-registers.ads	Definiciones con los parámetros de configuración de los <i>timers</i> .
gcc-2.8.1/ada/kernel-time.adb	Cuerpo de las funciones <i>Clock</i> , <i>Delay Until</i> y menejador de la alarma.
gcc-2.8.1/ada/kernel-time.ads	Definición del tipo <i>Time</i> y las funciones <i>Clock</i> y <i>Delay Until</i> .

6.4.5 Soporte para las tareas.

En este punto se explica las partes del código de bajo nivel, que dan soporte a la gestión de las tareas. El punto crítico de esta parte es el contexto de bajo nivel de la tarea, que contiene el estado de la máquina necesario para que una tarea pueda ejecutarse, y las rutinas que gestionan el contexto de bajo nivel. Para realizar el cambio de contexto ha sido necesario modificar el tipo *Context_Buffer* donde se almacena el contexto de cada tarea. El actual buffer del contexto contiene los siguientes campos.

- 7 registros generales (32 bits): eax, ebp, edi, esi, edx, ecx, ebx.
- FPU (Unidad de Coma Flotante): 108 bytes.
- Dirección de retorno (32 bits).
- Puntero de pila (32 bits).
- Límites de la pila (8 bytes). En x86 no tiene mucho sentido pero se conserva por mantener la máxima compatibilidad con la versión de ORK para ERC32,
- Prioridad (32 bits).
- FLAGS (32 bits).

Otro aspecto importante es el procedimiento que inicializa los contextos (*Initialize_Context*) de las tareas, es decir inicializa los datos fundamentales de la estructura anterior para que una tarea puede empezar a funcionar. Los campos que son necesarios inicializar son los siguientes:

- Inicializar el **puntero de pila**.
- Inicializar la **dirección de retorno**. En el caso de la inicialización será la primera instrucción de la tarea en cuestión, se debe ejecutar cuando el cambio de contexto le ceda el control.
- Colocar en el contexto la **prioridad** de la tarea. Esta prioridad está relacionada directamente con el rango de prioridades de las interrupciones, ya que el objetivo de este campo es poder ajustar la mascara de los PIC, bajo determinadas circunstancias (concretamente cuando se produce el cambio de contexto dentro del tratamiento de una interrupción) a partir del valor de este campo.
- Aunque se siguen manteniendo el campo de los **límites de la pila**, y se inicializan estos datos, su funcionalidad es nula en la actualidad.

Cambio de contexto.

Un aspecto fundamental en cualquier sistema multitarea es la implementación de la rutina de cambio de contexto, una forma habitual de implementar los cambios de contexto es mediante las funciones estándar *setjmp* y *longjmp*, que salvan y restauran respectivamente un contexto, en nuestro caso todo lo hace la misma rutina *Context_Switch*, las características de la rutina son las siguientes

1. La rutina de cambio de contexto está íntimamente relacionada con la rutina de tratamiento de interrupciones, por lo que deben compartir información para lo que se añaden nuevas variables globales, que pueden ser accedidas desde rutinas en ensamblador. A esta información acceden tanto la rutina de cambio de contexto como la rutina de interrupciones:
 - *Num_Interrupt_Nest*: número de interrupciones anidadas.
 - *Current_Task*: se almacena el puntero al buffer de la tarea en ejecución.
 - *Next_Task*: se almacena el puntero al buffer de la próxima tarea en ejecución.
 - *Thread_Id*: se almacena el identificador del actual thread en ejecución
 - *Context_Switch_Necessary*: flag que indica si es necesario el cambio de contexto.
 - *Ptr_Stack_Int*: variable donde se almacena el puntero de pila de una tarea interrumpida.
2. La rutina de cambio de contexto salva y restaura todo el estado de la máquina, concretamente actúa sobre:
 - Registros generales: eax, ebp, edi, esi, edx, ecx, ebx.
 - Puntero de pila.
 - Dirección de retorno
 - Unidad de coma flotante: FPU. Actualmente siempre se salva y restaura el estado de la FPU. En un futuro es posible que esto se optimice y solo se salve y restaure en los casos necesarios.
3. Actualiza la variable *Running_Thread*, que indica la tarea que esta ejecutando actualmente.
4. Implementación de la rutina de cambio de contexto mediante ensamblador en línea, siguiendo el esquema que se muestra en la figura 6.8.

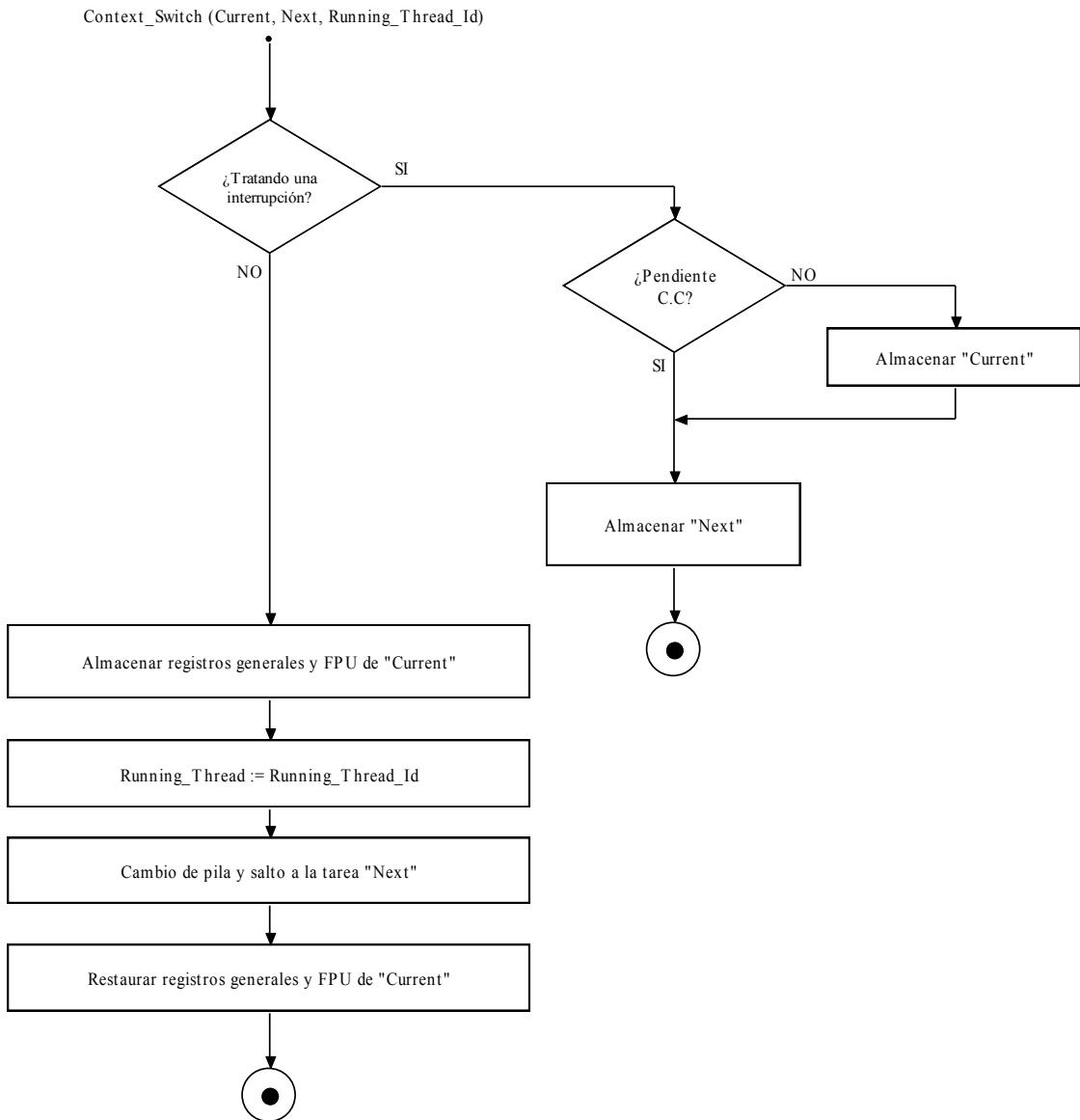


Figura 6.8: Cambio de Contexto en ORK-i386

6.4.5.1 Gestión de prioridades.

La arquitectura x86 no soporta directamente las prioridades de interrupción desde el registro de estado. Con lo cual para permitir el anidamiento de interrupciones correctamente, es decir para que una tarea con prioridad en el rango de las prioridades de las interrupciones, pueda bloquear a una interrupción con menor prioridad, pero no al revés, hay que acceder directamente a las máscara de los controladores de interrupciones (PIC) e inhibir un rango de interrupciones. Para este fin la rutina *Enable_Interrupts* debe llamar a otra función llamada *Interruption_Level* que calcula la máscara adecuada de los PIC, en función del nivel de interrupción. Las prioridades de las interrupciones ya se han comentado anteriormente en el punto 6.4.2.1, para más

información se recomienda leer ese punto, y más concretamente la tabla 6.2, donde se muestra la prioridad de cada interrupción en el sistema ORK-i386.

ARCHIVOS:

Ficheros	Descripción
gcc-2.8.1/ada/kernel_cpu-primitives.adb	Cuerpo de las funciones <i>Context_Switch</i> , <i>Initialize_Context</i> , <i>Change_Priority</i> y <i>Enable_Interrupts</i> .
gcc-2.8.1/ada/kernel_cpu-primitives.ads	Definición de las funciones <i>Context_Switch</i> , <i>Initialize_Context</i> , <i>Change_Priority</i> , <i>Save_Flags</i> , <i>Restore_Flags</i> , <i>Disable_Interrupts</i> y <i>Enable_Interrupts</i>
gcc-2.8.1/ada/kernel_peripherals.adb	Cuerpo de las funciones <i>Interruption_Level</i> , <i>CLI</i> y <i>STI</i> .
gcc-2.8.1/ada/kernel_peripherals.ads	Definición de las funciones <i>Interruption_Level</i> , <i>CLI</i> y <i>STI</i> .

6.4.6 Soporte de depuración con GDB.

ORK-i386 ofrece soporte de depuración remota, mediante el depurador de GNU GDB, una vez más esta funcionalidad del sistema ORK-i386 esta inspirado en el sistema *OS_Toolkit* y la utilización que de este hace *MaRTe OS*. El soporte de depuración remota permite que el depurador pueda estar ejecutando en una máquina mientras que la aplicación empotrada se ejecuta en su máquina de ejecución, a pesar de que las dos máquinas pueden tener diferentes arquitecturas. Solo hace falta una pequeña capa de enlace con el sistema operativo, esta pieza de código debe manejar adecuadamente los *traps*, interrupciones y comunicaciones asociados a la depuración remota. En esta situación la aplicación que esta ejecutando actúa como esclavo del depurador y simplemente interpreta las órdenes que este le manda.

El depurador de GNU, GDB, soporta una gran variedad de protocolos de depuración remota. El más común de los protocolos soportados es de línea serie (RS-232), el cual opera sobre una línea serie (normalmente un cable modem-nulo) que conecta las dos máquinas implicadas en el proceso.

Se detalla, sin entrar en profundidad, como funciona la depuración remota. En primer lugar para activar la depuración remota es preciso indicarlo explícitamente en el código de la aplicación (Ver apéndice B, sección B.2.2), mediante el procedimiento *Init_Serial_Communication_With_Gdb* que recibe como parámetro *Serial_Port_1* o *Serial_Port_2*, este procedimiento invoca a otro procedimiento denominado *gdb_pc_com_init*, que inicializa el entorno que permitirá realizar la depuración remota a través de la línea serie. Los pasos que lleva a cabo son los siguientes:

- Selecciona la *IRQ* que se encarga de tratar la comunicación por línea serie.
- Coloca en la tabla de *traps*, un manejador o rutina de tratamiento de excepciones general para todas las excepciones de 0 a 32 denominado *gdb_trap_ss* (esto entra en conflicto, con el manejo de excepciones que se ha explicado en el punto 6.4.2.6, ya que ahora todas las excepciones son capturadas por la pieza de código que gestiona la depuración) que se encarga de preparar el sistema cada vez que ocurre una excepción del sistema y pasar el control a *gdb_trap*. Esta última función envía a GDB por la línea serie el estado del sistema en el momento de producirse la excepción.
- Inicializa la línea serie que ha sido seleccionada para realizar la comunicación entre los dos sistemas, seleccionando la velocidad y todos los datos necesarios.
- Modifica la entrada de la tabla IDT adecuada para colocar una rutina de tratamiento de interrupciones específica para atender las interrupciones de la línea serie mencionada en el punto anterior.

Después de lo visto anteriormente, la otra función útil en la depuración es *Set_Break_Point_Here*, que como su propio nombre indica sirve para provocar un breakpoint de forma explícita en el código, será necesario colocar al menos una llamada a esta función después de inicializar la línea serie que da soporte a la depuración, para poder sincronizar GDB con la aplicación que se quiere depurar. Esta llamada provoca un trap, al iniciar la depuración se coloco un manejador general para todos los trap, que será el que se encargue de tomar el control de la aplicación. A partir de este momento cada vez que se produzca un trap, el manejador de excepciones tomará el control de la aplicación, estableciendo comunicación a través de la línea serie con GDB, y esperando que este le envíe las órdenes oportunas que debe ejecutar.

ARCHIVOS:

Ficheros	Descripción
gcc-2.8.1/ada/kernel_peripherals.adb	Cuerpo de las funciones <i>Set_Break_Point_Here</i> y <i>Init_Serial_Communication_With_Gdb</i> .
gcc-2.8.1/ada/kernel_peripherals.ads	Definición de las funciones <i>Set_Break_Point_Here</i> y <i>Init_Serial_Communication_With_Gdb</i> .
boot/gdb_pc_com.c	Función <i>gdb_pc_com_init</i> .
boot/gdb_trap_ss.S	Rutina de tratamiento de excepciones general, en caso de depuración.
boot/gdb_trap.c	Función que implementa el tratamiento de un <i>trap</i> en caso de una excepción.
boot/gdb_serial.c	Soporte para la comunicación por línea serie de GDB.

6.4.7 Montaje y arranque del sistema.

El sistema ORK-i386 sigue las mismas pautas que el ORK original a la hora de compilar y montar una aplicación, a pesar de las diferencias entre las dos arquitecturas sobre las que se monta el sistema. Por lo tanto, la mayor parte de este punto ha sido extraído de la referencia bibliográfica [13]. El sistema de compilación y montaje como ya se ha mencionado en otros puntos está basado en las herramientas de GNU, las cuales son tremadamente configurables, sobre todo GCC y las facilidades y herramientas que este proporciona. El primer paso es indicarle al enlazador la memoria de la que dispone y en qué partes de dicha memoria debe colocar cada parte del código. Para este fin existe un fichero de órdenes del enlazador, que 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 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 6.4) y en el código de inicio de ORK. 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. En nuestro caso y dado que trabajamos sobre una arquitectura PC, la memoria será muy configurable a la vez que barata, por lo que no se necesita aprovechar completamente la memoria instalada, como si ocurre en el ERC-32. Se indica al enlazador que coloque todo el código y los datos a partir de la dirección 0x100000 (1 MB de memoria), y así se evita entrar en conflicto con las parte ROM, que existen en el PC.

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.
- **Datos (data)** En esta sección se encuentran las variables globales del programa inicializadas con 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.

A parte 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 (i386) y el formato del mismo, formato elf en nuestro caso

Archivo: commands.ld

```

OUTPUT_ARCH(i386)
/* Uncomment the next line if you want the linker to output srecords */
/* OUTPUT_FORMAT(srec) */

/* Entry point, found in file multiboot.S */
ENTRY(_start)
RAM_SIZE = 3M;

RAM_START = 0x00100000;
RAM_END = RAM_START + RAM_SIZE;

/* Memory map */
MEMORY
{
    ram (rwx) : ORIGIN = 0x00100000, LENGTH = 3M
}

/*
-----
-- This is the memory layout when using the simulator. --
-- Everything is placed in RAM. --
-----

    +-----+
    RAM_START | .text      |
    |   text_start      |
    |   *(.text)        | program instructions
    |   text_end        |
    +-----+
    | .data      |
    |   data_start      |
    |   *(.data)        | initialized data sections
    |   data_end        |
    +-----+
    | .bss       |
    |   bss_start      | start of bss, cleared
    |   *(.bss)        | uninitialized data sections
    |   bss_end        |
    +-----+
    |   _heap_start    | start of heap memory area
    |                   | used by sbrk.c for memory allocation
    /
    /
    RAM_END      |
    +-----+
*/
/*

```

```

        Assign segments to previously described memory addresses
*/
SECTIONS
{
    .text :                  /* Instructions must be word aligned */
    {
        CREATE_OBJECT_SYMBOLS /* Add a new symbol for each input file */
        text_start = .;
        *(.text) . = ALIGN (16);
        *(.eh_frame) . = ALIGN (16);
        *(.rodata) . = ALIGN (16);
        text_end = .;
    } > ram

    .data ALIGN(4):          /* Data is word aligned */
    {
        data_start = .;
        *(.data)
        data_end = .;
    } > ram

    .bss  ALIGN(8):          /* BSS is doubleword aligned for quick clearing */
    {
        bss_start = .;
        *(.bss)
        *(COMMON)
        bss_end = .;
    } > ram

    .stab . (NOLOAD):        /* Sections for the debugger */
    {
        [.stab]
    }

    .stabstr . (NOLOAD):
    {
        [.stabstr]
    }
}

/* Set the start of the heap area */
/*
The memory heap is used by sbrk.c to provide low level support for memory
allocation functions: malloc, calloc and realloc.

This functions should only be used during initialization of objects because
the Ravenscar profile does not allow dynamic memory allocation.
*/
heap_start = bss_end;
heap_end = RAM_END;

/* Stack */

```

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. 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:

```
$ i386-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-i386. Estos argumentos se extraen, precisamente, del fichero de especificaciones seleccionado (ver archivo “`ork_specs`”). Estos argumentos provocan las siguientes acciones:

1. Incluir los ficheros de apoyo mediante el uso de las bibliotecas:
 - libm: librería matemática con rutinas de apoyo.
 - libmc: librería mínima de C, funciones no implementadas y función `sbrk`, que 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.
 - libboot: librería con código de inicio, depuración e interrupciones.
 - libini: librería con parte del código de inicio y excepciones.
2. Señalar al archivo “`multiboot.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`).

Archivo: `ork_specs`

```
%rename cpp old_cpp
%rename lib old_lib
%rename endfile old_endfile
%rename startfile old_startfile
%rename link old_link

*cpp:
%(old_cpp)

*lib:
%{!k: %(old_lib)} %{k: --start-group -lc -lgcc -lm -lmc -lboot -lgnarl -lini --end-
group}

*startfile:
%{!k: %(old_startfile)} %{k: multiboot%0%s}

*link:
%{!k: %(old_link)} %{k: -dc -dp -N -e _start -T commands.ld%0}
```

Actualmente el arranque del sistema se lleva a cabo con el *bootloader* GRUB, a través de un disquete de arranque (ver apéndice A), que aunque es un sistema un poco rudimentario es suficiente para poder probar el sistema. También es posible arrancar el sistema a través de red mediante NETBOOT, pero esto actualmente no está instalado.

ARCHIVOS:

Archivos relacionados con la carga y montaje del sistema.

Ficheros	Descripción
commands.ld	Fichero de órdenes al enlazador.
ork_specs	Fichero de especificaciones de gcc.
libmc/non_implemented.c	Rutinas de apoyo, implementadas(sbrk y otras) y no implementadas (rutinas que no tienen ninguna funcionalidad, que pueden ser llamadas en algún momento desde otras librerías).

6.4.8 Prueba del sistema.

Una vez portado el sistema fue necesario realizar algunos programas sencillos que probaran la funcionalidad básica de ORK-i386. Para este propósito, se desarrolló un sencillo programa con tareas esporádicas y periódicas así como mecanismos de comunicación. Este programa se distribuye como aplicación de ejemplo con los fuentes del entorno de desarrollo.

El programa consta de tres tareas las cuales simulan su tiempo de computación mediante *whetstone benchmark*. Para probar la comunicación entre tareas se emplea un objeto protegido a través del cual interactúan dos de las tareas, una de ellas es esporádica y la otra es periódica, la tercera tarea también es periódica.

La tarea esporádica se activa mediante una interrupción hardware para lo cual la tarea permanece esperando en una entrada protegida. Un procedimiento protegido es usado como manejador de la interrupción el cual se encarga de abrir la barrera en la cual la tarea esporádica se encuentra bloqueada.

Para simular la interrupción hardware se ha creado una tarea periódica que se encarga de interrumpir periódicamente mediante la instrucción *int*.

Todas las tareas escriben el valor de Real_Time.Clock cada vez que inician y acaban su ejecución periódica, mediante la salida estándar por pantalla. El ejemplo cubre casi todos los aspectos de una aplicación embarcada, en concreto el ejemplo incluye:

- Gestión de tareas.
- Sincronización de tareas.
- Retardos absolutos.
- Gestión de interrupciones Ada.
- Cálculos en coma flotante.

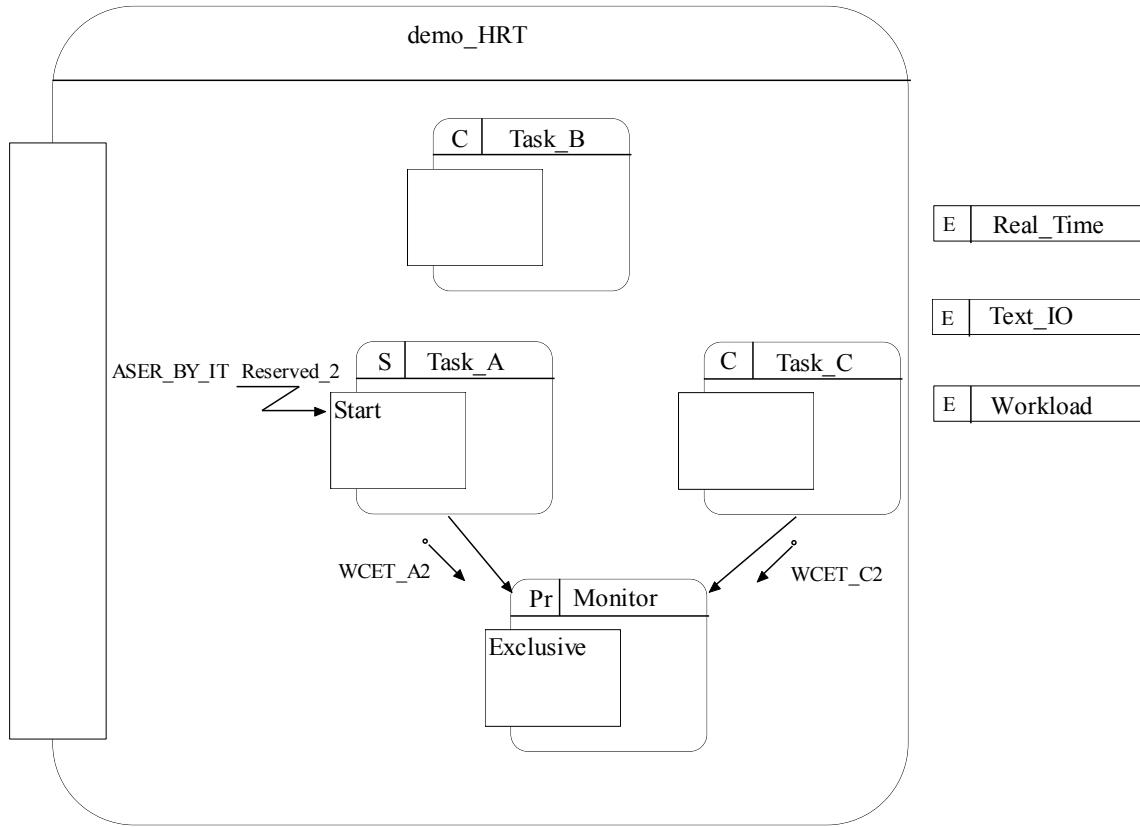


Figura 6.9: Conjunto de tareas del ejemplo.

6.4.8.1 Requisitos temporales de las tareas.

La figura 6.9 muestra la estructura del conjunto de las tareas. El conjunto de tareas será analizado por los requisitos temporales de las tareas que se muestran en la tabla 6.6. El periodo de la tarea A es interpretado como el mínimo tiempo entre activaciones.

Tabla 6.6: Requisitos temporales de las tareas.

Tareas	Periodo	Actividades
A	14	a1, a2
B	20	b1
C	36	c1, c2

La tarea A y C contienen dos bloques lógicos de actividades, mientras la tarea B solo contiene uno. La actividad a1 corresponde a un bloque de ejecución interno y la actividad a2 corresponde a un bloque de ejecución que se ejecuta dentro del objeto

protegido. Lo mismo le ocurre a la tarea C, c1 corresponde a ejecución interna y c2 corresponde a tiempo de ejecución dentro del objeto Monitor, finalmente b1 corresponde a tiempo de computo dentro de la tarea B. La tabla 6.7 muestra la asignación de prioridades a las tareas.

Tabla 6.7: Prioridad de las tareas y peores tiempos de computo (WCET).

Tareas	Prioridad	WCET	Recursos
A(a1)	Priority'Last	1	Ninguno
A(a2)	Priority'Last	2	Monitor
B(b1)	Priority'Last-1	6	Ninguno
C(c1)	Priority'Last-2	2	Ninguno
C(c2)	Priority'Last	6	Ninguno

Por lo tanto el máximo tiempo de respuesta se calcula usando la siguiente ecuación.

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_j}{T_j} \right\rceil \times C_j$$

La cual se resuelve aplicando la siguiente ecuación iterativa.

$$W_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i^n}{T_j} \right\rceil \times C_j$$

Como se emplea el protocolo de techo de prioridad, el máximo tiempo de bloqueo puede ser evaluado para cada tarea.

Tarea A: puede sufrir un tiempo de bloqueo igual al WCET de la actividad c2.

Tarea B: puede sufrir un tiempo de bloqueo igual al WCET de la actividad c2.

Tarea C: es la tarea con menor prioridad así que no puede sufrir bloqueos.

El máximo tiempo de respuesta para cada tarea puede ser ahora calculado. El mínimo tiempo entre llegadas será usado como el periodo para calcular el peor tiempo de respuesta de la tarea con menor prioridad. El valor inicial w_i es igual a la suma de los WCET de las tareas con mayor prioridad mas el WCET de la propia tarea.

$$W_a^1 = 3 + 6 = 9$$

$$\mathcal{W}_b^1 = 6 + 6 + \left\lceil \frac{9}{14} \right\rceil \times 3 = 15$$

$$\mathcal{W}_b^2 = 6 + 6 + \left\lceil \frac{15}{14} \right\rceil \times 3 = 18$$

$$\mathcal{W}_b^3 = 6 + 6 + \left\lceil \frac{18}{14} \right\rceil \times 3 = 18$$

$$\mathcal{W}_c^1 = 8 + \left\lceil \frac{17}{14} \right\rceil \times 3 + \left\lceil \frac{17}{20} \right\rceil \times 6 = 20$$

$$\mathcal{W}_c^2 = 8 + \left\lceil \frac{20}{14} \right\rceil \times 3 + \left\lceil \frac{20}{20} \right\rceil \times 6 = 20$$

La figura 6.10 muestra como se planifican las tareas, empezando en tiempo cero hasta tiempo igual a 60 donde cada unidad de tiempo son 100 ms. Las flechas hacia arriba muestran los periodos de activación y las flechas hacia abajo muestran los plazos de respuesta.

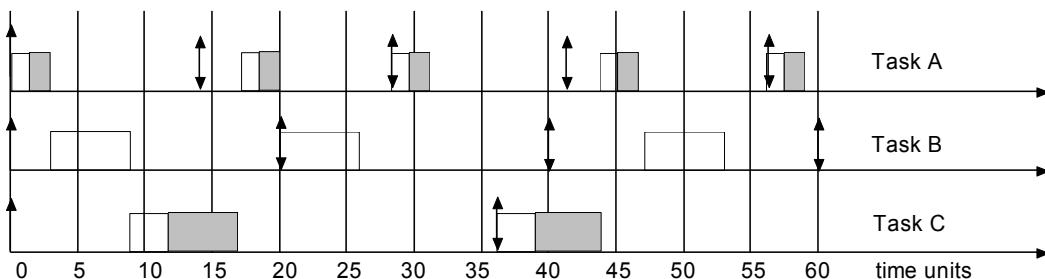


Figura 6.10: Planificación de las tareas.

La salida del programa que se obtiene es la siguiente donde se muestra los periodos de activación y finalización de cada tarea, para una unidad de tiempo igual a 100 ms.

