ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

Migración de un sistema operativo de tiempo real, MaRTE OS, a un microcontrolador

Para acceder al Título de

INGENIERO TÉCNICO INDUSTRIAL
ESPECIALIDAD ELECTRÓNICA INDUSTRIAL

Autor: Alberto Gutiérrez Castro
Santander, octubre de 2003

Quiero agradecerle muy sinceramente a Mario toda su ayuda y paciencia que a tenido, la ayuda que ha sido indispensable para realizar este proyecto, ha sido como una biblioteca de ayudas, y sobre todo la paciencia que ha tenido con todas y cada una de las consultas, preguntas y dudas a las que le he bombardeado continuamente.

Quisiera dar las gracias a Michael González Harbour por todo, por haberme dado la oportunidad de realizar este proyecto en el departamento, así como por su ayuda y orientación en la planificación y desarrollo del proyecto.

También quisiera agradecer y destacar el buen ambiente que se existe en el departamento de Electrónica y computadores de la facultad de Ciencias.

Y sobre todo se lo quiero agradecer a mi familia, por toda su comprensión y paciencia.

ÍNDICE

0 RESUMEN

MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) es un sistema operativo de tiempo real conforme con el subconjunto mínimo definido en el perfil del estándar POSIX 13. El estándar POSIX ("Portable Operating System Interface") define la interfaz que los sistemas operativos deben ofrecer a las aplicaciones, así como la semántica de los servicios ofrecidos por esa interfaz. En el estándar POSIX 13 se encuentra definido el perfil "Sistema de Tiempo Real Mínimo", pensado para aplicaciones empotradas pequeñas. Dicho perfil incluye un pequeño subconjunto de toda la funcionalidad definida en el estándar POSIX, lo que permite su implementación como un núcleo de sistema operativo pequeño y suficiente. MaRTE OS no es un sistema propietario, sino que tiene una licencia GPL (General Public License) con la que todo el código fuente se encuentra disponible para el público en general.

MaRTE OS se desarrolló originalmente para el microprocesadores de la familia intel x86. Nuestra intención será la de migrar a una plataforma diferente, que deberá cumplir unos requisitos mínimos tales como: trabajar con micros de 32 bits; tener un amplio juego de instrucciones que permitan optimizar las cualidades temporales de MaRTE OS; tener una capacidad de almacenamiento de al menos 1 MB, ya que una aplicación sencilla de gestión de 2 tareas en MaRTE OS utiliza unos 250 kbytes a los que se debe añadir el software de gestión de los periféricos y las librerías que requieren su correcto funcionamiento. Además deberá soportar compiladores de alto nivel de software libre. El compilador que se utilice deberá poder compilar tanto MaRTE OS como la aplicación de desarrollo, generando un único archivo listo para ser transmitido a la plataforma deseada, y esto implica compilar para lenguajes de alto nivel tales como Ada (MaRTE OS) y C (aplicación) por lo que deberemos tener acceso a su código fuente para adaptarlo a nuestras necesidades. Teniendo en cuenta todos estos requisitos se ha elegido el microcontrolador MC68332, y se ha encontrado un producto comercial, el controlador EyeBot, que nos servirá para el desarrollo del proyecto y para hacer una demostración del funcionamiento.

El controlador EyeBot trabaja con un potente microcontrolador MC68332 de 32 bits junto con una memoria ROM de 512 KBytes para la librería de funciones del controlador y 384 KBytes para la ejecución de programas, y una memoria RAM de 2 MBytes. Este microcontrolador está soportado por compiladores de alto nivel, y auque esta característica es esencial para la migración de MaRTE OS, conlleva la generación de un entorno de desarrollo cruzado que nos permita generar código para el MC68332, "target", en un ordenador de desarrollo PC, "host".

El objetivo principal del trabajo a realizar es el de portar MaRTE OS a la plataforma deseada. Esto implica generar un entorno de desarrollo cruzado que facilite la compilación, enlace y depuración tanto de MaRTE OS como de las aplicaciones de desarrollo que se diseñen.

El entorno de desarrollo cruzado consta de dos partes: herramientas de compilación, enlace de librerías y aplicaciones que generen un único archivo listo para ser ejecutado en la plataforma, y herramientas para la depuración de los archivos, a través del cable de comunicación entre el controlador y el PC de desarrollo.

Entre las herramientas de compilación nos basaremos en el compilador GCC adaptado para lenguajes de programación C y Ada, y unas BINUTILS cruzadas que contienen herramientas tan necesarias como son el GAS, para ensamblar el código generado en el lenguaje del microcontrolador, y el GLD, que nos permitirá enlazar todos los archivos compilados anteriormente y las librerías necesarias para generar un archivo único listo para ser ejecutado por el microcontrolador.

Para el desarrollo de una aplicación que demuestre el correcto funcionamiento del trabajo realizado, utilizamos el controlador EyeBot sobre la estructura SoccerBot, esto es, un pequeño robot con variedad de periféricos (sensores de presencia, motores, encoders, cámara digital color, LCD, etc...). El SoccerBot trabaja con RoBIOS, que es un conjunto de módulos software que nos permitirá acceder a los diferentes menús para la carga, ejecución y depuración de programas, así como de la configuración de los diferentes puertos y periféricos. Nos basaremos en los controladores y funciones que existen desarrollados e implementados en una amplia variedad de librerías que nos suministra el SoccerBot para el control de los diferentes periféricos. Así que cuando el SoccerBot ejecute una aplicación, MaRTE OS, se encargará de hacer todos los trabajos que corresponden al sistema operativo, control de tareas, temporización, etc. Y de RoBIOS utilizaremos funciones que controlen los periféricos que utilizaremos en cada una de las tareas.

Se ha realizado una pequeña aplicación que mueve el robot evitando los obstáculos que encuentra a su paso, con objeto de que sirva de demostración del trabajo realizado en el proyecto.

1 INTRODUCCIÓN

Para comprender mejor el desarrollo del proyecto debemos situarnos en su contexto, por un lado analizaremos los microprocesadores y los microcontroladores, por otro lado, lo que son los sistemas de tiempo real, los sistemas empotrados donde confluyen en el concepto de "sistemas empotrados de tiempo real", como evolución necesaria y obligada, nos introduciremos en los conceptos de sistema operativo, "System on Chip" y conjunto de estándares que definan el perfil de la interfaz entre la aplicación y el sistema.

Los sistemas basados en microprocesadores son sistemas separados en unidades cuya configuración mínima se basa: CPU, memorias y dispositivos de entrada/salida; y mantienen una comunicación entre unidades a través de los difentes buses: de direcciones, de datos y de control. Con estos sistemas podemos llegar a conseguir una gran variedad de configuraciones limitados por el peso, el tamaño, el consumo o el coste.

La otra opción que tenemos son los sistemas microcontroladores o sistemas cerrados, es decir, son sistemas en los que podemos encontrar la CPU, la memoria y los dispositivos de entrada/salida dentro de una misma pastilla o chips, suelen ser sistemas menos potentes pero con un menor consumo, una amplia variedad y más adecuados para sistemas empotrados, esto es, que incorporan el elemento de control dentro del propio sistema. Existe un gran nº de familias de microcontroladores que nos permitirán elegir el que más adecue a nuestra aplicación.

Muchos de los sistemas empotrados deben ser sistemas capaces de realizar tareas y responder a eventos asíncronos externos dentro de unos plazos temporales determinados que garanticen el correcto funcionamiento del sistema. Si el sistema de tiempo real lo aplicamos en un entorno empotrado manteniendo las características individuales de cada uno de los sistemas anteriores, estaremos desarrollando "sistemas empotrados de tiempo real".

1.1 SISTEMAS EMPOTRADOS DE TIEMPO REAL

A menudo se clasifica erróneamente un sistema empotrado como de tiempo real. Sin embargo, los sistemas empotrados no requieren por defecto de capacidades de tiempo real.

Podemos definir un sistema de tiempo real como aquel con restricciones temporales cuyo cumplimiento garantice el correcto funcionamiento; este tipo

de sistemas son capaces de realizar tareas concurrentes y responder a eventos asíncronos externos dentro de unos plazos temporales determinados.

Podemos clasificar los sistemas de tiempo real como:

- Sistemas de tiempo real estricto (Hard Real Time):
  Cuando el sistema debe responder siempre a los eventos de una forma determinista, con tiempo acotado. Un ejemplo sería el control electrónico de estabilidad de un automóvil (ESP), que debe asegurar la respuesta dentro de un tiempo acotado o de lo contrario la acción de control sería inútil.
- Sistemas de tiempo real no estricto (Soft Real Time):
  Cuando el sistema debe responder de una forma determinista, pero las restricciones temporales son suaves siendo suficiente que el comportamiento del sistema esté dentro de un rango de tolerancia. Por ejemplo, sistemas de comunicación donde el retraso en la entrega de un paquete de voz puede tolerarse descartándolo y procesando el siguiente.

Para implementar un sistema del primer tipo, necesitaremos un sistema operativo que garantice la ejecución del código generado con restricciones de tiempo real.

Los sistemas empotrados se caracterizan porque el computador constituye una parte más de un sistema mayor en el que se encuentra altamente integrado y en el que se dedica a realizar una función o un pequeño conjunto de ellas. Si este tipo de sistemas requiere capacidades de tiempo real, estaremos hablando de “Sistemas Empotrados de Tiempo Real”.

Todos estos sistemas están extendiéndose rápidamente gracias a los avances tecnológicos que desarrollan procesadores más baratos y más potentes, siendo cada vez más habitual encontrarse “computadores empotrados” en productos de consumo tan habituales como televisores, juguetes, reproductores DVD, etc.

El “World Semiconductor Trade Statistics” [WSTS] en su “Blue Book” de 2002 estima que en el mundo hay 5.000 millones de sistemas empotrados en uso, con unas expectativas de ventas de 2.000 millones de unidades / año para los próximos años.

Las limitaciones mencionadas eran mucho más relevantes hace unos años, debida a una menor potencia de los microcontroladores respecto a los microprocesadores, lo que exigía reducir al máximo los tiempos de cómputo y el tamaño de las aplicaciones empotradas. En consecuencia, la mayor parte de ellas se debían escribir en lenguaje ensamblador y sin utilizar ningún sistema operativo.

1.2 NECESIDAD DE UN SISTEMA OPERATIVO

Los sistemas de control demandan cada vez más servicios sofisticados que proporcionan los sistemas operativos modernos, estos son:

- Interfaces gráficas de alta resolución con el usuario (GUI). (X Windows, Web, etc…)
- Pilas de protocolos de red.
- Sistemas de ficheros y manejadores de dispositivos.

Además, el uso de un sistema operativo nos permitirá independizar el software de la plataforma (portabilidad), utilizar el soporte del sistema operativo (herramientas), y si consideramos esto, junto con la velocidad a la que se innova el hardware (CPU, chipset) y velocidad a la que este nuevo hardware queda obsoleto, podemos concluir que, los servicios deseados no pueden ser alcanzados utilizando la aproximación "antigua" de desarrollar todo el sistema o rescribir todo el código para cada diseño empotrado.

Por tanto, la solución eficaz es incorporar en nuestro sistema empotrado un sistema operativo que nos ofrezca estos servicios, con lo que logramos independizar el código de nuestra aplicación de control, y que además nos puede ofrecer la portabilidad del sistema empotrado una vez desarrollado.

En la actualidad, las mejoras tecnológicas y el consiguiente incremento en la complejidad de las aplicaciones han hecho casi imprescindibles la utilización de lenguajes de alto nivel y sistemas operativos en la mayoría de los entornos empotrados. Así, según datos de un estudio realizado por "Venture Development Corporation" [VDC00], ya en el año 2000 el 48% de los fabricantes de sistemas empotrados utilizaban sistemas operativos comerciales, el 26% sistemas operativos propios y solamente un 26% que no utilizaba ningún sistema operativo.

En este avance también ha influido la necesidad constante de los fabricantes de sistemas empotrados de reducir tiempos de desarrollo y aumentar la fiabilidad de las aplicaciones generadas. Estas mejoras se han logrado en gran medida mediante la aplicación en el desarrollo de las aplicaciones empotradas de los aspectos más avanzados de la ingeniería software para sistemas de tiempo real, como son la programación concurrente, estrategias sofisticadas de planificación de tareas, programación orientada a objetos, etc. Tecnologías, que difícilmente hubieran podido ser aplicadas sin la utilización de lenguajes de alto nivel y sistemas operativos.

En un principio, fueron los fabricantes de los sistemas hardware los que gradualmente procedieron a incorporar algunos de estos conceptos en sus sistemas de desarrollo específicos. La consecuencia de esta acción descoordinada y casi siempre competitiva, era que se requería por parte de las

empresas un gran esfuerzo para transferir o adaptar el software a las diferentes plataformas hardware de sus controladores industriales, que por su naturaleza son en general poco homogéneas. La solución a esta problemática se está buscando en el ámbito internacional mediante la elaboración de normas de estandarización, que permiten independizar los diferentes aspectos del diseño del sistema (arquitectura hardware, software de base, sistema operativo, aplicación), consiguiendo con ello una mayor modularidad, reusabilidad y, en definitiva, una reducción del tiempo de diseño y de los costos.

Entre las normas internacionales de estandarización destaca el estándar POSIX (Portable Operating System Interface) [PSX96] [PSX01] cuyo objetivo es permitir la portabilidad de aplicaciones a nivel de código fuente entre diferentes sistemas operativos.

Con este fin, y basándose en la interfaz del extensamente utilizado sistema operativo Unix, el estándar POSIX define la interfaz que los sistemas operativos deben ofrecer a las aplicaciones, así como la semántica de los servicios ofrecidos por esa interfaz.

La interfaz descrita en el estándar permite escribir aplicaciones de tiempo real de forma portable. Sin embargo, debido al tamaño del estándar POSIX, resulta inabordable su implementación completa en un sistema operativo destinado a computadores empotrados pequeños. Por esta razón en el estándar POSIX.13 se define el “Sistema de Tiempo Real Mínimo”, que está pensado para aplicaciones empotradas pequeñas. Dicho perfil incluye un pequeño subconjunto de toda la funcionalidad definida en el estándar POSIX, lo que permite su implementación en un núcleo de sistema operativo pequeño y eficiente.

Otro importante conjunto de estándares en el área de las aplicaciones de tiempo real está constituido por los lenguajes de programación. Así el C++ [C++98], y el Ada 95 [ADA95], aportan mecanismos para poder aplicar programación orientada a objetos. Ada 95 soporta los aspectos más avanzados de la ingeniería software para sistemas de tiempo real, como son la programación concurrente con tiempos de respuesta predecibles, estrategias sofisticadas de planificación de tareas y programación orientada a objetos. Su tipificación estricta permite detectar muchos problemas en tiempo de compilación, lo que incrementa considerablemente la fiabilidad del código generado.

El sistema operativo MaRTE OS está diseñado conforme con el perfil mínimo. Su estructura interna es modular y conocida de forma que, es sencillo incorporar nuevos servicios para proceder a su prueba y evaluación. Es importante destacar la fiabilidad, MaRTE OS se encuentra escrito en su mayor parte en lenguaje Ada (más una pequeña parte en lenguaje ensamblador del

MC68332), soporta la programación modular, con lo que se potencia la fiabilidad del código generado. MaRTE OS constituye una experiencia pionera, al se uno de los primeros sistemas operativos escritos en Ada que permite la ejecución de aplicaciones concurrentes escritas en C, Ada, o una mezcla de ambos.

Ciertamente existen sistemas operativos de tiempo real con interfaz POSIX cuyo código se encuentra disponible, como RTEMS [RTE03], pero el problema es que su diseño interno no sigue el modelo de 'threads' descrito en el estándar lo que, además de constituir una fuente de ineficiencia, complica la realización de modificaciones basadas en POSIX.

El sistema operativo MaRTE OS no consiste únicamente en un núcleo en el que se implementa la funcionalidad incluida en el subconjunto mínimo del POSIX, sino que además consta de una serie de aplicaciones y servicios que posibilitan la creación, carga y depuración de las aplicaciones. Ha sido necesario, por tanto, crear un entorno de desarrollo cruzado basado en Linux y en los compiladores de GNU [GNU02] GCC y GNAT.

1.3 LOS MICROPROCESADORES Y LOS MICROCONTROLADORES

La plataforma que soportaba originalmente MaRTE OS antes de la realización de este proyecto, era los microprocesadores intel x86, en la que se basa este proyecto, son los microcontroladores el motorota MC68332. A continuación trataremos de describir las características de cada una de las plataformas.

Un microprocesador es un circuito integrado que ejecuta instrucciones de forma secuencial y síncrona, que necesita una tensión continua y estable que lo alimente (tensiones típicas de alimentación: 5v, 3.3v, 2.5v, 7.2v), una señal de reloj y circuitos digitales adicionales, que configuren un sistema mínimo microprocesador.

El microprocesador saca al exterior las líneas de sus buses de direcciones, datos y control, para permitir conectarlo con la Memoria y los Módulos de E/S y configurar un computador implementado por varios circuitos integrados. Se dice que un microprocesador es un sistema abierto porque su configuración y sus componentes varían de acuerdo con la aplicación a la que se destine.

El microprocesador interpreta o decodifica combinaciones de bits (órdenes) y genera señales digitales internas y/o externas para el resto de los circuitos, para así ejecutar de manera continua una secuencia de órdenes (programa).

Un sistema mínimo microprocesador estará formado por:

- La Unidad Central de Proceso (CPU)
  + La Unidad de Control (CU)
  + La Unidad Aritmético-Lógica (ALU)
  + Los Registros
- Memoria
- Los dispositivos de entrada/salida

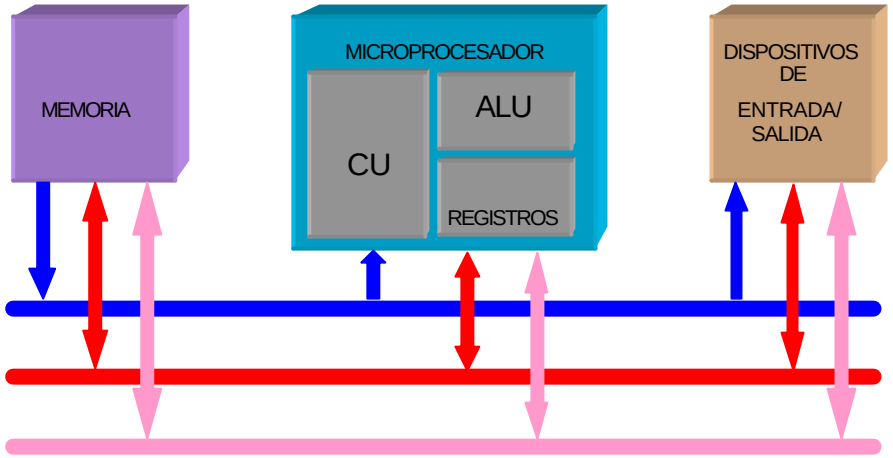


Figura. Sistema mínimo microprocesador

La comunicación entre unidades se realiza a través de:

- Bus de direcciones (unidireccional), el número de líneas (m) nos indica la capacidad para direccionar 2^m posiciones de memoria.
- Bus de datos (bidireccional), para la transmisión de datos, el número de líneas del bus nos indica el número de bits capaz de transmitir en paralelo.
- Bus de control, heterogéneo, depende de cada microcontrolador.

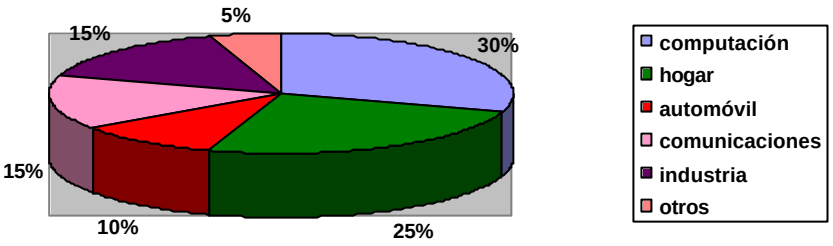
Teniendo en cuenta esta configuración básica de los microprocesadores, podemos encontrar gran variedad de configuraciones diferentes, tan sólo deberemos elegir la adecuada para la aplicación que se vaya a realizar. Las características de los microprocesadores vienen dadas por:

- Tamaño de los datos que maneja
- Frecuencia de trabajo
- Capacidad de direccionamiento
- Densidad de integración
- Número de registros internos

?? Juego de instrucciones que maneja
?? Arquitectura interna y externa

Podemos clasificar los sistemas basados en microprocesadores por el tamaño de los datos que maneja, entonces encontraremos desde pequeños microcontroladores de 4 bits hasta potentes computadores con procesadores digitales de señal (DSPs), procesadores integrados de aplicación específica (ASIP) que están optimizados para una aplicación concreta o procesadores RISC de 64 bits. Sin embargo, la mayor parte de los sistemas ven limitadas sus prestaciones por razones de tamaño, peso, consumo y coste.

La distribución de los microprocesadores por sectores:



Las compañías diseñadoras y fabricantes de circuitos integrados desde siempre han participado en la carrera por obtener dispositivos más potentes, basando esta potencia en un aumento de los recursos internos del dispositivo. Respecto a esto último, el avance ha sido vertiginoso. En los años 80 un simple controlador de disquetera contenía más del doble de los componentes que hoy en día tiene un ordenador.

Todas las decisiones arquitecturales como RISC vs. CISR, Firmware vs. Wired, etc..., se fueron concentrando gracias al avance de la VLSI (Very Large Scale Integration), hasta llegar a ser decisiones internas a un sólo chip, ya que en el mismo, es posible encontrar recursos tales como CPU, memorias, periféricos de comunicaciones, control, potencia, interfaz, etc... Esta necesidad de integración de los sistemas microprocesadores desarrolló el microcontrolador.

El microcontrolador es un sistema cerrado. Todas las partes del computador están contenidas en su interior y sólo salen al exterior las líneas que gobiernan los periféricos.

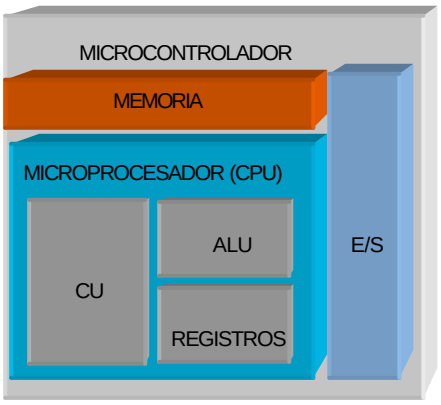
Si sólo se dispusiese de un modelo de microcontrolador, éste debería tener muy potenciados todos sus recursos para poderse adaptar a las exigencias de las diferentes aplicaciones. Esta potenciación supondría en muchos casos un despilfarro. En la práctica cada fabricante de microcontroladores oferta un elevado número de modelos diferentes, desde los

más sencillos hasta los más poderosos. Es posible seleccionar la capacidad de las memorias, el número de líneas de E/S, la cantidad y potencia de los elementos auxiliares, la velocidad de funcionamiento, etc. Por todo ello, un aspecto muy destacado del diseño es la selección del microcontrolador a utilizar.

Aunque en el mercado de la microinformática la mayor atención la acaparan los desarrollos de los microprocesadores, lo cierto es que se venden cientos de microcontroladores por cada microprocesador.

Existe una gran diversidad de microcontroladores. Quizá la clasificación más importante sea entre microcontroladores de 4, 8, 16 ó 32 bits. Aunque las prestaciones de los microcontroladores de 16 y 32 bits son superiores a los de 4 y 8 bits, la realidad es que los microcontroladores de 8 bits dominan el mercado y los de 4 bits se resisten a desaparecer. La razón de esta tendencia es que los microcontroladores de 4 y 8 bits son apropiados para la gran mayoría de las aplicaciones, lo que hace absurdo emplear micros más potentes y consecuentemente más caros. Uno de los sectores que más tira del mercado del microcontrolador es el mercado automovilístico. De hecho, algunas de las familias de microcontroladores actuales se desarrollaron pensando en este sector, siendo modificadas posteriormente para adaptarse a sistemas más genéricos. El mercado del automóvil es además uno de los más exigentes: los componentes electrónicos deben operar bajo condiciones extremas de vibraciones, choques, ruido, etc. y seguir siendo fiables. El fallo de cualquier componente en un automóvil puede ser el origen de un accidente. [DSECUPM]

La siguiente figura muestra la estructura de un microcontrolador



La demanda actual de las empresas fabricantes de ASICs , circuitos integrados para aplicaciones específicas, se está orientando hacia el diseño de

un sistema o subsistemas en un sólo chip aplicable a electrónica de consumo, por ejemplo, telefonía celular. Estos sistemas que hoy se engloban en la sigla SoC (System on Chip) contienen a menudo circuitos analógicos, circuitos de radio frecuencia (RF) y subsistemas de señales mixtas para satisfacer la demanda en aplicaciones de electrónica de comunicaciones. [CSIC00]

Cabe destacar de la tecnología SoC como la más floreciente para los computadores basados en sistemas empotrados, que tienen la característica de poder ser programados y configurados e incluso de ser reactivos (responden a eventos en el entorno en el cual se encuentran empotrados), desarrollan información del proceso y el control de tareas, siendo a menudo invisibles para el usuario. La implementación de estos sistemas, tanto los componentes hardware como software, es una parte esencial del diseño. Según la *Informatics and Mathematical Modelling Technical University of Denmark* [IMM03]

Los SoC son en su mayor parte microcontroladores que trabajan bajo un sistema operativo para entornos empotrados. El avance de los SoC, tiene una frontera, no siendo esta funcional, sino arquitectural. Aun así, con los SoC se puede llegar a conseguir una reducción en el tamaño de los equipos del 50% al 80% respecto a las tecnologías microelectrónicas convencionales, lo que permitirá un alto nivel de integración. Por otro lado, una aplicación basada en un SoC puede consumir hasta un 60 % menos de energía respecto a las convencionales, lo que permitirá poner baterías más pequeñas o que éstas duren más tiempo. Todo ello contando además con considerables reducciones en los costes de los equipos y productos finales.

Los SoC, con el fin de abordar el problema que supone el diseño de sistemas empotrados se han caracterizado por ofrecer soluciones con una mayor capacidad de interfaz en detrimento de la capacidad de procesar o cálculo, esto es, mediante el uso del microcontrolador más adecuado a la aplicación a desarrollar.

1.4 EL MICROCONTROLADOR MC68332

En la elección del microcontrolador se ha optado por el MC68332 ya que contiene un microprocesador de 32 bits, que trabaja a una frecuencia de 33 MHz, tratándose de un microcontrolador de alta densidad de integración desarrollado con la tecnología metal-óxido semiconductor (HCMOS). Tiene un alto desarrollo en el manejo de datos con poderosos subsistemas periféricos a través del Bus de Comunicación entre Módulos (IMB). Además es un microcontrolador de la familia motorola 68300 de controladores empotrados de 32 bits, con muchas características de los MC68010 y del MC68020, esto es, compatible con la familia M68000.

El IMB es el encargado de la interconexión de los módulos SIM, TPU y QSM con la CPU32 (el microprocesador), entre otras características el módulo SIM, que es el modulo integrado del sistema, se encargará de la configuración y protección del sistema, el módulo TPU, la Unidad Procesadora del Tiempo, que dispone de 16 canales de propósito general para el manejo de funciones de tiempo y el módulo QSM, Módulo de Cola de Periféricos Serie, que es el encargado de facilitar la interconexión de más periféricos o comunicación entre procesadores.

El microcontrolador dispone de una memoria ROM de 512 KBytes (128 Kbytes para la librería de funciones del controlador, y 384 Kbytes para la ejecución de programas) y una memoria RAM de 2 MBytes, memoria suficiente para desarrollar con éxito nuestra aplicación.

El MC68332 con el que hemos trabajado se encuentra integrado en el controlador Eyebot, elemento de control de la plataforma de desarrollo SoccerBot.

En el SoccerBot podemos encontrar 1 puerto paralelo, 3 puertos serie, 8 entrados digitales, 8 salidas digitales, 8 entradas analógicas, una cámara digital, un LCD de gráficos, un micrófono, 2 motores DC y sus servos, 6 sensores de posición por infrarrojos, el módulo de depuración (BDM) y cable para conectarlo al 'Host', puerto de infrarrojos, micrófono, speaker y golpeador.

La facilidad de programación del un microcontrolador es una importante característica a destacar. Dispone de ocho registros de datos multifunción y siete registros de direcciones de propósito general. Los registros de datos soportan datos de 8 bits (tamaño byte), 16 bits (tamaño word) y 32 bits (tamaño longword) como longitud de la palabra para todas las operaciones. Aunque el 'contador del programa' (PC) y el 'puntero de la pila' (SP) son registros de propósito especial, también se encontrarán disponibles para la mayoría de los direccionamientos. Facilidad de programar el chequeo y diagnosis mediante las instrucciones 'trace&trap'.

Como las aplicaciones en el controlador pueden llegar a extenderse y volverse bastante complejas, la el MC68332 es capaz de trabajar con compiladores para lenguajes de alto nivel, característica básica para poder desarrollar el proyecto.

A la hora de programar el microcontrolador, hay que tener en cuenta que podemos encontrarnos en dos estados de privilegio, como usuario y como supervisor, para la implementación de MaRTE OS en el microprocesador, deberemos encontrarnos en el modo supervisor, en cambio, cuando realicemos la aplicación bastará con encontrarse en el modo usuario.[MCPU]

1.5 ENTORNO DE DESARROLLO CRUZADO

Uno de las características del sistema operativo MaRTE OS es que su código fuente debe estar accesible para cualquier usuario, de un modo más preciso, se trata de software abarcado por General Public License (GPL)[GNU02], esto es un conjunto de específico de términos de distribución para proteger un programa de 'software libre'. Software libre se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software. Para que las libertades de hacer modificaciones y de publicar versiones mejoradas tengan sentido, debemos tener acceso al código fuente del programa. Por lo tanto, la posibilidad de acceder al código fuente es una condición necesaria para el software libre.

El software libre debe su impulso y creación a la Fundación para el Software Libre (FSF)[GNU02], que está dedicada a eliminar las restricciones sobre el copiado, redistribución, entendimiento, y modificación de programas de computadoras, promocionando el desarrollo y uso del software libre en todas las áreas de la computación, pero muy en particular en el proyecto GNU.

Linux es un sistema operativo derivado de UNIX desarrollado originalmente por Linux Torvals asistido por la comunidad de desarrolladores a lo largo de todo el mundo a través de Internet. Amparado bajo la licencia pública, GNU General Public License (GPL), todo el código fuente está disponible para el público en general y podemos recalcar las siguientes características:

- Altamente probado y estable.
- Código público GPL.
- Desarrollado en C, lo que lo hace altamente portable.
- Basado en el compilador GNU C, lo que permite utilizar una gran cantidad de herramientas de desarrollo.
- Disponible para sistemas Intel PC.
- Soporte de gran número de periféricos.
- Muy documentado (Linux Documentation Project)

MaRTE OS tiene una licencia GPL, tendremos acceso al código fuente, trabajaremos también con herramientas desarrolladas en el proyecto GNU, como son los compiladores GCC y GNAT, compiladores de alto nivel de lenguajes C y Ada, las BINUTILS, conjunto de utilidades que ayudan en la compilación y la NEWLIB, que son un conjunto de librerías C.

La ventaja de trabajar con estas herramientas es que está probada sobradamente su fiabilidad, debido a su amplia utilización, además el conocimiento del código fuente sin restricciones nos deja la posibilidad de adaptarlas y modificarlas de la forma que más convenga.

Buscaremos la configuración más adecuada para nuestro propósito, e incluso, una vez se adquiera un conocimiento sobre las herramientas que generen el entorno de desarrollo, se deberá buscar la versión más adecuada que permita el mejor acoplamiento entre ellas, aplicando 'parches' y modificando los *Makefile*, que son unos pequeños archivos con permiso de ejecución, que utilizaremos para ejecutar una serie de instrucciones de

Una peculiaridad de los sistemas empotrados es la falta de un interfaz con el usuario y/o programador, por lo que para desarrollar cualquier clase de aplicación o simplemente acceder al control del microcontrolador directamente en la máquina. Se hace necesario generar un entorno de desarrollo cruzado que permita el control deseado sobre el microcontrolador.

Un entorno de desarrollo cruzado estará formado por una parte hardware y otra software, en la parte hardware encontraremos el 'host' máquina que tendrá instalada el compilador y depurador cruzados, teniendo como misión generar código para el microcontrolador o plataforma de desarrollo. El compilador cruzado se hace necesario ya que la plataforma de desarrollo por sí misma no podría entender una aplicación escrita en un lenguaje de alto nivel. Cabe otra posibilidad, y es que si la plataforma de desarrollo tuviera una interfaz con el usuario, este debería ser capaz de programar en el lenguaje de la máquina de desarrollo.

El 'host' será una máquina que trabaje con un sistema operativo (Linux, en nuestro caso) que soporte las herramientas necesarias para poder generar el código para la plataforma de desarrollo. Herramientas tales como compilador de ada *gnat*, el compilador *gcc*, las herramientas *binutils* y las librerías *newlib*. La idea será compilar código en lenguaje Ada o C (también podremos utilizar

ensamblador) basándonos en un correcto uso de: las herramientas de las *binutils* y las librerías suministradas por las *newlib*, para generar código binario en un formato entendible para la plataforma de desarrollo (*file*.hex),es decir, un compilador cruzado y depurador cruzados con los que podemos compilar una aplicación escrita en lenguaje de alto nivel (Ada o C) o en lenguaje ensamblador, y enlazarla con las librerías necesarias (librerías propias del SoccerBot) para la obtención de un archivo listo para ejecutar en la plataforma de desarrollo.

El depurador cruzado servirá para corregir los errores (tanto de compilación como de ejecución) y poder continuar con la depuración sin necesidad de compilar, enlazar y descargar el programa o aplicación otra vez.

Una vez que tengamos las herramientas que nos permitan obtener archivos para el MC68332, tendremos que preparar a éste para que trabaje bajo el sistema operativo deseado: MaRTE OS.

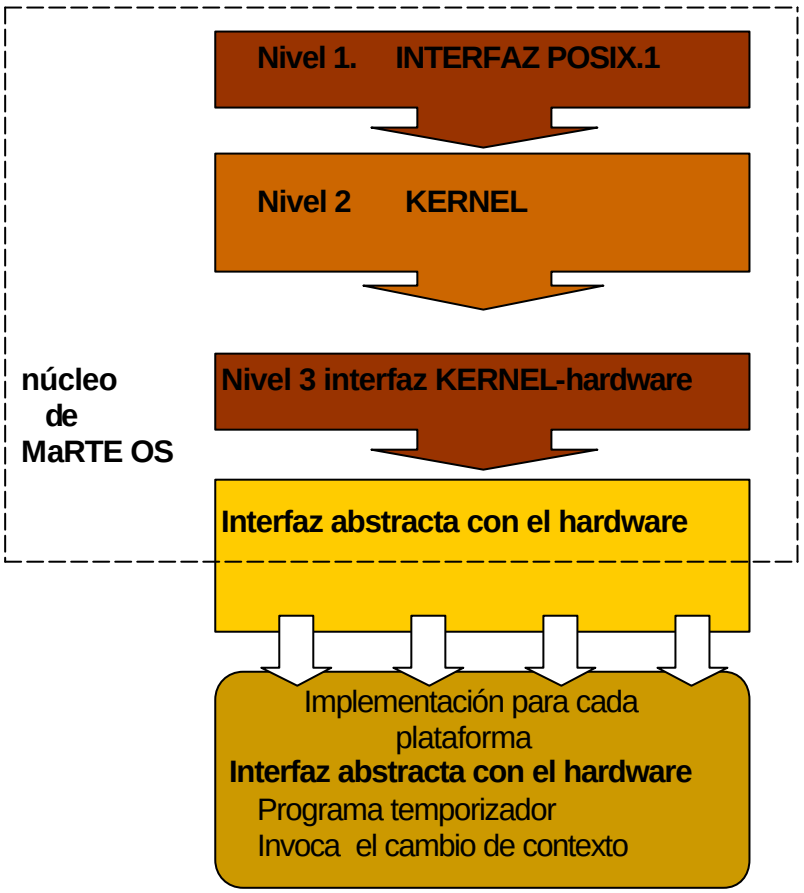
El núcleo de MaRTE OS se encuentra escrito en tres lenguajes: Ada, C y ensamblador (en el hardware_interface y en alguno de sus paquetes). El compilador cruzado deberá entender los paquetes escritos en estos lenguajes y generar para cada uno de ellos un *'object file'* (*.o), que uniremos a las librerías estándar adecuadamente compiladas para el MC68332 (para cada uno de los archivos de las librerías también se generarán *.o, aunque los podremos encontrar librerías compiladas para esta máquina, con la forma *.a).

Esto se puede hacer gracias a las características del sistema operativo MaRTE OS, por las que en el computador de desarrollo (host), la aplicación es compilada y posteriormente enlazada con las librerías que forman el núcleo de MaRTE OS. El ejecutable así generado constituye un programa independiente listo para ser cargado en el microprocesador del sistema empotrado.

1.6 PORTABILIDAD DE MaRTE OS

En MaRTE OS, en el núcleo se incluye una interfaz abstracta de bajo nivel para acceder al hardware. En ella se define la visión que del hardware tienen las partes del núcleo que son independientes de la plataforma de ejecución. Esta interfaz constituye la única parte del núcleo que es dependiente del hardware, lo que facilita el portado de MaRTE OS a distintas plataformas.

La siguiente figura nos muestra la estructura de niveles del núcleo de MaRTE OS:



La figura constituye una representación esquemática de la arquitectura interna del núcleo de MaRTE OS. Podemos diferenciar tres niveles, el *nivel 1* corresponde a la parte de la interfaz POSIX.1 aportada por el núcleo, que corresponde a los ficheros cabecera C de sus mismos nombres, los citados paquetes están constituidos en su mayor parte por renombrados de las funciones y subtipos de los tipos definidos en los paquetes del núcleo que

implementan la interfaz POSIX. El nivel 2, contiene los paquetes que componen el núcleo están organizados jerárquicamente a partir del paquete raíz *kernel*, en el que se definen los tipos, constantes y operaciones básicas que serán utilizados por el resto del núcleo. Existe un nivel 3, que será el que se encuentre más relacionado con la interfaz abstracta del hardware, será el nivel que se encargue de la funcionalidad básica del núcleo, encargado de la gestión directa de las tareas y los eventos temporales, siendo desde este nivel desde donde se realizaran las llamadas para la programación del temporizador hardware y las llamadas a la rutina de cambio de contexto.

Por lo tanto, teniendo en cuenta la estructura interna de MaRTE OS, para que el sistema operativo sea portado de una plataforma a otra, tan sólo deberemos adaptar los paquetes que se definen en la interfaz abstracta del hardware a la plataforma en la que queremos implementar MaRTE OS, manteniéndose los paquetes correspondientes al núcleo inalterados.[MAR02]

1.7 OBJETIVOS DEL PROYECTO

Los objetivos de este proyecto son:

- Desarrollo del entorno de desarrollo cruzado para el MC68332

  Adaptados según nuestras necesidades, comenzando con la creación de un entorno de desarrollo simple, capaz de generar código para el MC68332, compilando programas codificados en lenguaje C, lo ampliaremos para que sea capaz de codificar programas escritos en Ada, y en la última evolución haremos que las aplicaciones que implementen alguna de las características POSIX de MaRTE OS, mediante el correcto enlazando con las diferentes librerías y paquetes propios de MaRTE OS.
  - Adaptación de un compilador cruzado para que soporte aplicaciones escritas en lenguaje Ada y C.
  - Adaptación de un depurador cruzado para el MC68332.

- Migración de MaRTE OS al MC68332

  La Interfaz Abstracta con el Hardware proporciona al resto del núcleo de MaRTE OS una visión abstracta de la plataforma y debe de constar de los siguientes elementos: temporizador hardware, un reloj que permita leer y modificar la hora actual, un dispositivo que permita la habilitación o deshabilitación de interrupciones de forma individual o conjunta, operaciones de cambio de contexto, así como operaciones de conversión de tiempos. Este es precisamente el objetivo principal del presente trabajo, demostrar que esta portabilidad es posible.

- Implementación de la Interfaz Abstracta con el Hardware en el MC68332. Identificará la plataforma para la cual se encuentra versionado MaRTE OS.
- Generación de un conjunto de utilidades que automaticen la compilación, enlazado y carga de aplicaciones que utilizan MaRTE OS en MC68332.
  Adaptaremos algunas herramientas proporcionadas por MaRTE OS para este propósito.

?? Desarrollo de una aplicación de demostración que implemente alguna de las características POSIX de MaRTE OS sobre la plataforma SoccerBot.

Antes de la aplicación de demostración se demostrará el correcto funcionamiento de MaRTE OS en el controlador EyeBot compilando y ejecutando los diferentes ejemplos proporcionados por MaRTE OS.