```

Task A running RT.Clock      = 0.001300500
Task A finishing RT.Clock   = 0.306737700
Task B running RT.Clock     = 0.308817300
Task B finishing RT.Clock   = 0.915951300
Task C running RT.Clock     = 0.917942600
Task C finishing RT.Clock   = 1.728120300
Task A running RT.Clock     = 1.730182000
Task A finishing RT.Clock   = 2.035818800
Task B running RT.Clock     = 2.037934000
Task B finishing RT.Clock   = 2.645087400

```

```
Task A running RT.Clock = 2.801267200
Task A finishing RT.Clock = 3.106838900
Task C running RT.Clock = 3.600678000
Task C finishing RT.Clock = 4.410930900
Task A running RT.Clock = 4.413009400
Task A finishing RT.Clock = 4.718632300
Task B running RT.Clock = 4.720762600
Task B finishing RT.Clock = 5.327842300
Task A running RT.Clock = 5.601311000
Task A finishing RT.Clock = 5.906893700
...
...
```

Puede haber pequeñas variaciones con respecto a los tiempos de la figura 6.10 pero es debido al tiempo empleado en la realización de operaciones del kernel y el acceso a recursos compartidos que incrementan los WCET calculador en la tabla 6.7.

6.4.9 Otros aspectos.

Prácticamente han quedado explicados todos los aspectos fundamentales del sistema ORK-i386, el único aspecto del ORK original que no ha sido posible simular en la arquitectura i386 actualmente, es el que hace referencia a la protección de la memoria, con lo cual en el caso de que la pila de una tarea sea desbordada no se detectará. Aunque se proponen soluciones a este problema en el capítulo 9, de momento la única “solución”, consiste en asignar una cantidad generosa de memoria a la pila de las tareas, de tal manera que sea improbable que las tareas utilicen toda su pila, si se tiene en cuenta que la memoria en una arquitectura PC, no es muy cara, esta no es una solución muy descabellada, siempre y cuando se tenga un poco de cuidado a la hora de diseñar las aplicaciones, evitando grandes recursividades o el uso de variables locales y parámetros muy grandes.

Para la prueba del sistema se ha utilizado a parte de una maquina desnuda(i486 y Pentium) el simulador de PC VMware©, con un resultado bastante satisfactorio, este último no es gratuito pero se puede usar solicitando licencias gratuitas a través de su pagina Web, que hay que renovar cada mes aproximadamente.

Capítulo 7

Desarrollo de una aplicación sobre ORK-i386

7.1 Introducción.

Una vez que el sistema ha sido portado a la nueva arquitectura, se tomó la decisión de desarrollar una aplicación empotrada real, a través de la cual se pudiera probar la eficiencia del entorno de desarrollo cruzado ORK-i386. Además el construir una aplicación real sirve para depurar el sistema de forma más completa y encontrar algún *bug* oculto del sistema.

El sistema construido, consiste en controlar un brazo robótico con cuatro grados de libertad, para lo cual es preciso controlar la velocidad de los motores de corriente continua que mueven el brazo, así como controlar la posición en que se encuentra el brazo en cada momento. Para el desarrollo del hardware del sistema se ha partido del trabajo realizado por Roberto Rica [19], pero con varias modificaciones al trabajo de éste. Primero se define el entorno hardware del sistema desarrollado, para después pasar a explicar el diseño del software desarrollado para el control del brazo robótico, cuya principal característica es que está desarrollado en Ada95 y bajo el perfil Ravenscar.

7.2 Entorno hardware.

Se puede decir que el sistema desarrollado está dividido en dos grandes partes, que son: hardware y el software. Para empezar, se pensó en los componentes hardware que serían necesarios para llevar a cabo el control del brazo. Puesto que se trata de controlar un brazo robótico, una parte imprescindible en el hardware que se necesita es la parte mecánica de un brazo. Por otro lado, para controlarlo hace falta un sistema empotrado. El sistema empotrado debe dar las órdenes correspondientes a los distintos motores de que consta el brazo y debe recoger la información que le permite conocer el estado de este, pero las señales que necesitan los motores para encenderse requieren de una cierta

potencia que no puede suministrar un Target común que normalmente tiene entradas/salidas digitales de tipo TTL. Por lo tanto, seguramente, dependiendo del Target escogido, se necesitará una interface que “traduzca” unas señales en otras. Según el entorno descrito en el capítulo anterior, para realizar el desarrollo de la aplicación que se encargará de tomar las decisiones de cómo encender los motores y en qué momentos darles o no tensión, es necesario un equipo informático que haga las funciones de Host. Por último, todas estas partes deben ir conectadas de alguna forma para que la información de una llegue a la otra. Esta organización se puede ver de forma gráfica en la figura 7.1.

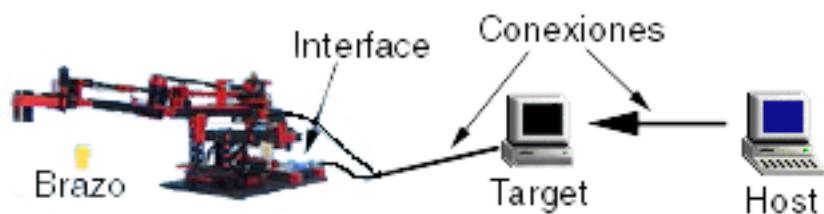


Figura 8.1: Representación del entorno de desarrollo y ejecución.

Más concretamente, los módulos de que consta el hardware son los siguientes:

- **Brazo:** Es el sistema que se tiene que controlar mediante la activación o desactivación de sus motores usando para ello la información que es facilitada a través de sus sensores (que como se verá más adelante, en este caso son interruptores) que indican, de alguna forma, que es lo que está pasando en el exterior.
- **Target:** El target de ejecución es un PC con microprocesador Pentium a 100 MHz, sobre el que ejecuta la aplicación desarrollada con el entorno de desarrollo cruzado GNAT/ORK. dicha aplicación se encarga de dar las señales de control al brazo, y por lo tanto, de tomar las decisiones pertinentes de cuando activar o desactivar alguno de los motores del brazo y en qué sentido.
- **Interface:** Se ha usado este nombre para describir los circuitos diseñados para realizar las necesarias conversiones de señales entre el brazo (motores y sensores) y el Target, y que, por lo tanto, su diseño dependerá de qué tipo de motores y sensores tenga el brazo elegido y del Target escogido. Por otro lado, también se puede decir que depende de cómo se plantee hacer el control desde la aplicación, ya que por ejemplo, a esta interface se le podría añadir un módulo hardware que hiciera el control de velocidad PWM y por lo tanto, no sería necesario implementarlo por software, entre otras cosas.

- **Host:** Debido al entorno de desarrollo elegido para llevar a cabo este proyecto, en el que el control del brazo será realizado mediante un sistema empotrado, es necesario tener un sistema, que será denominado como Host, donde se encuentre instalado el entorno de desarrollo cruzado ORK-i386, con todas las herramientas necesarias para desarrollar la aplicación.

Como se verá, el Host elegido en este proyecto es una estación de trabajo común (basado en un Pentium II 400Mhz). El sistema operativo instalado en el host es un Linux (Red-Hat 7.1), sobre el que está instalado todo el entorno de desarrollo cruzado, y las herramientas necesarias para la codificación de la aplicación, tales como editores de texto, o el depurador GDB.

- **Conexiones:** En este apartado es donde se explicará como están conectados todos estos subsistemas, cables y otras formas de intercambio de información usadas para que todo funcione correctamente, se puede decir que lo que en este apartado se cuente será el pegamiento de todos los módulos comentados anteriormente, y por lo tanto, será lo que cree el sistema total.

Una vez se tiene una visión general del sistema, se pasará a describir de una forma más detallada cada uno de estos componentes.

7.2.1 Sistema físico: Brazo.

Lo primero que se necesita es el brazo a controlar, entendiendo como brazo a la parte mecánica constituida por varios segmentos rígidos unidos mediante sistemas mecánicos que permiten el cambio de posición de uno de estos segmentos respecto a otro, normalmente del ángulo que forman. De la elección de este, dependerá en gran medida tanto la interface que se diseñe e implemente como el Target escogido. En el caso de la interface, esta deberá ser capaz de dar las señales con la potencia necesaria al brazo además de transformar las señales que recibamos a medida que se mueve el brazo, y que dependerán de los sensores utilizados y de los niveles utilizados por el Target. En el caso del Target, este deberá tener la suficiente potencia para llevar a cabo los cálculos necesarios para controlar el brazo, no hay que olvidar que cuanto más rápido se mueva el brazo, menos tiempo tiene el Target para tratar las señales periódicas que envía al realizar un movimiento para controlar su posición.

El brazo que se utilizará en la aplicación nos ofrece cuatro grados de libertad, giro, avance y retroceso, elevación de la pinza y apertura y cierre de la pinza, esta basado en un kit desarrollado por la empresa Fischertechnik llamado “Industry Robots”. Este kit incluye una serie de planos para poder construir 4 tipos distintos de brazos, como ventajas que proporciona esta opción frente a otros brazos encontrados, se pueden destacar las siguientes:

- Al ser un sistema de construcción basado en ir encajando las distintas piezas disponibles, no se limita a un único modelo, ya que se puede desmontar sin ningún problema y montar otro modelo distinto para experimentar con otro tipo de movimientos y sistemas.
- El brazo escogido para el proyecto consta de cuatro grados de libertad, pero gracias a la modularidad del sistema, la ampliación a cinco “solo” requeriría diseñar una nueva colocación de las piezas, además de adquirir las nuevas que fuesen necesarias, reutilizando las que ya se tienen y por lo tanto, sin perder la inversión realizada.
- Las piezas utilizadas son de plástico bastante resistente, lo que hace que sea un sistema robusto a la vez que barato.
- Los motores utilizados son de 9V, de corriente continua, con lo cual el sistema es bastante realista.

El modelo que se ha construido tiene el siguiente aspecto:

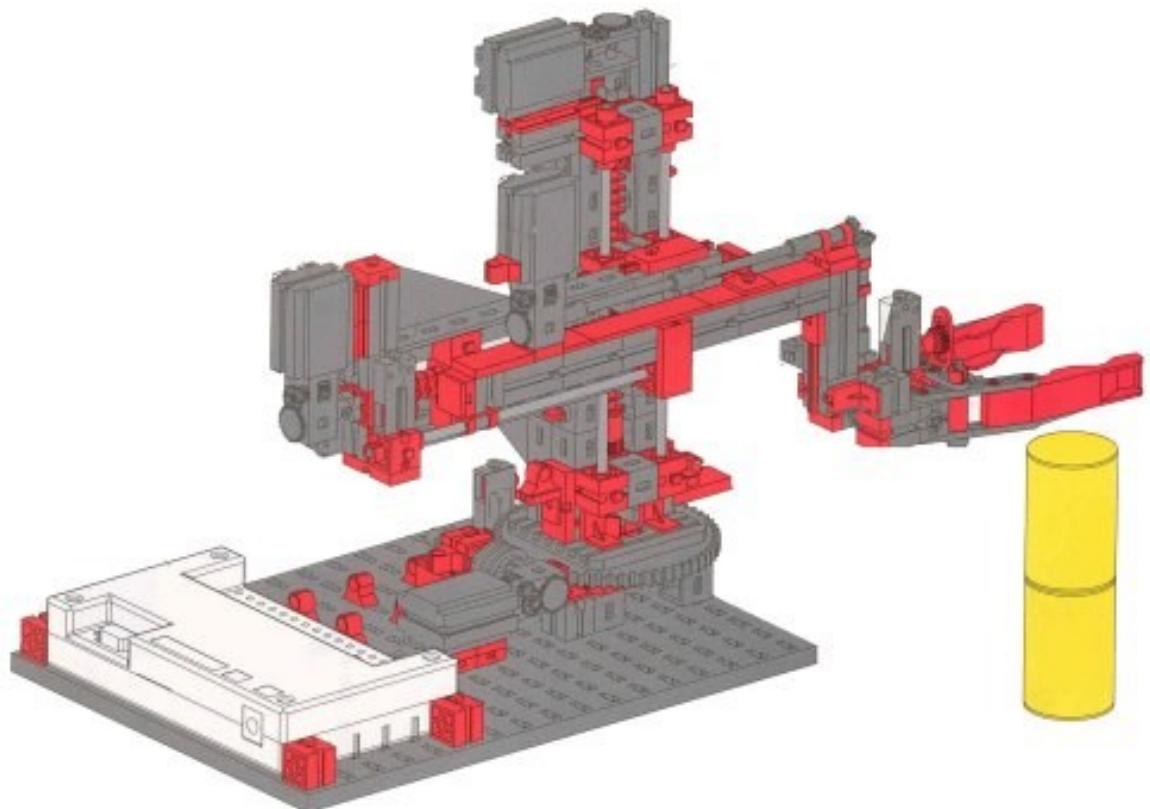


Figura 7.2: Brazo robótico construido.

En cuanto a las desventajas,

- Los motores tienen más bien poca potencia, con lo cual se puede simular casi cualquier movimiento (dentro de los posibles con los grados de libertad que se tienen) pero no se pueden realizar simulaciones con piezas que sean demasiado pesadas.
- La velocidad de movimientos que se consigue no es muy elevada debido a que en motor de giro, son necesarias unas reductoras para conseguir una potencia mínima, con lo cual pueden quedar excluidos algún tipo de experimentos que requieran unos rápidos movimientos. Sin embargo la velocidad es suficiente como para hacer prácticas de control de velocidad mediante PWM (Modulación del Ancho de Pulso, ver hilo Velocidad más adelante).
- Al ser las piezas de plástico y llevar todas uniones por presión, a veces se desajusta el sistema por el propio uso de este, o por que se fuerza algún movimiento más de lo permitido, por algún fallo en la programación por ejemplo.
- Los sensores encargados de hacer posible que el sistema empotrado pueda llevar el control de la posición en la que se encuentra el brazo en todo momento son interruptores que provocan un ruido innecesario además de una falta de precisión si se compara con otro tipo de sensores como podrían ser sensores ópticos

El brazo consta de un interruptor de fin de carrera por cada grado de libertad que indica cuando está el brazo en la posición inicial de movimiento y otro que va marcando la posición en la que se encuentra, ya que irá dando una serie de pulsos (cuatro pulsaciones por vuelta de la reductora) a medida que los distintos motores hacen que el brazo se mueva en sus posibles direcciones. La detección del fin de carrera en el otro extremo del movimiento, es decir, en el punto final, tendrá que ser realizada mediante la cuenta de los pulsos recibidos por el interruptor de pasos que indica la posición del brazo y por lo tanto, sabiendo de antemano cuantos pasos puede dar el brazo antes de llegar al final. Esto puede ser un problema, ya que hay que asegurarse de que todos los pulsos son recibidos y tratados, porque si el Target perdiere uno, el brazo estaría en una posición distinta a la que este cree y por lo tanto se habría perdido la precisión y no sería posible detectar convenientemente el fin de carrera en uno de las direcciones de dicho movimiento.

7.2.2 Entorno de ejecución: Target.

El siguiente componente que forma el sistema y que se va a explicar es el Target. Se usa esta palabra para identificar el sistema informático que ejecutará el programa de control del brazo y que, por lo tanto, será el encargado de dar las órdenes a los motores para que se enciendan o apaguen, teniendo en cuenta para ello la información recibida de los sensores (que como se ha visto, en este caso son interruptores) que indican en qué posición se encuentra el brazo. Se podría decir, por tanto, que este componente es el cerebro del sistema.

El target elegido es una arquitectura i386-PC ya que el objetivo del desarrollo de la aplicación es probar el nuevo sistema ORK-i386. El microprocesador deberá ser un 386 o superior, se probará tanto la versión que usa el TSC del Pentium, como la versión que utiliza los PITs para mantener la hora del sistema. El sistema básico contiene una disquetera, una tarjeta de vídeo y un monitor (estos dos últimos componentes solo son necesarios para la depuración del sistema), es decir, no hace falta que tenga un disco duro, ya que el sistema se puede arrancar desde un disquete de 3 ½''. Actualmente se está generando otro proyecto en el que se está incluyendo una tarjeta de red ethernet para el arranque y depuración remota del sistema.

Finalmente se escogió una máquina con las siguientes características:

- Microprocesador: Pentium 100Mhz.
- Memoria: 32Mb de RAM (aunque solo se usan 3 MB).
- Una disquetera de 3 ½ '' de alta densidad.
- Una tarjeta de vídeo genérica.
- Monitor VGA genérico.

A este sistema, se le insertó en uno de los buses de expansión ISA una tarjeta de entrada/salida de datos digitales con nivel de señales TTL que será explicada más adelante en el apartado dedicado a la Interface.

7.2.3 Adecuación de señales: Tarjeta de interface.

Una vez se tiene el brazo mecánico y el Target que lo controlará, se plantea el problema de conectarlos para que las órdenes dadas por el Target sean cumplidas por el brazo.

Para realizar esta conexión, lo primero es pensar en el número de señales que tiene que producir el Target para controlar el brazo. Puesto que se tiene un motor por grado de libertad, el brazo tiene cuatro grados de libertad (giro, fondo, altura y pinza), y para controlar un motor el Target puede darle la orden de parar, girar a la izquierda o girar a la derecha (que se puede codificar con dos bits) se puede deducir que se necesitan 8 salidas digitales en el Target (4 motores x 2 bits). Por otro lado, puesto que cada grado de libertad tiene de dos sensores que indican el fin de carrera y los pulsos que se producen con el movimiento, se necesitan 8 entradas digitales (4 grados de

libertad x 2 bits). También habrá otra entrada adicional que será la que genere la interrupción cada vez que uno de los interruptores cambie de posición, y otra salida adicional para indicar al controlador de las interrupciones que ya se ha tratado la interrupción. Por último, ya que se quiere que el brazo pueda moverse en todos sus grados de libertad al mismo tiempo, será necesario que estas entradas y salidas digitales anteriormente comentadas sean distintas, es decir, no se pueden conectar todos los interruptores de pulso a la misma entrada porque en caso de que así se hiciese, al mover al brazo en dos grados de libertad a la vez, no se tendría forma de reconocer que grado de libertad dio el pulso.

Todo esto hizo que se requiriese una tarjeta de entrada/salida digital capaz de generar interrupciones y con el número adecuado de puertos de entrada/salida.

Por lo tanto, teniendo en cuenta lo anteriormente expuesto, para realizar la interface entre el Target y el brazo robótico, se ha utilizado una placa genérica de entrada/salida con niveles de señal TTL. Concretamente la placa utilizada es el modelo PCL-730, que para nuestro sistema ofrece 32 canales de entrada/salida TTL (16 entradas y 16 salidas) además de la capacidad de generar una interrupción. A la placa se le ha conectado un bornero PCLD-780 que hace más sencillo la conexión de los cables a la placa.

A la placa de E/S se le han acoplado dos pequeños circuitos, uno de ellos esta basado en el chip l293b que junto con una fuente de alimentación de 9V se encarga de suministrar la suficiente potencia a los motores. Su esquema electrico se muestra en la siguiente página.

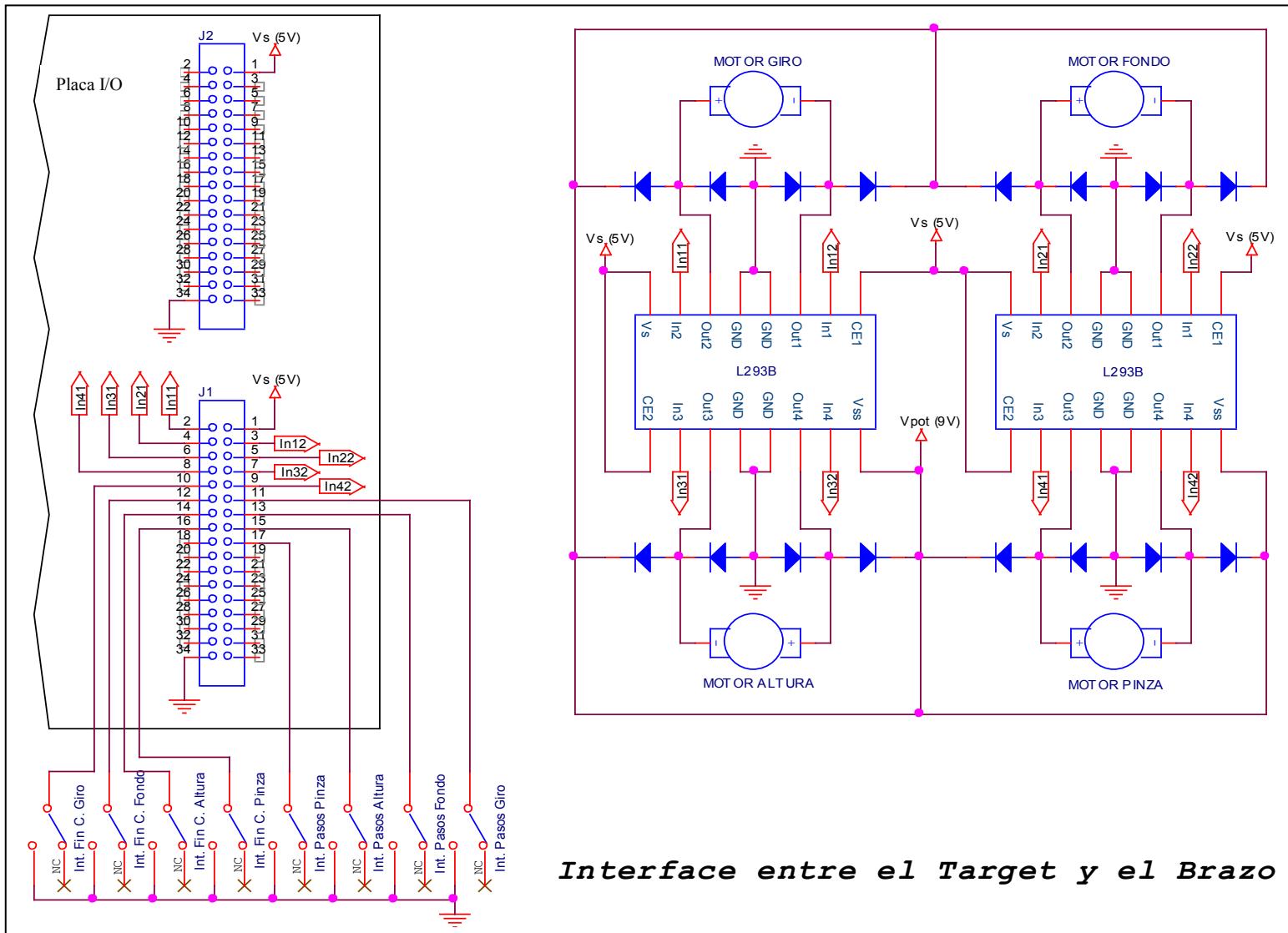


Figura 7.3: Esquema eléctrico de los motores y el circuito L393b.

En cuanto a los interruptores, estos se han conectado directamente a la placa de entrada/salida, a través de la placa de potencia y del bornero PCLD-780, de esta manera basta con leer en el puerto al que se conecten los interruptores para saber el estado de cada uno. Puesto que queremos que el sistema produzca una interrupción cada vez que uno de los interruptores cambie de estado, ha sido necesario desarrollar un segundo circuito que se encargue de testear continuamente los interruptores para ver cuando cambia alguno, y producir la interrupción. El circuito se ha construido sobre el controlador PIC-16F84A (ver Apéndice C).

El programa que ha sido cargado en el controlador se muestra en la página siguiente. Como se puede apreciar el controlador tiene 13 puertos de entrada/salida en dos direcciones de memoria, a 8 de los puertos se conectan los interruptores, otro se usa para producir la interrupción y por último otro puerto para recibir la señal de reconocimiento de la interrupción (ACK). El controlador se limita a testear continuamente los puertos de los interruptores, en caso de que alguno haya cambiado provoca una interrupción y permanece parado hasta que reciba la señal ACK. Aunque el controlador no es excesivamente rápido, es lo suficiente como para no perder ningún cambio en los interruptores. Además se supone que el target es bastante rápido por lo que el tiempo de tratamiento de una interrupción no será excesivo, y no bloqueará al generador de interrupciones mucho tiempo.

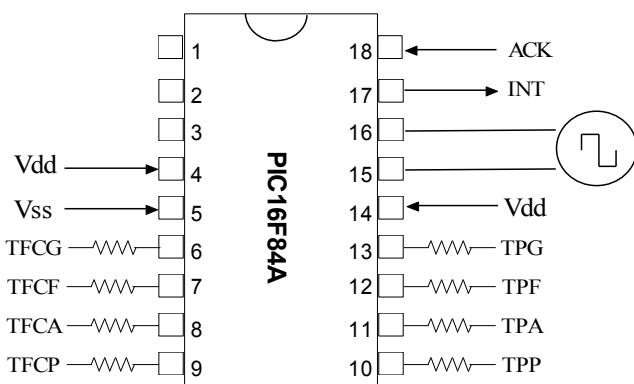


Figura 7.4: Conexiones del PIC16F84A.