La aplicación trabajará con tareas concurrentes con diferentes prioridades implicando diferentes periféricos del SoccerBot de tal manera que no con su correcto funcionamiento quedará probada la viabilidad de este proyecto.

1.8 ESTRUCTURA DE ESTA MEMORIA

En el capítulo 2 se hace una introducción a sistema operativo MaRTE OS, comentando las principales características, donde descubriremos la posibilidad de portado a diferentes plataformas gracias a su implementación como un núcleo con estructura monolítica. En la estructura hay una descripción de los diferentes niveles estructurales, desde la aplicación hasta, nivel más alto, hasta el hardware de la plataforma, nivel más bajo. El capítulo finaliza con una descripción de los diferentes componentes de la interfaz abstracta con el hardware, que serán requisitos mínimos indispensables comunes a cualquier plataforma elegida.

En el capítulo 3 hablamos del microcontrolador MC68332, características principales, estructura, veremos una descripción y configuración de los principales registros tanto del módulo SIM, como del TPU.

En el capítulo 4, una vez conocido el sistema operativo que se va a portar y la plataforma que lo va a recibir, se plantea como será la nueva interfaz abstracta con el hardware con los diferentes paquetes que completarán su implementación.

En el capítulo 5, utilizando el software de control de periféricos en de la plataforma de aplicación y ejecución se desarrolla la aplicación de demostración que trabaje con el sistema operativo MaRTE OS.

En el capítulo 6, se trata el entorno de desarrollo, su justificación e instalación. Encontraremos también ejemplos de aplicación para probar el correcto funcionamiento del compilador cruzado y depurador cruzado, así como la configuración de los diferentes puertos de comunicación del PC y del SoccerBot.

Terminaremos en el capítulo 7 con conclusiones y trabajo futuro.

2 SISTEMA OPERATIVO MaRTE OS

MaRTE OS [MaR03] (Minimal Real-Time Operating System for Embedded Applications) es un sistema operativo de tiempo real conforme con el subconjunto mínimo definido en el estándar POSIX 13 desarrollado en el Departamento de Electrónica y Computadores de la Universidad de Cantabria.

Son muchos los sistemas operativos de tiempo real que proporcionan a las aplicaciones una interfaz POSIX, pero la mayoría son sistemas propietarios para los cuales no es posible disponer de su código fuente. Ciertamente existen sistemas operativos de tiempo real cuyo código se encuentre disponible, como RTEMS, RT-Linux, y entre otros, pero el problema es que en todos los casos su diseño interno no sigue el modelo de 'threads' POSIX, lo que complica la realización de modificaciones basadas en POSIX, además de constituir una fuente de ineficiencia a la hora de usar las interfaces estándar.

El lenguaje en el que está implementado de forma eficiente es el sistema operativo es Ada95.

MaRTE OS no consiste únicamente en un núcleo en el que se implementa la funcionalidad incluida en el subconjunto mínimo del POSIX (concurrencia, sincronización, temporización, etc), sino que además consta de una serie de aplicaciones y servicios que posibilitan la creación, carga y depuración de aplicaciones. Se dispone por tanto a crear un entorno de desarrollo cruzado basado en Linux y en los compiladores de GNU GCC y GNAT. [MAR02]

2.1 CARÁCTERÍSTICAS PRINCIPALES

En sistemas los sistemas empotrados de tiempo real el número de ‘threads’ y demás recursos que precisará la aplicación en el tiempo de ejecución es limitado y conocido de antemano. Durante la configuración del sistema es posible especificar el número máximo de recursos de cada tipo que serán prealojados durante la inicialización del sistema. De esta forma es posible ajustar el tamaño del ejecutable en función de las necesidades de cada aplicación. Esto se encontrará implementado dentro de ‘*configuracion_parameters.ads*, como veremos más adelante. Así, entre otros parámetros es posible ajustar:

El máximo número de ’threads’ que pueden existir al mismo tiempo
El número de niveles de prioridad diferentes
La longitud máxima de la cola de eventos temporales
El número máximo de temporizadores
El número máximo de señales pendientes

El número de datos específicos de cada 'thread'
El tamaño de los 'stacks' de los 'threads'
El número máximo de 'threads' con política de servidor esporádico
El tamaño del área de memoria dinámica

Permite ejecutar aplicaciones Ada y C.

El núcleo presenta las interfaces POSIX Ada y C. El entorno de desarrollo permite el desarrollo cruzado de aplicaciones Ada y C utilizando los compiladores de GNU GNAT y GCC.

El compilador GNAT debe implementar toda la semántica propia de las tareas Ada en su librería de tiempo de ejecución GNARL (GNU Ada Runtime Library). La librería de tiempo de ejecución es en su mayor parte independiente de la plataforma sobre la que se ejecuta. Sólo una pequeña parte, denominada GNULL (GNU Low-level Library), debe ser modificada para adaptarse a las diferentes plataformas de ejecución.

Portable a distintas arquitecturas

El núcleo presenta una interfaz abstracta con el hardware que permite independizar su funcionamiento del hardware específico sobre el que se ejecuta. Dicha interfaz proporciona al núcleo operaciones que permiten acceder al reloj y al temporizador del sistema, instalar manejadores de señal y realizar el cambio de contexto entre tareas. Éste es precisamente el objetivo principal del presente trabajo, demostrar que esta portabilidad es posible.

Núcleo monolítico

Las estructuras de datos del kernel deben ser protegidas ante la posibilidad de acceso simultáneo por parte de varias tareas ejecutando llamadas al sistema. Para lograr esa protección, MaRTE OS ha sido implementado como un núcleo con estructura monolítica, deshabilitándose las interrupciones durante los periodos de tiempo en los que se precisa acceder a estructuras globales.

Toma la forma de una librería para ser enlazada con la aplicación

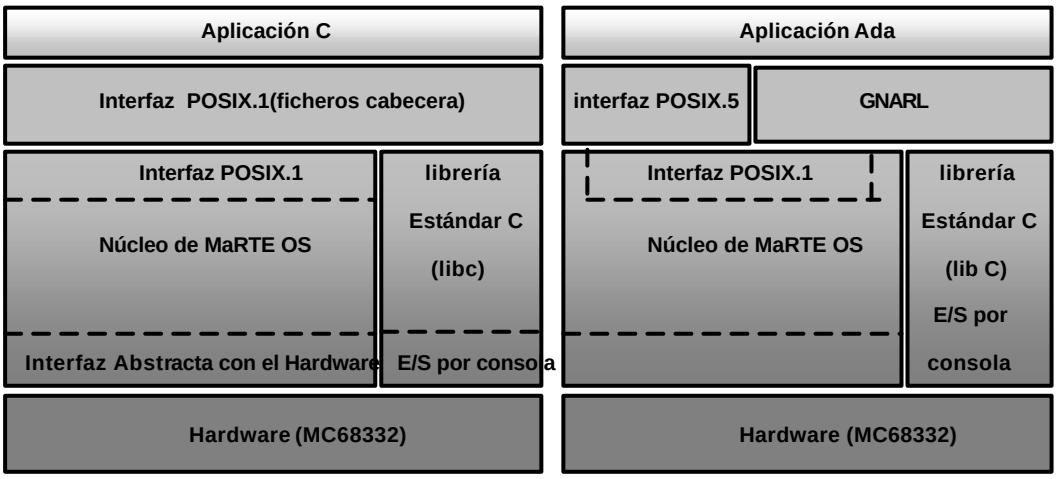
En el computador de desarrollo la aplicación de usuario es compilada y posteriormente enlazada con las librerías que forman el núcleo de MaRTE OS. El ejecutable así generado constituye un programa independiente listo para ser cargado en el computador empotrado y comenzar la ejecución de usuario de forma automática. Puesto que MaRTE OS está pensado para sistemas empotrados pequeños en los que lo normal es que no exista terminal para interfaz con el usuario, no tiene sentido el disponer de ningún tipo de intérprete de comandos, cuya inclusión aumentaría el tamaño del ejecutable, *según la*

tesis doctoral "planificación de tareas en sistemas operativos de tiempo real estricto para aplicaciones empotradas" de Mario Aldea Rivas [MAR02]

2.2 ARQUITECTURA

Visión general

El núcleo de MaRTE OS, para la implementación del núcleo en el PC, ha sido escrito principalmente utilizando el lenguaje de programación Ada 95, habiéndose utilizado también los lenguajes C y ensamblador. El uso de estos dos últimos lenguajes representa un porcentaje muy pequeño del tamaño total del núcleo, y corresponde principalmente a código tomado de otros sistemas operativos o librerías de código libre, tales como Linux, RT-Linux y el conjunto de librerías para desarrollo de sistemas operativos denominado OSKit. El lenguaje C, entre otros usos, es utilizado para el código encargado de la inicialización del PC, de la salida de caracteres por consola mediante la escritura directa en la memoria de vídeo. Así mismo durante la inicialización del sistema y para operaciones de muy bajo nivel como la rutina de cambio de contexto.



El núcleo incluye una interfaz abstracta de bajo nivel para acceder al hardware. En ella se define la visión del hardware que tienen las partes del núcleo que son independiente de la plataforma de ejecución. Esta interfaz constituye la única parte del núcleo que es dependiente del hardware, lo que facilita el portado de MaRTE OS a distintas plataformas. Además de la dependencia de la plataforma encapsulada en la interfaz abstracta con el hardware, también la librería estándar C depende del hardware sobre el que se ejecuta en lo referente a las operaciones de entrada/salida por consola. La E/S

por consola no está incluida dentro de la interfaz abstracta con el hardware porque no es común a todas las posibles arquitecturas (los sistemas empotrados normalmente no disponen de dispositivos de interfaz con el usuario). Lo normal es que esta funcionalidad sea aportada por el código encargado de manejar el dispositivo que se desea haga las veces de consola. [MAR02]

2.3 INTERFAZ ABSTRACTA CON EL HARDWARE

Esta interfaz proporciona al resto del núcleo de MaRTE OS una visión abstracta de la plataforma que consta de los siguientes elementos:

Temporizador: dispositivo que permite ser programado para provocar una interrupción cuando transcurra el intervalo de tiempo elegido, siendo el intervalo máximo programable finito y conocido por MaRTE OS. Cuando el intervalo deseado supera el máximo, el núcleo se encargará de programar varias veces consecutivas el temporizador hasta cubrir dicho intervalo. Tras la inicialización del sistema el temporizador no se encuentra programado.

Reloj: dispositivo que permite leer y modificar la hora actual. No se le exige que desde la inicialización del sistema marque la hora absoluta o tiempo transcurrido desde la "época", pero sí que debe tener capacidad para almacenar una fecha de este tipo, de forma que le pueda ser asignada a posteriori. Conviene que tenga una resolución igual o mayor que la del temporizador y no es necesario que exista como dispositivo físico. En el caso de que el sistema carezca de reloj, MaRTE OS realizará la programación periódica del temporizador incluso cuando no haya ningún evento temporal pendiente. De esta forma la capa de interfaz podrá llevar la cuenta del tiempo transcurrido actualizando una variable global tras cada programación. La hora actual en un instante dado se podrá obtener como la hora de la última programación más el estado de la cuenta del temporizador en ese instante.

Operaciones de conversión de tiempos: se proporcionan operaciones que permiten convertir tiempos medidos en cuentas del reloj a valores del tipo *duration* y viceversa. MaRTE OS utiliza internamente el formato usado por el reloj para representar el tiempo. Se prefiere utilizar las unidades del reloj en lugar de las del temporizador puesto que las operaciones matemáticas en las que interviene el valor actual del reloj son muy frecuentes con lo que se evitan innecesarias conversiones de tipo.

Interrupciones hardware: existen una o más fuentes de interrupciones hardware identificadas en la interfaz, debiendo existir como mínimo la correspondiente al temporizador. Se facilita una operación que permite asociar un procedimiento con una interrupción de forma que sea ejecutado inmediatamente cada vez que ésta se produzca. Por defecto a todas las

interrupciones les es asignado un manejador que indica al usuario la interrupción producida y finaliza la aplicación.

Dispositivo controlador de las interrupciones: permite la habilitación o deshabilitación de las interrupciones de forma individual o conjunta. En el caso de que no exista un dispositivo que centralice el control de todas las interrupciones las operaciones de habilitación o deshabilitación actuarán, siempre que sea posible, directamente sobre los dispositivos generadores de interrupciones. El dispositivo comienza con todas las interrupciones deshabilitadas.

Cuando las operaciones del núcleo son invocadas desde un manejador de interrupción deben retrasar la ejecución de los cambios de contexto hasta después de que haya sido completado correctamente el ciclo de reconocimiento de la interrupción, con la consiguiente rehabilitación del dispositivo controlador de interrupciones. La interfaz abstracta con el hardware proporciona una operación para saber cuándo se está ejecutando dentro de un manejador y otra para indicar cuándo se desea que sea llamada la función de planificación una vez rehabilitada la interrupción en el controlador.

Operación de cambio de contexto entre tareas: toma como argumentos las cimas de las pilas de las tareas entre las que se realizará el cambio. Tras su ejecución el estado de los registros del procesador de la tarea saliente queda almacenado en su pila, de forma que el valor correspondiente a su contador de programa corresponde a una dirección interna de la propia rutina de cambio de contexto. Por otro lado, el estado de la tarea entrante ha sido restaurado.

Registros del procesador: existen operaciones para lectura y escritura del registro de estado del procesador. También se proporcionan operaciones para habilitar o deshabilitar todas las interrupciones mediante la modificación de uno o más bits de ese registro. Estas operaciones son más rápidas que las proporcionadas por el dispositivo controlador de interrupciones. Las operaciones sobre los registros son utilizadas por el núcleo para crear secciones críticas: la entrada a una de estas secciones se realiza almacenando el registro de estado y deshabilitando las interrupciones, mientras que la salida consiste en restaurar el antiguo valor del registro de estado. [MAR02]

En el capítulo 4 se explicará en detalle la implementación del la interfaz abstracta del hardware para el microcontrolador MC68332.

3 EL MC68332

El MC68332 [MCPU], un microcontrolador de alta integración de 32 bits, combina alto rendimiento en la manipulación de datos con unos poderosos subsistemas periféricos. El microcontrolador está formado por módulos estándar y se controlan a través de un interfaz, el 'common intermodule bus' (IMB). La estandarización facilita un rápido desarrollo de dispositivos confeccionados para aplicaciones específicas.

El microcontrolador incorpora una CPU de 32 bits (CPU32), un 'System Integration Module' (SIM)[SIMRM], un 'Time Processor Unit' (TPU)[TPURM], un 'Queued Serial Module' (QSM), y un módulo de tamaño de 2 Kbytes de RAM estática con capacidad para emular la TPU (TPURAM) [TPURAM].

El microcontrolador puede incluso sintetizar una señal interna de reloj desde una referencia externa o si se usa una entrada de reloj externo directamente. La frecuencia de la señal de referencia es estándar y de valor 32.768 KHz. El hardware y el software permiten cambios en la frecuencia del reloj mientras se encuentra en uso. Porque la operación en el microcontrolador es completamente estática, los contenidos del registro y memoria no son afectados por los cambios en la frecuencia del reloj.

La alta densidad complementaria a una arquitectura metal-oxido semiconductor (HCMOS) provoca un bajo nivel de consumo de energía. Aún así, este puede ser minimizado mediante la parada del reloj del sistema. La CPU32 incluye una instrucción que implementa con eficiencia esta capacidad (LPSSTOP), 'low-power stop'.

3.1 CARACTERÍSTICAS PRINCIPALES

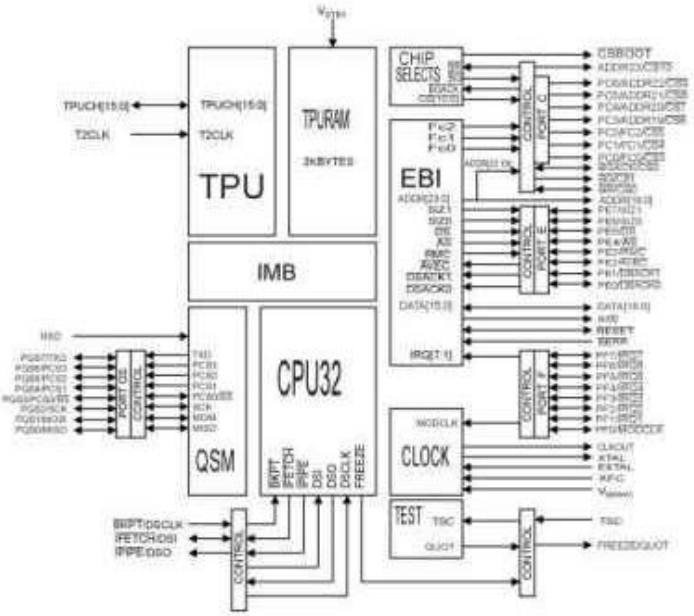
· Unidad Central de Proceso (CPU32)
- arquitectura de 32 bits
- implementación de memoria virtual
- manejador de excepciones para aplicaciones del controlador
- soporte para lenguajes de alto nivel
- modo depuración en segundo plano
- operación completamente estática

· Módulo de Integración de Sistema (SIM)
- soporte para bus externo
- salidas programables para el chip-select
- lógica de protección del sistema
- watchdog time, clock monitor y bus monitor
- 2 puertos de 8 bits de entrada/salida de función dual
- 1 puerto de 7 bits de salida con función dual

· Unidad de Procesado del Tiempo (TPU)
- capacidad de operación independiente de la CPU32
- 16 canales y pins independientes y programables
- algún canal puede desempeñar alguna función de tiempo
- 2 registros de cuenta de temporizador con prescalers programables
- nivel de prioridad de los canales seleccionables

· Módulo de Cola de Periféricos Serie (QSM)
- interfaz de comunicación serie mejorada
- interfaz para la cola de periféricos serie
- 1 puerto de 8 bit de función dual

En la figura: diagrama de bloques del MC68332.



3.2 CONFIGURACIÓN

A continuación se detallan los registros de configuración de los módulos SIM y TPU, así como los diferentes registros que se utilizarán para la adaptación de MaRTE OS al microcontrolador. Los registros de configuración, tanto de la SIM como de la TPU, se analizarán con los valores que mantienen cuando se hace un 'reset' en el SoccerBot, esto es, los valores predeterminados que establece RoBIOS.[TB03]

3.3 System Integration Module (SIM)

El módulo SIM lo forman cinco bloques funcionales que controlan el sistema de control de arranque, inicialización, configuración y bus externo. Hablaremos con detenimiento del bloque funcional de configuración, que será el que nos interese dentro de la SIM. A continuación hacemos una pequeña descripción de cada uno de los bloques.

El sistema de configuración y bloque de protección controla la configuración y el modo de operación.

El reloj del sistema genera señales de reloj utilizadas por el SIM, otros módulos IMB, y recursos externos. Por ello, un generador de interrupciones periódicas soporta el control de rutinas de tiempo de ejecución crítico.

El interfaz de bus externo maneja la transferencia de información entre los módulos IMB y el espacio de direcciones externo.

El bloque de selector de chip (chip-select) produce once señales 'chip-select' de propósito general y una 'boot ROM'. Ambos de encuentran asociados al registro base de direcciones (Base Address Register) y a los Registros de Opciones (options registers).

El bloque de test del sistema incorpora el hardware necesario para testear el microcontrolador.

El mapa de direcciones de registros de control del SIM ocupa 128 bytes. Los registros que dentro de este mapa no son utilizados devuelven ceros cuando son leídos. El bit SUPV en el registro SIMCR nos indicará en que modo nos encontramos, si en modo supervisor o usuario. Existen registros que tan sólo podrán modificarse en modo supervisor.[SIMRM]

3.3.1 REGISTROS DE CONFIGURACIÓN SIM

Este bloque funcional se encarga del control de la configuración de todo el microcontrolador. A continuación se describen los registros de configuración del Módulo SIM que nos interesan, el SIMCR [SIMRM], porque allí se activa/desactiva el modo supervisor y el Módulo de Mapeado, bits que nos interesa controlar, y el SYNCR [SIMRM], para la configuración del reloj del sistema.

SIMCR – REGISTRO DE CONFIGURACIÓN DE SIM

Controla la configuración del sistema, puede ser leído y escrito en cualquier momento.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXOFF	FRZSW	FRZBM	0	SLVEN	0	SHEN		SUPV	MM	0	0	IARB			
0	1	1	0	0	0	0	0	1	1	0	0	1	1	1	1

SUPV = 1, los registros con acceso controlado por este bit están restringidos. Solo pueden ser accedidos en modo supervisor.
MM =1, los registros internos se encuentran direccionados desde la posición de memoria $FFF000 hasta la $FFFFFF.
IARB = $F = '1111', es la más alta prioridad, permite la arbitración de interrupciones del mismo nivel que se provoquen a la vez. Este valor evita que interrupciones provocadas en la SIM durante la inicialización sean descartadas.

SYNCR – CLOCK SYNTHESIZER CONTROL REGISTER

Determina la frecuencia de operación del reloj del sistema así como su modo de funcionamiento.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
W	X	Y						EDIV	0	0	SLIMP	SLOCK	RSTEN	STSIM	STEXT
1	1	0	1	1	1	1	1	0	0	0	0	1	0	0	0

Cuando el 'sintetizador de reloj' es utilizado, los bits del [15:8] del SYNCR determinan la frecuencia de operación.
Fsystem = Freference[4 (Y+1)($2^{2W + X}$)]=32768[4(31+1)($2^{2 \cdot 1+1}$)]=33.554432 MHz
SLIM = 0, 'Limp Mode Status', el microcontrolador se encuentra operando en modo normal.

SLOCK = 1,'Synthesizer Lock', VCO se encuentra en la frecuencia deseada o el reloj del sistema es externo.
RSTEN = 0, 'Reset Enable', pérdida de memoria de las causas por las que el microcontrolador actúa en modo 'limp'.
STSIM = 0,'Stop Mode Sim Clock', el 'SIM clock' generado por la referencia externa se apaga cuando nos encontremos en modo de bajo consumo.
STEXT = 0, 'Stop Mode External Clock', el CLKOUT se pondrá a 'low' cuando ocurra una parada en el modo de bajo consumo.

3.4 Time Processor Unit (TPU)

La TPU [TPURM] es un inteligente, semiautónomo coprocesador diseñado para el control del tiempo. Opera simultáneamente con la CPU, la TPU procesa microinstrucciones, programas y procesos de eventos hardware en tiempo real, se encarga de la 'entrada' y 'salida', y accede a datos compartidos sin la intervención de la CPU. Consecuentemente, por cada prueba del temporizador hardware, el 'CETUR' de la CPU y el tiempo de servicio son minimizados o eliminados.

El módulo TPU incluye el microcódigo de un conjunto de funciones preprogramadas en ROM.

3.4.1 ESTRUCTURA TPU

El módulo TPU consiste en 2 contadores de 16 bits, dieciséis canales independientes temporizados, un programador de tareas, un 'microengine' y un 'host interface'.Dos contadores de 16 bits que son tiempos base referencia para todas el 'output compare' e 'input capture events'. Factores de escala que controlan estas dos bases de tiempos a través del registro TPUMCR. MaRTE OS utilizará el factor de escala TCR1, que siempre deriva del reloj del sistema.

La TPU tiene 16 canales independientes, cada uno conectado a un pin del microcontrolador. Los canales tienen idéntico hardware con excepción del canal 15. Cada canal consiste e un registro de evento y un pin de control lógico. El registro de evento contiene un registro de captura de 16 bits, un registro comparador, un comparador 'mayor o igual que' de 16 bits. La dirección de cada pin, si entrada o salida, es determinada por el microcódigo de la TPU.

El 'host interface' permite al CPU 'host' controlar la operación de la TPU. El host CPU debe inicializar la TPU escribiendo en el apropiado registro de interfaz del host para asignar una función, nivel de interrupción, y prioridad de cada canal.

El CPU host hace a un canal activo cuando a éste le asigna una de las prioridades: alta, media o baja. El programador de tareas determina el orden en

el que los canales son atendidos basados en número de canal y la prioridad asignada.

El 'microengine' esta compuesto por un control de almacenaje y una unidad de ejecución. En el 'control-store ROM' se guarda el microcódigo de cada función mascarada. [TPURM]

3.4.2 REGISTROS DE CONFIGURACIÓN TPU

A continuación se hace referencia a los registros de configuración de la TPU. [TPURM]

TPUMCR – TPU MODULE CONFIGURATION REGISTER

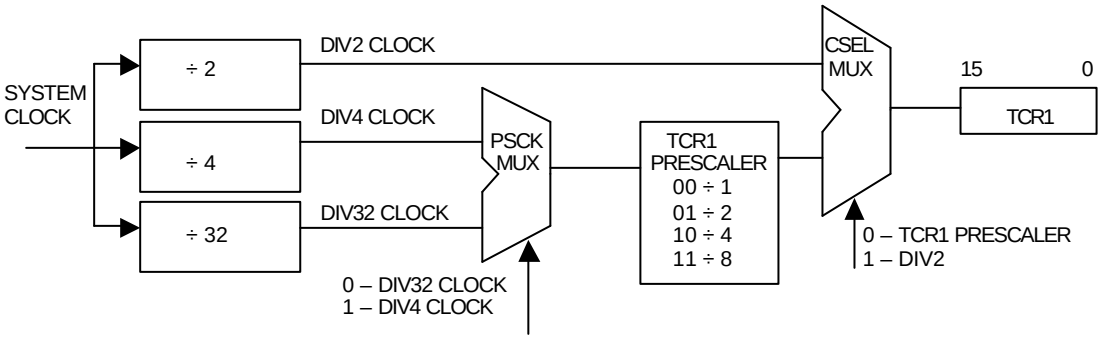
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
STOP	TCR1P		TCR2P		EMU	T2CG	STF	SUPV	PSCK	TPU2	T2CSL	IARB			
0	0	1	1	1	0	1	0	0	1	0	0	1	1	1	0

De este registro nos interesa lo siguiente
STOP = 0, 'Stop Bit' la TPU opera con normalidad.
TCR1P = '0 1', 'Timer Count Register 1 Prescaler Control', MaRTE OS utiliza este.
SUPV = 0,'Supervisor Data Space', los registros asignables son accesibles tanto como usuario como supervisor.
PSCK = 1, 'Prescaler Clock', system clock/4 es la entrada del prescaler TCR1

TPUMCR2 – TPU MODULE CONFIGURATION REGISTER 2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	DIV2	SOFTRST	ETBANK		FPSCK			T2CF	DTPU
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

DIV2 = 0, Divide by 2 Control, el TCR1 Incrementa la frecuencia determinada por los campos TCR1 y PSCK.



Así que el registro TCR1 se incrementará cada:

$(33554432\ Hz \div DIV4 \div TCR1P)^{-1} = (33554432\ Hz \div 4 \div 2)^{-1} = 238.41\ ns$

3.4.3 INTERRUPCIONES TPU

La CPU compara el nivel de cada petición de interrupción que recibe con el valor de la máscara de interrupción. Las peticiones de interrupción de niveles mayores que el de la máscara son aceptadas; Los niveles menores o iguales al valor de la máscara son ignorados. La excepción a la norma es el nivel 7 de petición de interrupción no mascarable, el cual es aceptado y dado servicio incluso si las máscara de interrupción de la CPU es 7.

Todos los periféricos que se encuentren implementados en el chip tienen asignados un nivel de petición de interrupción de 0 a 7. Siete es la más alta prioridad, uno es la más baja, y 0 deshabilita cualquier petición de interrupción.

El nivel de petición de interrupción de canal se encuentra en el registro de configuración de interrupciones de la TPU.

Para estar seguros de donde debe buscar la CPU la rutina de servicio de interrupción para cada tipo de interrupción está el CIBV (Channel Interrupt Base Vector) que determina donde encontrar las 16 rutinas de interrupción que puede servir la TPU, una para cada uno de los 16 canales de la TPU. CIBV es el *nibble*, cada una de las mitades de una palabra de 8 bits, más significativo del número del vector de interrupción en tamaño byte; el nibble menos significativo es el propio numero de canal. CIBV debería contener un número de vector que no esté reservado.

La CPU puede enmascarar interrupciones para los canales de forma individual en el CIER (Channel Interrupt Enable Register). Un cero en un bit de este registro enmascara la interrupción de un canal específico, un uno habilita la interrupción en ese canal. Si la interrupción del canal se encuentra enmascarada, podemos leer el registro CISR (Channel Interrupt Status Register), si el canal tiene pendiente la petición de interrupción.

TICR – TPU INTERRUPT CONFIGURATION REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	CIRL			CIBV							
0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0

CIRL =’100’, ‘Channel Interrupt Request Level, estos tres bits especifican el nivel de petición de interrupción para todos los canales de la TPU. El nivel siete en este campo indicaría interrupción no mascarable; nivel cero indicaría que todos los canales de interrupción están desactivados.

CIBV = '0101', 'Channel Interrupt Base Vector, la TPU tiene asignado únicamente dieciséis vectores de interrupción, un vector para cada canal. Este campo especifica el 'nibble' más significativo de los números de los vectores que son asignados como vectores de interrupción. El 'nibble' menos significativo viene dado por el número de canal.

Para saber en que posición de memoria va a buscar la CPU la rutina manejadora de interrupciones se debe hacer el siguiente cálculo:

Dir. Vector interrupción = 4 x nº vector interrupción + dirección del VBR

VBR – Vector Base Register, se encuentra en la posición de memoria $0, es un registro de la CPU32 que contiene los vectores de excepción. Para más información *Reference Manual MC68332*.
Por ejemplo, si CIBV = $5 y canal 1, la dirección de memoria donde se debería encontrar la rutina de interrupción de este canal será $51 x $ = $144

CIER – CHANNEL INTERRUPT ENABLE REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH15	CH14	CH13	CH12	CH11	CH10	CH9	CH8	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

En la inicialización todos los canales deben tener deshabilitadas las interrupciones, requisito de MaRTE OS. En este registro se habilitan las interrupciones de la TPU individualmente.

0 – interrupción deshabilitada
1 – interrupción habilitada

CISR – CHANNEL INTERRUPT STATUS REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH15	CH14	CH13	CH12	CH11	CH10	CH9	CH8	CH7	CH6	CH5	CH4	CH3	CH2	CH1	CH0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Este registro indica algún canal de la TPU ha provocado interrupción.
No debe haber interrupciones pendientes durante la inicialización, requisito de MaRTE OS.

0 – interrupción no se ha producido
1 – se ha producido interrupción

CFSR# – CHANNEL FUNCTION SELECT REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CHANNEL 15,11,7,3				CHANNEL 14,10,6,2				CHANNEL 13,9,5,1				CHANNEL 12,8,4,0			
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

El '#' corresponde a 0, 1,2 ó 3. Dependiendo del canal que se quiera utilizar.
El valor que se puede asignar a cada canal corresponde es un valor de 4bits que corresponderá a una de las funciones predefinidas en la TPU ROM.
Ver Pág. 56 de *TPU Reference Manual.*

HSQR# – HOST SEQUENCE REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH 15,7		CH14,6		CH13,5		CH12,4		CH11,3		CH10,2		CH9,1		CH8,0	
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

En el HSQR# el '#' corresponde a 0 ó 1 dependiendo del canal que se quiera utilizar.
Este registro ayuda a especificar la operación de la función de tiempo seleccionada en el canal deseado. El significado de los 'host sequence bits' dependerá de la función de tiempo especificada.
Ver tabla en Pág. *56 de TPU Reference Manual.* .[TPURM]

HSRR# – HOST SERVICE REQUEST REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH 15,7		CH14,6		CH13,5		CH12,4		CH11,3		CH10,2		CH9,1		CH8,0	
#	#	#	#	#	#	#	#	#	#	#	#	#	#	#	#

En HSRR# el '#' corresponde a 0 ó 1 dependiendo del canal que se quiera utilizar.
Este registro selecciona el tipo de 'host service request' de la función de tiempo seleccionada en el canal deseado. El significado de los 'host service request bits' dependerá de la función de tiempo especificada.
Ver tabla en Pág. *56 de TPU Reference Manual.*[TPURM]

CPR# – CHANNEL PRIORITY REGISTER

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CH 15,7		CH14,6		CH13,5		CH12,4		CH11,3		CH10,2		CH9,1		CH8,0	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

El '#' corresponde a 0 ó 1 dependiendo del canal que se quiera utilizar.

Se asignan una de las los tres prioridades a los canales o se deshabilita el canal

00 – canal deshabilitado
01 – prioridad baja
10 – prioridad media
11 – prioridad alta

CHANNEL PARAMETER RAM

El 'channel parameter RAM' está organizado como cien palabras de 16 bits de RAM. Los canales de 0 – 13 tienen hasta 6 parámetros de configuración; los canales 14 y 15 tienen 8 parámetros de configuración.

El 'channel parameter RAM' constituye un espacio de trabajo compartido para la comunicación entre la CPU y la TPU y de almacén de datos para la TPU. Cada función preprogramada utiliza estos parámetros de forma diferente. Ver Pág. 55 *TPU Reference Manual* para más información. .[TPURM]

Los demás registros que faltan de analizar y que se utilicen para la implementación de MaRTE OS son específicos del canal de TPU en el que se encuentren y la función interna de la TPU a la que se haga referencia.

4 MIGRACIÓN DE MaRTE OS A MOTOROLA MC68332

4.1 DESARROLLO DE MaRTE OS EN MC68332

Como ya se comentó con anterioridad MaRTE OS es un sistema operativo mínimo de tiempo real para sistemas empotrados. Gracias a la estructura y a la interfaz abstracta con el hardware que presenta el núcleo se demostrará la facilidad de portabilidad de una plataforma de desarrollo a otra, en este caso se trata de portar a una CPU32 [MCPU] montada sobre el microcontrolador MC68332 de Motorola.

La interfaz abstracta del hardware proporciona al resto del núcleo de MaRTE OS una visión abstracta de la plataforma, esto es un conjunto de elementos con unos requisitos mínimos para que MaRTE OS se adapte correctamente a la plataforma elegida.

Como ya se comentó, la interfaz está formada por funciones Ada que siguen los prototipos propuestos en el estándar POSIX.1, para ello era necesario que los tipos utilizados por las aplicaciones coincidan con los utilizados por el núcleo, estos tipos se encuentran en:

/MaRTE/kernel/basic_integer_types.ad[s,b]
/MaRTE/kernel/basic_console_io.ad[s,b]

Estos tipos serán utilizados en la implementación de MaRTE OS en el MC68332.

4.2 ESTRUCTURA E IMPLEMENTACIÓN DE MaRTE OS EN MC68332

De acuerdo con lo expuesto anteriormente en el paquete de instalación de MaRTE OS existirá un directorio específico para cada una de las plataformas para las que el sistema operativo se encuentre implementado.

Dentro del directorio que se haya creado, en nuestro caso: *~/MaRTE/MC68332/* nos encontraremos los siguientes paquetes:

Tpu_reg.ads

Contiene las direcciones de memoria que corresponden a todos los registros de la TPU.

Interrupt_tables.ads&Interrupt_tables.adb

Encontraremos todo lo relacionado con las interrupciones, fuentes de interrupción, instalación de manejadores de interrupción, tratamiento de interrupciones y habilitación/deshabilitación de interrupciones del módulo TPU

Tpu.ads&Tpu.adb

Generación del reloj del sistema, programación del temporizador

`Proccessor_registers.ads&Proccessor_registers.adb`

Habilitación/deshabilitación de todas las interrupciones, comprobación de interrupciones. Operaciones en tamaño bit, cambio de contexto de tareas

`Hardware_interface.ads&Hardware_interface.adb`

Este paquete se hará toda comunicación entre la interfaz del hardware y el resto del sistema operativo, por lo que deberemos mantener los nombres de tipos, subtipos, constantes, procedimientos y funciones. Desde aquí se llamarán al resto de los paquetes anteriormente nombrados.

4.2.1 INTERRUPT_TABLES

Se define el tipo *interrupt*

```
Type interrupt is new integer range 0... 255;
```

Este tipo corresponde a la longitud de la tabla de los vectores de interrupción del microcontrolador. El usuario puede definir sus propios vectores de interrupción a partir del vector 64 hasta el 255. En esta primera implementación las interrupciones serán provocadas por el módulo de TPU, y más en concreto por cualquiera de sus 16 canales que se encuentren mapeados a partir del vector 80, serán los únicos para los que el usuario podrá definir sus manejadores.

Procedure install_interrupt_handler

```
Procedure install_interrupt_handler
     ( num_int : in interrupt
     Handler : in interrup_handler_type) is
Begin
     Install_handler_ada(num_int);
     Vector_interrupt_table(num_int):=handler;
End install_interrupt_handler;
```

`Install_handler_ada (num_int)`, es la función que se encarga de crear un vector de interrupción correspondiente al número de interrupción

pasado como parámetro. El código de la función en ensamblador es el que sigue:

```
install_handler_asm:
     1    MOVEM.L    A1/D0 , -(SP)
     2    MOVE.L     12(SP), D0
     3    ASL.L      #2 , D0
     4    MOVE.L     D0, A1
     5    MOVE.L     #wrapper,(A1)
     6    MOVEM.L    (SP)+, A1/D0
     7    RTS
wrapper:
     8    MOVEM.L    SP,-(SP)
     9    JSR        reset_tpu_asm
     10   MOVEM.L    (SP)+,SP
     11   RTE
```

(1) todos los registros que vamos a utilizar en la subrutina se guardan en 'la pila'

(2) el parámetro que pasa la función se encuentra ahora 12 'words' por encima de donde señala el SP actualmente, y en esta posición se guarda el contenido de los siguientes 32 bit (tamaño long_word (.L)) en el registro D0, con esto, ya tenemos el parámetro en D0.

(3) la dirección del vector de interrupción se calcula como:
4*nºinterrupción+vbr
vbr se encuentra en 0x0. Luego es correcto calcularlo como un desplazamiento lógico hacia la izquierda de 2 bits. Ahora D0 contendrá la dirección del vector d interrupción.

(4) como lo que se encuentra en D0 es realmente una dirección de memoria y vamos a trabajar con ella, es más apropiado guardar lo que hay en D0 en un registro de direcciones, A1.

(5) para el tratamiento de las interrupciones utilizaremos una envoltura o 'wrapper' que ejecutará cuando se provoque la interrupción

(6) restauramos los valores de los registros, tal y como estaban antes de llamar a la función

(7) termina la subrutina

(8) cuando se ejecute el 'wrapper', se habrá provocado una excepción, en nuestro caso una interrupción programada de la TPU, la cual tendremos que manejar.

(9) se ha provocado una interrupción, para tratarla se saltará a una segunda envoltura que nos desvelará que interrupción se ha provocado y que rutina manejadora va a ejecutar.

(10) como queremos que continúe el programa donde lo dejamos antes de la excepción, recuperamos ese SP

(11) retorno de rutina de excepción

```
Vector_interrupt_table(num_int):=handler;
```

Se trata de un array de manejadores de interrupción seguirá la misma distribución que sigue el registro 'vector_base_register' [REF], por lo tanto las posiciones correspondientes a la TPU serán de la 80 a la 95. En principio, los 16 canales de la TPU pueden provocar interrupciones, y todos ellos puede haber instaladas rutinas de interrupción, aunque para utilizar MaRTE OS sólo utilizaremos un canal, el encargado del 'timer'.

Procedure reset_tpu

```
Procedure reset_tpu (state: in trap_state_ac) is
     Num_int : interrupt;
     Reg : unsigned_16;
begin
     Processor_registers.interrupts_disabled;
     Reg:=cisr and 16#0080#;
     If reg = 0 then
          Put ("error no TPU interrupt");
          New_line;
     End if;
     For i in 0..15 loop
          If reg = (reg or 16#0001#) then
               Num_int := interrupt(80+i);
               Processor_registers.bit_reset
                    (unsigned_32(cisr),
                    unsigned_32(i));
               exit;
          else
               reg := processor_registers.shift_right
                    (reg, 1)
          end if;
     end loop;
     base_irq_nest := base_irq_nest + 1;
     vector_interrupt_table
          (interrupt(num_int)).all(state);
     base_irq_nest := base_irq_nest – 1;
     if base_irq_nest = 0 then
          base_irq_nest = 16#80#;
          do_scheduling;
     end if;
end reset_tpu;
```

Este procedimiento es llamado cuando se provoca una interrupción, se deshabilitan las interrupciones (requerido por MaRTE OS)comprueba que la interrupción es provocada por la TPU, localiza el canal fuente de interrupción, ejecuta la rutina de interrupción correspondiente y limpia el 'flag'de interrupción

correspondiente. En caso de que la fuente de interrupción no sea conocida, se mostrará por pantalla un mensaje de error. Una vez que se termine de ejecutar este procedimiento se devolverá el control a la rutina de excepción hasta que ésta finalice.