```

#define byte port_a = 5
#define port_b = 6

// RB0 - RB3 por nivel
#define NIVEL 0x0F
// RB4 - RB7 por cambio
#define CAMBIO 0xF0

//byte=1
#define INTERRUPT 1

// bit 1 , byte=2
#define INT_ACK 1

void main() {

    int ultimo_estado,estado;

    setup_counters(RTCC_INTERNAL,RTCC_DIV_2);

    set_tris_b(0xff); //puerto B todo como entradas
    set_tris_a(0x02); //RA0 -> Salida RA1 -> Entrada

    port_a=0;

    ultimo_estado=port_b;

    while (1) {
        estado=port_b;
        if (ultimo_estado ^ estado) {

            port_a=INTERRUPT;

            while (!bit_test(port_a,INT_ACK)); //esperamos un ACK del PC

            port_a=0;
        }
        ultimo_estado=estado;
    }
}

```

En la siguiente página puede apreciarse el diagrama completo del hardware que controla el brazo: interruptores, placa de control, controlador de interrupciones y placa de entrada/salida.

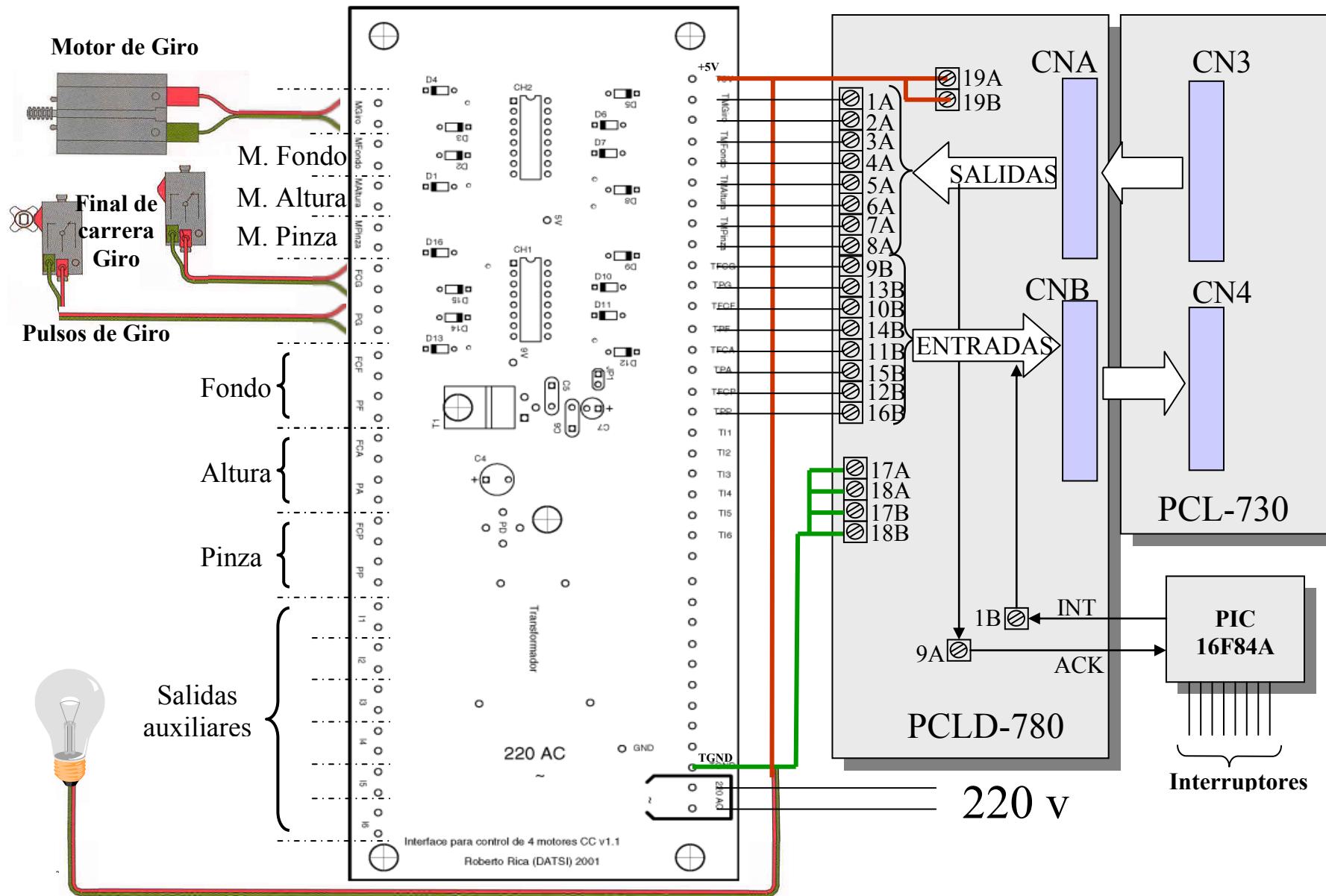


Figura 7.5: Diagrama global del hardware para el control del robot.

Como se puede apreciar en la figura 7.3 , cada motor es controlado por dos bits de uno puertos de la tarjeta de entrada/salida, que al tener 8 bits, hace posible el control de los 4 motores sin necesidad de usar más puertos.

Por otro lado, la entrada de datos exige otros ocho bits, al tener dos interruptores por grado de libertad (uno para el fin de carrera y otro que irá informando de los pasos que se van dando para poder saber donde está situada la pinza). Para este fin se ha utilizado el segundo puerto. Además se precisa otra entrada para generar la interrupción, por último hace falta una salida para generar la señal ACK. Con lo cual quedan aún disponibles 7 canales libres para salidas, y 7 canales libres para la entrada.

Como se puede apreciar en el esquema de conexión entre, lo que se ha llamado, el circuito de potencia (el basado en el l293b) y la tarjeta de entrada/salida (a la derecha en el diagrama anterior), el significado de los distintos pins de los dos puertos usados es el reflejado en la siguiente tabla, siendo base la dirección base de los registros de la tarjeta:

Tabla 7.1: Significado de los puertos del hardware.

I/O	Puerto	Bit	Descripción
OUT	base + 2	0 y 1	Control del motor de Giro
OUT	base + 2	2 y 3	Control del motor de Fondo
OUT	base + 2	4 y 5	Control del motor de Altura
OUT	base + 2	6 y 7	Control del motor de la Pinza
IN	base + 3	0	Interruptor Fin Carrera Giro
IN	base + 3	1	Interruptor Pasos Giro
IN	base + 3	2	Interruptor Fin Carrera Fondo
IN	base + 3	3	Interruptor Pasos Fondo
IN	base + 3	4	Interruptor Fin Carrera Altura
IN	base + 3	5	Interruptor Pasos Altura
IN	base + 3	6	Interruptor Fin Carrera Pinza
IN	base + 3	7	Interruptor Pasos Pinza
IN	base + 2	0	Señal INT
OUT	base + 3	0	Señal ACK

La lógica correspondiente a las entradas y salidas es la siguiente: En cuanto a las entradas (lectura de los interruptores del brazo por parte del Target):

Tabla 7.2: Significado de los valores del puerto de los interruptores.

Interruptor (Bit)	Valor	
	1	0
Fin Carrera Giro (0)	Pulsado → Fin carrera giro	No pulsado → posición > 0

Fin Carrera Fondo (1)	Pulsado → Fin carrera fondo	No pulsado → posición > 0
Fin Carrera Altura (2)	Pulsado → Fin carrera altura	No pulsado → posición > 0
Fin Carrera Pinza (3)	Pulsado → Fin carrera Pinza	No pulsado → posición > 0
Pasos Giro (4)	Pulsado	Suelto
Pasos Fondo (5)	Pulsado	Suelto
Pasos Altura (6)	Pulsado	Suelto
Pasos Pinza (7)	Pulsado	Suelto

Es decir, cuando un interruptor de fin de carrera está pulsado (valor = 1), se sabrá que el brazo está en la posición inicial en ese grado de libertad, lo que forzará que, como se verá en la parte del software, se debe de parar el motor que controla ese sentido y permitir su giro solo en el sentido que provoque posiciones mayores que cero. Ya que como se ha dicho, de lo contrario, se podría quemar el motor debido a que se le estaría forzando a girar en un sentido en el que no puede o se podría estropear cualquier otra pieza. Al ser de plástico su resistencia es bastante limitada y en el mejor de los casos, se desencajará alguna pieza de su sitio permitiendo el movimiento.

En cuanto a la salida (órdenes enviadas del Target al brazo), la lógica es la siguiente:

Tabla 7.3: Órdenes al puerto de los motores.

Interruptores (Pto:1, Pto_2)	Valores		
	0 1	0 0	1 0
Giro (0,1)	Izquierda	Paro	Derecha
Fondo (2,3)	Adelante	Paro	Atrás
Altura (4,5)	Arriba	Paro	Abajo
Pinza (6,7)	Abrir	Paro	Cerrar

Nota: los valores 1 1, no se usan, pero darían como resultado lo mismo que el 0 0, es decir, paro del motor.

Por último en cuanto al control de los motores se refiere, solo destacar que como se puede apreciar en la hoja de características del l293b, este se podría configurar usando un pin (y por lo tanto un bit) más para que el paro fuese libre o rápido. En este caso, se ha configurado fijo (ver esquema eléctrico) para que cuando se dé la orden de parar el motor, este pare lo más rápido posible y de esta forma alcanzar la mayor precisión posible. De todas formas, con este sistema, no habría existido ninguna diferencia ya que la inercia que generan los movimientos de brazo es mínima y la fuerza de rozamiento la vencería de forma instantánea.

Como se puede ver, se ha integrado en la misma placa una fuente de alimentación de 9V con el fin de dar la tensión y corriente necesaria a los motores.

Resulta importante destacar que los límites de alimentación soportados por la placa son de 1A de corriente y 36V de tensión, limitados por los chips que se encargan

de controlar los motores, es decir, los l293b que se pueden ver como CH1 y CH2. Aunque en el caso límite sería conveniente poner disipadores de calor debido a que en este caso, los chips tendrían que disipar el calor generado por el consumo de 36W, y por lo tanto se calentarían en exceso (en el caso probado de alimentación de 9V y un consumo por motor de aproximadamente 400mA, no se han mostrado indicios de aumento térmico en los chips, y por lo tanto no se ha montado ningún tipo de disipador de calor sobre estos).

En cuanto a la interface de la placa, según se ve la imagen de la serigrafía de la figura 7.5, a la izquierda se tienen las conexiones con el brazo robótico o regleta de periférico, con el siguiente significado, de arriba a abajo:

Tabla 7.4: Significado de las conexiones de la placa de interfaz con el brazo.

Señal	Descripción
Mgiro	Dos conexiones preparadas para conectar los dos bornes del motor que se encarga de hacer el giro del brazo.
Mfondo	Dos conexiones en las que se conectarán los bornes del motor encargado de hacer que el brazo se mueva de adelante a atrás.
Maltura	Dos conexiones correspondientes al motor encargado de llevar a cabo el movimiento ascendente o descendente de la pinza.
Mpinza	Dos conexiones para enchufar el motor encargado de la apertura y cierre de la pinza.
FCG	Dos conexiones para el interruptor de Fin de Carrera de Giro, es decir, el interruptor que indica cuando está el brazo lo más a la izquierda permitido.
PG	Dos conexiones reservadas para el interruptor de Paso de Giro, es decir, el interruptor que va cerrándose y abriéndose a medida que el brazo gira a izquierda o derecha.
FCF	Dos conexiones en las que se conectarán los bornes del interruptor de Fin de Carrera de Fondo que indica cuando está lo más atrás posible.
PF	Dos conexiones dedicadas al interruptor de Paso de Fondo.
FCA	Dos conexiones para el interruptor que indica el Fin de Carrera de Altura, que informa cuando está la pinza lo más alta posible.
PA	Dos conexiones a las que se tiene que conectar el interruptor de Paso de Altura.
FCP	Dos conexiones para el interruptor de Fin de Carrera de la Pinza, que sirve para conocer cuando está la pinza abierta del todo.
PP	Dos conexiones dedicadas al interruptor de Paso de Pinza.
I1	Dos conexiones para conectar un interruptor para uso del usuario.
I2	Dos conexiones para conectar un interruptor para uso del usuario.
I3	Dos conexiones para conectar un interruptor para uso del usuario.
I4	Dos conexiones para conectar un interruptor para uso del usuario.
I5	Dos conexiones para conectar un interruptor para uso del usuario.
I6	Dos conexiones para conectar un interruptor para uso del usuario.

En cuanto a la parte de conexión con el Target, se pueden ver en la serigrafía de la figura 7.5 a la derecha y de arriba abajo las siguientes conexiones pertenecientes a la regleta del Target:

Tabla 7.5: Significado de las conexiones del target.

Señal	Descripción
T5V	Una sola conexión en la que se conectará el cable procedente del Target que dará la referencia de 5V. Esta referencia les servirá a los I293b para diferenciar si las distintas entradas que tiene están a nivel alto o bajo.
TMGiro	Dos conexiones que servirán para hacer que el motor de giro del brazo se mueva en uno u otro sentido o simplemente se pare, según la codificación vista anteriormente. El más cercano a la señal T5V es el bit de menor peso.
TMFondo	Dos conexiones en las que se enchufarán los hilos procedentes del Target correspondientes al control del motor de fondo. Según la codificación vista anteriormente.
TMAltura	Dos conexiones en las que se conectan los dos hilos que transportan las señales necesarias para el control del motor de altura. Según la codificación vista anteriormente.
TMPinza	Dos conexiones a las que se unirán los hilos que utiliza el Target para el control del giro del motor que provoca la apertura o cierre de la pinza. Según la codificación vista anteriormente.
TFCG	Una conexión que está unida al interruptor de Fin de Carrera de Giro, y por lo tanto, a ella se conectará el hilo procedente del Target al que se tiene que pasar dicha información.
TPG	Una conexión en la que se tiene la información sobre el estado del interruptor de Paso de Giro.
TFCF	Una conexión a la que se conectará el hilo procedente del Target que necesita transportar la información del estado del interruptor de Fin de Carrera de Fondo.
TPF	Una conexión donde se unirá el hilo del Target que transportará el estado del interruptor de Pulso de Fondo.
TFCA	Una conexión a la que se conectará el hilo que necesita el estado del interruptor de Fin de Carrera de Altura.
TPA	Conexión a la que se conectará el hilo del Target que informará del estado del interruptor de Paso de Altura.
TFCP	Conexión que informa del estado del interruptor de Fin de Carrera de la Pinza.
TPP	Conexión en la que se unirá el hilo que informará del estado del interruptor de Paso de Pinza al Target.
TI1	Conexión que indica el estado del primer interruptor del usuario.
TI2	Conexión que indica el estado del segundo interruptor del usuario.
TI3	Conexión que indica el estado del tercer interruptor del usuario.
TI4	Conexión que indica el estado del cuarto interruptor del usuario.
TI5	Conexión que indica el estado del quinto interruptor del usuario.
TI6	Conexión que indica el estado del sexto interruptor del usuario.

TGND	Punto en el que se conectará la señal de masa procedente del Target.
220AC	Conexión de tensión de 220 V AC 50Hz

7.2.4 Entorno de desarrollo: Host.

El único componente hardware que queda por explicar es el Host. Como ya se ha explicado anteriormente, este sistema es un ordenador de propósito general que será utilizado para realizar el código fuente de control del brazo (y el del programa que se quiera ejecutar en el Target), compilarlo, crear la documentación, y todas las demás tareas necesarias para hacer el proyecto que no sea ejecutar el programa. Además cumplirá una función fundamental a la hora de depurar la aplicación ya que será en el Host donde se ejecute GDB. El Host utiliza como sistema operativo Linux (Red-Hat 7.1). El software del entorno de desarrollo así como su instalación se detalla en el apéndice B.

7.3 Diseño del software.

El software a diseñar debe controlar el brazo robótico, más concretamente debe controlar los dispositivos que componen el brazo robótico, es decir los interruptores que sirven para monitorizar los movimientos del brazo y los motores que sirven para controlar la velocidad. Como es lógico el software se desarrolla bajo el perfil Ravenscar, lo cuál tampoco supone un gran inconveniente aunque se esté acostumbrado a trabajar sobre entornos sin ninguna restricción. Ya que generalmente, este tipo de sistemas no son dinámicos y por tanto las condiciones estáticas que impone el perfil son perfectamente asumibles.

Como ya se vio en los capítulos 2 y 3, entre las condiciones más importantes que impone Ravenscar es que no se pueden crear tareas ni objetos protegidos dinámicamente ya que todos deben estar declarados a nivel de biblioteca, aparte de otros aspectos que se irán viendo más adelante.

La metodología seguida en el diseño lógico de la aplicación es HRT-HOOD, aunque no se ha llegado a realizar un análisis temporal calculando los peores tiempos de respuesta.

7.3.1 Gráfico de dispositivos externos.

Como se observa en la figura 7.7 los dispositivos externos se pueden enumerar en 8 interruptores y 4 motores. Los interruptores son de dos tipos:

- **Interruptores de Paso:** A medida que el brazo se mueve en un grado de libertad el interruptor de paso correspondiente a ese grado de libertad va abriendose y cerrándose periódicamente, cuando uno de estos interruptores

cambie de estado provocará una interrupción. Esta será la manera de monitorizar el movimiento, contando el numero de cambios de estado que se van produciendo.

- **Interruptores de Final de Carrera:** Cuando el brazo llegue al final de la carrera de algún grado de libertad se producirá una interrupción.

El sistema de control también debe suministrar las órdenes oportunas a los cuatro motores para que se muevan en el sentido adecuado y a la velocidad adecuada, o pararlos en función del algoritmo de control.

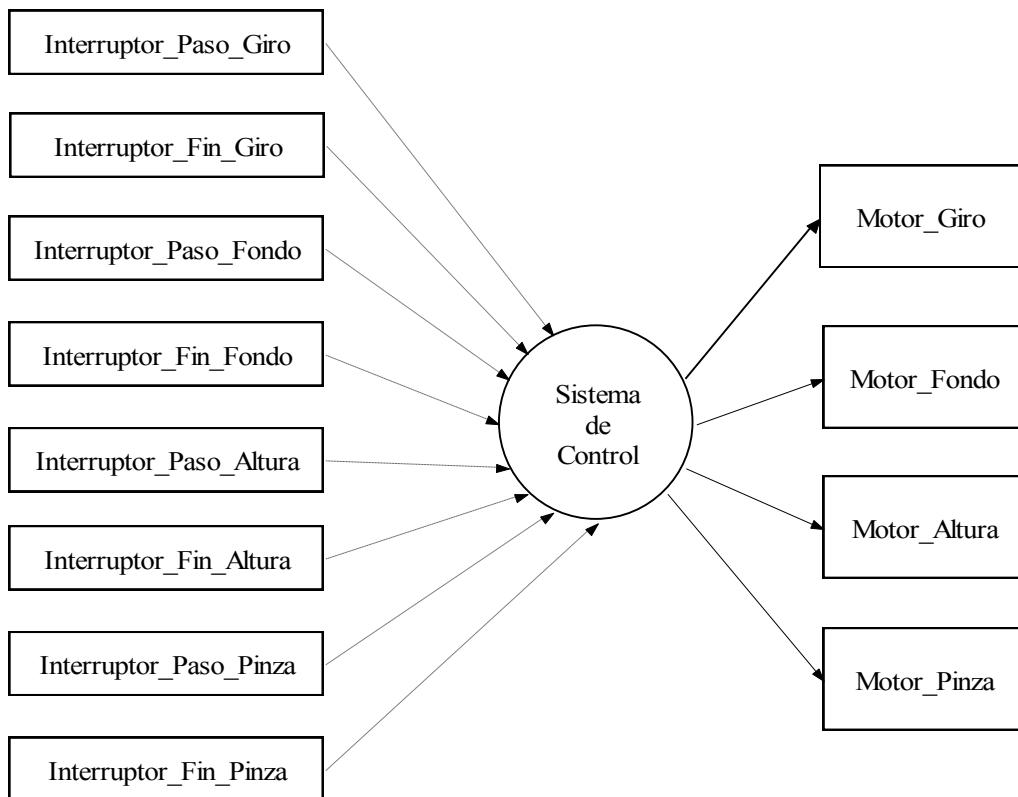


Figura 7.6: Gráfico de dispositivos externos.

- Flechas discontinuas: Los interruptores interrumpen al sistema.
- Flechas continuas: El sistema actúa sobre los motores, ya sea arrancándolos o parándolos.

7.3.2 Diseño de la arquitectura lógica.

El software para control del brazo robótico está diseñado a modo de librería que debe importar la aplicación que vaya a usar el brazo (ver figura 7.7). Es decir, lo que se suministra al usuario es un paquete que contiene una serie de procedimientos para

controlar el brazo robótico, de forma que haga completamente transparente los aspectos internos de control del brazo, como el control de la velocidad, el tratamiento de las interrupciones o el acceso a los puertos. Dado que casi toda la funcionalidad que provee el paquete aparece repetida cuatro veces, una vez por cada grado de libertad, se ha tomado la decisión de implementar paquetes genéricos que puedan ser instanciados con los valores adecuados para poder así reutilizar el código escrito. Esto va a provocar que paquetes instanciados tengan que invocar operaciones de otros paquetes que también están instanciados, para lo cual deben recibirlas como parámetros. De este modo aunque en los diagramas existan flechas de relación entre paquetes como el de la figura 7.9, estas relaciones realmente se producen a través de la operación que un paquete recibe como parámetro.

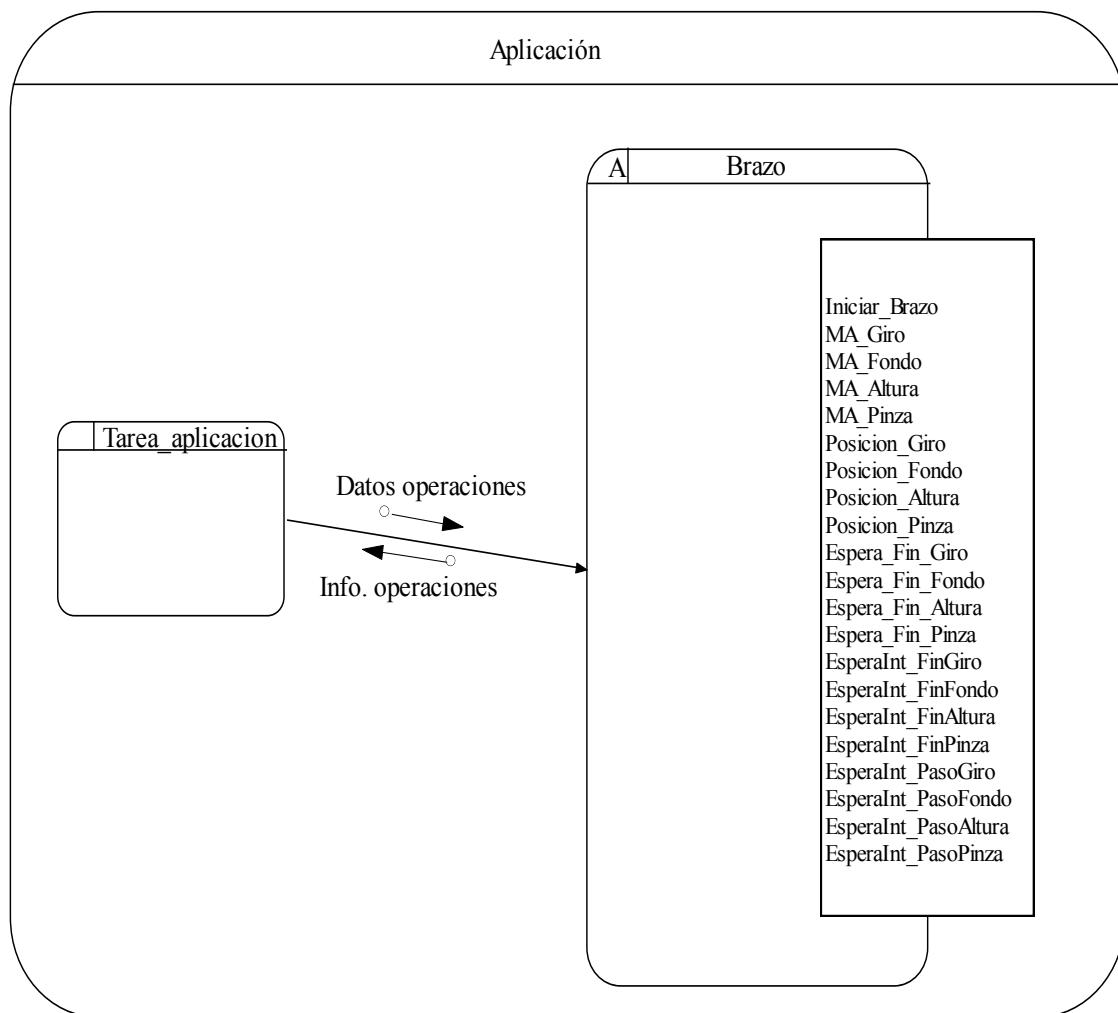


Figura 7.7: Diagrama de la aplicación.

7.3.2.1 Brazo.

OBJETO Brazo ES ACTIVO

DESCRIPCIÓN

El paquete brazo encapsula toda la funcionalidad del brazo. Provee las primitivas básicas para gestionar el brazo sin necesidad de conocer detalles técnicos del hardware.

INTERFAZ OFRECIDA

• TIPOS

-- Amplitud máxima del grado de libertad giro.

```
subtype Amplitud_Giro    is Integer range 1 .. PB.Max_Giro
```

-- Amplitud máxima del grado de libertad fondo.

```
subtype Amplitud_Fondo   is Integer range 1 .. PB.Max_Fondo
```

-- Amplitud máxima del grado de libertad altura.

```
subtype Amplitud_Altura is Integer range 1 .. PB.Max_Altura
```

-- Amplitud máxima del grado de libertad pinza.

```
subtype Amplitud_Pinza  is Integer range 1 .. PB.Max_Pinza
```

-- Rango de los valores de velocidad.

```
subtype T_Velocidad is Integer range 1 .. 99
```

• OPERACIONES

-- Procedimiento para iniciar el estado general del brazo.

```
procedure Iniciar_Brazo;
```

-- Procedimiento para solicitar un movimiento absoluto de giro.

```
procedure MA_Giro      (posicion : in Amplitud_Giro;
                        velocidad : in T_Velocidad);
```

-- Procedimiento para solicitar un movimiento absoluto de fondo.

```
procedure MA_Fondo    (posicion : in Amplitud_Fondo;
                        velocidad : in T_Velocidad);
```

-- Procedimiento para solicitar un movimiento absoluto de altura.

```
procedure MA_Altura   (posicion : in Amplitud_Altura;
                        velocidad : in T_Velocidad);
```

-- Procedimiento para solicitar un movimiento absoluto de pinza.

```
procedure MA_Pinza   (posicion : in Amplitud_Pinza;
                        velocidad : in T_Velocidad);
```

-- Función que devuelve la posición del brazo en el grado de libertad giro.

```
function Posicion_Giro return Amplitud_Giro;
```

```
-- Función que devuelve la posición del brazo en el grado de libertad fondo.  
function Pocision_Fondo return Amplitud_Fondo;  
  
-- Función que devuelve la posición del brazo en el grado de libertad altura.  
function Posicion_Altura return Amplitud_Altura;  
  
-- Función que devuelve la posición del brazo en el grado de libertad pinza.  
function Posicion_Pinza return Amplitud_Pinza;  
  
-- Procedimiento para esperar el final de un movimiento de giro.  
procedure Espera_Fin_Giro;  
  
-- Procedimiento para esperar el final de un movimiento de fondo.  
procedure Espera_Fin_Fondo;  
  
-- Procedimiento para esperar el final de un movimiento de altura.  
procedure Espera_Fin_Altura;  
  
-- Procedimiento para esperar el final de un movimiento de pinza.  
procedure Espera_Fin_Pinza;  
  
-- Procedimiento para esperar que el interruptor FinGiro se cierre.  
procedure EsperaInt_FinGiro;  
  
-- Procedimiento para esperar que el interruptor PasoGiro se cierre.  
procedure EsperaInt_PasoGiro;  
  
-- Procedimiento para esperar que el interruptor FinFondo se cierre.  
procedure EsperaInt_FinFondo;  
  
-- Procedimiento para esperar que el interruptor PasoFondo se cierre.  
procedure EsperaInt_PasoFondo;  
  
-- Procedimiento para esperar que el interruptor FinAltura se cierre.  
procedure EsperaInt_FinAltura;  
  
-- Procedimiento para esperar que el interruptor PasoAltura se cierre.  
procedure EsperaInt_PasoAltura;  
  
-- Procedimiento para esperar que el interruptor FinPinza se cierre.  
procedure EsperaInt_FinPinza;  
  
-- Procedimiento para esperar que el interruptor PasoPinza se cierre.  
procedure EsperaInt_PasoPinza;
```

INTERFAZ REQUERIDA

- **TIPOS**

Tipos provistos por el paquete *Parametros_Brazo*.

PARTE INTERNA

- **OBJETOS**

El paquete brazo crea cuatro instancias del paquete genérico *Motor*, una por cada grado de libertad.

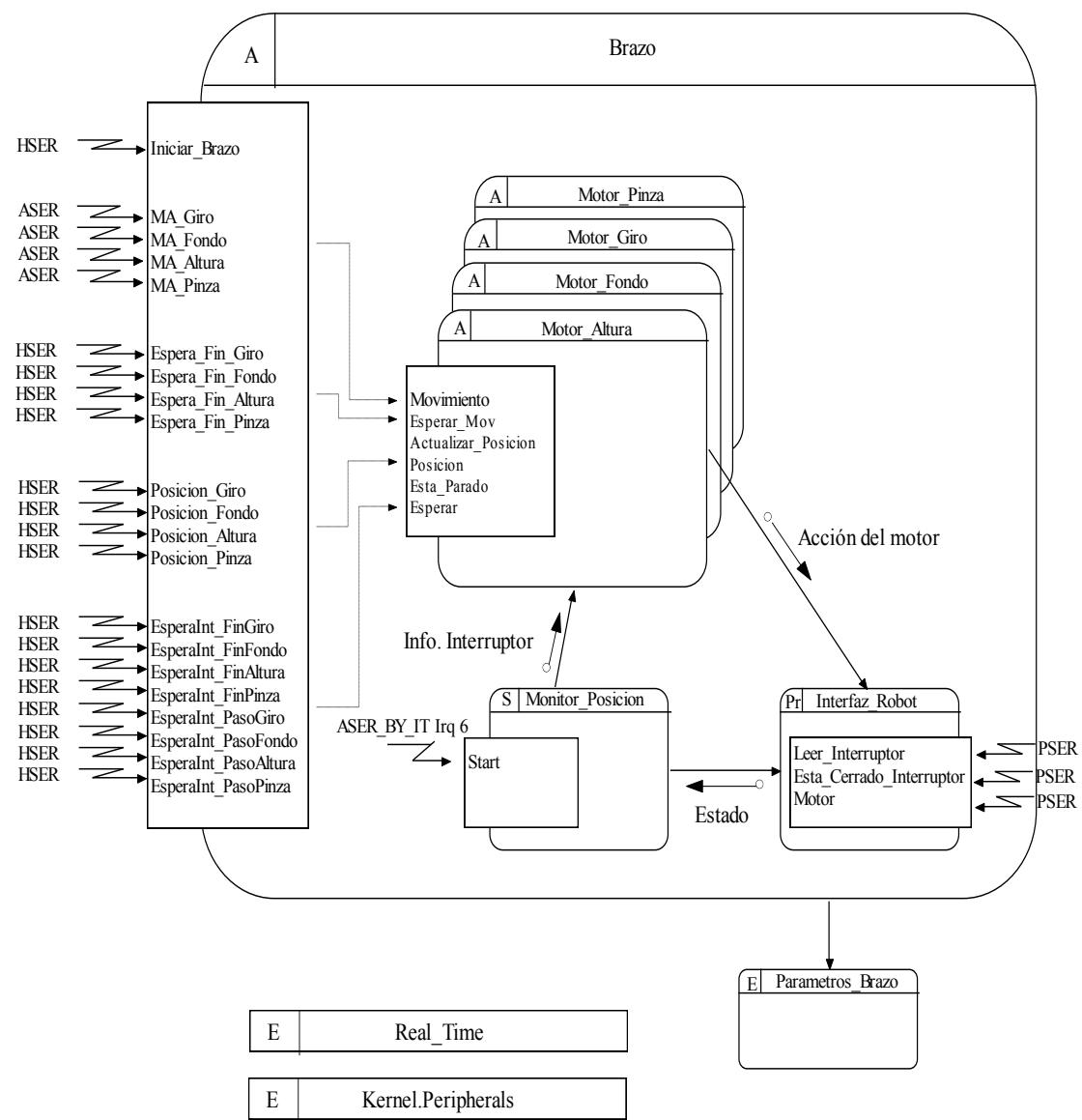


Figura 7.8: Brazo.

7.3.2.2 Monitor_Posicion.

OBJETO Monitor_Posicion ES ESPORADICO

DESCRIPCIÓN

El objeto Monitor_Posicion es el que se encarga de atender a las interrupciones (IRQ_6) que producen los interruptores y de informar al objeto motor adecuado de que uno de sus interruptores ha provocado la interrupción. Debe mantener y actualizar el estado de cada interruptor cada vez que se produce una interrupción para poder decidir que interruptor ha provocado la interrupción, y de esta forma poder saber en qué posición se encuentra en cada momento el brazo.

INTERFAZ OFRECIDA

Ninguna.

INTERFAZ REQUERIDA

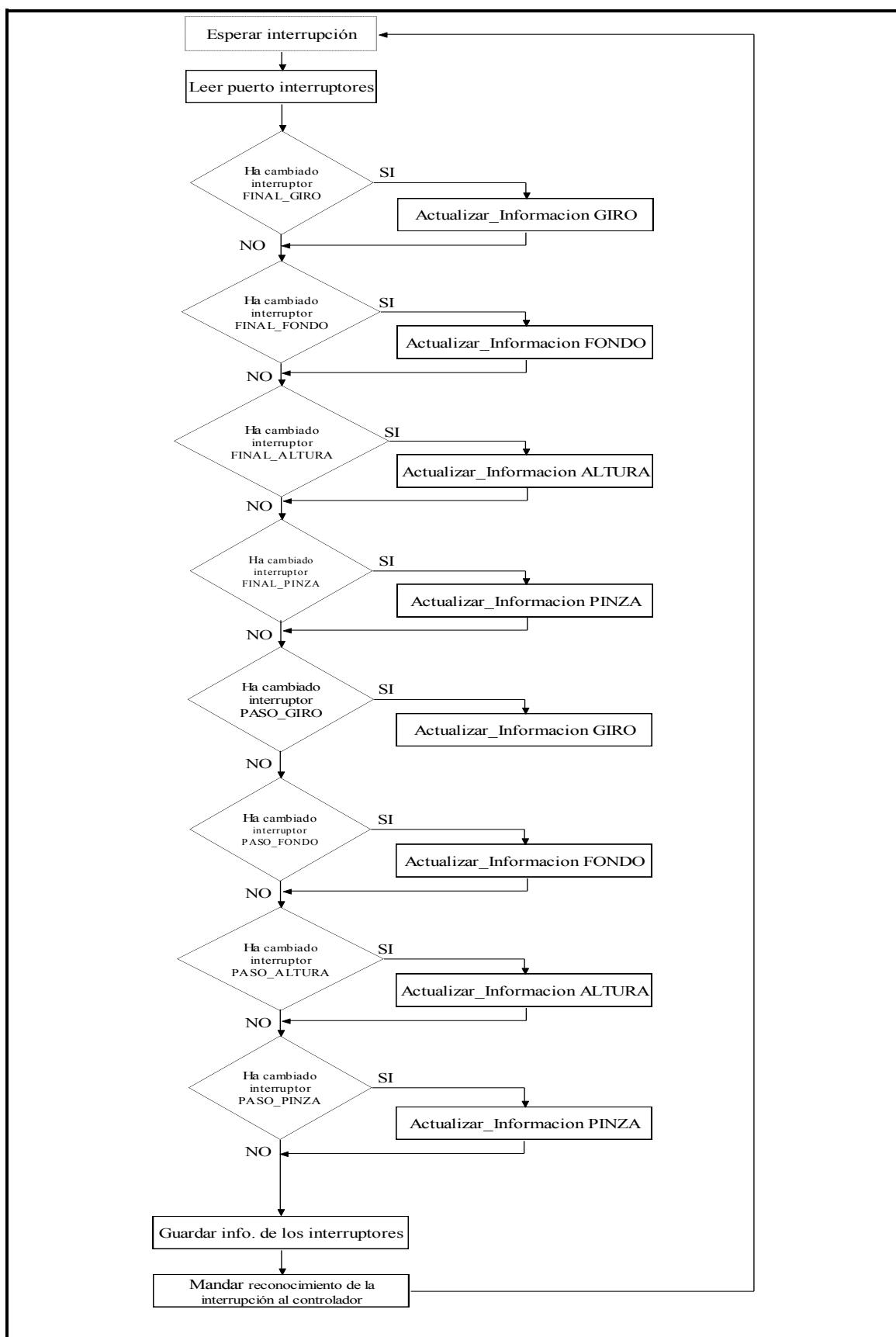
- TIPOS

Tipos provistos por el paquete *Parametros_Brazo*.

El objeto Monitor_Posición utiliza el procedimiento *Actualizar_Posicion*, de los objetos *Motor* declarados en el paquete Motores, para actualizar el estado de cada motor.

PARTE INTERNA

Interiormente esta compuesto por un objeto protegido que implementa el manejador de la interrupción y de una tarea esporádica que se activa cada vez que llega una interrupción (IRQ_6). Mantiene el estado de cada interruptor después de la última interrupción ocurrida. El comportamiento de la tarea esporádica es el siguiente.



7.3.2.3 Interfaz_Robot.

OBJETO Interfaz_Robot ES PROTEGIDO

DESCRIPCIÓN

El objeto Interfaz_Robot, ofrece los servicios que se pueden realizar sobre los puertos, es decir leer del puerto de los interruptores y escribir en el puerto donde se encuentran las señales de salida a los motores..

INTERFAZ OFRECIDA

- TIPOS

```
Type Tipo_Pulsador is (No_Pulsado, Pulsado);

Type Tipo_Motor_Giro is (Parado, Giro_Antihorario,
                         Giro_Horario);

Type Tipo_Motor_Fondo is (Parado, Atras, Adelante);

Type Tipo_Motor_Altura is (Parado, Arriba, Abajo);

Type Tipo_Motor_Pinzas is (Parado, Abrir, Cerrar);

Type Tipo_Entradas_Robot is
record
    Final_Carrera_Giro : Tipo_Pulsador;
    Final_Carrera_Fondo : Tipo_Pulsador;
    Final_Carrera_Altura : Tipo_Pulsador;
    Final_Carrera_Pinzas : Tipo_Pulsador;
    Pasos_Giro : Tipo_Pulsador;
    Pasos_Fondo : Tipo_Pulsador;
    Pasos_Altura : Tipo_Pulsador;
    Pasos_Pinzas : Tipo_Pulsador;
end record;

Type Tipo_Salidas_Robot is
record
    Motor_Giro: Tipo_Motor_Giro;
    Motor_Fondo: Tipo_Motor_Fondo;
    Motor_Altura: Tipo_Motor_Altura;
    Motor_Pinzas: Tipo_Motor_Pinzas;
end record;
```

- OPERACIONES

-- Función que devuelve el valor de los puertos

```
function Leer_Puerto_Interruptores return
                                         Tipo_Entradas_Robot
```

-- Procedimiento para mandar ordenes al motor

```
procedure Motor (grado      : Tipo_Grado;
                sentido   : Tipo_Sentido);
```

INTERFAZ REQUERIDA• **TIPOS**

Tipos provistos por el paquete *Parametros_Brazo*.

PARTE INTERNA

El acceso a los puertos se hace a través de objetos protegido, ya que si varias tareas intentan escribir en el mismo puerto a la vez se pueden producir problema de coherencia.

7.3.2.4 Parametros_Brazo.**OBJETO Parametros_Brazo ES PASIVO****DESCRIPCIÓN**

El objeto Parametros_Brazo contiene definiciones generales de tipos, dirección de los puerto, y las prioridades de cada una de las tareas y objetos protegido del sistema.

INTERFAZ OFRECIDA• **TIPOS**

-- Prioridades base asignadas a cada tarea y objeto protegido del sistema, posteriormente cada grado de libertad asigna un *offset* acada prioridad.

```
P_Cola_Peticiones      := System.Priority'First + 20;
P_Tarea_Posicion       := System.Priority'First + 20;
P_Control_Posicion    := Names.DISKETTE_IRQ_Priority;
P_Tarea_Velocidad      := System.Priority'First + 30;
P_Control_Velocidad   := Names.DISKETTE_IRQ_Priority;
P_Puerto_Lectura       := Names.DISKETTE_IRQ_Priority;
P_Puerto_Motor         := Names.DISKETTE_IRQ_Priority;
P_Estado                := Names.DISKETTE_IRQ_Priority;
P_Interrupcion         := Names.DISKETTE_IRQ_Priority;
P_Tarea_Monitor_Posicion := Names.DISKETTE_IRQ_Priority;
```

-- Este es el *Offset* que se suma a cada una de las prioridades de las tareas y los objetos compartidos de un determinado grado de libertad, para que todas las prioridades de las tareas sean distintas.

```

Offset_Prio_Giro    : constant := 0;
Offset_Prio_Fondo   : constant := 1;
Offset_Prio_Altura  : constant := 2;
Offset_Prio_Pinza   : constant := 3;

-- Amplitud máxima en cada grado de
Max_Giro    : constant := 250;

Max_Fondo   : constant := 163;

Max_Altura  : constant := 115;

Max_Pinza   : constant := 30;

-- Puertos disponibles en la tarjeta E/S
-- Puerto Isolated
p_isolated_IO_1 : constant Tipo_Puerto_IO := 16#200#;

p_isolated_IO_2 : constant Tipo_Puerto_IO := 16#201#;

-- Puerto TTL
p_ttl_IO_1      : constant Tipo_Puerto_IO := 16#202#;

p_ttl_IO_2      : constant Tipo_Puerto_IO := 16#203#;

-- Tamaño de la cola de peticiones de movimientos.
Tamano_Cola := 100;

-- Grados de libertad del robot.
type Tipo_Grado is (PINZA, ALTURA, FONDO, GIRO);

-- Distintos tipos de movimientos asociados al robot.
type Tipo_Sentido is (PARAR, HORARIO, ANTIHORARIO,
                      ADELANTE, ATRÁS, ARRIBA,
                      ABAJO, ABRIR, CERRAR);

-- Puertos usados por los interruptores y los motores.
Salidas_Digitales_Robot : Tipo_Puerto_IO := p_ttl_IO_1;

Entradas_Digitales_Robot :Tipo_Puerto_IO := p_ttl_IO_2;

-- Tipos asociados a los interruptores.
type Tipo_Clase Interruptor is (FINAL, PASO);

```

```
type Tipo_Interruptor is (FINGIRO, PASOGIRO, FINFONDO,
                           PASOFONDO, FINALTURA, PASOALTURA,
                           FINPINZA, PASOPINZA);
```

-- Interrupción utilizada por los interruptores.

-- IRQ_6

```
Int_Interruptores : constant :=
    Ada.Interrupts.Names.DISKETTE_IRQ;
```

7.3.2.5 Motor.

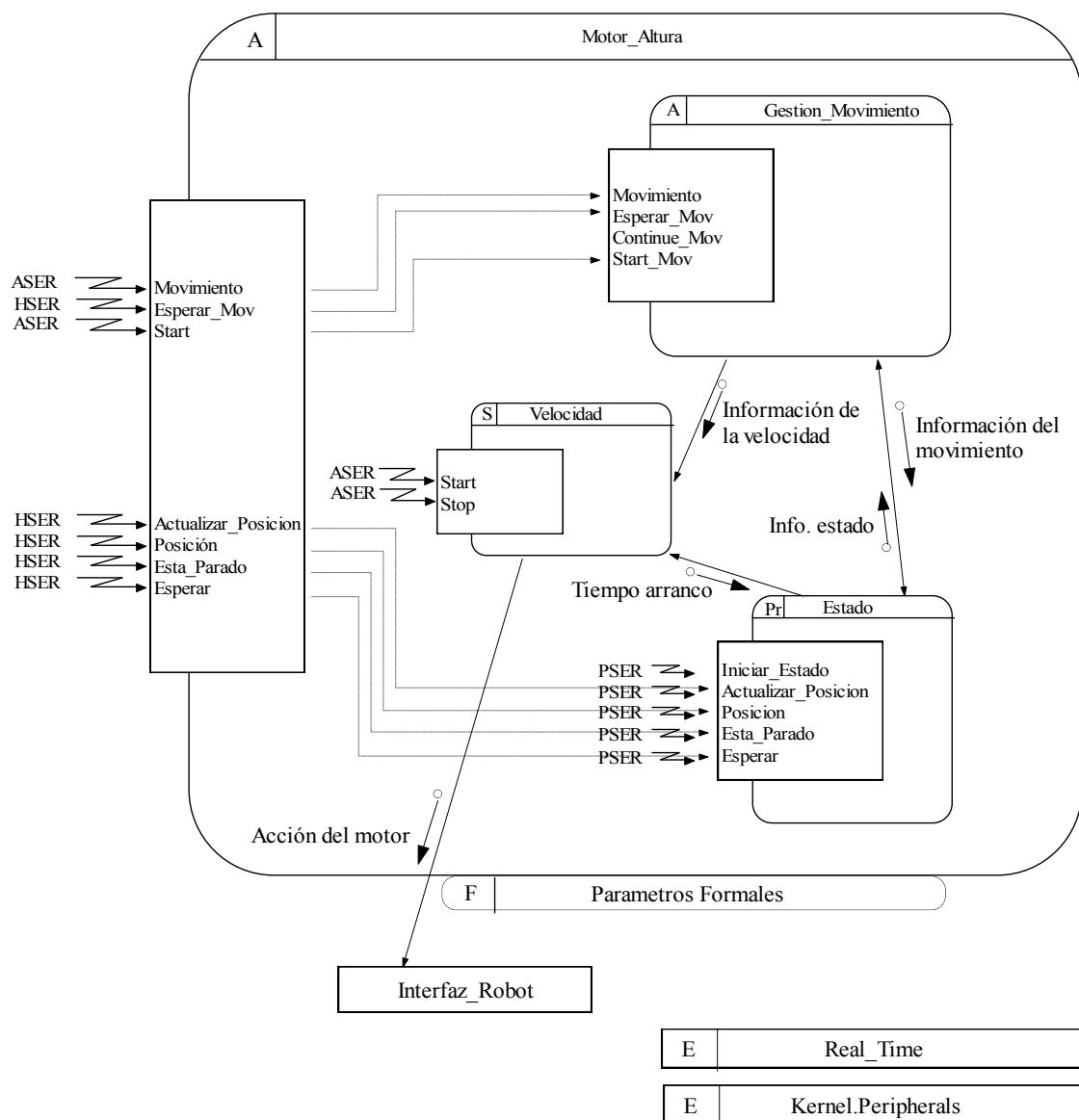


Figura 7.9: Motor_Altura.

OBJETO Motor ES ACTIVO**DESCRIPCIÓN**

El objeto motor encapsula toda la funcionalidad asociada a un motor, fundamentalmente los aspectos referentes al control de la velocidad del motor, al estado actual del motor, es decir su posición y estado de movimiento actual, y se encarga de gestionar las peticiones de movimiento que se le solicitan. Esta compuesto de tres paquetes instanciados *Estado*, *Velocidad* y *Gestion_Movimiento*.

INTERFAZ PAQUETE GENERICO

-- Grado de libertad sobre el que actúa.

BM_g_libertad : in Tipo_Grado

-- Amplitud máxima de movimiento.

BM_amplitud : in Integer

-- Interruptor de paso del grado de libertad.

BM_int_paso : in Tipo_Interruptor

-- Interruptor de final de carrera del grado de libertad.

BM_int_final : in Tipo_Interruptor

-- Movimiento en el sentido del interruptor de final de carrera.

BM_sentido1 : in Tipo_Sentido

-- Movimiento en el sentido de la amplitud máxima de movimiento.

BM_sentido2 : in Tipo_Sentido

-- Offset para el cálculo de la prioridad de los elementos del paquete.

BM_Offset_Priority : in System.Priority

INTERFAZ OFRECIDA**• OPERACIONES**

-- Indica al motor una petición del movimiento a realizar.