Procedure initialize

```
Procedure initialize is
Begin
     Install_interrupt_handler (interrupt(87) ,
          default_interrupt_handler'access);
End initialize;
```

Por defecto a todas las interrupciones les es asignado un manejador que indica al usuario la interrupción producida y finaliza la operación. Así es que, este procedimiento se llamará en la inicialización que se haga en el *hardware_interface.adb* asignando el manejador por defecto al temporizador. Se ha asignado al temporizador el canal 7 de la TPU.

4.2.2 TPU

Este paquete corresponde principalmente a la utilidad que se ha dado al módulo TPU, consta de dos procedimientos y una función. Los procedimientos van a utilizar 2 funciones específicas del módulo TPU, para generar un pulso cuadrado de un periodo constante que hará las veces de nuestro reloj software, y otra para la programación del temporizador. La función nos servirá para saber el tiempo en cualquier momento con la resolución que tenga el 'clock'. Existe un tercer procedimiento escrito en lenguaje ensamblador que realizará la inicialización del temporizador.

Además MaRTE OS pide las siguientes constantes

```
TPU_HZ: constant := 4201680;
```

Esta constante nos da el número 'ticks' de TCR1 por segundo, esto es, de la base de tiempos.

```
TPU_NS : constant := 1000000000/TPU_HZ;
```

Esta constante nos dará los nanosec/ticks.

```
Safe_longest_timer_interval : constant :=31447;
```

Esta constante es muy importante porque será el tiempo máximo de programación del temporizador hardware 7.5 ms, es decir, MaRTE OS deberá programar el temporizador aproximadamente 133 veces por segundo, siempre

que no esté pendiente ningún evento temporal, esta reprogramación periódica se realiza con el fin de llevar correctamente la variable que lleva el tiempo, tal y como ya se había explicado anteriormente.

Procedure clock;

En esta función se utiliza el factor de escala TCR1 en el temporizador del módulo TPU, que tiene un periodo de 238.5 nanosec/ticks, pero este timer no tiene accesible el registro donde realiza el conteo de los 'ticks', esta razón nos lleva a producir una señal cuadrada en el canal 8 de la TPU con un periodo de 8 ticks de TCR1, esto es, el 'clock' quedará con una resolución aproximada de 2 microsegundos (8x238.5 ns˜ 2 µs).

```
Procedure clock is
Begin
     CPRO := CPRO AND 16#FFFC#;
     CFSR1 :=CFSR1 AND 16#FFF0#;
     CFSR1 :=CFSR1 OR 16#0009#;
     CH_8_PAR_0:= 16#0091#;
     CH_8_PAR_2:= 16#0004#;
     CH_8_PAR_3 := 16#0008#;
     HSRR0 := HSRR0 AND 16#FFFC#;
     HSRR0 := HSRR0 OR 16#0002#;
     CPR0 := CPR0 OR 16#0003#;
End clock;
```

La TPU puede generar una onda con ancho de pulso modulado (PWM) Para definir la PWM la CPU proporciona un parámetro que indica el periodo y otro parámetro que indica el 'tiempo en alto' de la onda. [REF] Para más información ver TPURM Pág...73.

Procedure initialize_timer

Se va a preparar al canal 7 de la TPU para que sea nuestro temporizador programable, para ello debemos configurar la función OC (Output Compare) la cual permitirá programar una interrupción con un periodo máximo de 7.5 ms. La base de tiempos seguirá siendo TCR1.

```
Initialize_timer:    ANDI.W    #0x0FFF , CFSR2
                     ORI.W     #0xE000 , CFSR2
                     MOVE.W    #0x0089 , CH7PAR0
                     MOVE.W    #0x00EC , CH7PAR2
                     ANDI.W    #0x3FFF , HSQR1
                     ORI.W     #0x0000 , HSQR1
                     ORI.W     #0xC000 , CPR1
                     RTS
```

La configuración de éste canal será completa cuando llamemos al procedimiento `program_tpu_timer`. Hacer la configuración por separado permitirá a MaRTE OS acceder a cambiar la cuenta del temporizador sin tener que configurar el canal en cada ocasión.

Esta función será llamada tan sólo en la inicialización del `hardware_interface.adb`

Para más información de la función ir al [REF] TPURM Pág... 61

Procedure program_tpu_timer

```
Procedure program_tpu_timer (count :in unsigned_16) is
Begin
     CIER:=CIER AND 16#FF7F#;
     CISR:=CISR AND 16#FF7F#;
     CH7_PAR_1:= count;
     CIER:=CIER OR 16#0080#;
     HSRR1 := HSRR1 OR 16#4000#;
     CPR1 := CPR1 OR 16#C000#;
End program_tpu_timer;
```

Este procedimiento completa la configuración del canal 7 de la TPU, para reprogramar habrá que ejecutar este procedimiento con un valor de cuenta adecuado, es decir inferior o igual al máximo periodo de programación dado por la constante `safe_longest_timer_interval`. MaRTE OS ya tiene en cuenta el tiempo que se tarda en la reprogramación como se explicará en el paquete `harware_inteface`.

Function read_counter

Debe de tratarse de una función muy rápida ya que será llamada muy frecuentemente (cada vez que se desee conocer la hora actual). Para ello lee un parámetro de la TPURAM (el cual incrementa cada vez que se produzca un periodo completo del reloj), como este parámetro es un registro de 16 bits, podemos llegar a contar 2^16 x 238.5e-9 s ˜ 15.63 ms, con una resolución de 8 x 238.5e-9 s = 1.908 µs, como el mayor periodo que MaRTE OS necesita contar es de 7.5 ms la elección es correcta.

```
Procedure read_counter return unsigned_16 is
Begin
     Return CH_8_PAR_4;
End read_counter;
```

Realmente lo que ocurre en este parámetro es que cada vez que se cumple un periodo completo del clock, el cuarto parámetro (registro de 16 bits) de esta función actualiza su valor, en este registro se va a actualizar el valor de TCR1, luego se trata de un registro de acumula 'ticks' de TCR1 pero actualiza

este registro cada periodo, por lo tanto tendremos un reloj con una resolución de aprox. 2 µs, como ya se había comentado.

4.2.3 PROCESSOR_REGISTERS

Procedure interrupts_enabled

Este procedimiento habilitará todas las interrupciones en la CPU, por ello debemos actuar sobre el 'registro de estado' (SR), se trata de un registro de la CPU, así que no está mapeado en memoria, lo cual se hará mediante instrucciones en ensamblador. [MCPU] Para más información ir al CPU32RM Pág...25

```
Procedure interrupts_enabled is
Begin
     Asm ("andi.w #0xf0ff,  %%SR", no_output_operands,
     no_input_operands, "", True);
End interrupts_enabled;
```

La mascará de interrupciones lo que hace es permitir las interrupciones del mismo nivel y siguientes. Con esta instrucción permitimos que se provoquen interrupciones de todos los niveles, ya que la mascará la situamos en el nivel 0.

Procedure interrupts_disabled

Va a ser la función contraria a la anterior, debemos deshabilitar todas las interrupciones, luego siguiendo la misma lógica deberemos habilitar la máscara al nivel 7.

```
Procedure interrupts_disabled is
Begin
     Asm ("ori.w #0xf7ff,  %%SR", no_output_operands,
     no_input_operands,"", True);
End interrupts_disabled;
```

Tanto la instrucción 'andi' como 'ori' trabajarán sobre cada bit de la palabra implicada, como la máscara pasará siempre de nivel '0' a nivel '7' no habrá ningún problema en la escritura de los bits ya que con 'ori' pasaremos de 'X ori 7 = 7' esto es 'XXX' ? '111'; con 'ori' pasaremos de 'X andi 0 = 0' esto es 'XXX' ? '000';

Function are_interrupts_enabled

```
Function are_interrupts_enabled return boolean is
     Flags: unsigned_16;
Begin
```

```
        Asm ("move.w   %%SR, %0",
             unsigned_16'asm_output ("=d", flags),
             no_input_operands, "", true);
        Return (flags and 16#0700#) <= 16#0500#;
End are_interrupts_enabled;
```

Esta función servirá para preguntar si tenemos activadas las interrupciones de la CPU, la TPU en su configuración original produce interrupciones de nivel 5, la función devolverá el valor TRUE siempre que la máscara de interrupciones sea de un nivel inferior o igual a 5.

Function save_flags & function restore_flags

Estas funciones sirven para salvar y restablecer los 16 bits del SR. La variable `flags` será del tipo `integer` para mantener una coherencia con las llamadas que hace MaRTE OS a esta función.

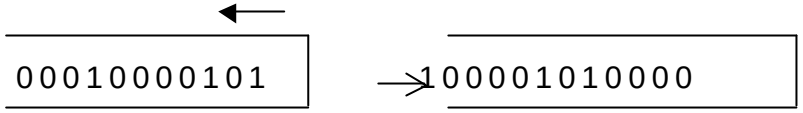
```
Function save_flags return integer is
     Flags:unsigned_16;
Begin
     Asm ("move.w   %%SR, %0", unsigned_16'asm_output
     ("=d", flags), no_input_operands, "", true);
     Return integer(flags);
End save_flags;

procedure restore_flags (flags : in integer) is
begin
     Asm ("move.w   %0,%%SR", no_output_operands,
     integer'asm_output ("=d", flags), "", true);
End restore_flags;
```

Function get_stack_pointer_register

Esta función devuelve los 32 bits correspondientes al 'stack pointer'. Debido a que este registro también es propio de la CPU32, se requiere una instrucción en ensamblador para acceder a su contenido.

```
Function get_stack_pointer register return
          unsigned_32 is
     SP_register: unsigned_32;
Begin
     Asm ("move.l %%SP, %0",
     unsigned_32'asm_output("=d", SP_register) ,
     no_input_operands,"", true);
     Return SP_register;
End get_stack_pointer_register;
```

A continuación se este paquete se desarrollan funciones y procedimientos para realizar 'operaciones de bit'. Estas funciones serán utilizadas tanto por MaRTE OS como en el paquete `interrupt_tables`.
La función `shift_left & shift_right` se encuentran en: *~/MaRTE/kernel/basic_integer_types.adb*
Shift_left: `shift_left (133,4)`, *desplazamiento lógico hacia la izquierda de 4 bits del número 133.*



Procedure bit_set & bit_reset

```
     Procedure bit_set (bit_field: in out unsigned_32; bit:
in unsigned_32) is
          Aux : unsigned_32;
     Begin
          Aux : = shift_left (1, natural (bit));
          Bit_field := bit_field or Aux;
     End bit_set;
```

Ejemplo: `bit_set (133,4)`

Bit_field : '0..010000101' ; aux :'00..010000' ; Bit_field or aux : '0..010010101'

```
     Procedure bit_reset (bit_field: in out unsigned_32;
                          bit: in unsigned_32) is
          Aux : unsigned_32;
     Begin
          Aux : = not shift_left (1, natural (bit));
          Bit_field := bit_field and Aux;
     End bit_set;
```

Ejemplo: `bit_reset (133,7)`

Bit_field : '0..010000101' ; aux :'11..101111111' ; Bit_field and aux : '0..000000101'

En el procedimiento bit_set activará el bit número, 'bit,' en un campo de bits de tamaño 32 bits, 'bit_field', análogamente el procedimiento bit_reset deshabilitará el bit número,'bit', del campo de bits,'bit_field'.
La salida será el propio campo de bits modificado.

Procedure bit_scan_reverse & bit_scan_forward

MaRTE OS asegura que nunca llamará a estos dos procedimientos si el campo de bits en el que hay que realizar la búsqueda es nulo.

Este procedimiento va a trabajar sobre un campo de bits, su salida nos indicará la posición del bit más significativo de ese campo de bits.

```
Procedure bit_scan_reverse (bit_field: in unsigned_32;
                            bit: out unsigned_32) is
     Field : unsigned_32;
Begin
     Field := bit_field;
     For i in 0..31 loop
          If (field and 16#8000_0000#) /= then
               Bit := unsigned_32(31-i);
               Exit;
          Else
               Field := shift_left (field,1);
          End if;
     End loop;
End bit_scan_reverse;
```

Para la búsqueda del bit más significativo se va a hacer un bucle que compruebe el bit 31, si se encuentra a '1' quiere decir que se ha encontrado el bit más significativo y que se encontrará en la posición (31 – número de vueltas del bucle), si lo encuentra a '0', desplazaremos lógicamente el campo de bits una posición hacia la izquierda.

Ejemplo:`bit_scan_reverse(133,bit_num)`
Field:'0..010000101'; i : 24 ; bit_num= 7;

Análogamente se encuentra la función bit_scan_forward

```
Procedure bit_scan_forward (bit_field: in unsigned_32;
                            bit: out unsigned_32) is
     Field : unsigned_32;
Begin
     Field := bit_field;
     For i in 0..31 loop
          If (field and 16#0000_0001#) /= then
               Bit := unsigned_32(i);
               Exit;
          Else
               Field := shift_right (field,1);
          End if;
     End loop;
End bit_scan_forward;
```

En este procedimiento se hará la búsqueda del bit menos significativo, es decir desplazamos lógicamente el campo de bits hasta que el bit que se

encuentra en la posición '0' corresponde a un '1'. El número de desplazamientos que se deban realizar corresponderán a la posición en la que se encuentra el bit menos significativo.

Ejemplo: `bit_scan_forward (133,bit_num)`
`Field:'0..010000101';  i : 0 ; bit_num= 0;`

Los procedimientos que se muestran a continuación tienen que ver con el cambio de contexto de tareas el contexto de una tarea se almacena en un registro (el cual es característico de la plataforma que vaya a soportar a MaRTE OS):

```
Type preempted_task_stack is record
     Return_address       : system.address;
     D0                : unsigned_32;
     D1                : unsigned_32;
     D2                : unsigned_32;
     D3                : unsigned_32;
     D4                : unsigned_32;
     D5                : unsigned_32;
     D6                : unsigned_32;
     D7                : unsigned_32;
     A0                : unsigned_32;
     A1                : unsigned_32;
     A2                : unsigned_32;
     A3                : unsigned_32;
     A4                : unsigned_32;
     A5                : unsigned_32;
     Context_switch_ret   : system.address;
End record;
```

Donde **return**_address & context_switch_ret son campos del registro que serán comunes a cualquiera de las plataformas elegidas, ya que contienen las direcciones de memoria correspondientes a la dirección de retorno desde una tarea dentro de la subrutina del cambio de contexto y la dirección de retorno después de realizar el cambio de contexto entre tareas. Los registros D0...D7 y A0...A5 son propios del microcontrolador MC68332. Los registros A6 y A7 corresponden al FP (frame_pointer) y SP (puntero de la pila), que se 'salvan' automáticamente cuando se utiliza el 'la pila', cuando hay un salto a una subrutina en la pila lo primero que se hace es guardar la dirección de retorno y el 'frame pointer'.

Se define el siguiente tipo:

```
Type preempted_task_stack_ac is access
      preempted_task_stack;
```

Para usar en las siguientes funciones:

Function to_preempted_task_stack_ac

```
Function to_preempted_task_stack_ac is
     new ada.unchecked_conversión (unsigned_32,
preempted_task_stack_ac);
```

procedure change_return_address_of_preempted_task

Este procedimiento es requerido por MaRTE OS se ha dejado la misma implementación que había para la plataforma x86.

```
procedure change_return_address_of_preempted_task(
          top_of_stack : unsigned_32;
          new_ret_sddress : in system.address ) is

     stack : preemped_task_stack_ac :=
           to_preempted_task_stack_ac(top_of_stack);
begin
     stack.return_address := new_ret_address;

end change_return_address_of_preempted_task;
```

procedure context_switch

Este procedimiento va a realizar el cambio de contexto entre tareas, la función que va a hacer el cambio de contexto, propiamente dicho, será la `context_switching` implementada en lenguaje ensamblador, ya que es más adecuado para realizar lecturas y escrituras en los registros del microcontrolador. Se exportarán dos variables que serán la cima de las pilas de la tarea nueva y la vieja. Aunque se podrían haber pasado estas variables por la pila, y acceder a ellas (mirando +12 y +16 en el SP), la función quedará más clara y estructurada de esta otra forma.

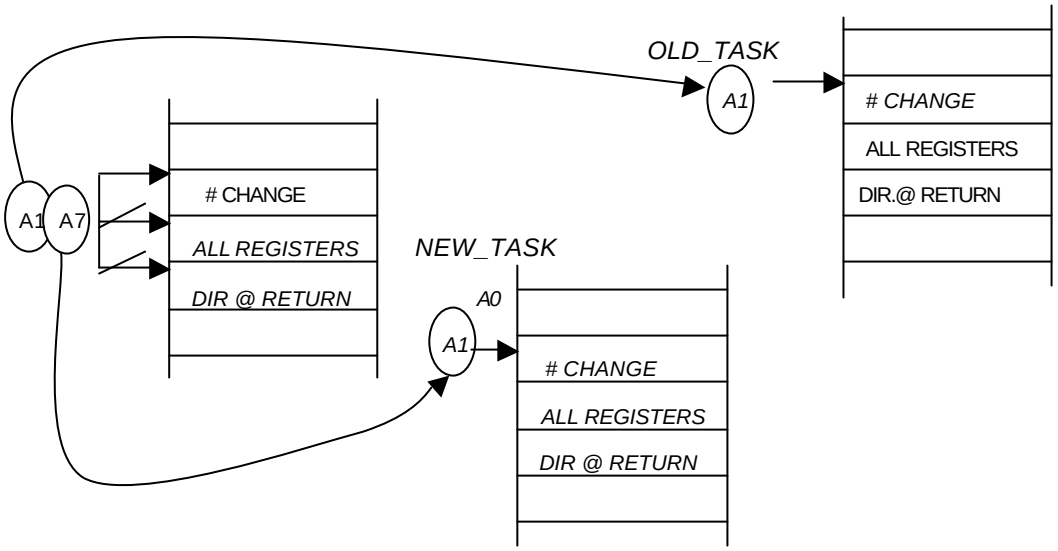
```
procedure context_switch (old_task :in system.address;
                     new_task : in system.address) is
begin
     oldtask := old_task;
     newtask := new_task;
     context_switching;

end context_switch;
```

Los parámetros de entrada de la función son las cimas de las pilas de las tareas que van a realizar el cambio de contexto. El cambio de contexto va a consistir en salvar los registros de datos y de direcciones correspondientes a

cada una de las tareas, guardándoles en la pila de forma que se pueda, a partir de la cima de su pila (dirección de la memoria correspondiente a la cima de la pila), restaurar el contexto adecuado.
La función de cambio de contexto en ensamblador queda como la que sigue:

```
context_switching:
     (1)        MOVEM.L    D0-D7/A0-A6, -(SP)
     (2)        MOVE.L     old_task, A1
     (3)        MOVE.L     new_task, A0
     (4)        MOVE.L     #change,-(SP)
     (5)        MOVE.L     SP, (A1)
     (6)        MOVE.L     (A0), SP
     (7)        RTS
change:
     (8)        MOVEM.L    (SP)+,D0-D7/A0-A6
     (9)        RTS
```

(1) guardamos en la pila todos los registros de la 'tarea vieja'
(2)(3) guardamos los parámetros de entrada correspondientes a la dirección de memoria de la cima de la pila de la 'tarea vieja' y 'tarea nueva' en los registros de direcciones A1 y A0.
(4) preparamos la pila para que coincida con la estructura de la pila de la tarea como hemos previsto.
(5) La tarea en la que nos encontramos pasará a ser la 'tarea vieja'.
(6) El SP señalará a la 'tarea nueva'.
(7) Devolvemos el control al principal como nos indica la dirección de retorno. Saltará a la etiqueta "`change`"
(8) Se repondrán los registros que corresponden a la 'tarea nueva'
(9) Se sale de la rutina a la dirección de retorno que se indica en 'tarea nueva'



4.2.4 HARDWARE_INTERFACE

Este paquete constituye la visión abstracta de la plataforma, un conjunto de procedimientos, funciones y tipos, que requiere MaRTE OS para que funcione. La mayor parte de las funciones y procedimientos del `hardware_interface` son 'llamadas' a procedimientos y funciones desarrollados en el `interrupt_tables, processor_registers` y `tpu`, por lo que tan sólo haremos mención para demostrar que se cumplen los requisitos mínimos. La estructura que se sigue es la siguiente: interrupciones, time&timers, bit_operations y context_switch.