```
procedure Movimiento (posicion : in Integer;
                      velocidad : in Integer)
```

-- Operación bloqueante para esperar al final del movimiento.

```
procedure Esperar_Mov
```

-- Inicialización del motor.

```
procedure Start;
```

```
-- Procedimiento para actualizar la posición del motor.
procedure Actualizar_Posicion
    (int      : in Tipo_Clase_Interruptor;
     tiempo  : in Ada.Real_Time.Time)

-- Función que devuelve la posición actual del brazo.
function Posicion return Integer

-- Función que informa de si el motor esta parado en este momento.
function Esta_Parado return Boolean

-- Operación bloqueante para esperar que se cierre un interruptor.
procedure Esperar (interruptor : in Tipo_Clase_Interruptor)
```

INTERFAZ REQUERIDA

- **TIPOS**

Tipos provistos por el paquete *Parametros_Brazo*.

PARTE INTERNA

- **OBJETOS**

Interiormente se instancian un objeto *Estado*, un objeto *Velocidad* y un objeto *Gestion_Movimiento*.

7.3.2.6 Estado.

OBJETO Estado ES PROTEGIDO

DESCRIPCIÓN

Fundamentalmente mantiene todos los datos de un grado de libertad como posición. Es informado cada vez que ocurre una interrupción, y comprueba si la interrupción se ha producido por uno de los interruptores de su grado de libertad. Desbloquea a la aplicación cuando esta está esperando por el cierre de un interruptor, y a la tarea Movimiento, cuando se llega a la meta

PARAMETROS PAQUETE GENERICO

-- Amplitud máxima de movimiento.

BME_ampli_max : in Integer;

-- Interruptor de paso del grado de libertad.

BME_sent_ajuste : in Tipo_Sentido;

-- Interruptor de final de carrera del grado de libertad.

BME_int_final : in Tipo_Interruptor;

```
-- Operación para informar a la tarea que gestiona los movimientos que el
-- movimiento actual ha finalizado.
procedure BME_op_fin_movimiento;

-- Procedimiento para detener el motor.
procedure BME_op_stop_velocidad;

-- Offset para el cálculo de la prioridad de los elementos del paquete.
BME_Offset_Priority    : in System.Priority
```

INTERFAZ OFRECIDA

- **OPERACIONES**

-- Procedimiento para iniciar el estado de un movimiento.

```
procedure Iniciar_Estado (sentido      : in Tipo_Sentido;
                           meta        : in Integer;
                           g_libertad : in Tipo_Grado;
                           ret         : out Boolean);
```

-- Procedimiento para actualizar la posición en el estado.

```
procedure Actualizar_Posicion
  (int      : in Tipo_Clase_Interruptor;
   tiempo  : in Ada.Real_Time.Time);
```

-- Función que devuelve la posición actual del brazo en el grado de libertad.

```
function Posicion return Integer;
```

-- Función que devuelve True si el motor esta parado.

```
function Esta_Parado return Boolean;
```

-- Procedimiento bloqueante para esperar hasta que un interruptor determinado
-- se cierre.

```
procedure Esperar (interruptor : in Tipo_Clase_Interruptor);
```

INTERFAZ REQUERIDA

- **TIPOS**

Tipos provistos por el paquete *Parametros_Brazo*.

El objeto necesita el procedimiento *Stop* del paquete *Velocidad* para poder apagar el motor cuando se llegue al final de un movimiento.

PARTE INTERNA

El objeto Estado esta implementado como un objeto protegido.

También se declara un objeto *Synchronous_Task_Control* por cada interruptor del grado de libertad, que sirve para implementar la operación *Esperar*.

7.3.2.7 Velocidad.

OBJETO Velocidad ES ESPORÁDICO

DESCRIPCIÓN

El objeto velocidad es el encargado de controlar la velocidad de un motor en concreto, esta implementado como un paquete genérico. Básicamente provee dos operaciones una para arrancar un motor en una determinada dirección y a una velocidad determinada y otra para pararlo.

En cuanto a la técnica utilizada para controlar la velocidad del motor para el que se configure el hilo, se ha optado por la utilización de la modulación del ancho de pulso o PWM. Esta técnica consiste en el control de la potencia aplicada al motor mediante una modulación de pulsos, es decir, el motor es apagado y encendido con unas frecuencias altas. De esta forma dependiendo de la relación entre el tiempo que se mantiene encendido y el tiempo que se mantiene apagado durante un intervalo de tiempo más o menos largo, la potencia eléctrica aplicada al motor será mayor o menor, y por lo tanto, la velocidad media a la que gira también será más o menos rápida.

INTERFAZ PAQUETE GENERICO

-- Recibe el grado de libertad al que pertenece como parámetro
BMV_g_libertad : in Tipo_Grado

-- Offset para el cálculo de la prioridad de los elementos del paquete.
BMV_Offset_Priority : in System.Priority

INTERFAZ OFRECIDA

• OPERACIONES

-- Arranca el motor

```
procedure Start (p_velocidad : in Integer;  
                 p_sentido    : in Tipo_Sentido);
```

-- Parar el motor

```
procedure Stop;
```

INTERFAZ REQUERIDA

• TIPOS

Tipos provistos por el paquete *Parametros_Brazo*.

El objeto *Velocidad* necesita el procedimiento *Motor* del paquete *Interfaz_Robot*.

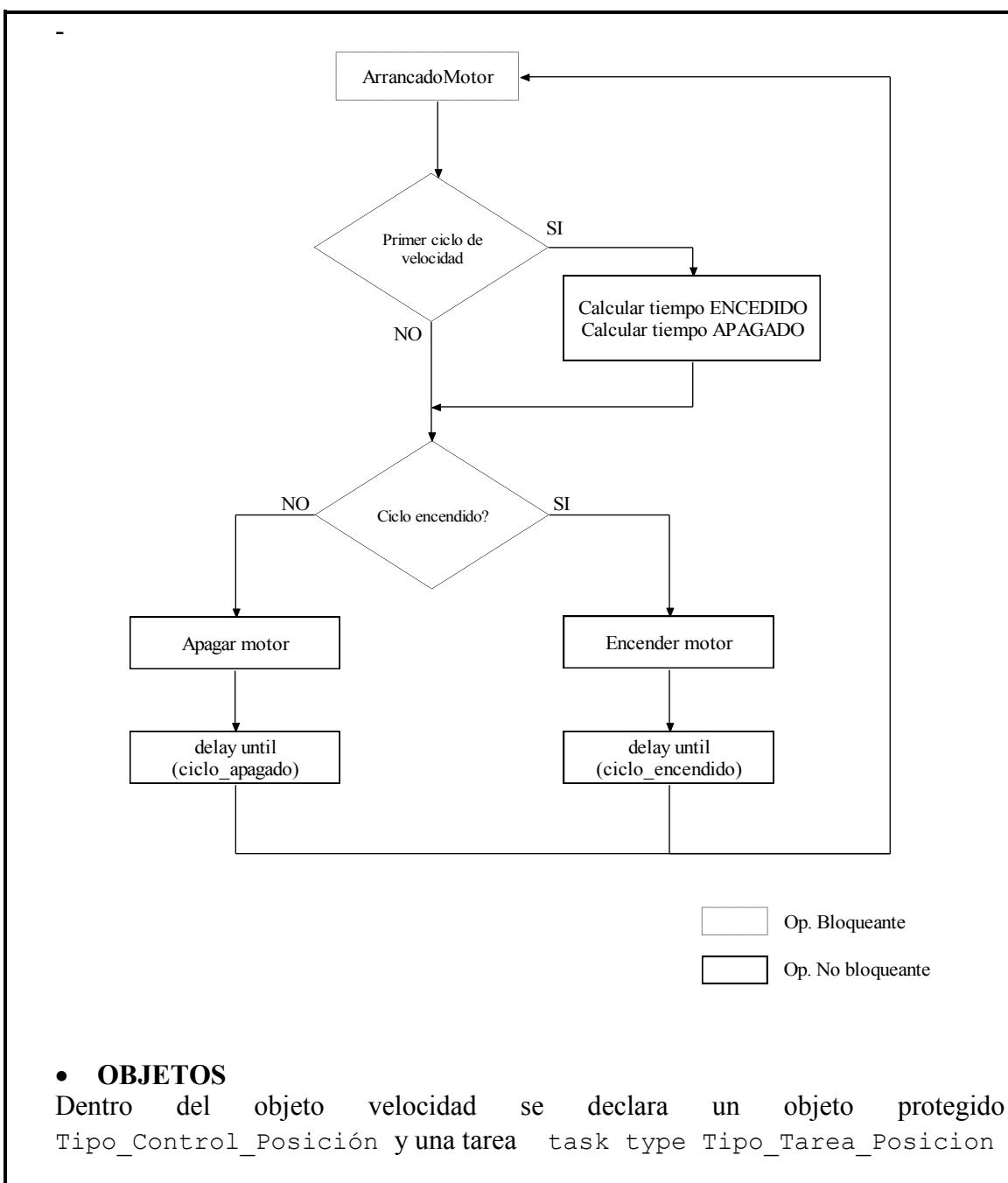
PARTES INTERNAS

• TIPOS

-- Objeto protegido que sincroniza a la tarea que controla la posición.
protected type Tipo_Control_Posicion

-- Tarea que gestiona la posición del brazo.

```
task type Tipo_Tarea_Posicion
```



7.3.2.8 Gestion_Movimiento.

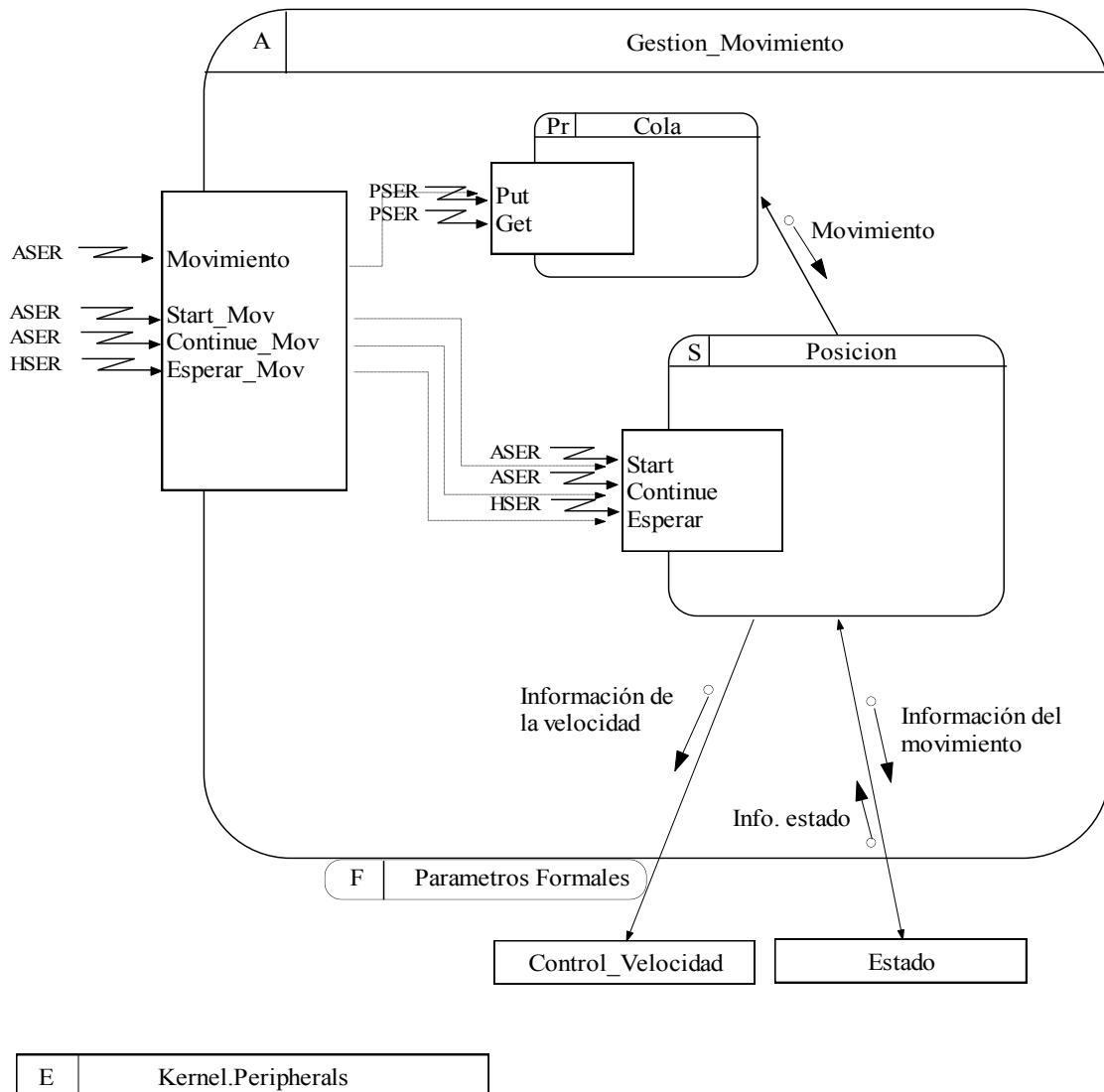


Figura 7.10: Gestion_Movimiento.

OBJETO Gestion_Movimiento ES ACTIVO

DESCRIPCIÓN

El objeto Gestion_Movimiento se encarga de gestionar las peticiones de movimientos para un determinado grado de libertad. Básicamente su función consiste en almacenar en la cola de peticiones las peticiones que van llegando, para posteriormente procesarlas y enviar las ordenes oportunas a la tarea que controla la velocidad y el estado. Hasta que el movimiento no acabe no se pasa a procesar otra petición de movimiento.

INTERFAZ PAQUETE GENERICO

```
-- Grado de libertad.
BMG_g_libertad : in Tipo_Grado;

-- Procedimiento para iniciar el estado de un movimiento.
procedure BMG_op_ini_estado
    (sentido : in Tipo_Sentido;
     periodo : in Ada.Real_Time.Time_Span;
     meta    : in Integer);

-- Función que devuelve la posición del brazo.
function BMG_op_posicion return Integer;

-- Función que devuelve True en caso de que el motor este parado.
function BMG_op_esta_parado return Boolean;

-- Procedimiento para arrancar el motor con unos valores determinados.
procedure BMG_op_start_motor (p_velocidad : Integer;
                               p_sentido   : Tipo_Sentido);

-- Offset para el cálculo de la prioridad de los elementos del paquete.
BMG_Offset_Priority : in System.Priority
```

INTERFAZ OFRECIDA

- **OPERACIONES**

```
-- Procedimiento para solicitar un movimiento del brazo.
procedure Movimiento (posicion : in Integer;
                      velocidad : in Integer);

-- Procedimiento bloqueante para esperar que acabe un movimiento.
procedure Esperar_Mov;

-- Procedimiento para iniciar la gestión de movimientos.
procedure Start_Mov;

-- Procedimiento para informar de que el actual movimiento ha finalizado y ya se
-- puede procesar el siguiente movimiento.
procedure Continue_Mov;
```

INTERFAZ REQUERIDA

- **TIPOS**

Tipos provistos por el paquete *Parametros_Brazo*.

PARTE INTERNA

- **TIPOS**

-- Objeto protegido que implementa la cola de peticiones

```
protected type Tipo_Cola_Peticiones
```

- **OBJETOS**

Está compuesto internamente de dos hijos creados de paquetes genéricos que son un paquete *Cola* y un paquete *Posición*, que se instancian de los paquetes genéricos con ese mismo nombre.

7.3.2.9 Cola.

OBJETO Cola ES PROTEGIDO

DESCRIPCIÓN

El objeto cola es un objeto pasivo protegido, como su propio nombre indica es una cola donde se almacenan peticiones de un determinado grado de libertad. Esta implementado como un paquete genérico ya que hay una cola por cada grado de libertad,

INTERFAZ PAQUETE GENERICO

-- Offset para el cálculo de la prioridad de los elementos del paquete.
 BMGC_Offset_Priority : in System.Priority

INTERFAZ OFRECIDA

- **OPERACIONES**

-- La operación Put devuelve el primer movimiento de la cola.
 procedure Put (elemento : in Tipo_Movimiento);

-- La operación Get es bloqueante, en caso de que la cola esta vacia bloqueará a la
 -- tarea que la invoque.

```
function Get return Tipo_Movimiento;
```

INTERFAZ REQUERIDA

- **TIPOS**

-- Necesita el tipo Movimiento del paquete Parametros_Brazo

```
type Tipo_Movimiento (Velo : Integer := 0;
                      Pabs : Integer := 0) is record
    velocidad      : Integer := Velo;
    posicion_abs   : Integer := Pabs;
end record;
```

PARTE INTERNA**• TIPOS**

-- Objeto protegido que implementa la cola de peticiones
protected type Tipo_Cola_Peticiones

• OBJETOS

Se declara un objeto Cola_Peticiones.

7.3.2.10 Posicion.**OBJETO Posicion ES ESPORADICO****DESCRIPCIÓN**

El objeto Posicion se encarga básicamente, de recoger las Órdenes de movimiento de la cola, calcular los movimientos a realizar, iniciar el nuevo estado, y arrancar la tarea velocidad. Su implementación es en forma de paquete genérico por lo tanto las operaciones que ejecuta de otros objetos las recibe como parámetro.

INTERFAZ PAQUETE GENERICO

-- Grado de libertad sobre el que actua el paquete.

BMGP_g_libertad : in Tipo_Grado;

-- Recibe el procedimiento para iniciar el estado.

```
procedure BMGP_op_ini_estado
    (sentido : in Tipo_Sentido;
     periodo : in Ada.Real_Time.Time_Span;
     meta    : in Integer);
```

-- Recibe la función para preguntar a *Estado* si el motor esta parado.

function BMGP_op_esta_parado return Boolean;

-- Recibe el procedimiento para arrancar el motor.

```
procedure BMGP_op_start_motor(p_velocidad : in Integer;
                               p_sentido   : in Tipo_Sentido);
```

-- Recibe la función para sacar de la cola el siguiente movimiento.

function BMGP_op_get return Tipo_Movimiento;

-- Offset para el cálculo de la prioridad de los elementos del paquete.

BMGP_Offset_Priority : in System.Priority

INTERFAZ OFRECIDA**• OPERACIONES**

-- La operación *start* debe ejecutarse para inicializar la tarea que gestiona la -- posición.