INTERRUPCIONES

```
Subtype HW_Interrupt is interrupt_tables.interrupt;
```

Se debe declarar la constante para la interrupción del 'timer':

```
TIMER_INTERRUPT : constant HW_Interrupt :=
        interrupt_tables.TIMER_INTERRUPT;
```

Se va a renombrar a subrutinas de otros paquetes, esto implica crear subtipos de tipos que tienen como origen otros paquetes, para mantener la coherencia en las funciones y procedimientos utilizados en el hardware interface.

```
Subtype trap_state is interrupt_tables.trap_state;
Subtype trap_state_ac
     is interrupt_tables.trap_state_ac;
```

Procedure default_HW_interrupt_handler

```
Procedure default_HW_interrupt_handler
     (state : in trap_state_ac) renames
     interrupt_tables.default_interrupt_handler;
```

procedure install_HW_interrupt_handler

```
procedure install_HW_interrupt_handler
     ( int_num : in HW_interrupt;
     handler : in HW_interrupt_handler) is
begin
     interrupt_tables.install_interrupt_handler
          (interrupt_tables.interrupt (int_num),
          interrupt_tables.interrupt_handler_type(hand
          ler));
end install_HW_interrupt_handler;
```

procedure interrupts_disabled

```
procedure interrupts_disabled renames
     processor_registers.interrupts_disabled;
```

procedure interrupts_enabled

```
procedure interrupts_enabled renames
     processor_registers.interrupts_enabled;
```

procedure interrupt_controller_hardware_enable_interrupt

Se habilitan las fuentes de interrupción hardware de un canal del TPU en particular.

```
procedure interrupt_controller_hardware_
     enable_interrupt (int : in HW_interrupt) is
begin
     bit_set(unsigned_32(tpu.cier),
```

```
                unsigned_32(int-HW_interrupt(80)));
    end interrupt_controller_hardware_enable_interrupt;
```

Las interrupciones que va a manejar MaRTE OS son provocadas por el módulo TPU, y más en particular por el temporizador hardware (cumpliendo los requisitos mínimos), en este procedimiento accedemos al registro de activación de interrupciones para activar el canal que provoca la interrupción del temporizador hardware en el módulo TPU (cier). Se aprovecha la función `bit_set` implementada en el paquete `processor_registers.`

procedure interrupt_controller_hardware_disable_interrupt

Se deshabilitan las fuentes de interrupción hardware de un canal del TPU en particular.

```
    procedure interrupt_controller_hardware_disable_
              interrupt(int : in HW_interrupt) is
    begin
         bit_reset(unsigned_32(tpu.cier),
                   unsigned_32(int-HW_interrupt(80)));
    end interrupt_controller_hardware_disable_interrupt;
```

procedure interrupt_controller_hardware_enable

Este procedimiento se encarga de la activación del controlador de las interrupciones hardware.

```
    procedure interrupt_controller_hardware_enable is
    begin
         tpu.cisr := 16#0000#;
    end interrupt_controller_hardware_enable;
```

En el módulo se encuentra implementado un registro que realiza esta función, este registro informa del estado de los 'flags' de interrupción de cada uno de los canales de la tpu, el cual antes de activar las interrupciones deberá encontrarse inicializado a '0'. Este procedimiento debe afectar a todas las fuentes de interrupción hardware, por ello que se inicializa todo el registro y no sólo el canal del 'timer'.

TIME&TIMER

```
Last_HW_Timer : constant integer_32 := 0;
```

Sólo habrá implementado un timer;

```
Type HW_Timers is new integer_32;
HW_timer_0 : constant HW_timers := 0;
```

```
Type HWTime is new unsigned_64;
```

Se define el tipo HWTime, como un nuevo unsigned_64, este tipo es utilizado por MaRTE OS para llevar la cuenta del tiempo transcurrido utilizando como unidad el "tick" del reloj hardware proporcionado por la plataforma, con un unsigned_64 podremos almacenar y llevar el tiempo desde la inicialización sin ningún problema, ya que en el peor caso 238.5ns sería la mejor resolución que podríamos obtener, luego esto nos llevaría a una cuenta de:

238.5ns x 2^{64} / (3600 sec/hora x 24 h/día x 365 dia/año) ~ 139508 años

```
HWT_HZ : constant := tpu.TPU_HZ;
```

Es el número de 'HWTime' por segundo.

```
NS_per_S : constant  HWTime := 10#1#E9;

Total_Time : HWTime;
```

Total_time es la variable que va a llevar el tiempo del sistema en 'HWTime', se inicializará con la inicialización de MaRTE OS.

Function safe_longest_timer_interval

```
Function safe_longest_timer_interval return HWTime is
Begin
     Return tpu.safe_longest_timer_interval;
End safe_longest_timer_interval;
```

Las siguientes cuatro funciones son internas al paquete y no aparecen en la especificación del paquete hardware_interface, pero ya que serán utilizadas en otros procedimientos y funciones de dicho paquete.

Function to_HWT_NS

```
Function to_HWT_NS is new
      ada.unchecked_conversion (duration, HWTime);
```

Function to_duration

```
Function to_HWT_NS is new
     ada.unchecked_conversion (HWTime, duration);
```

Function HWTime_to_unsigned_64

```
Function HW_to_unsigned_64 is new
```

```
        ada.unchecked_conversion (HWTime, unsigned_64);
```

Function du_to_unsigned_64

```
    Function HW_to_unsigned_64 is new
        ada.unchecked_conversion (duration, unsigned_64);
```

Function HWTime_to_duration

```
    Function HWTime_to_duration (th : in HWTime)
              return Duration is
        S : HWTime;
    Begin
        S := th / HWT_HZ;
        Return to_duration (S * NS_Per_S + ((th – S *
        HWT_HZ) * NS_Per_S) / HWT_HZ);
    End HWTime_to_duration;
```

Esta función hace la conversión en dos partes, primero la parte entera en nanosegundos (S * NS_Per_S), después se le añade la parte decimal, la parte decimal será el valor completo menos su parte entera y como estamos sumando nanosegundos, el segundo sumando queda como sigue:
((th – S * HWT_HZ) * NS_Per_S) / HWT_HZ, una vez realizada la suma el valor se convierte al tipo *duration*, y se devuelve ese valor.

Function duration_to_HWTime

```
    Function duration_to_HWTime (D : in duration)
              return HWTime is
        S : HWTime;
    Begin
        S := to_HWT_NS (D) / NS_per_S;
        Return S * HWT_HZ + ((to_HWT_NS (D) – S *
        NS_per_S) * HWT_HZ) / NS_per_S;
    End duration_to_HWTime;
```

La variable ‘S’ el valor en segundos del parámetro de entrada ‘D’. El valor de retorno se realiza en dos partes diferenciadas, la entera y la decimal, la parte entera es el resultado de convertir a segundos la parte entera del parámetro de entrada ‘D’, la parte decimal en nanosegundos corresponde al valor de la variable ‘D’ convertido a nanosegundos menos la parte entera, convertido a segundos. Luego las unidades de retorno quedan de la siguiente forma:
‘ticks’ se les puede considerar como las unidades del HWTime.
(segs* ticks/segs + ((nsec – sec * nsec/sec) * ticks/sec / nsec/sec) = ticks

```
time_before: Unsigned_16 := 0;
```

Corresponde a una variable global, interna al paquete `hardware_interface`, que nos va a ayudar a llevar correctamente el tiempo, debe estar inicializada a cero y se modificará en el procedimiento 'program_timer'. Su función se analizará a continuación.

procedure program_timer

Este procedimiento va a ser el encargado de programar el temporizador hardware que utilice MaRTE OS, tendrá como parámetros de entrada `timer`, MaRTE OS está preparado para admitir más de un 'timer', *interval*, que corresponde al valor de cuenta en 'ticks' de HWTime con el que se programa el 'timer', y un parámetro de salida, que indicará cuando se debe realizar la siguiente programación del timer antes de que espire su cuenta.

```
     procedure program_timer ( timer : in HW_Timers ;
                               interval : in HWTime;
                               next_activation : out HWTime) is
          time_after: unsigned_16;
     begin
(1)       tpu.program_tpu_timer(unsigned_16(interval));
(2)       time_after := tpu.Read_counter;
(3)       total_time := total_time + HWTime(time_after –
                        time_before);
(4)       time_before := time_after;
(5)       next_activation := total_time + interval ;
     end program_timer;
```

(1) Se llama a la función que programa el canal del TPU asignado para programar el temporizador hardware, el valor de cuenta será el parámetro de entrada `interval`.
(2) Antes de calcular el `'total_time'`, debemos hacer una lectura del contador de 'ticks' de TCR1.
(3) La variable 'total_time'se calcula como el 'total_time' acumulado más el valor de la diferencia de `'time_after'` y `'time_before'`. Como se inicializa la variable `'time_before'` a cero, inicialmente cometeremos un error máximo de valor 1.908 µs, despreciable frente al uso que se le va a dar a esta variable en la medición de tiempos, aún así esto tan sólo supone tener un tiempo inicial de ese valor en vez de ser '0'.
(4) Al hacer esto nos estamos asegurando que no existe perdida de tiempo en la programación del temporizador hardware, ya que asignamos exactamente a `'time_before'` el valor de `'time_after'`.
(5) El valor de la próxima activación viene dado por el valor de cuenta que se haya dado en la programación del temporizador hardware.

Function get_HWTime_Slow

Función que se encarga de la captura de tiempos medidos en HWTime, esta función salva el CCR (Registro de condición de código) y deshabilita las interrupciones, perteneciente al registro de estado del MC68332 antes de hacer la captura de tiempo, después lo restaura, y devuelve el valor actual de 'total_time'.

```
Function get_HWTime_Slow return HWTime is
     Flags : integer;
     Ret : Hwtime;
Begin
     Flags := processor_registers.save_flags;
     TPU.interrups_disabled;
     Ret := total_time + HWTime( time_before –
           tpu.read_counter);
     Processor_registers.restore_flags (flags) ;
     Return ret;
End get_HWTime_slow;
```

Function get_HWTime

Se encarga de la captura de tiempos, pero asumiendo que se hace con las interrupciones deshabilitadas, es más rápido que `get_HWTime_slow`, pero hay que estar seguro con las interrupciones;

```
Function get_HWTime return HWTime is
Begin
     Pragma assert (not are_interrupts_enabled,
     "get_HWTime: called with interrupts enabled");
     Return Total_time + HWTime(time_before –
          tpu.Read_counter);
End get_HWTime;
```

El `pragma assert` nos va a permitir detectar si la función es ejecutada con las interrupciones habilitadas.

Function get_HWclock_frequency

```
Function get_HWclock_frequency return HWTime is
Begin
     Return HWT_HZ;
End get_Hwclock_frequency;
```

Function compulsory_timer_reprogramming

La medida del tiempo se hace usando el módulo de la TPU, y este debe de estar siempre programado para permitir almacenar el tiempo transcurrido en la variable `'total_time'`, luego esta función debe retornar TRUE para indicar a MaRTE OS que es necesario realizar la reprogramación del temporizador, incluso cuando no haya programado ningún evento. Esto podría ocurrir, por ejemplo, cuando las interrupciones del timer son utilizadas tan solo por el reloj del sistema.

```
Function compulsory_timer_reprogramming return
          boolean is
Begin
     Return TRUE;
End compulsory_timer_reprogramming;
```

BIT_OPERATIONS

Procedure bit_set

```
Procedure bit_set (bit_field : in out unsigned_32;
                   bit : in unsigned_32)
          renames processor_registers.bit_set;
```

Procedure bit_reset

```
Procedure bit_reset (bit_field : in out unsigned_32;
                     bit : in unsigned_32)
          renames processor_registers.bit_reset;
```

Procedure bit_scan_forward

```
Procedure bit_scan_forward (bit_field :in unsigned_32;
                            bit : out unsigned_32)
     renames processor_registers.bit_scan_foward;
```

Procedure bit_scan_reverse

```
Procedure bit_scan_reverse (bit_field :in unsigned_32;
                            bit : out unsigned_32)
     renames processor_registers.bit_scan_reverse;
```

CONTEXT SWITCH

Procedure context_switch

```
Procedure context_switch(old_task : system.address;
```

```
                                new_task : system.address)
          renames processor_registers.context_switch;
```

procedure change_return_address_of_preemted_task

```
     procedure change_return_address_of_preemted_task
          ( top_of_stack : in unsigned_32;
          new_ret_address : in system.address)
          renames processor_registers.
               change_return_address_of_preempted_task;
```

Procedure finish_hardware_use

```
     Procedure finish_hardware_use is
     Begin
          Null;
     End finish_hardware_use;
```

Procedure breakpoint

```
     Procedure breakpoint is
     Begin
          Put ("fatal_error_68332");
          New_line;
     End breakpoint;
```

Al final del paquete `hardware_interface.adb` se abre un bloque **`begin...end`**. Dicho bloque solo se ejecutará una vez, por lo que se utilizará para inicializar procedimientos, funciones y variables.

```
          Begin
     (1)  Tpu.clock;
     (2)  Total_time := 0;
     (3)  Tpu.cisr := tpu.cisr and 16#FF7F#;
     (4)  Tpu.cier := tpu.cier and 16#FF7F#;
     (5)  Processor_registers.interrrupts_disabled;
     (6)  Interrupt_tables.initialize;
     (7)  Tpu.initialize_timer;
     (8)  Tpu.program_tpu_Timer(31447);
     (9)  Tpu.cisr := tpu.cisr and 16#FF7F#;
          End hardware_interface;
```

(1) el procedimiento '`clock`' se ejecuta, esto hace que comience a generarse el pulso diseñado, no se interrumpe nunca.
(2) Se inicializa a '0' la variable que va a llevar el conteo del tiempo, '`total_time`'.

(3) Se limpia el registro de los 'flags' de interrupción del canal del temporizador hardware en la tpu.
(4) Se deshabilitan las interrupciones en canal del 'timer de la tpu.
(5) Se deshabilitan las interrupciones en la CPU. Requisito de MaRTE OS.
(6) Se instalan los manejadores de interrupción por defecto en los canales de la TPU.
(7) Este procedimiento preparará el canal del TPU encargado del temporizador hardware, dejándolo pendiente del valor de cuenta con el que sería programado el temporizador hardware.
(8) La primera vez que se programa el temporizador hardware, se programa al valor máximo que puede aceptar como parámetro, aprox. 7.5 ms.
(9) Nos aseguramos que las interrupciones en el canal del timer están deshabilitadas. Requisito de MaRTE OS.

5 PROTOTIPO DE PLATAFORMA DE APLICACIÓN Y EJECUCIÓN

La plataforma de aplicación será SoccerBot [TB03], que es un robot móvil inteligente, esto es, contiene un microcontrolador de 32 bits (motorola MC68332) el cual tiene a su disposición una variedad de sensores y actuadotes: cámara digital, sensores de distancia por infrarrojos, motores DC, y servos, etc. La interacción con el usuario se realiza a través de una pantalla LCD y cuatro botones.

Originalmente el EyeBot (el controlador del SoccerBot) trabaja bajo el sistema operativo de tiempo real RoBiOS, que se encuentra en ROM (128Kb), incluye un conjunto de librerías de funciones de operaciones de entrada/salida muy básicas. MaRTE OS es capaz de independizar su funcionamiento de el hardware específico sobre el que se ejecuta, esto es, se compilarán por separado tanto MaRTE OS como la APLICACIÓN junto con las librerías necesarias en cada caso, luego se enlazarán todos los archivos del tipo **.o* junto con las librerías estándar necesarias para crear un archivo único del tipo *.hex, que será un archivo binario preparado para ser transmitido al SoccerBot, será capaz de ejecutar de forma independiente , en principio, sin necesidad de utilizar RoBIOS como sistema operativo.

5.1 RoBIOS, LIBRERÍA DE FUNCIONES PARA PERIFÉRICOS

Esto no significa que no vayamos a utilizar RoBIOS en nuestras aplicaciones, de hecho, si se ejecuta una aplicación C que contenga tareas que controlar los diferentes sensores/actuadores, motores, etc... que se encuentren acoplados al controlador (EyeBot en este caso) sería necesario disponer de las librerías de funciones controladoras de periféricos que permitan un control sobre ellos, por esto, se utilizaran las funciones de RoBIOS, recayendo el control de tareas, manejadores de interrupciones y control del tiempo bajo el sistema operativo MaRTE OS.

5.2 RoBIOS & MaRTE OS

Debido a esta necesidad que tenemos de mantener RoBIOS junto con MaRTE OS en el microcontrolador, no deberemos modificar la configuración básica del microcontrolador. No podremos modificar nada del módulo SIM, ni 'chip-selects' , ni los registros de configuración SIMCR y SYNCR, ni los canales del TPU que se utilicen para motores, controlador de motores, micrófono,

'speaker', golpeador, cámara y cualquier otro periférico al que se le asigne un canal, así del TPU mantendremos los registros de configuración TPUMCR, TPUMCR2 y el TICR sin variación alguna.

Para la migración de MaRTE OS al microcontrolador es necesario un canal del TPU, para el temporizador hardware, y otro que produzca una señal periódica, que se utilice como reloj de sistema, así como la capacidad de producir interrupciones por parte del temporizador hardware. Todo esto se encuentra resuelto y MaRTE OS utilizará los canales 8 y 9 de la TPU. Estos canales en principio reservados por RoBIOS para unos servos, no se encuentran conectados en nuestro robot, luego los podremos utilizar para nuestro fin.

RoBIOS no provoca interrupciones con la TPU, aun así MaRTE OS debe ser capaz de saber que canal ha provocado la interrupción (el canal del 'timer'). Para satisfacer los requisitos mínimos de MaRTE OS, también se deberá tener en cuenta otra fuente de interrupción, la *KEYBOARD INTERRUP,* aunque el SoccerBot no la provoque nunca.

El modo de actuar de nuestro compilador cruzado será el siguiente: para una determinada función definida dentro de una aplicación C, primero buscará en las librerías que le proporciona MaRTE OS, si no la encuentra, buscará en las librerías que tiene RoBIOS, y de no encontrarse allí, buscará por último en las librerías estándar C. Esta es una forma de estar seguros que se va a trabajar con las librerías que se encuentren implementadas para MaRTE OS.

5.3 Hardware Description Table (HDT)

El HDT [TB03] describe de forma detallada la configuración del hardware que se encuentra instalado en el SoccerBot, junto con sus correspondientes 'drivers' y configuración, de aquí se obtienen los canales de la TPU que puede utilizar MaRTE OS.

Para poder observar qué conectores tiene el SoccerBot, ocupados y que 'driver' utiliza debemos analizar el HDT (Tabla de Descripción Hardware) que se ha descargado en la plataforma de desarrollo.

En el HDT encontramos tres partes bien diferenciadas: una primera de calibración/inicialización (la calibración que se hace para los diferentes sensores de infrarrojos y calibración de los motores); una segunda de definición /configuración (de los motores, de los servos, de los sensores de infrarrojos, de la barra de estado de la batería, del sensor de control remoto por infrarrojos así como de la información general del robot); y una tercera parte donde se define la composición de la HDT con todos sus componentes.

El HDT deberá ser modificado en función de los periféricos que se quieran habilitar. No se deberá deshabilitar la variable *roboinfo,* ya que de ser así, por tratarse de la variable de configuración del SoccerBot, éste tendría una configuración mínima. Esto es, la configuración del robot sería la más básica por la que se podría optar en condiciones normales en cuanto a velocidad de transmisión de datos a 9600, protocolo de conexión a NONE, puerto de conexión a SERIAL1, etc...

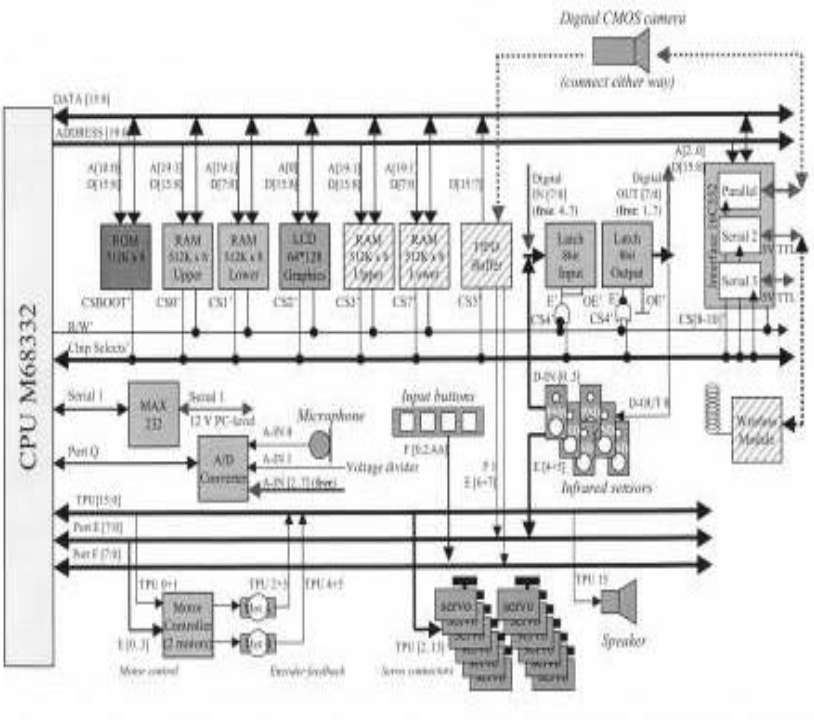
La compilación del HDT se hará un compilador especial llamado *gcchdt,* tan sólo tenemos que transmitirlo al SoccerBot a través del cable serie RS-232 y ejecutarlo, se detectará que se trata de un HDT automáticamente.