```
procedure Start;  
  
-- La operación continue informa a la tarea que gestiona la posición que ya  
-- finalizado un movimiento.  
procedure Continue;  
  
-- La operación esperar es empleada por la aplicación para esperar que acabe un  
-- movimiento por lo tanto es bloqueante y se encarga de desbloquearla la  
-- Tarea_Posición cuando acabe el movimiento.  
procedure Esperar;
```

INTERFAZ REQUERIDA

- **TIPOS**

El Objeto *Posición* interactúa tanto con *Velocidad* como con *Estado*, las operaciones que usa de estos las recibe a través de los parámetros del paquete genérico.

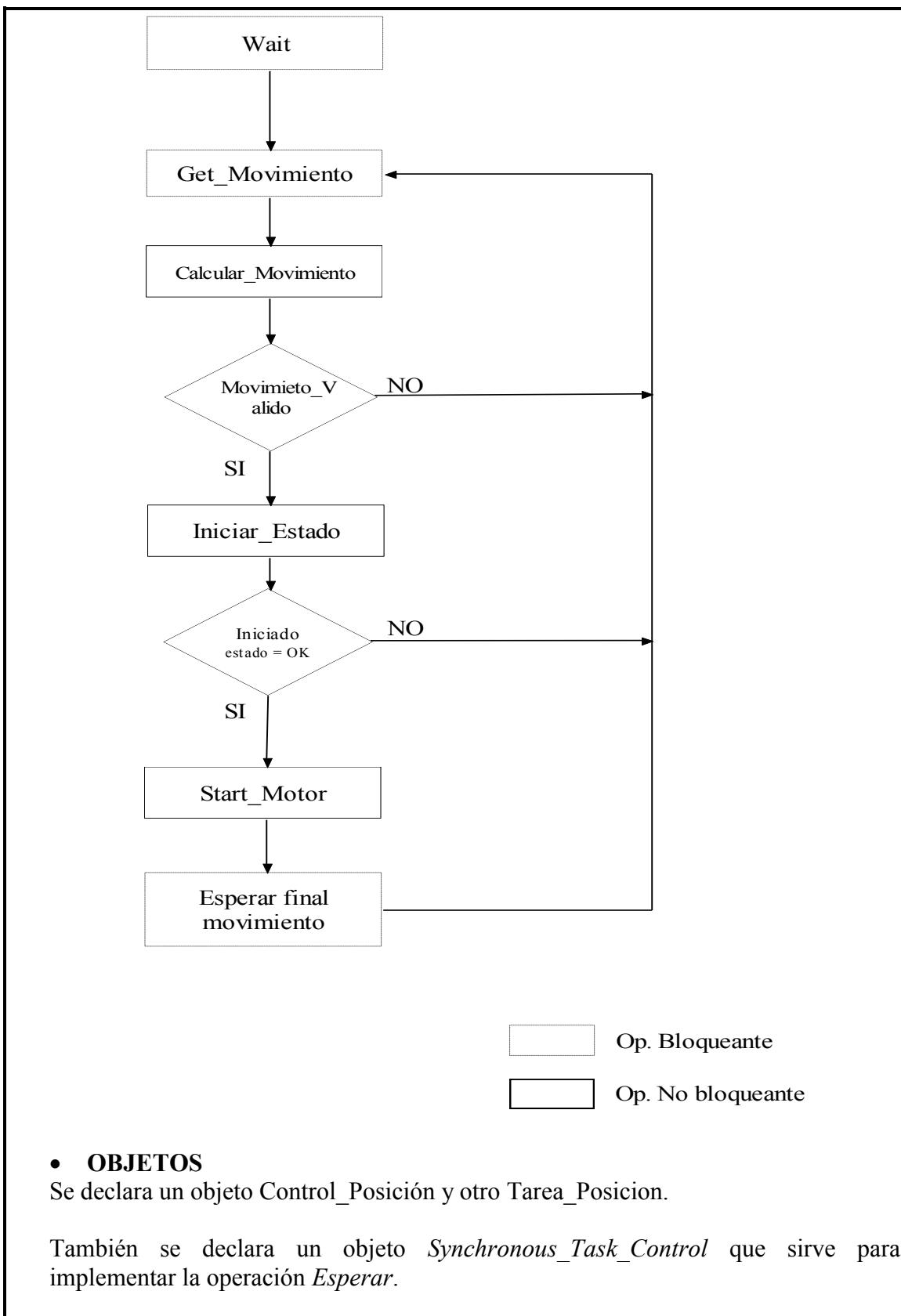
Del paquete Ada.Synchronous_Task_Control usa:
Suspension_Object

PARTE INTERNA

- **TIPOS**

-- Objeto protegido que sincroniza a la tarea que controla la posición.
protected type Tipo_Control_Posicion

-- Tarea que gestiona la posición del brazo.
task type Tipo_Tarea_Posición



7.3.3 Diagrama de procesos.

El diagrama de procesos representado por la figura 7.11 muestra las tareas y objetos que formarán el sistema final y las interacciones entre ellos. Para que el diagrama quede más claro algunos objetos protegidos y tareas se han representado en forma de cascada, estos son las entidades que se repiten por cada grado de libertad. Dentro de líneas punteadas se encuentran los objetos esporádicos compuestos por la tarea y el objeto protegido que controla a la tarea. Un caso especial es la tarea *Monitor_Posición* la cual se encarga de comprobar que interruptor ha provocado la interrupción y actualizar el estado del grado de libertad afectado, esta tarea es despertada por una interrupción externa, aunque la funcionalidad de la tarea podría haberse incluido directamente en el manejador.

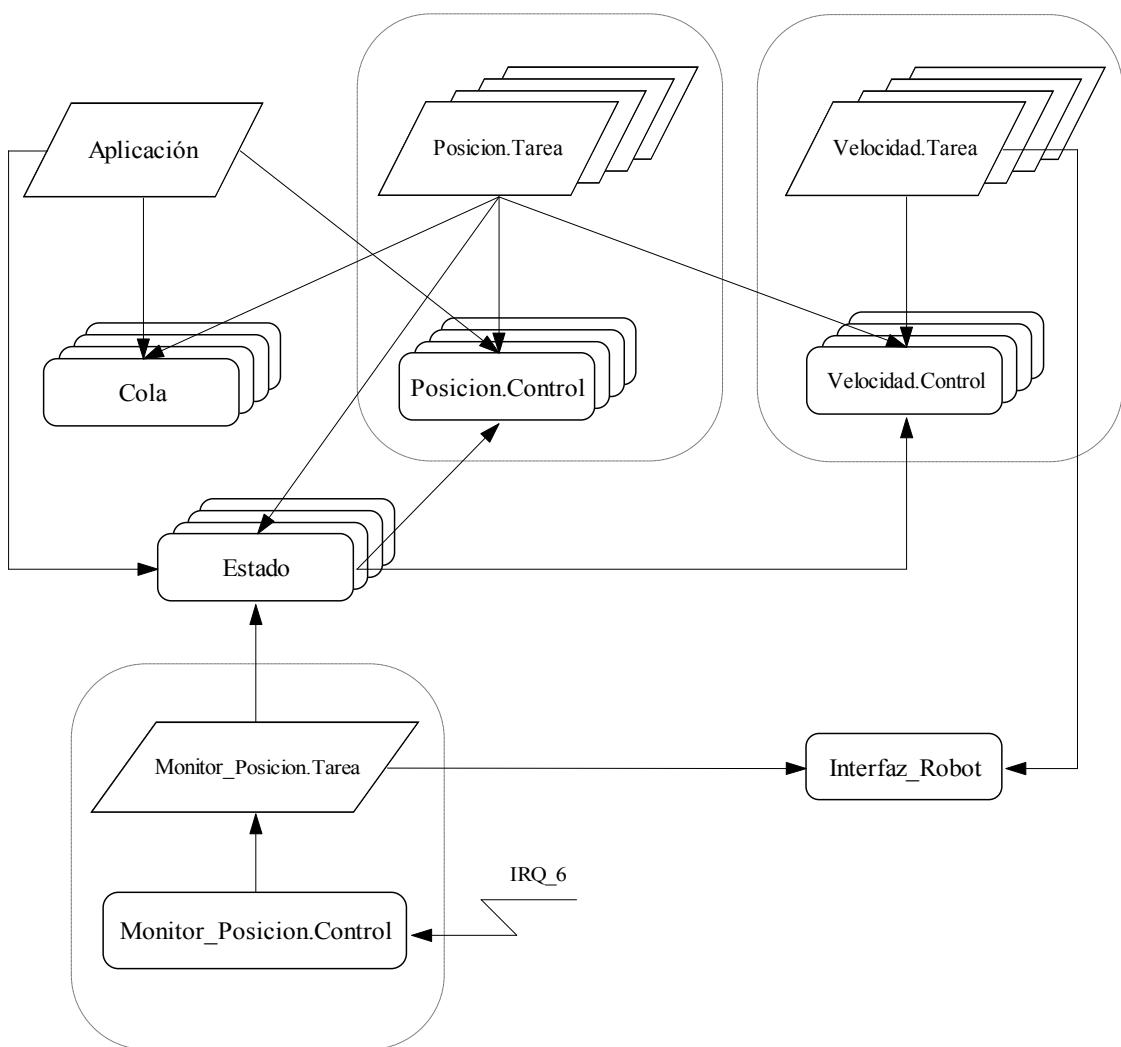


Figura 7.11: Diagrama de procesos.

7.3.4 Diagrama de módulos.

La figura 7.12 muestra el diagrama de módulos de la librería y la aplicación. El paquete *Parametros_Brazo* es importado por casi todos los paquetes de la librería aunque no se refleje en el diagrama por motivos de claridad del mismo. Un aspecto a resaltar en el diagrama es el de los paquetes hijos, en este caso los paquetes hijos realmente son instanciaciones de un paquete genérico que se realizan en el cuerpo de otro paquete. Tal es el caso de los hijos del paquete *Brazo* que son *Motor_Altura*, *Motor_Giro*, *Motor_Altura* y *Motor_Pinza* (estos tres no se representan en el diagrama), que son instanciaciones del paquete genérico *Motor*. Si tomamos como ejemplo *Motor_Altura* vemos que también tiene tres hijos, *Estado*, *Gestión_Movimiento* y *Velocidad*, que son respectivamente instanciaciones de sus respectivos paquetes genéricos. A su vez *Gestión_Movimiento* también tiene otros dos hijos que son *Cola* y *Posición* que vuelven a ser instancias del paquete genérico correspondiente. Como se puede ver los paquetes hijos no son realmente ficheros físicos sino instanciaciones de un paquete genérico.

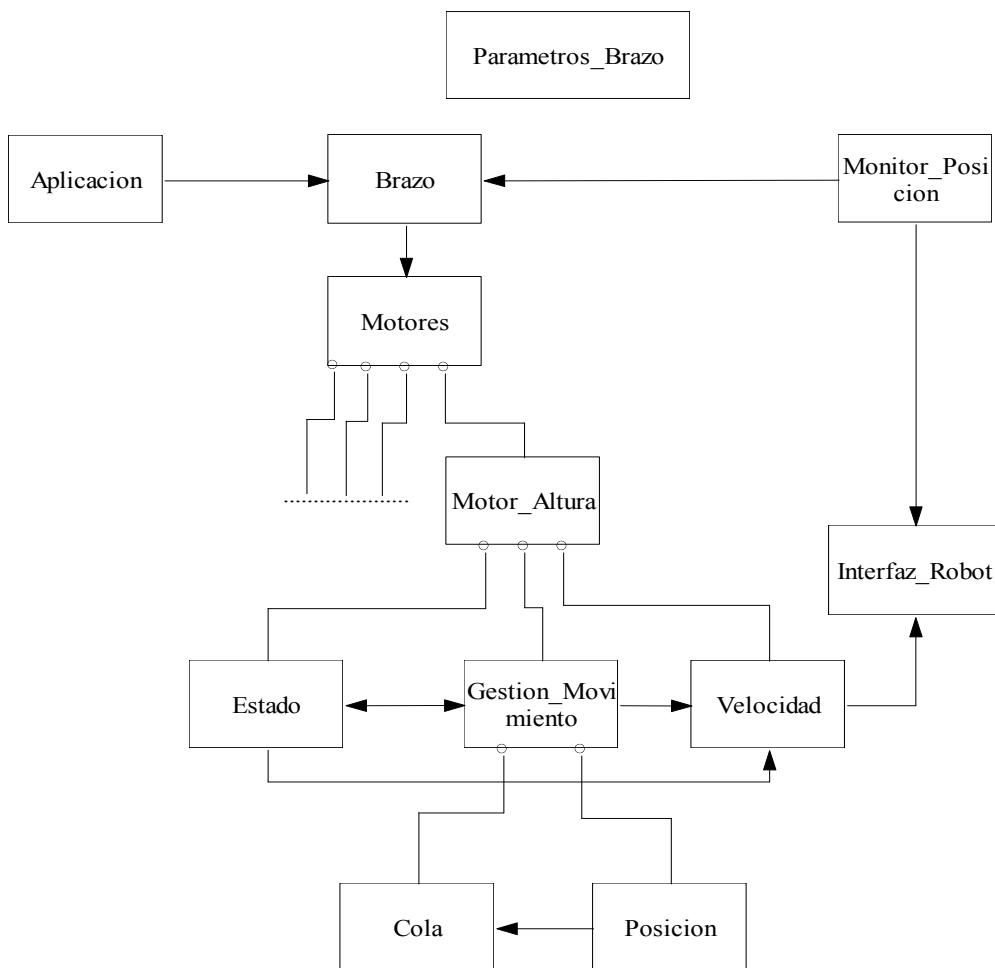


Figura 7.12: Diagrama de módulos.

7.4 Conclusiones.

El diseño de este sistema ha permitido verificar la funcionalidad del entorno de desarrollo cruzado basado en GNAT/ORK-i386 para el desarrollo de sistemas empotrados.

El sistema se ha probado tanto en la versión que usa los PIC, como la versión que ya emplea el TSC del Pentium, en ambos casos el sistema se ha comportado correctamente, aunque no se ha hecho un estudio exhaustivo por supuesto el calculo de los tiempos siempre será más exacto en el caso de la versión que emplea el TSC, la exactitud del reloj mejora mucho, esto se ha podido comprobar al realizar calculos de los tiempos de computo de las diferentes rutinas que componen el programa, así como los tiempos entre interrupciones. Por otra parte el diseño a través de paquetes genéricos permite que el sistema sea muy escalable en el caso de que se añadieran más grados de libertad al brazo, siendo necesario retocar muy poco código en este supuesto.

La realización de una aplicación real ha permitido encontrar algunos fallos del compilador GNAT, más concretamente de la implementación de algunos aspectos del *runtime* que han sido comunicados a través del correspondiente *bug report*. En concreto se encontró un fallo en el comportamiento del pragma Atomic, cuando se aplica a tipos *record*, este fallo parece que ha sido solucionado en versiones de posteriores GNAT como la 3.14 y siguientes.

Un fallo bastante importante y que parece a día de hoy que no ha sido solucionado, tiene que ver con la implementación de los objetos protegidos en el *restricted runtime*, que es el se utiliza cuando se aplica pragma Ravenscar.

Capítulo 8

Conclusiones y posibles mejoras.

8.1 Conclusiones.

El objetivo fundamental de este proyecto consistía en portar el sistema operativo ORK y todo el entorno de desarrollo cruzado GNAT/ORK con las facilidades para depuración a la plataforma i386, y se puede decir que el objetivo se ha cumplido. Aunque el sistema se encuentra en una versión beta, actualmente. Ya puede usarse para desarrollar sistemas reales como se ha demostrado en la aplicación realizada y descrita en el capítulo 7.

El desarrollo de este proyecto ha permitido familiarizarse con el mundo de los entornos de desarrollo cruzados basados en núcleos de tiempo real como RTEMS y MaRTe OS, así como con los aspectos relativos a la programación de bajo nivel y la programación sobre dispositivos físicos.

Uno de los mayores logros del proyecto sea el poder hacer accesible el sistema ORK a un amplio segmento de profesionales y estudiantes que no están en disposición de adquirir materiales caros como es el caso de los sistemas basados en el ERC-32. De esta manera puedan practicar y ensayar con el desarrollo de aplicaciones para sistemas de tiempo real críticos bajo el perfil de Ravenscar en una arquitectura más accesible.

Algo que es incluso muy ventajoso para la propia ESA, pues permite dar una mayor difusión a sus trabajos en el campo de la ingeniería del software, y contar en el futuro con un más amplio espectro de profesionales que conozcan sus métodos de análisis y diseño en el campo que nos ocupa, el software embarcado, como son actualmente HOORA y HRT-HOOD, con las limitaciones impuestas por el perfil Ravenscar, y que las puedan poner en práctica sobre una arquitectura muy accesible.

Actualmente los núcleos construidos a medida del perfil Ravenscar son muy pocos, por lo que las expectativas de futuro del sistema ORK parecen bastante buenas. Sus ventajas sobre otros sistemas operativos de tiempo real son evidentes ya que la ESA apostará por los sistemas que implementen directamente el perfil Ravenscar, por lo que

muchos profesionales y empresas del sector espacial ya se están interesando el sistema ORK, siendo una vez más ORK-i386 una buena primera toma de contacto con ORK.

A corto plazo tanto el entorno de desarrollo cruzado GNAT/ORK-i386 como el brazo robótico que se ha construido será utilizado en las prácticas de la asignatura de Sistemas de Tiempo Real, que se imparte en la facultad.

Por último se ha desarrollado una aplicación real sobre el sistema usando para su desarrollo el perfil Ravenscar, el cual ha demostrado tener la suficiente expresividad como para desarrollar sistemas embarcado a pesar de las restricciones que impone. Dado que ORK esta basado en GNAT, la manera de programar la aplicación no dista mucho de cómo se desarrollaría una aplicación convencional sobre un sistema operativo de propósito general, con la única excepción de poder acceder directamente al hardware del sistema, y concretamente la gestión de las interrupciones directamente, lo que permite la programación de manejadores de dispositivos de manera bastante sencilla.

8.2 Mejoras a ORK-i386.

Un proyecto de este tipo no se puede considerar como cerrado ya que la arquitectura Intel está en continua evolución y el entorno hardware sobre el que puede actuar el sistema es de lo más variado desde tarjetas de expansión a periféricos hechos a medida de un determinado problema.

Como ya se ha mencionado anteriormente el sistema se puede considerar que ya es operativo, pero a pesar de ello todavía se pueden realizar unas cuantas modificación al núcleo ORK-i386 que le darían un aspecto bastante competitivo con respecto a los sistemas que actualmente se encuentran en el mercado. Entre las principales modificaciones que se podrían realizar se mencionan las siguientes:

- Salvar el contexto de la unidad de coma flotante (FPU) solo en los casos necesarios: Actualmente la FPU se salva y restaura en todos los cambios de contexto, si solo se salvara en los casos estrictamente necesarios los cambios de contexto podrían ser mucho más rápidos ya que no todas las tareas van a usar la unidad de coma flotante. Una fuente de inspiración para realizar esto podría ser el sistema RTEMS, el cual en la arquitectura i386 marca las tareas como usuarias o no usuarias de la FPU y en función de los cambios de contexto que se vayan produciendo se salva y restaura la FPU o no. Otra posibilidad sería detectar en tiempo de ejecución que tareas usan la FPU y cuales no.
- Actualmente se está estudiando la posibilidad de colocar todo el contexto necesario para el tratamiento de interrupciones (RTI y manejadores) en posiciones de memoria no cacheables. Para el cálculo de los peores tiempos de computo, es necesario tener en cuenta, el caso de que cada vez que se produce una interrupción, el código que trata la interrupción puede haber alterado el contenido de la cache provocando expulsiones. Por lo que ahí que tener en

cuenta el tiempo necesario en llenar de nuevo la memoria cache cuando se vuelve de la interrupción. Si se pudiera asegurar que el tratamiento de una interrupción no ha producido ninguna expulsión, se mejoraría la planificabilidad del sistema mejorando el calculo del tiempo de respuesta de las tareas, ya que cuando se devuelve el control a la tarea interrumpida esta no necesita llenar de nuevo la cache, ya que el tratamiento de la interrupción no ha desalojado ningún bloque de la cache.

- Usar el APIC del Pentium II como temporizador: esto mejoraría drásticamente el tratamiento de los *delays*. Actualmente utilizando los temporizadores estandar que lleva un PC, los cuales tienen un ancho de 16 bits, el máximo intervalo de tiempo de espera que se puede programar es de 54.9 ms, lo que hace que para esperas mayores sea necesario interrumpir periódicamente, hasta que se cumpla el plazo. Con el nuevo hardware del pentium II, que tiene un ancho de 64 bits, permitiría programar esperas mucho mayores.
- Protección de las pilas: Uno de los aspecto que se contempla el ORK original era el de la protección de las pilas, es decir en caso de desbordamiento se activa la excepción predefinida Storage_Error. Aunque en la actual implementación de ORK-i386 esto no ocurre, podría implementarse la protección de las pilas creando segmentos de datos específicos para cada pila de cada tarea. Para esto sería necesario cargar los nuevos segmentos en la tabla de descriptores globales (GDT), y cada vez que se produzca un cambio de contexto o una interrupción será necesario cargar en el registro “ss” la dirección del segmento que apunta al espacio de memoria asignado a una pila concreta. Dado que un descriptor de segmento contiene la base y el límite de este, en caso de violar los límites, saltaría una excepción.
- Depuración remota a través de *Ethernet*: Actualmente la depuración cruzada se realiza a través de línea serie. Pero puede haber usuarios del sistema que soliciten mayores prestaciones de velocidad, para esto es posible adaptar la depuración a través de tecnología *Ethernet*. Una buena fuente de inspiración podría ser el sistema RTEMS, que dispone de esta posibilidad.

Apéndice A

Creación de un disquete de arranque con GRUB

Un *bootloader* es el primer programa software que corre cuando se arranca el computador, es el responsable de cargar y transferir el control al sistema operativo. Se ha elegido GRUB por su flexibilidad y facilidad de manejo para arrancar sistemas en el entorno ix86. Se van a explicar dos formas distintas y sencillas de construir el disquete de arranque con GRUB.

La primera forma consiste en descargarse los fuentes de la pagina web <http://www.gnu.org/grub>. Una vez tenemos el fichero llamado “**grub-0.5.96.tar.gz**” lo descomprimimos mediante la llamada “**tar zxvf grub-0.5.96.gz**”. Posteriormente entramos en el directorio donde se ha descomprimido el archivo y se encuentran los fuentes de GRUB y ejecutamos las siguientes ordenes:

1. **\$./configure**
2. **\$ make**
3. Creamos un sistema de ficheros en el disquete. Por ejemplo:

```
$ mke2fs /dev/fd0
```

4. Se monta el disquete en el directorio “**/mnt**”
5. Se crea el directorio “**/mnt/boot/grub**” y se copia en el los ficheros “**./grub-0.5.86/stage1/stage1**” y “**./grub-0.5.86/stage2/stage2**”.
6. Editamos el fichero “**menu.lst**” y se copia al directorio anterior también. El fichero debe contener al menos las siguientes líneas.

```
title= Hello World Test
kernel= (fd0)/hello.exe
```

7. Ejecutamos las siguientes ordenes:

```
$ grub --batch <<EOT
root (fd0)
setup (fd0)
quit
EOT
```

8. Por ultimo copiamos el fichero ejecutable al disquete. Según el ejemplo actual el fichero ejecutable deberá llamarse “**hello.exe**”.

Otra forma de construir un el disquete de arranque con el GRUB un poco más complicada, se muestra a continuación. La siguiente información ha sido recogida de la distribución del sistema RTEMS 4.5.0, concretamente del fichero situado en “**\$/rtems-4.5.0/c/src/lib/libbsp/i386/pc386/howto**”.

Lo primero que tenemos que hacer es bajar la herramienta mediante ftp en la dirección: <ftp://ftp.uruk.org/public/grub/>.

Una vez se tiene el archivo llamado “**grub-0.4.tar.gz**”, se deberá crear un nuevo directorio temporal (que se puede borrar una vez acabado el proceso) en el que se descomprimirá dicho archivo con la orden “**gunzip -c grub-0.4.tar.gz | tar xvof -**” (recordar que previamente se deberá estar situado en el directorio temporal creado, y si el fichero .tar.gz se tiene en otro directorio, será suficiente con poner el camino completo donde lo puede encontrar la herramienta gunzip).

Una vez llevado a cabo esto, se deberá cambiar al directorio **grub-0.4/bin_std** donde se tendrán dos ficheros llamados “**stage1**” y “**stage2**”.

En este momento, lo que se necesita son dos discos de alta densidad disponibles para ser formateados. Uno de ellos solo será usado de forma temporal para crear el otro que será con el que realmente arranquemos nuestro equipo más tarde y el que contendrá el programa que creemos con RTEMS. Al temporal se le llamará disco “RAW GRUB” y al definitivo, disco “GRUB FS”. Este último debe estar formateado con alguno de los sistemas de archivos soportados por GRUB, que son: DOS FAT, BSD FFS y ext2fs de Linux. En el proyecto, puesto que se ha usando Linux en el Host para el desarrollo y para crear este disco, se usará el sistema ext2fs de Linux, por lo tanto, para formatearlo se introduce el disco en la unidad fd0 (o a:), y se teclea una de estas dos ordenes:

```
mke2fs /dev/fd0
ó
mkfs.ext2 /dev/fd0
```

Lo siguiente es crear el disco temporal “RAW GRUB” con el que se creará el disco “GRUB FS”, para esto, y estando en el directorio `grub-0.4/bin_std`, se debe introducir el disco “RAW GRUB” en fd0 (o a:) y ejecutar los siguiente:

```
dd if=stage1 of=/dev/fd0 bs=512 count=1  
dd if=stage2 of=/dev/fd0 bs=512 seek=1
```

Una vez acabado este punto, se deberán copiar los ficheros `stage1` y `stage2` al disco “GRUB FS”, con Linux consiste en introducir el disco en fd0 (o a:), montarlo, con la orden mount (por ejemplo, “`mount /dev/fd0 /mnt/floppy`”), copiar los archivos con la orden cp (“`cp ./stage1 /mnt/floppy`” y “`cp ./stage2 /mnt/floppy`”) y por último desmontar el disco con umount (“`umount /mnt/floppy`”). Es muy importante desmontar el disco antes de dar el siguiente paso de arrancar la máquina, ya que si no se hace así, por las características de sistema de ficheros de Linux, la copia podría no haberse realizado o haberlo hecho a medias, con los consiguientes problemas que podría generar.

El siguiente paso es arrancar con el disco “RAW GRUB”. Una vez arrancado, se tendrá una interface de línea de comandos. Se cambia el disco, introduciendo el disco “GRUB FS” y se da la siguiente orden:

```
$ install=(fd0)/stage1 (fd0) (fd0)/stage2 0x8000 (fd0)/grubmenu
```

Este comando hará el disco “GRUB FS” de arranque y no se volverá a necesitar el disco “RAW GRUB”. También se podrá borrar el archivo `stage1` del disco “GRUB FS” para tener más espacio para la aplicación.

Como penúltimo paso, se tiene que copiar el archivo .exe al disco (se monta el disco con mount, se copia el archivo con cp y se desmonta el disco). Si se quiere o resultase necesario, se puede comprimir este archivo antes de copiarlo con gzip, puesto que GRUB carga este tipo de archivos de forma transparente.

Y por último se deberá crear un archivo llamado “`grubmenu`” con tantas entradas como se quiera del estilo de la siguiente:

```
title= Hello World Test  
kernel= (fd0)/hello.exe (si estuviese comprimido es:  
"kernel= (fd0)/hello.exe.gz")
```

De esta forma, si se tiene más de una entrada de este tipo en el archivo, al arrancar se presentará un menú con todos los title que permite seleccionar el archivo

a ejecutar. En caso de que el archivo `grubmenu` solo tenga una entrada de este tipo, se ejecutará sin preguntar nada.

Por fin, ya solo queda arrancar con el disco “GRUB FS” y seleccionar el programa que se quiere ejecutar en caso de que el `grubmenu` tenga varias opciones.

Apéndice B

Instalación y uso del entorno ORK-i386

B.1 Instalación y estructura de directorios.

B.1.1 Obtener ORK-i386.

ORK se distribuye vía ftp. La dirección es la siguiente <http://www.operavenscar.org>. El sistema de compilación cruzado y sus herramientas relacionadas pueden ser encontradas pinchando en el enlace “*software*”. Los fuentes para construir el entorno ORK-i386 pueden ser obtenidos en la misma localización. Las distribuciones ORK-i386 incluidas son:

- **openravenscar-beta-0.1-pc-linux-gnu-bin.tgz**: fichero tar comprimido con gzip que contiene la distribución binaria para GNU/Linux. La actual distribución ha sido compilada sobre Red-Hat 6.2 y Red-Hat 7.1 usando la librería glibc2. Para evitar problemas con diferentes versiones de la libc, todos los ficheros binarios han sido enlazados estaticamente.
- **openravenscar-beta-0.1-src.tgz**: fichero tar comprimido con gzip que contiene los fuentes del sistema así como los procedimientos para construir ORK.

El sistema soporta PC-compatibles con sistema operativo GNU/Linux como plataforma de desarrollo.

B.1.2 Instalación de ORK-i386.

B.1.2.1 Instalación de los binarios de ORK-i386.

El árbol de directorios de openravenscar-beta-0.1 ha sido compilado para residir en el directorio `/usr/local/openravenscar-i386`. Después de obtener el fichero

con la distribución binaria se debe descomprimir en el directorio anteriormente mencionado.

Los pasos para descomprimir el fichero serían los siguientes:

```
$ cd /usr/local
$ tar -zxvf openravenscar-beta-0.1-pc-linux-gnu-bin.tgz
```

Después de instalar el sistema de compilación cruzado, se debe añadir al path la siguiente ruta `/usr/local/openravenscar-i386/bin`.

B.1.2.2 Instalación de los fuentes de ORK-i386.

En este caso asumimos que los fuentes (`openravenscar-beta-0.1-src.tgz`) están instalados y descomprimidos mediante tar y gzip en el directorio `/usr/local/openravenscar/src`, aunque pueden estar en cualquier otra localización.

Los pasos para descomprimir el fichero serían los siguientes:

```
$ cd /usr/local
$ tar -zxvf openravenscar-beta-0.1-src.tgz
```

B.1.3 Compilación del kernel.

Los procedimientos para compilar el sistema de compilación cruzado se encuentran en el directorio `/usr/local/openravenscar/src`.

Para reconstruir todo el sistema de compilación cruzado ORK, tienes que hacer lo siguiente:

1. Editar el fichero `user.cfg` en caso que deseas cambiar la localización de ORK. Originalmente el entorno se instalará en el directorio `/usr/local/openravenscar-i386`.
2. Ejecutar el comando.

```
./build_ada i386
```

Como este procedimiento toma mucho tiempo, es posible seleccionar que solo se compile la `adalib`, en todo caso el sistema debe estar previamente compilado para poder hacer esto. Si el sistema de compilación ya ha sido previamente compilado los subdirectorios `build-i386-tools` y `src` ya deben existir en `/usr/local/openravenscar/src`. El procedimiento ha ejecutar es el siguiente.

```
./build_adalib i386
```

B.2 Desarrollo de software en el entorno ORK-i386.

Para desarrollar programas basados en ORK, deberías llevar a cabo las siguientes actividades:

1. Escribir el código para la aplicación.
2. Compilar y enlazar el programa.
3. Crear un disquete de arranque para ejecutar el programa.
4. Depurar el programa en la plataforma de ejecución.

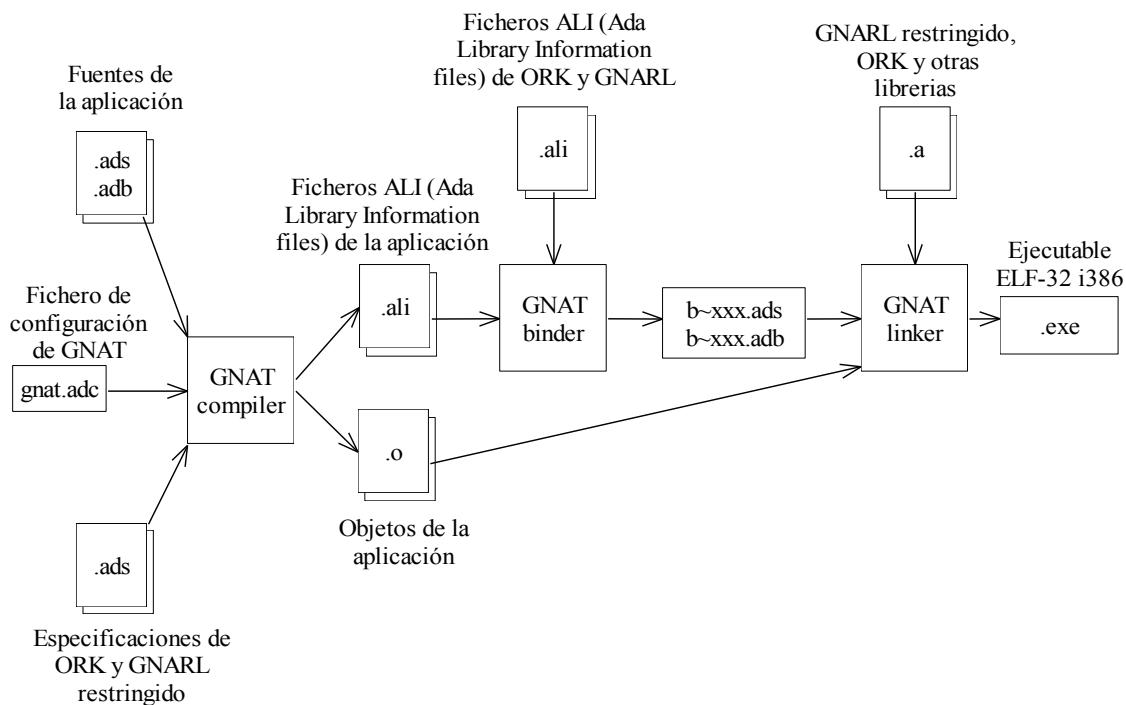


Figura B.1: Flujo de compilación de una aplicación GNAT/ORK.

B.2.1 Escribir y compilar programas Ada95.

El primer paso para desarrollar aplicaciones Ada es escribir el código fuente, usando para ello tu editor favorito. Dado las características del entorno en que nos encontramos, hay que tener en cuenta que los programas deben cumplir las restricciones del perfil Ravenscar. Por lo tanto el fichero `gnat.adc` debe tener un aspecto parecido al siguiente:

```
-- gnat.adc - configuration file template for the Ravenscar profile
pragma Ravenscar;
pragma Restrictions (Max_Tasks => 4);
-- N must be equal to the number of tasks of the application
pragma Restrictions(No_Allocators); -- does not work properly in
.                                         -- GNAT 3.13
pragma Restrictions(No_IO);
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy          (Ceiling_Locking);
```

Supongamos que tenemos un programa compuesto por los ficheros `hello.adb`, `tasking.adb` y `tasking.ads`. Para compilarlo y enlazarlo con `gnatmake`, se hace lo siguiente:

```
$ i386-ork-gnatmake hello \
-largs -k -specs ork_specs
```

La línea de comandos anterior puede ser sustituida por la siguiente lista de comandos, donde se compila y enlazan los fuentes por separado, como se describe en el manual *GNAT User's Guide*:

```
$ i386-ork-gcc -c hello.adb
$ i386-ork-gcc -c tasking.adb
$ i386-ork-gnatbind -x hello.ali
```

```
$ i386-ork-gnatlink -k -specs ork_specs hello.ali
```

Podemos obtener información de todos los símbolos enlazados en el ejecutable así como información de los lugares de localización de los símbolos añadiendo las siguientes opciones a `gnatlink`:

```
$ i386-ork-gnatmake -g hello.adb -largs -Xlinker -Map \
hello.map -k -specs ork_specs
```

Como resultado se ha creado un fichero `hello.map` con información de los ficheros y librerías enlazados, este tipo de ficheros son muy útiles en el desarrollo de sistemas empotrados. Después de los pasos de compilación obtenemos un fichero ejecutable en formato ELF-32 i386.

B.2.2 Ejecución y depuración de programas Ada95.

Lo primero será crear un disquete de arranque con GRUB de acuerdo a como se explica en el apéndice A. Una vez creado el disquete de arranque podemos ejecutar la aplicación, simplemente arrancando el PC de manera que este configurado para arrancar desde la disquetera.

La depuración de aplicaciones es llevada a cabo usando una línea serie (RS-232) que conecte el “host” y el “target”. En un host con GNU/Linux el COM1 recibe el nombre de `/dev/ttys0` y el COM2 de `/dev/ttys1`. Para poder depurar las aplicaciones estas deben ser compiladas con el flag `-g`. Para sincronizar el ejecutable con el depurador en el host, se deben añadir estas líneas al código de la aplicación (lo más cerca posible del punto de inicio de la aplicación).:

```
with Kernel.Peripherals; use Kernel.Peripherals;
procedure My_Proc is
begin
    Init_Serial_Communication_With_Gdb (Serial_Port_1);
    Set_Break_Point_Here;

    -- Application's code;

end My_Proc;
```

La aplicación ejecuta hasta el primer “Set_Break_Point_Here”, en este punto la ejecución se para y el target está listo para sincronizarse con el depurador.

En el directorio donde se encuentra el ejecutable de la aplicación arranca gdb:
\$ `gdb mprogram`

La conexión con el target se lleva a cabo con el siguiente comando de gdb:

```
(gdb) target remote /dev/ttys0
```


Apéndice C

Hojas de características del hardware.

C.1 PIC-16F84A.

MICROCHIP

PIC16F84A

18-pin Enhanced FLASH/EEPROM 8-Bit Microcontroller

High Performance RISC CPU Features:

- Only 35 single word instructions to learn
- All instructions single-cycle except for program branches which are two-cycle
- Operating speed: DC - 20 MHz clock input
DC - 200 ns instruction cycle
- 1024 words of program memory
- 68 bytes of Data RAM
- 64 bytes of Data EEPROM
- 14-bit wide instruction words
- 8-bit wide data bytes
- 15 Special Function Hardware registers
- Eight-level deep hardware stack
- Direct, indirect and relative addressing modes
- Four interrupt sources:
 - External RB0/INT pin
 - TMRO timer overflow
 - PORTB<7:4> interrupt-on-change
 - Data EEPROM write complete

Peripheral Features:

- 13 I/O pins with individual direction control
- High current sink/sources for direct LED drive
 - 25 mA sink max. per pin
 - 25 mA source max. per pin
- TMRO: 8-bit timer/counter with 8-bit programmable prescaler

Special Microcontroller Features:

- 10,000 erase/write cycles Enhanced FLASH Program memory typical
- 10,000,000 typical erase/write cycles EEPROM Data memory typical
- EEPROM Data Retention > 40 years
- In-Circuit Serial Programming™ (ICSP™) - via two pins
- Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST)
- Watchdog Timer (WDT) with its own On-Chip RC Oscillator for reliable operation
- Code protection
- Power saving SLEEP mode
- Selectable oscillator options

Pin Diagrams

PDIP, SOIC

SSOP

PIC16F84A

1.0 DEVICE OVERVIEW

This document contains device specific information for the operation of the PIC16F84A device. Additional information may be found in the PICmicro™ Mid-Range Reference Manual, (DS33023), which may be downloaded from the Microchip website. The Reference Manual should be considered a complementary document to this data sheet, and is highly recommended reading for a better understanding of the device architecture and operation of the peripheral modules.

The PIC16F84A belongs to the mid-range family of the PICmicro® microcontroller devices. A block diagram of the device is shown in Figure 1-1.

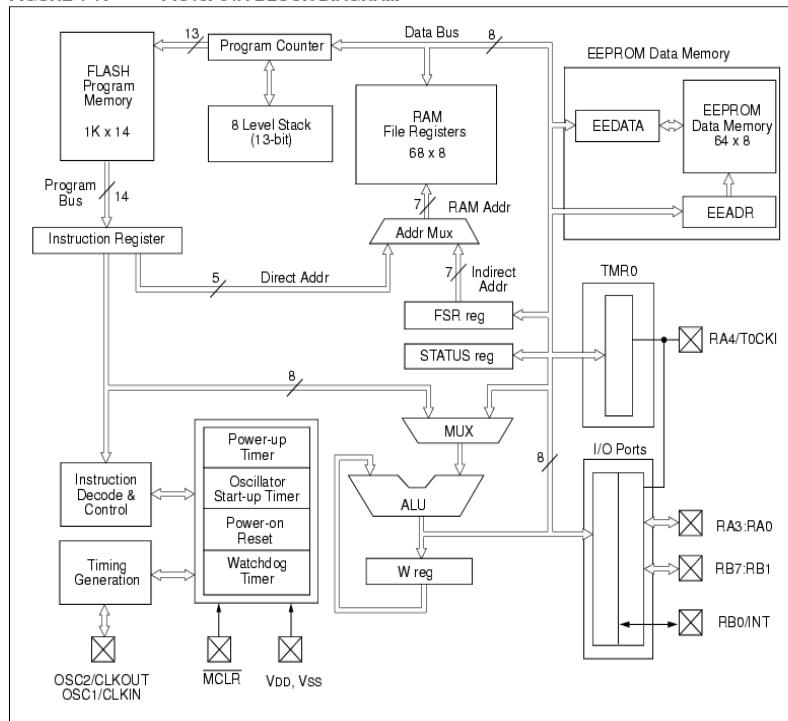
The program memory contains 1K words, which translates to 1024 instructions, since each 14-bit program memory word is the same width as each device instruction. The data memory (RAM) contains 68 bytes. Data EEPROM is 64 bytes.

There are also 13 I/O pins that are user-configured on a pin-to-pin basis. Some pins are multiplexed with other device functions. These functions include:

- External interrupt
- Change on PORTB interrupt
- Timer0 clock input

Table 1-1 details the pinout of the device with descriptions and details for each pin.

FIGURE 1-1: PIC16F84A BLOCK DIAGRAM



PIC16F84A

TABLE 1-1: PIC16F84A PINOUT DESCRIPTION

Pin Name	PDIP No.	SOIC No.	SSOP No.	I/O/P Type	Buffer Type	Description
OSC1/CLKIN	16	16	18	I	ST/CMOS ⁽³⁾	Oscillator crystal input/external clock source input.
OSC2/CLKOUT	15	15	19	O	—	Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. In RC mode, OSC2 pin outputs CLKOUT, which has 1/4 the frequency of OSC1 and denotes the instruction cycle rate.
MCLR	4	4	4	I/P	ST	Master Clear (Reset) input/programming voltage input. This pin is an active low RESET to the device.
RA0	17	17	19	I/O	TTL	PORTA is a bi-directional I/O port.
RA1	18	18	20	I/O	TTL	
RA2	1	1	1	I/O	TTL	
RA3	2	2	2	I/O	TTL	
RA4/T0CKI	3	3	3	I/O	ST	Can also be selected to be the clock input to the TMR0 timer/counter. Output is open drain type.
RB0/INT	6	6	7	I/O	TTL/ST ⁽¹⁾	PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs. RB0/INT can also be selected as an external interrupt pin.
RB1	7	7	8	I/O	TTL	
RB2	8	8	9	I/O	TTL	
RB3	9	9	10	I/O	TTL	
RB4	10	10	11	I/O	TTL	Interrupt-on-change pin.
RB5	11	11	12	I/O	TTL	Interrupt-on-change pin.
RB6	12	12	13	I/O	TTL/ST ⁽²⁾	Interrupt-on-change pin. Serial programming clock.
RB7	13	13	14	I/O	TTL/ST ⁽²⁾	Interrupt-on-change pin. Serial programming data.
Vss	5	5	5,6	P	—	Ground reference for logic and I/O pins.
Vdd	14	14	15,16	P	—	Positive supply for logic and I/O pins.

Legend: I = Input O = Output I/O = Input/Output P = Power
 — = Not used TTL = TTL input ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.

2: This buffer is a Schmitt Trigger input when used in Serial Programming mode.

3: This buffer is a Schmitt Trigger input when configured in RC oscillator mode and a CMOS input otherwise.

PIC16F84A

2.0 MEMORY ORGANIZATION

There are two memory blocks in the PIC16F84A. These are the program memory and the data memory. Each block has its own bus, so that access to each block can occur during the same oscillator cycle.

The data memory can further be broken down into the general purpose RAM and the Special Function Registers (SFRs). The operation of the SFRs that control the "core" are described here. The SFRs used to control the peripheral modules are described in the section discussing each individual peripheral module.

The data memory area also contains the data EEPROM memory. This memory is not directly mapped into the data memory, but is indirectly mapped. That is, an indirect address pointer specifies the address of the data EEPROM memory to read/write. The 64 bytes of data EEPROM memory have the address range 0h-3Fh. More details on the EEPROM memory can be found in Section 3.0.

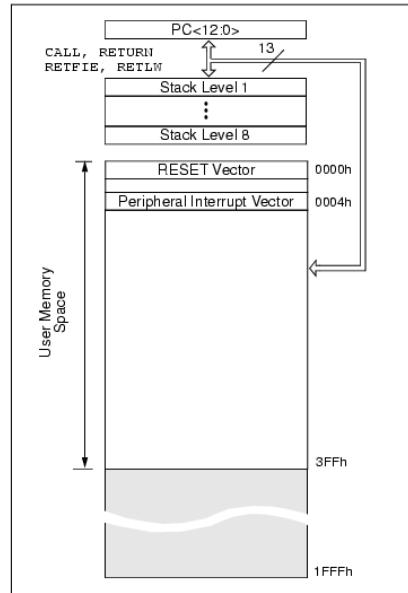
Additional information on device memory may be found in the PICmicro™ Mid-Range Reference Manual, (DS33023).

2.1 Program Memory Organization

The PIC16FXX has a 13-bit program counter capable of addressing an 8K x 14 program memory space. For the PIC16F84A, the first 1K x 14 (0000h-03FFh) are physically implemented (Figure 2-1). Accessing a location above the physically implemented address will cause a wraparound. For example, for locations 20h, 420h, 820h, C20h, 1020h, 1420h, 1820h, and 1C20h, the instruction will be the same.

The RESET vector is at 0000h and the interrupt vector is at 0004h.

FIGURE 2-1: PROGRAM MEMORY MAP AND STACK - PIC16F84A



PIC16F84A

2.2 Data Memory Organization

The data memory is partitioned into two areas. The first is the Special Function Registers (SFR) area, while the second is the General Purpose Registers (GPR) area. The SFRs control the operation of the device.

Portions of data memory are banked. This is for both the SFR area and the GPR area. The GPR area is banked to allow greater than 116 bytes of general purpose RAM. The banked areas of the SFR are for the registers that control the peripheral functions. Banking requires the use of control bits for bank selection. These control bits are located in the STATUS Register. Figure 2-2 shows the data memory map organization.

Instructions `MOVWF` and `MOVF` can move values from the W register to any location in the register file ("F"), and vice-versa.

The entire data memory can be accessed either directly using the absolute address of each register file or indirectly through the File Select Register (FSR) (Section 2.5). Indirect addressing uses the present value of the RP0 bit for access into the banked areas of data memory.

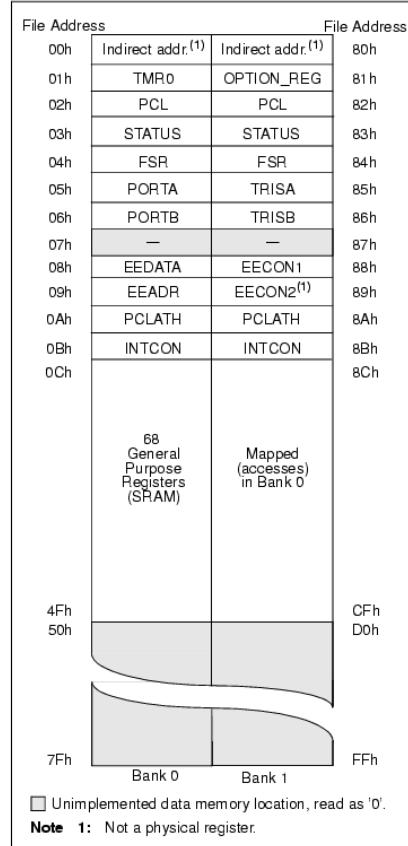
Data memory is partitioned into two banks which contain the general purpose registers and the special function registers. Bank 0 is selected by clearing the RP0 bit (`STATUS<5>`). Setting the RP0 bit selects Bank 1. Each Bank extends up to 7Fh (128 bytes). The first twelve locations of each Bank are reserved for the Special Function Registers. The remainder are General Purpose Registers, implemented as static RAM.

2.2.1 GENERAL PURPOSE REGISTER FILE

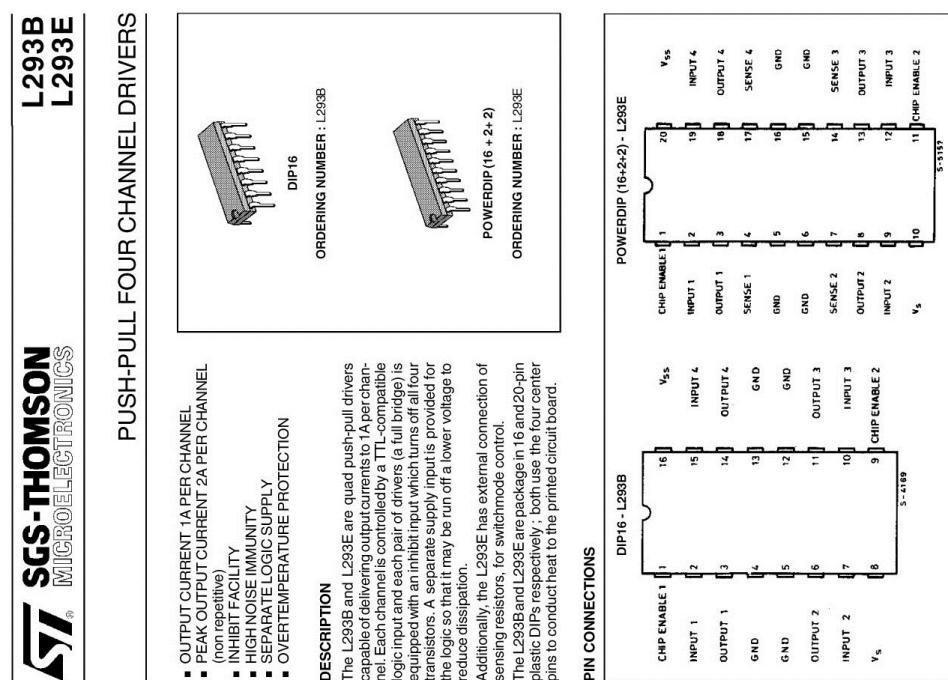
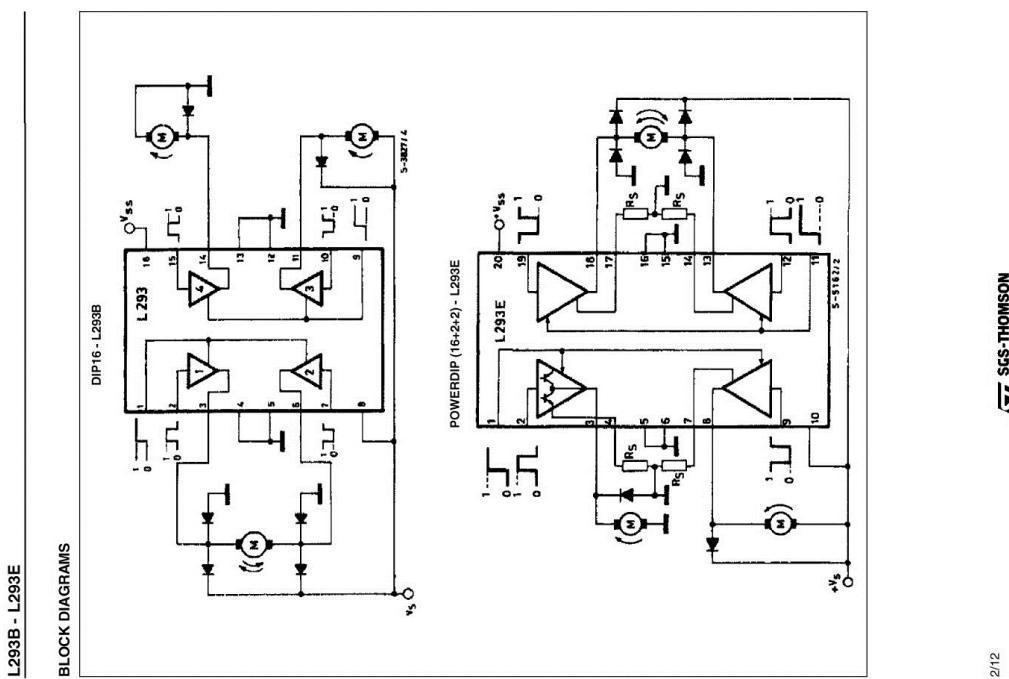
Each General Purpose Register (GPR) is 8-bits wide and is accessed either directly or indirectly through the FSR (Section 2.5).

The GPR addresses in Bank 1 are mapped to addresses in Bank 0. As an example, addressing location 0Ch or 8Ch will access the same GPR.

FIGURE 2-2: REGISTER FILE MAP - PIC16F84A

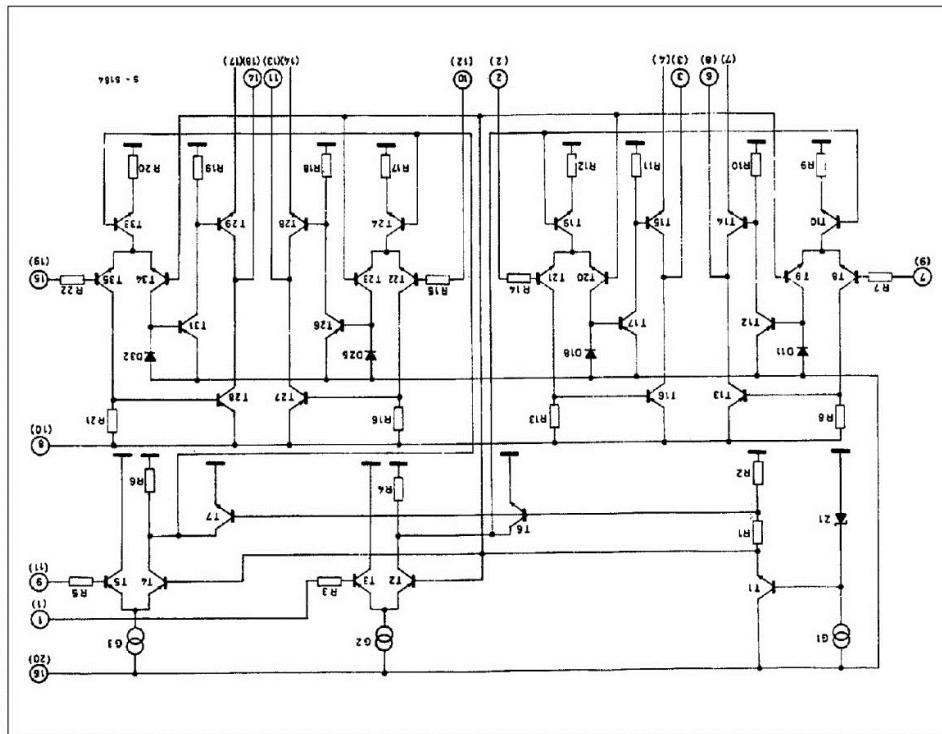


C.1 Circuito L293b.



L293B - L293E

SCHEMATIC DIAGRAM



(*) In the L293 these points are not externally available. They are internally connected to the ground (substrate).
 O Pins of L293

ABSOLUTE MAXIMUM RATINGS

Symbol	Parameter	Value	Unit
V_s	Supply Voltage	36	V
V_{ss}	Logic Supply Voltage	36	V
V_i	Input Voltage	7	V
V_{inh}	Inhibit Voltage	7	V
I_{out}	Peak Output Current (non repetitive $t = 5ms$)	2	A
P_{tot}	Total Power Dissipation at $T_{jmax} = 80^\circ C$	5	W
$T_{stg.}, T_j$	Storage and Junction Temperature	-40 to +150	°C

THERMAL DATA

Symbol	Parameter	Value	Unit
$R_{th(j-case)}$	Thermal Resistance Junction-case	Max. 14	°C/W
$R_{th(j-amb)}$	Thermal Resistance Junction-ambient	Max. 80	°C/W

ELECTRICAL CHARACTERISTICS

For each channel, $V_s = 24V$, $V_{ss} = 5V$, $T_{amb} = 25^\circ C$, unless otherwise specified

Symbol	Parameter	Test Conditions	Min.	Typ.	Max.	Unit
V_s	Supply Voltage	$V_{ss} = 0$	$V_{ss} = H$	4.5	36	V
V_{ss}	Logic Supply Voltage	$V_i = L$	$V_{in} = H$		2	mA
I_s	Total Quiescent Supply Current	$V_i = H$	$V_{in} = L$		16	24
I_{ss}	Total Quiescent Logic Supply Current	$V_i = L$	$I_o = 0$	44	60	mA
V_{il}	Input Low Voltage	$V_i = H$	$I_o = 0$		16	22
V_{ih}	Input High Voltage	$V_{ss} \leq 7V$	$V_{in} = H$		16	24
I_{il}	Low Voltage Input Current	$V_{ss} > 7V$	$V_{in} = L$	-03	1.5	V
I_{ih}	High Voltage Input Current	$V_{ss} \leq 7V$	$V_{in} = H$	2.3	V_{ss}	V
V_{inh}	Inhibit Low Voltage	$V_{ss} > 7V$	$V_{in} = L$	2.3	7	V
I_{inh}	Inhibit High Voltage	$V_{ss} \leq 7V$	$V_{in} = H$	-30	-100	mA
$V_{ce(sat)}$	Source Output Saturation Voltage	$I_o = 1A$	30	100	-	V
$V_{ce(sat)}$	Sink Output Saturation Voltage	$I_o = -1A$	-0.3	1.5	V	
V_{sens}	Sensing Voltage (pins 4, 7, 14, 17 (*))		2.3	V_{ss}	V	
t_r	Rise Time	0.1 to 0.9 V_o (*)		250	ns	
t_f	Fall Time	0.9 to 0.1 V_o (*)		250	ns	
t_{on}	Turn-on Delay	0.5 V to 0.5 V_o (*)		750	ns	
t_{off}	Turn-off Delay	0.5 V to 0.5 V_o (*)		200	ns	

* See figure 1
 ** Referred to L293E

TRUTH TABLE

V_i (each channel)	V_o	V_{inh} (*)
H	H	H
L	H	X
L	L	X

(*) High output impedance
 (*) Relative to the common-emitter channel

4/12

SGS-THOMSON
Integrated Circuits Division

3/12

SGS-THOMSON
Integrated Circuits Division

L293B - L293E

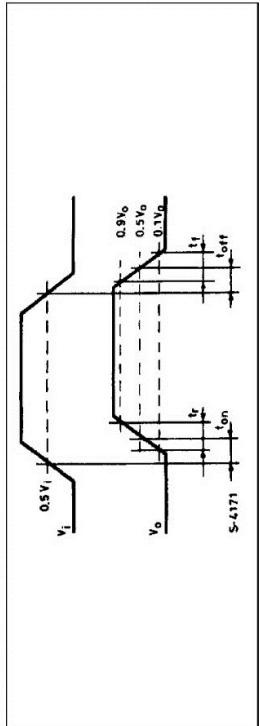


Figure 2 : Saturation voltage versus Current

Figure 3 : Source Saturation Voltage versus Ambient Temperature

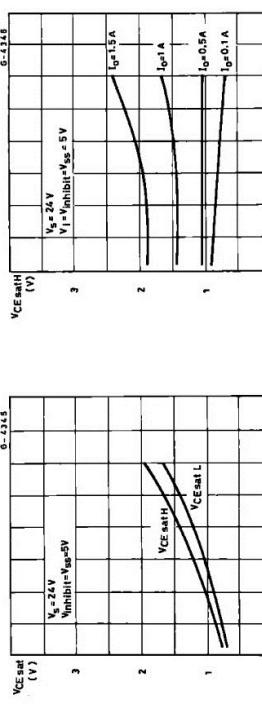
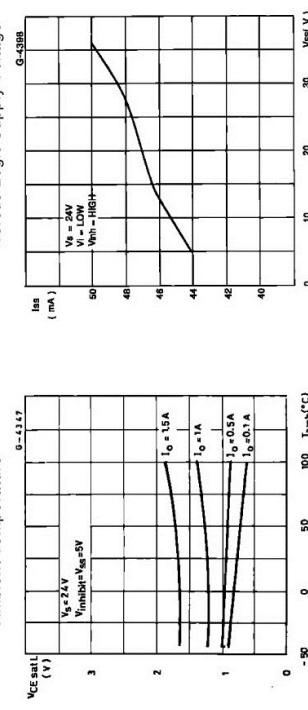


Figure 4 : Sink Saturation Voltage versus Ambient Temperature

Ambient Temperature (T_{amb}) [°C]	Sink Saturation Voltage (V_{dsat}) [mV]
0	-35
25	-38
50	-42
75	-46
100	-50

Figure 5 : Quiescent Logic Supply Current versus Logic Supply Voltage

Logic Supply Voltage (V_{cc}) [Volts]	Quiescent Logic Supply Current (I_q) [mA]
0.5	0.0
0.75	0.2
1.0	0.4
1.25	0.6
1.5	0.9



SCS-THOMSON

Figure 1 : Switching Timers

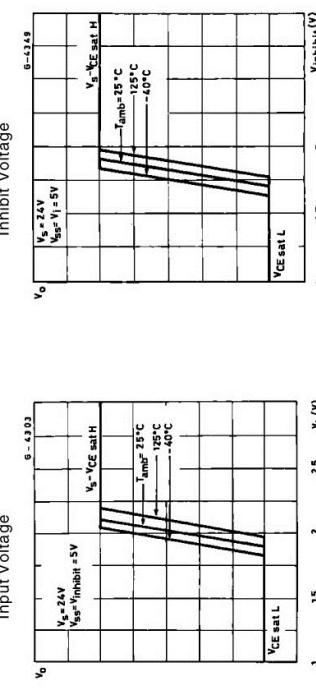


Figure 6 : Output Voltage versus Input Voltage

Figure 7: Output Voltage versus Inhibit Voltage



Figure 9 : Bidirectional DC Motor Control

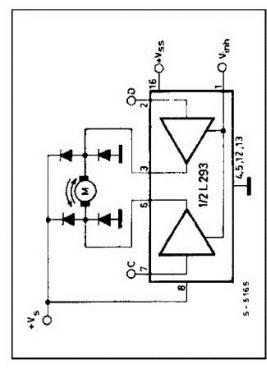
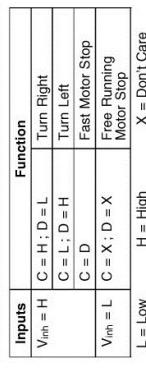


Figure 9: Bivariate HAC M8t8f BC



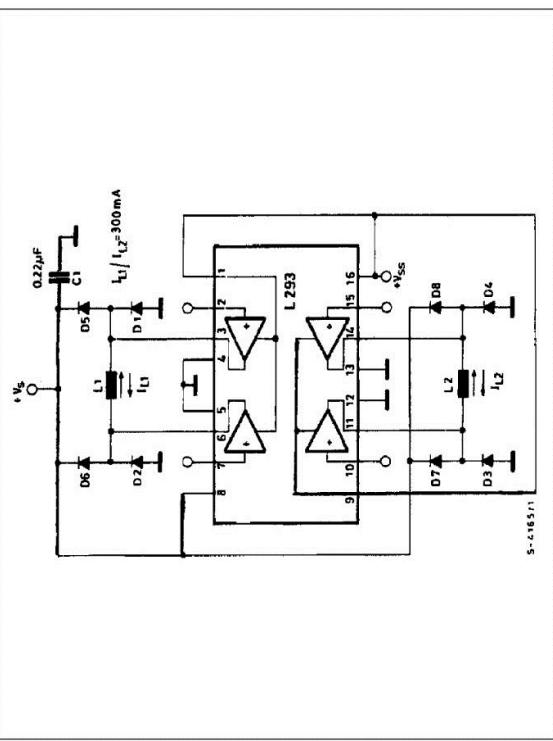
Inputs	Function
$V_{in} = H$	$C = H \cdot D = L$ Turn Right
$V_{in} ; D = H$	Turn Left
$C = D$	Fast Motor Stop
$V_{in} = L$	$C = X \cdot D = X$ Free Running
$L = Low$	$H = High$ Motor Stop $X = Don't Care$

6/12

5/12

L293B - L293E

Figure 10 : Bipolar Stepping Motor Control

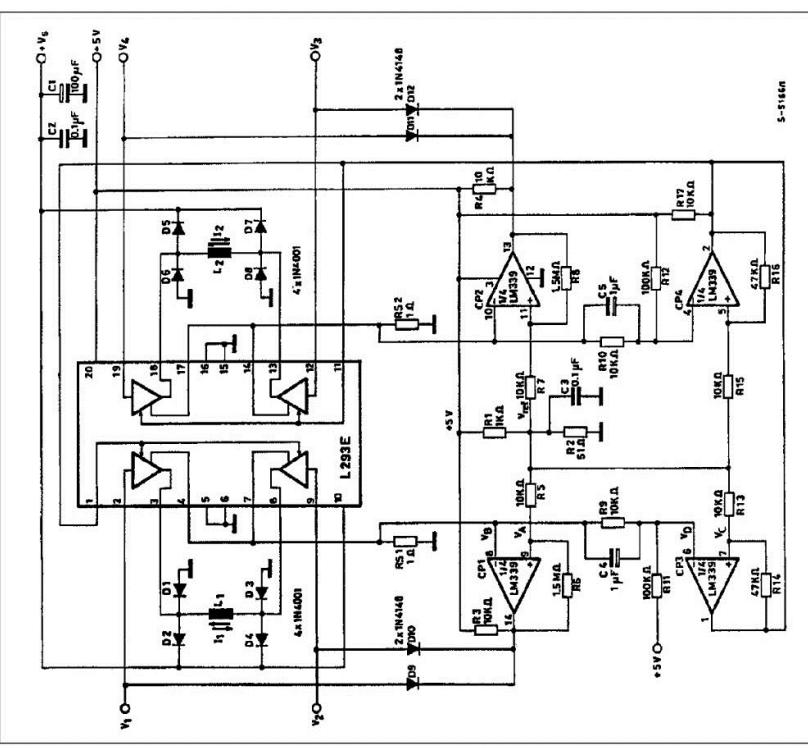


7/12

SGS-THOMSON**SGS-THOMSON**

Ref ID: 293B-293E-133-0001-00

Figure 11 :Stepping Motor Driver with Phase Current Control and Short Circuit Protection



8/12

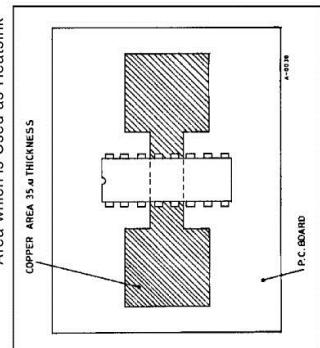
SGS-THOMSON

Ref ID: 293B-293E-133-0001-00

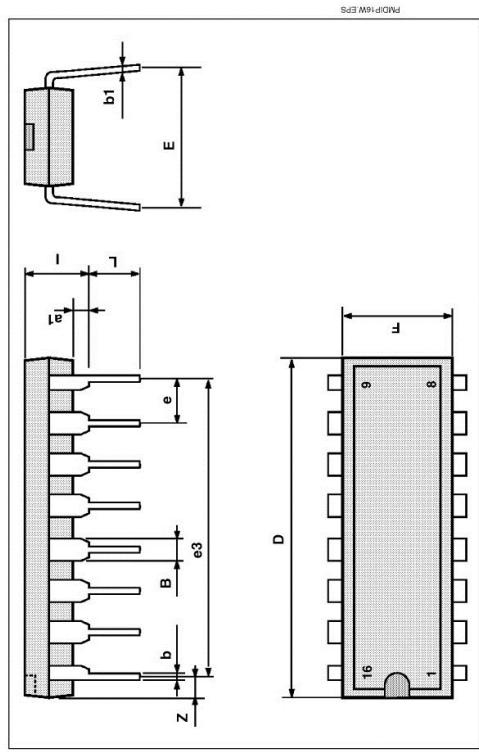
L293B - L293E**Mounting Instructions**

The R_{th} value of the L293B and the L293E can be reduced by soldering the GND pins to a suitable copper area of the printed circuit board as shown in figure 12 or to an external heatsink (figure 13).

Figure 12 : Example of P.C. Board Copper Area which is Used as Heatsink

**DIP16 PACKAGE MECHANICAL DATA**

Dimensions	Millimeters			Inches		
	Min.	Typ.	Max.	Min.	Typ.	Max.
a1	0.51			0.020		
B	0.77		1.65	0.030		0.065
b		0.5			0.020	
b1		0.25			0.010	
D		20			0.787	
E	8.5			0.335		
e	2.54			0.100		
e3	17.78			0.700		
F		7.1		0.280		
i		5.1			0.201	
L		3.3		0.130		
Z		1.27		0.050		

DIP16 PACKAGE MECHANICAL DATA

10/12

9/12

SGS-THOMSON

Semiconductors

ICs

GaAs

Laser

LED

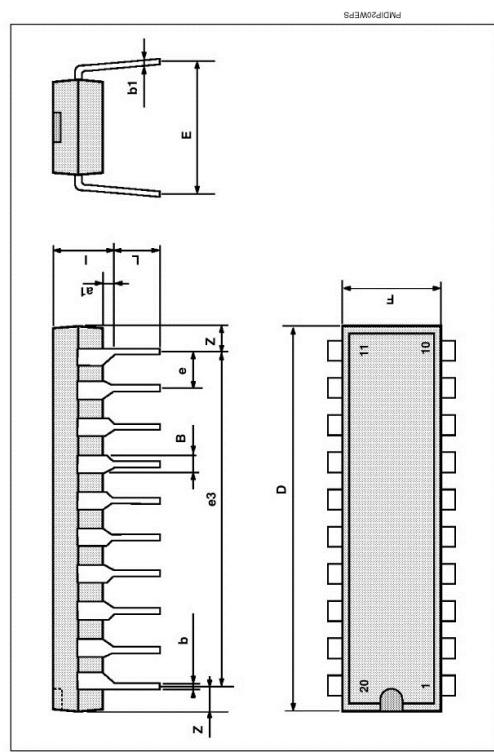
Optronics

Optoelectronics

L293B - L293E

POWERDIP (16+2) PACKAGE MECHANICAL DATA

Dimensions	Millimeters					Inches	
	Min.	Typ.	Max.	Min.	Typ.	Max.	
a1	0.51			0.020			
B	0.85		1.4	0.033		0.055	
b		0.5			0.020		
b1	0.38		0.5	0.015		0.020	
D			24.8			0.976	
E		8.8		0.346			
e		2.54		0.100			
e3		22.86		0.900			
F		7.1		0.280			
i		5.1		0.201			
L		3.3		0.130			
Z		1.27		0.050			



Information furnished is believed to be accurate and reliable. However, SGS-THOMSON Microelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of SGS-THOMSON Microelectronics. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. SGS-THOMSON Microelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of SGS-THOMSON Microelectronics.

© 1994 SGS-THOMSON Microelectronics - All Rights Reserved
SGS-THOMSON Microelectronics GROUP OF COMPANIES

Australia - Brazil - France - Germany - Hong Kong - Italy - Japan - Korea - Malaysia - Malta - Morocco - The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - Thailand - United Kingdom - U.S.A.

11/12

SGS-THOMSON
Microelectronics

12/12

ST
Microelectronics

Bibliografía

- [1] ANTONIO GARCÍA GUERRA, *Los microprocesadores xx86 y la arquitectura del PC*, Sistemas y Servicios de Comunicaciones S.L.
- [2] INTEL CORPORATION, *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1997.
- [3] INTEL CORPORATION, *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1997.
- [4] INTEL CORPORATION, *Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 1997.
- [5] JOSÉ FRANCISCO RUIZ MARTÍNEZ, *Plataforma de Desarrollo de Sistemas de Tiempo Real Empotrados en Ada 95*, Proyecto Fin de Carrera ETSIT-UPM. Abril-1998.
- [6] THE FLUX RESEARCH GROUP, *The OSKit: The Flux Operating System Toolkit*, Department of Computer Science, University of Utah.
<http://www.cs.utah.edu/flux/oskit/>
- [7] ON-LINE APPLICATIONS RESEARCH CORPORATION. *RTEMS Intel i386 Applications Supplement*. Edition 1, for RTEMS 4.5.0. September 2000.
- [8] ON-LINE APPLICATIONS RESEARCH CORPORATION. *RTEMS Porting Guide*. Edition 1, for RTEMS 4.5.0. September 2000
- [9] GRUPO DE COMPUTADORES Y TIEMPO REAL, *MaRTe OS*, Departamento de Electrónica y Computadores, Universidad de Cantabria.
<http://ctrpc17.ctr.unican.es/marte.html>
- [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 Plödereder (Eds.) *Reliable Software Technologies. Ada-Europe 2000*. Lecture Notes in Computer Science, 1845. Springer-Verlag (2000).

- [11] 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.
- [12] JOSÉ F. RUIZ, JUAN A. DE LA PUENTE, JUAN ZAMORANO AND RAMÓN FERNÁNDEZ-MARINA. *Exception Support for the Ravenscar Profile*. *Ada Letters*, XXI, 3, September 2001.
- [13] ALEJANDRO ALONSO AND JUAN A. DE LA PUENTE, *Implementation of Mode Changes with the Ravescar Profile*. *Ada Letters*, XXI, 1, March 2001.
- [14] *Open Ravenscar Real-Time Kernel*. Software Design Document.
- [15] *Open Ravenscar Real-Time Kernel*. Interface Control Document.
- [16] *Open Ravenscar Real-Time Kernel*. Operation Manual.
- [17] RODRIGO GARCÍA GARCÍA, *ORK (Open Ravenscar Real Time Kernel) Implementación sobre máquinas ERC-32*, Proyecto Fin de Carrera ETSIT-UPM. Septiembre-2001.
- [18] ANDREW S. TANENBAUM AND ALBERT S. WOODHULL, *Sistemas Operativos: Diseño e implementación*, Prentice Hall.
- [19] ROBERTO RICA GUTIÉRREZ, *Construcción de un brazo robótico e interface de control para un sistema empotrado basado en el sistema operativo RTEMS*, Proyecto Fin de Carrera FI-UPM. Julio-2001.
- [20] GORDON MATZIGKEIT AND OKUJI YOSHINORI, *The GRUB Manual*. Version 0.5.96, 5 October 2000.
<http://www.gnu.org/grub/>.
- [21] ADA CORE TECHNOLOGIES, *GNAT Reference Manual*. Version 3.13a. March 2000.
- [22] ADA CORE TECHNOLOGIES, *GNAT User's Guide*. Version 3.13a. March 2000.
- [23] ALAN BURNS AND ANDY J. WELLINGS, *Real-Time Systems and Programming Languages*. Addison-Wesley, 2 edition, 1996.
- [24] JOHN BARNES, *Programming in Ada95*. Addison Wesley. 2nd edition, 1998.
- [25] ALAN BURNS AND ANDY J. WELLINGS, *Concurrency with Ada*. Addison-Wesley, 2nd edition, 1996.
- [26] MICHAEL A. SMITH, *Object Oriented Software in Ada95*, Second edition. School of Computing, University of Brighton .

- [27] JERRY VAN DIJK. *GNAT Inline Assembler Tutorial. For Intel x86 processor, Version 1.1.* <http://www.adapower.com/articles>.
- [28] 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.
- [29] 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.
- [30] RICHARD M. STALLMAN AND ROLAND H. PESCH, *Debugging with GDB*. Seventh Edition, for GDB version 4.18, February 1999.
- [31] RICHARD M. STALLMAN, *Using and Porting the GNU Compiler Collection*. For gcc-2.95, July1999.
- [32] HOOD TECHNICAL GROUP. *Hood Reference Manual*, Release 4, June 1995.
- [33] HOOD TECHNICAL GROUP. *Hood User Manual*, Release 1.0, July 1994.
- [34] TULLIO VARDANEGA. *Development of On-Board Embedded Real-Time Systems. An Engineering Approach*. Technical Report ESA-STR-260. European Space Agency 1999
- [35] ALAN BURNS AND ANDY WELLING. *HRT-HOOD: A Structurated Design Method for Hard Real-Time Ada Systems*. Elsevier Science, 1995.
- [36] ALAN BURNS, *The Ravenscar Profile*. Ada Letters XIX(4), 49-52.
- [37] ADA, *Guidance for the use of the Ada Programming Language in High Integrity Systems*. ISO/IEC TR 15942:2000.