Para ver una descripción detallada del HDT que se encuentra instalado en el SoccerBot ir a anexo.

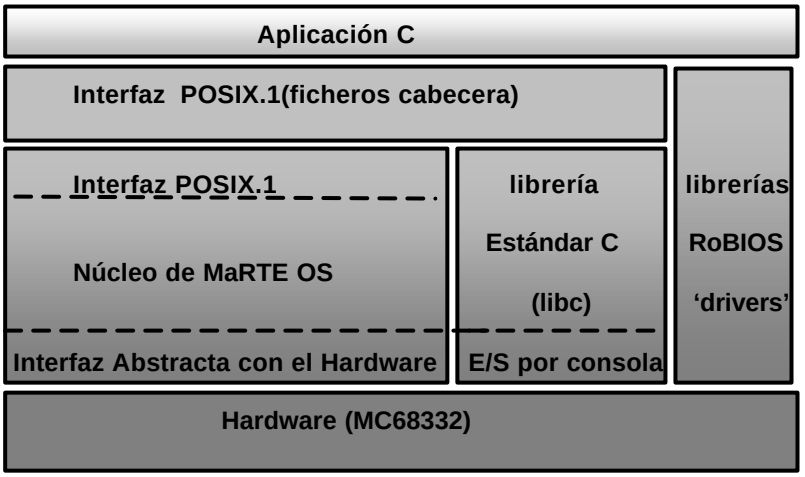
5.4 CONTROLADOR EyeBot

El EyeBot es un potente controlador de 32 bits en el que se encuentran ensamblados en un espacio reducido tanto el microcontrolador MC68332 como memorias, sensores, actuadores, convertidores A/D e interfaces de comunicación (puertos paralelo, serie, bdm).

Diagrama de bloques



5.5 DESARROLLO DE LA APLICACIÓN



En el desarrollo de la aplicación trataremos de comprobar que MaRTE OS ha sido portado con éxito al microcontrolador MC68332.

En la aplicación el Soccerbot se moverá en línea recta hasta que se aproxime a un objeto, que lo evitará girando hasta que pueda continuar su marcha en línea recta, pero si mientras trata de reorientarse se acercase demasiado a algún objeto, inmediatamente retrocedería hasta poder evitarlo en su giro, los recursos que utilizaremos del SoccerBot serán: PSDfrontal, Motores izquierdo y derecho.

Se trata de desarrollar un programa en el que se desean controlar diferentes sensores del SoccerBot mediante tareas, `'threads'` (la aplicación estará escrita en C), concediéndoles una prioridad a cada una de ellas. La tarea de prioridad más alta tendrá que, inicializar el PSD frontal (la primera vez que se llame a esta tarea), después entrará en un bucle infinito en el que se realizan lecturas con el PSD, (según el HDT, el PSD está configurado para detectar objetos que se encuentran a distancias de entre 60 mm y 999 mm), la tarea se bloqueará al ejecutarse la función *nanosleep,* las tareas son planificadas bajo la política POSIX denominada SCHEA_FIFO: los threads son planificados en base a su prioridad, respetándose el orden FIFO (First In First Out) para las tareas de una misma prioridad. Es entonces y teniendo en cuenta el valor

registrado, y siempre y cuando MaRTE OS permita que se activen las tareas de menor prioridad, se activan los motores que impulsan al robot dependiendo del valor obtenido por el PSDFrontal, existirá una tercera tarea, la de menor prioridad, que se encargará de escribir en la pantalla del LCD la distancia en mm a la que se encuentra el obstáculo detectado.

La tarea de prioridad media se encargará de mover los motores que mueven al SoccerBot dependiendo del valor de lectura del PSD.

Utilizando el 'mutex' (entre la tarea de prioridad alta y media), evitaremos que se produzcan errores entre las tareas, por ejemplo, si la tarea de máxima prioridad detecta un objeto cercano, la tarea prioridad media deberá activar los motores para que retrocedan, pero si antes de poder ejecutarse se despierta la tarea de mayor prioridad con otra lectura del PSD diferente a la anterior, podría llegar a provocar que el SoccerBot se choque con el objeto. Con el `mutex`, aunque la tarea de máxima prioridad se 'despierte' no se ejecutará hasta que se 'libere' el `mutex`.

La tarea de menor prioridad tratará de mostrar en pantalla el valor actual de lectura del PSD, esta tarea se activará siempre que las otras dos se encuentren bloqueadas.

Se muestra el código de la aplicación:

```
#include "eyebot.h"
#include "/home/agc/marte/include/stdio.h"
#include "/home/agc/marte/include/signal.h"
#include "/home/agc/marte/include/time.h"
#include "/home/agc/marte/include/pthread.h"
#include "/home/agc/marte/include/sched.h"

extern void adainit();
extern void adafinal();

struct shared_variable
     {
          pthread_mutex_t mutex;
          int psdf;
     }variable;

void * inicia (void *arg)
{
     PSDHandle psd_handler_front;
     struct timespec ts;
     ts.tv_sec = 0;
     ts.tv_nsec = 150000000;
     psd_handler_front = PSDInit(PSD_FRONT);
```

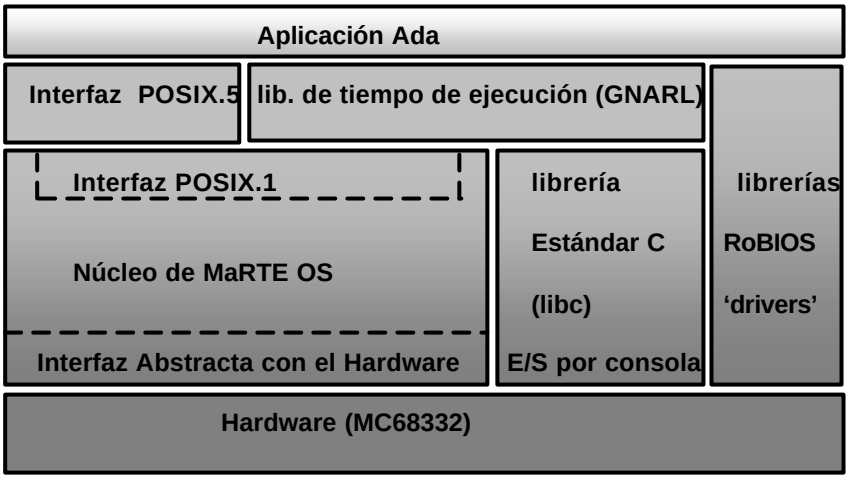
```
      while (1)
      {
            pthread_mutex_lock(variable);
            PSDStart(psd_handler_front,FALSE);
            variable.psdf=PSDGet(psd_handler_front);
            pthread_mutex_unlock(variable);
            nanosleep(&ts,NULL);
      }
}

void * mueve (void *arg)
{
      MotorHandle leftmotor;
      MotorHandle rightmotor;
      int psd_front;
      struct timespec ts;
      ts.tv_sec = 0;
      ts.tv_nsec = 200000000;
      leftmotor = MOTORInit(LEFTMOTOR);
      rightmotor = MOTORInit(RIGHTMOTOR);
      while (1)
      {
            pthread_mutex_lock(variable);
            if (variable.psdf<100)
            {
             MOTORDrive(leftmotor,-30);
             MOTORDrive(rightmotor,-30);
            }
                  else if (variable.psdf >=100)&&
                        (variable.psdf<165)
            {
            MOTORDrive(leftmotor,30);
            MOTORDrive(rightmotor,-30);
            }
                  else if ((variable.psdf >=165 )&&
                        (variable<980)
            {
            MOTORDrive(leftmotor,30);
            MOTORDrive(rightmotor,30);
            }
                  else

            {
            MOTORDrive(leftmotor,0);
            MOTORDrive(rightmotor,0);
            }
            pthread_mutex_unlock(variable);
```

```
                nanosleep(&ts,NULL);
     }
}

void * pon (void *arg)
{
  struct timespec ts;
  int psd_front;
  ts.tv_sec = 0;
  ts.tv_nsec = 400000000;
  while (1)
  {
     pthread_mutex_lock(variable);
     psd_front == variable.psdf;
     pthread_mutex_unlock(variable);
          LCDSetPos(5,0);
          LCDPrintf("Front:%3d\n",psd_front);
      nanosleep(&ts,NULL);
    }
}
int main()
{
  pthread_mutexattr_t mutexattr;
  pthread_t t0,t1,t2;
  pthread_attr_t attr;
  struct sched_param sch_param;
  adainit();
  if (pthread_attr_init(&attr) != 0 ){
    perror ("error creacion atributo");
  }
  pthread_mutexattr_init(&mutexattr);
  pthread_mutexattr_setprioceiling(&mutexattr,
          sched_get_priority_max(SCHED_FIFO) - 2);
  if (pthread_mutex_init(&variable.mutex,&mutexattr) != 0){
    perror ("error mutex init");
  }
 if (pthread_attr_setinheritsched
(&attr,PTHREAD_EXPLICIT_SCHED) != 0){
    perror("error in attribute inheritsched");  }
  }
 if (pthread_attr_setdetachstate
(&attr,PTHREAD_CREATE_DETACHED) != 0){
    perror("error in attribute detachstate");
  }
  if (pthread_attr_setschedpolicy (&attr,SCHED_FIFO) != 0){
    perror("error in attribute schedpolicy");
  }
```

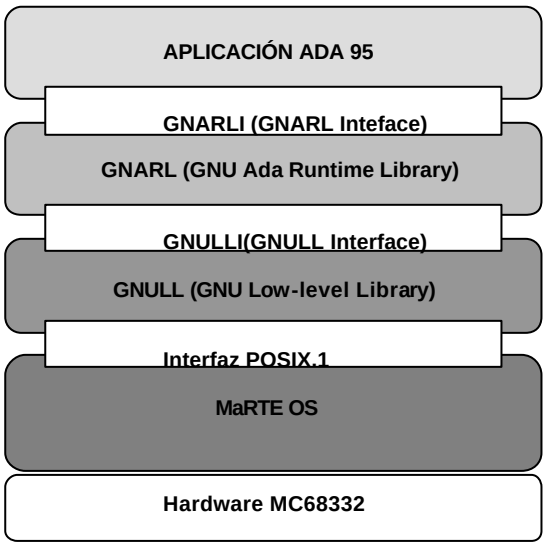
```
 sch_param.sched_priority =
(sched_get_priority_max(SCHED_FIFO) - 2 );
  if (pthread_attr_setschedparam (&attr,&sch_param) != 0){
    perror("error in attribute schedparam");
  }
  if (pthread_create(&t0, &attr, inicia, NULL) != 0){
    perror ("error creacion inicia");
  }
 sch_param.sched_priority =
(sched_get_priority_max(SCHED_FIFO) - 3);
  if (pthread_attr_setschedparam (&attr,&sch_param) != 0){
    perror("error in attribute schedparam");
  }
  if (pthread_create(&t1, &attr, mueve, NULL) != 0){
    perror ("error creacion mueve");
  }
 sch_param.sched_priority =
(sched_get_priority_max(SCHED_FIFO) - 6);
  if (pthread_attr_setschedparam (&attr,&sch_param) != 0){
        perror("error in attribute schedparam");
  }
  if (pthread_create(&t2, &attr, pon, NULL) != 0){
    perror ("error creacion pon");
  }

  adafinal();
  return (0);
}
```

5.6 PROBLEMAS Y SOLUCIONES

Otro de los objetivos a desarrollar por en esta memoria, correspondería a las aplicaciones escritas en lenguaje Ada, estas aplicaciones tienen una estructura un poco diferente a las aplicaciones C y es la siguiente:



Para poder desarrollar una aplicación basada en tareas Ada, '*tasks*', debemos tener adaptada una librería de tiempo de ejecución que es la encargada de proporcionar el soporte para tareas incluido en el lenguaje C. en nuestro caso realizaríamos la adaptación de la librería del compilador GNAT (GNARL). La mayor parte de la librería GNARL es independiente de la plataforma, por lo que nos faltaría, según se explica el propio sistema MaRTE OS, la adaptación del paquete GNULL (GNU Low-level Library). De una forma más detallada una aplicación Ada sigue la siguiente estructura:



Se muestra el esquema que sigue una aplicación Ada ejecutando sobre GNARL y sobre MaRTE OS. El compilador GNAT genera el código correspondiente a la semántica de tareas realizando llamadas a GNARL. La interfaz GNARLI (GNARL Inteface) facilita la comunicación entre el código generado por el compilador y la librería de tiempo de ejecución. La librería de tiempo de ejecución es en su mayor parte independiente de la plataforma sobre la que se ejecuta. Sólo una pequeña parte, denominada GNULL (GNU Low-level Library), deberá ser modificada para adaptarse al MC68332. La interfaz entre las partes de la librería dependientes e independientes de la plataforma se denomina GNULLI (GNULL Interface) [MAR02].

Los archivos que componen parte de GNULL y se basan en las librerías estándar de GNAT en nuestro caso los archivos que deben ser modificados son los siguientes, que son los que indica el compilador cuando queremos compilar la librería '*runtime*' de Ada: `s-osinte.ads, s-osinte.adb, s-osprim.adb, s-taprop.adb, a-calend.ads, a-calend.adb, a-textio.adb. Y los ficheros cabecera escritos en C: a-ada.h, a-adainit.h, a-cio.c, a-init.c, a-raise.c, a-raise.h, a-type.h, localtime.c.`

Esta adaptación se deja como trabajo futuro.

Otro problema se nos planteó al tratar de usar recursos del SoccerBot, tales como la captura de imágenes a través de la cámara digital, se trató de hacer un programa con dos tareas: una tarea(la de mayor prioridad) donde se realizase la inicialización de la cámara y se entrase un bucle infinito que capturase la imagen (en color) en una variable y 'durmiéramos' la tarea, la variable se compartiría mediante un mútex entre las dos tareas; la segunda tarea convertiría la variable compartida (la imagen capturada) en escala de grises para mostrarlo en el LCD. La función que realiza la captura de la imagen es un tanto compleja, ya que tiene que inicializar la cámara, hacer una calibración y ajustar los parámetros de contraste y brillo, teniendo en cuenta que el código nos es desconocido; es en esta función donde durante la ejecución de la aplicación se genera un error, el compilador genera una instrucción ilegal, por lo que se optó por utilizar sensores/actuadores más sencillos para la implementación de la aplicación, los PSDs, sensores de posición, donde la calibración del sensor se realiza en cuando se define en el HDT(`Hardware Description Table`). En el futuro debería probarse a compilar la aplicación con una versión más moderna del compilador GCC para comprobar si el error se repite.

5.7 CONCLUSIONES

Se ha desarrollado una aplicación capaz de utilizar diferentes sensores/actuadores del robot SoccerBot . La aplicación está formada por tres threads POSIX y en ella se utilizan numerosos primitivas de la interfaz POSIX proporcionada por MaRTE OS. El objetivo de la aplicación no era el de utilizar el mayor número de dispositivos del SoccerBot, sino los suficientes para desarrollar una aplicación sencilla donde se aprecie el control que realiza MaRTE OS sobre los '`pthreads` 'planteados.

La aplicación no ha sido posible desarrollarla en lenguaje Ada debido a no haberse realizado la adaptación de la librería runtime GNARL, la cual está formada por archivos que regulan el más bajo nivel, para realizar estas modificaciones se requiere unos amplios conocimientos de sistemas operativos, de MaRTE OS en nuestro caso, debido a las modificaciones que requieren estos archivos.

Al utilizar las funciones de la librería de RoBIOS, estamos asumiendo que con ciertas funciones se puede perder un cierto control en las aplicaciones, se desconocen posibles fuentes de interrupción generadas por las funciones que controlan los diferentes sensores/actuadores, no se conoce como funciona el manejador/es de interrupción, sólo podemos asegurar el control de las interrupciones que se provoquen a partir del módulo TPU, (provoca interrupciones de nivel 5), y de la interrupción periódica en la SIM (no la utilizaremos, luego no se provocará, la utiliza RoBIOS para llevar el tiempo con una precisión de centésimas). Cualquier otra interrupción no sería provocada por MaRTE OS, luego deberán ser provocadas por las funciones de la librería de RoBIOS. Se plantean varias soluciones, la primera, crear nosotros los *controladores* que controlen a estos sensores; la segunda, utilizar funciones de RoBios que no nos generen esta clase de problemas. La primera sería la mejor solución para obtener un control total sobre los sensores/actuadores, pero queda fuera de los objetivos de este proyecto, y la segunda es la más adecuada para la situación en la que nos encontramos. Se ha comprobado que los periféricos más sencillos de controlar son los motores y los PSDs.

6 ENTORNO DE DESARROLLO

El propósito de este apartado es guiar en el proceso de instalación del entorno de desarrollo cruzado GNU para ser usado con MaRTE OS. Las referencias que se hacen en este apartado a RTEMS se deben a que se seguirán los pasos necesarios para la creación del entorno de desarrollo cruzado para RTEMS [RTE02] pero sin llegar a instalar este sistema operativo, y si MaRTE OS.

Se verá como instalar los siguientes componentes:

Herramientas de compilación cruzada GNU C/Ada en PC
Depurador GDB

6.1 HERRAMIENTAS DEL ENTORNO DE DESARROLLO CRUZADO

Como ya se había comentado con anterioridad, el software utilizado será en su mayor parte desarrollado por el proyecto GNU, bajo la licencia GPL. A continuación se realizará una breve explicación del contenido y funcionalidad de las herramientas elegidas: GCC, GNAT, BINUTILS y GDB.

GCC

Se trata de una colección de compiladores para lenguajes como ANSI C, C++, Objetive C, Java y Fortran. El GCC es capaz de proporcionar muchos niveles de error de código fuente tradicionalmente suministrados por otras herramientas (tales como *lint*), produce información para la depuración, y puede desarrollar diferentes optimizaciones en el código objeto (*object files*) resultantes.

GNAT

El GNAT es un completo compilador para Ada95 integrado en la colección de compiladores GCC.

Ada es un lenguaje perfecto para sistemas grandes y complejos, o en proyectos donde la reusabilidad y mantenabilidad del código son esenciales. GNAT ahora incluye el GLIDE, un entorno de desarrollo integrado que incluye un editor y un depurador.

Binutils

Las binutils son una colección de herramientas para código binario. Las principales son:

`ld:`	enlazador de archivos objeto de GNU (object files linker)
`as:`	ensamblador de GNU, se encargará de 'compilar' los archivos escritos en lenguaje ensamblador

Aunque también utilizaremos los siguientes:

`ar:`	una utilidad para la creación, modificación y extracción de archivos, por ejemplo, obtener un object file determinado (`*.o`) de una librería (`*.a`).
`nm:`	muestra una lista de los objetos que se encuentran definidos en un `*.o`, nos será útil para conocer el nombre dado por el compilador a un procedimiento o función escrito en lenguaje Ada.
`objcopy:`	copia y traduce los archivos objeto (`*.o`),
`objdump:`	muestra la información contenida en los archivos objeto. (`*.o`).

GNU GDB

Éste es el depurador de GNU, nos va a permitir observar que ocurre dentro de otro programa mientras se ejecuta, o bien, que hace el programa en el mismo instante que se produce un error de ejecución.

El GDB permite ejecutar el programa, especificar algo que podría modificar su comportamiento, colocar puntos de ruptura en condiciones especificadas, examinar que ha ocurrido cuando el programa se ha parado y cambiar el valor de las variables, y continuar depurando.

6.2 INSTALACIÓN DE HERRAMIENTAS DE COMPILACIÓN CRUZADA GNU C/ADA

En este apartado realizaremos la instalación paso a paso de las herramientas de compilación cruzada GNU.

Creación de los directorios `'archive' & 'tools'`

El primer paso para la instalación de las herramientas de desarrollo cruzado es la creación de los directorios 'archive', el cual se utilizará para la ubicación de códigos fuentes, y el directorio 'tools', el cual se utilizará como directorio de trabajo para la construcción del entorno.

Comandos que hay que ejecutar:

```
mkdir cruzado
cd cruzado
mkdir archive tools
cd archive
```

Las herramientas necesarias para desarrollar el compilador cruzado se obtienen a través de internet [RTE03]

Aunque las herramientas descargadas están pensadas para funcionar con el sistema operativo RTEMS, no se llegará nunca a instalar este sistema operativo, lo que nos interesa es su entorno de desarrollo.

Descargo tanto las herramientas necesarias para el compilador como para de depurador, así como los 'parches' necesarios para su correcta instalación:

```
ada_build_scripts-4.5.0.tgz
binutils-2.9.1-rtems-gnat-3.12p-20000429.diff
binutils-2.9.1.tar.gz
gcc-2.8.1-rtems-gnat-3.12p-20000429.diff
gcc-2.8.1.tgz
gdb-4.17-bdm-patches.tar.gz
gdb-4.17-rtems-gnat-3.12p-20000429.diff
gnat-3.12p-rtems-20000429.diff
gnat-3.12p-src.tar.gz
newlib-1.8.2-rtems-20000606.diff
newlib-1.8.2.tar.gz
```

Ahora ya podemos descomprimir todos los archivos:

```
cd ..
cd tools
tar xzf ../archive/gcc-2.8.1.tar.gz
     tar xzf ../archive/gnat-3.12p-src.tar.gz
     tar xzf ../archive/binutils-2.9.1.tar.gz
     tar xzf ../archive/newlib-1.8.2.tar.gz
     tar xzf ../archive/ada_build_scripts-4.5.0.tgz
```

Ahora debemos instalar las binutils cruzadas, es un requisito para poder comenzar la instalación del gcc, para ello seguimos los pasos de instalación del README del `'binutils-2.9.1'`

Una vez que tengamos las binutils instaladas, aplicamos los parches:

```
cd gcc-2.8.1/
patch -p1 <../../archive/gcc-2.8.1-rtems-gnat-3.12p-
20000429.diff
cd ..
cd binutils-2.9.1/
```

```
patch -p1 <../../archive/binutils-rtems-gnat-3.12p-
20000429.diff
cd ..
cd newlib-1.8.2/
patch -p1 <../../archive/newlib-1.8.2-rtems-
20000606.diff
cd ..
cd gnat-3.12p-src/
patch -p1 <../../archive/gnat-3.12p-rtems-gnat-
20000429.diff
cd ..
```

Como queremos que el compilador `gcc` compila código tanto en lenguaje C como en Ada hacemos lo siguiente:

```
cp -rf gnat-3.12p-src/src/ada/ gcc-2.8.1/
cp gnat-3.12p-src/src/gcc-281.diff/ gcc-2.8.1/
cd gcc-2.8.1/
patch -p0 <gcc-281.diff
touch cstamp-h.in
cd ada/
touch treeprs.ads a-[es]info.h nmake.ad[bs]
cd ..
```

Ahora se modifica el INSTALL POINT en el archivo 'user.cfg' para instalarlo donde convenga, en este caso en `/usr/local/MaRTE`

Además se debe aplicar el parche que viene en las fuentes del 'gnat 3.12p' para el gcc 2.8.1, como se indica en el Readme.build de las fuentes del GNAT

Para comenzar la instalación del compilador tenemos que ejecutar `'bit_ada'` teniendo en cuenta que nuestra plataforma de desarrollo será `'m68k-coff'`. Deberá ser ejecutado en modo 'supervisor'

```
# ./bit_ada m68k-coff
```

El script `'bit_ada'` se ha generado al descomprimir:

```
ada_build_scripts-4.5.0.tgz
```

Este script nos evitará tener que instalar y configurar por separado el gcc, el gnat y newlib como se indica en las 'ayudas' o documentos 'README' de cada uno de ellos, siguiendo los pasos necesarios para crear un compilador cruzado, que generará código para máquinas del tipo *`m68k-coff`,* tal como es el MC68332.

Una vez que se ha desarrollado el entorno cruzado, el nombre de todas las herramientas cruzadas serán de la forma: `m68k-rtemscoff-herramienta`

Ahora instalaremos las herramientas disponibles para el SoccerBot, se creará un directorio donde guarden las librerías *libc,* junto con herramientas necesarias para el correcto funcionamiento del SoccerBot.

```
mkdir soccerbot
cd soccerbot
```

Se utiliza el siguiente *url* para descargar el archivo '*.tgz' que va a tener las herramientas y las *libc* para nuestra versión del SoccerBot [TB03]:

http://www.ee.uwa.edu.au/~braunl/eyebot/ftp/robios41.usr.tgz

```
tar zxf robios41.usr.tgz
```

Con lo que el directorio quedará de la siguiente forma

```
/soccerbot/mc
/soccerbot/libc
```

En el directorio mc encontraremos lo siguiente:

COPYING	DPROGS	HOWTO	VERSION	hex	ldfiles	pc-appl
ChangeLog	Execlib	Makefile	cmd	html	lib	
ChangeLog.hdtdata	HDT	README	hdtdata	include	libc	

Los directorios que nos interesan son:

`cmd/` que contiene los *shell scripts* que compilan, ensamblan y que enlazan aplicaciones para el SoccerBot.

`ldfiles/` contiene los archivos del tipo `*.ld` , que indicarán al compilador la forma de estructurar los datos de los `*.hex` para el Soccerbot

`hdtdata/` contiene un hdt adecuado para cada configuración del robot en el que pueda ser montado el EyeBot.

Lo primero hay que hacer es copiar las herramientas

```
soccerbot/mc/cmd/linux/gcc68o
                      /gas68o
                      /gld68o
```

A un directorio que se encuentre en nuestro PATH, o crear un directorio donde poder copiar y modificarlas, y añadir este directorio al PATH, lo más sencillo en este caso es añadir todo al directorio `~/bin`.

Ahora ya podemos modificar los shell scripts sin peligro de modificar los originales. Los shell scripts a modificar son los llamados `gcc68o`, `gld68o` y `gas68o`.

gcc68o

Se trata del script encargado de compilar programas C y Ada. Esta herramienta va a permitir crear un `'*.o'` a partir de un `'*.c'` ó `'*.ad[sb]'` Para el `gcc68o` se harán los siguientes cambios:

Donde se utilicen las herramientas del tipo '`m68k-coff-*'` habrá que sustituirlas por las que se han generado en el entorno cruzado, es decir por las `m68k-rtemscoff-*`, para poder hacer esto debemos modificar lo siguiente.

```
export basedir=/usr/local/
export gccdir=$basedir/rtems
export gccparts=$gccdir/lib/gcc-lib/m68k-rtemscoff/2.8.1
```

Tanto el directorio `libc` como el *mc* corresponderán a los directorios con el mismo nombre de `~/soccerbot/`

También se ha modificado para que el compilador reconozca si el archivo a compilar está escrito en Ada.

gas68o

Se trata del script encargado de ensamblar ficheros escritos en lenguaje ensamblador. Esta herramienta va a permitir crear un *`object file`* (`*.o`), a partir de un `'*.s'` (archivo escrito en el lenguaje ensamblador del MC68332)

Para poder utilizar la herramienta `m68k-rtemscoff-as` se hacen los siguientes cambios:

```
setenv gccdir   /usr/local/rtems
setenv PATH $gccdir/bin
setenv mc /home/agc/soccerbot/mc
```

gld68o

Se trata del script encargado de enlazar aplicaciones y convertirlas al formato esperado por el MC68332. Esta herramienta tiene creará un `'*.hex'` una vez haya enlazado todos los '*.o' que se le hayan suministrado.

Para enlazar los '*.o' tendrá que utilizar la herramienta m68k-rtems-ld, en la que se indicará que *ldscript* es el apropiado, junto con las librerías específicas para el SoccerBot, y las librerías de Ada y C que necesita el compilador.

6.3 EJEMPLO DE APLICACIÓN C *hola_mundo.c*

Ya se puede comprobar el correcto funcionamiento del entorno de desarrollo, se trabajará con el programa de prueba *hola.c*

```
#include "eyebot.h"
#include <stdio.h>

int main ()
{
     printf("hola\n");
     return 0;
}
```

Este programa de prueba se va a encargar de mostrar por pantalla del SoccerBot " hola " y terminar. El comando encargado de mostrar caracteres por pantalla es printf, el cual se encuentra en la librería estándar C.

Se han obtenido los siguientes resultados:

```
hola.c                 ->          ->  ocupa 87 bytes
gcc68o hola.c   ->  hola.o      ->  ocupa 466 bytes
gld68o hola.o   ->  hola.hex    ->  ocupa 23677 bytes
```

6.4 EJEMPLO DE APLICACIÓN ADA *hola_mundo.adb*

Probamos que el compilador 'gcc' también es capaz de compilar un programa simple, sin hacer uso de tareas, escrito en lenguaje Ada.

Tendremos que seguir los siguientes pasos:

1 Crear un programa Ada (hola.adb)que muestre un mensaje por pantalla. Para ello utilizará los siguientes paquetes, que se encuentran en el directorio:

~/MaRTE/kernel/

basic_console_io.adb

```
basic_console_io.ads
basic_console_io_c.c
basic_integer_types.ads
```

2 Compilar el *hola.adb* con el 'gnatmake' cruzado:

```
m68k-rtemscoff-gnatmake –c –f –gnatp hola.adb
```

`-c` -> sólo realiza la compilación
`-f` -> fuerza la compilación
`-gnatp` -> compila sin tener en cuenta las excepciones
el 'gnatmake' compila el principal y las librerías a las que se hace referencia en el principal.

3 Se crea un programa en C (`hola_c.c`) que llame al procedimiento principal (`hola.adb`)

3.1 se utiliza la herramienta `'m68k-rtemscoff-nm'` para saber como se llama al procedimiento principal del programa Ada

```
m68k-rtemscoff-nm hola.o
```

Del listado que se muestre al ejecutar este comando, se buscara el `*.T`
-> *`_ada_hola.T`*

3.2 Crear el programa `hola_c.c`

```
extern void _ada_hola (void);

int main ()
{
_ada_hola ();

return 0;
}
```

4 Compilar los `*.c` con el gcc cruzado

```
gcc68o  –c hola_c.c
gcc68o  –c basic_console_io_c.c
```

Así se genera un `hola_c.o` y un `basic_console_io_c.o` que se enlazarán junto con los demás `*.o` en el siguiente orden:

```
5    gld68o    hola_c.o hola.o basic-console_io.o
     basic_console_io_c.o
```

Generando un hola_c.hex listo para ser transmitido al SoccerBot

Ahora tan sólo hay que transmitir al SoccerBot cualquier archivo `*.hex`, por ejemplo el `hola.hex`. Se presentan dos alternativas, mediante el cable serie RS-232 y mediante el cable de depuración utilizando el depurador cruzado.

La configuración del depurador se desarrolla en el apartado siguiente: depurador cruzado.

6.5 DESCARGA DE APLICACIONES POR LA LINEA SERIE

Se deberá configurar tanto en el PC (*host*) como en el SoccerBot (*plataforma de desarrollo)*, en parámetros tales como la velocidad de transmisión y el protocolo de comunicación en puerto COM1. Se van a realizar las transmisiones a 115200 bauds y con el protocolo `rts/cts`. Si se observa que hay una pérdida de información de los caracteres transmitidos se variará la velocidad a la inmediatamente inferior.

Cuando se usa el protocolo RTS/CTS para el control del flujo hace que sólo se pueda producir un “informe de progreso de la transmisión”. El uso de esta característica, sin embargo, significa un incremento del tiempo de transmisión. De cualquier modo, las opciones son: transferencia más lenta con un “informe de progreso”, o la transferencia más rápida sin ello.

Cuando el flujo de control es RTS/CTS, la rutina de descarga apaga la señal *‘Clear To Send’* (CTS) para prevenir envíos desde el PC mientras se está escribiendo en el 'display' del SoccerBot.

6.5.1 CONFIGURACIÓN DEL PUERTO SERIE EN LINUX

```
stty –a /dev/ttyS0
```

Así se obtiene la configuración actual del puerto serie, tan sólo hay que preocuparse de la velocidad de transmisión y del protocolo de comunicación, se hace lo siguiente:

```
stty speed 115200 < /dev/ttyS0
```

Así se cambia la velocidad de transmisión del puerto, esta tiene que ser la misma que tenga asignada el SoccerBot. Se podrá elegir entre las siguientes velocidades: 9600, 19200, 38400, 57600 y 115200.

```
stty crtscts < /dev/ttyS0
```

Ahora se cambia el protocolo de comunicación entre el PC y el SoccerBot, podremos elegir entre none y `rts/cts`

Para transmitir el archivo hola.hex se utilizará el comando *cat:*

```
cat hola.hex > /dev/ttyS0
```

6.5.2 CONFIGURACIÓN DEL SoccerBot

Se selecciona Set/Ser en el menú del EyeBot para configurar el puerto a utilizar, la velocidad y protocolo de transferencia de datos.

```
SerSetup:
Serial Speed: *
   9600
Handshake:
   RTS/CTS

 +  -   Nxt END
```

Se selecciona Usr/Ld, para dejar al EyeBot en modo '*start server*'

```
Ready to
download at
115200, RTS/CTS:
Downloading:


           END
```

Se transmite el archivo hola.hex al SoccerBot

```
Ready to
download at
115200, RTS/CTS:
Downloading:
     hola.hex
Bytes:   23677
Ld Run Rom END
```

Llegados a este punto el EyeBot avisa cuando se ha completado la transmisión de los datos. Si ocurriera algún error, el programa lo mostraría.

```
User Prog.:

Program loaded:
    hola.hex


Ld Run Rom END
```

6.6 DEPURADOR CRUZADO

El depurador es otra herramienta necesaria dentro del entorno de desarrollo de MaRTE OS. El depurador cruzado utiliza el puerto BDM del robot. Debemos incluir en linux el soporte para este tipo de puerto. Debemos crear un nuevo módulo que añadiremos a directorio *../dev/* que corresponde a los diferentes módulos que maneja el PC.

El depurador cruzado también nos permitirá la descarga de programas directamente al robot, gracias a su BDM y al cable que conecta el puerto paralelo con el conector de 10 pins situado en la reverso del EyeBot.

El depurador a instalar será el *gdb-4.17* que al igual que el compilador pertenece a GNU.

6.6.1 INSTALACIÓN DEL DEPURADOR GDB-4.17

Los pasos que se han dado para la instalación del depurador son los siguientes:
1º - se crea y añade el módulo que va a identificar al depurador, soporte para BDM en Linux, en el PC en *../dev/*

El depurador se instalará una vez se haya instalado el compilador cruzado, por lo que los archivos comprimidos correspondientes al compilador y sus parches ya los debemos tener descargados. Para comenzar debemos encontrarnos en una consola con permisos de súper usuario.

```
cd /usr/src/
tar –zxf ~/cruzado/archivo/gdb-4.17-bdm-
          patches.tar.gz
cd gdb-4.17-bdm-patches/bdm-driver
```

Debemos modificar el 'Makefile':

```
/sbin/rmmod bdm
/sbin/insmod bdm
```

En la carpeta `/tools/gdb-4.17-bdm-patches/bdm-driver/` se encuentran las fuentes listas para compilar el módulo BDM. Es necesario que este módulo sea compatible con la versión de kernel del sistema operativo, por

ello se debe compilar el módulo. La primera operación que se ha de ejecutar es MAKEDEV, el cual crea los dispositivos :

```
pd_bdm0, 1, 2
icp_bdm0, 1, 2
```

Una vez modificado el 'Makefile' ya lo podemos ejecutar correctamente. Para esto necesitaremos tener permisos de súper usuario.

```
#     ./MAKEDEV
```

Con esta última línea lo que conseguiremos será ejecutar el script que instale el módulo que corresponderá al depurador. Es decir, creará lo siguiente:

```
/dev/pd_bdm0,1,2
```

Que corresponderá a la preinstalación para el BDM de dominio público, este será el que corresponda al módulo que tenemos en el robot.

```
/dev/icp_bdm0,1,2
```

Éste otro corresponde a la preinstalación para el BDM específico de motorola Con esto ya queda preparado el PC para poder comenzar la instalación del depurador cruzado.

```
#     make
```

Esta instrucción se encarga de compilar el módulo bdm, así como de copiar el módulo ya compilado (bdm.o) en la carpeta donde se le haya indicado en el *'Makefile'.*

2º - preparación e instalación de los archivos que corresponden al depurador

```
      cd /usr/src
#     tar –zxf ~/cruzado/archivo/gdb-4.17.tar.gz
       cd gdb-4.17/
      patch –p1 <~/cruzado/archivo/gdb-4.17-bdm-
      patches/gdb-4.17.patch
      make
#     make install
```

Con esto el programa del depurador queda instalarlo, ahora sólo hay que configurarlo para que utilice el módulo que se ha creado en el PC para el depurador.

Para comenzar a funcionar con el depurador debemos ejecutar el comando del gdb tal y como sigue:

```
            m68k-bdm-coff-gdb
(gdb)       _
```

activamos el puerto paralelo para el gdb, con esto nos conectamos al 'driver' del bdm

```
(gdb)       target bdm /dev/pd_bdm0
```

deshabilitamos el 'caching' de los test iniciales

```
(gdb)       set remotecache off
```

Las siguientes líneas seleccionan el tiempo de espera antes de realizar la conexión. Debe hacerse para permitir el acceso a memoria y la inicialización de la SIM. También se selecciona la velocidad de comunicación entre el driver bdm y el SoccerBot

```
(gdb)       bdm_timetocomeup 600000
(gdb)       bdm_autoreset off
(gdb)       bdm_setdelay 70
```

La siguiente instrucción dejará el 'LCD' en blanco

```
(gdb)       bdm_reset
```

la activación de los siguientes registros a 5 significa que el depurador podrá acceder al espacio de la memoria en modo supervisor

```
(gdb)       set $sfc=5
(gdb)       set $dfc=5
```

6.6.2 COMANDOS bdmcommands, initgdb & gdb_marte

Con todo esto el gdb ya estaría configurado para poder depurar cualquier programa, el problema es que todos estos comandos se deberían ejecutar cada vez que se inicie el depurador, para evitarlo se ha optado crear un script que se encargue de inicializar el depurador y que se necesite un solo comando para comenzar a depurar cualquier programa que previamente se haya compilado correctamente.

Esto se hace en tres partes:

Se crea un primer script que será la fuente de comandos que se deben ejecutar una vez se haya inicializado el gdb:

bdmcommands contiene lo siguiente:

```
define configura
set remotecache off
bdm_timetocomeup 600000
bdm_autoreset off
bdm_setdelay 70
bdm_reset
set $sfc=5
set $dfc=5
end
```

Un segundo script contendrá los comandos que pueden ejecutarse en la inicialización del depurador y tendrá permiso de ejecución, *initbdm* contiene lo siguiente:

```
file /tmp/a.out
```

Este comando indica donde se encuentra el archivo a depurar. El `a.out` contiene siempre una copia del último *.hex obtenido.

```
br main
```

se instalará un punto de ruptura en el *main*

```
target bdm /dev/pd_bdm0
```

Indicamos que existe una fuente de comandos, que será el primer script

```
source bdmcommands
```

Y un tercer script que contenga el ejecutable del gdb con capacidad de ejecutar el script *initbdm*. Éste se llamará *gdb_marte*** y contiene lo siguiente:

```
m68k-bdm-coff-gdb –x initbdm
```

***gdb_marte* debe tener privilegios de ejecución.

6.6.3 MODO DE EMPLEO DEL DEPURADOR gdb-4.17

Entonces, siempre que se quiera utilizar el depurador los pasos a seguir son los siguientes:

Debemos estar seguros que los tres scripts que se han almacenado en algún directorio que pertenezca al PATH, sino debemos añadir el PATH completo de donde se encuentren creados. Lo más sencillo sería crearlos en el directorio */bin/,* que ya pertenece al PATH.

1 – compilar el programa a depurar
2 - gdb_marte

```
(gdb)
    br in main()
```

Se habrá instalado el 'breakpoint' en el *main*

```
3 – (gdb) configura
```

Configuramos el gdb

```
4 – (gdb) run
```

Comienza la depuración

6.6.4 INSTALACIÓN DEL MÓDULO 'BDM' EN EL PC

Cada vez que se reinicie el PC tendremos que instalar el módulo que corresponderá al controlador bdm, habrá que seguir los siguientes pasos

```
1-    cd /usr/src/gdb-4.17-bdm-patches/bdm-driver
2- #  /sbin/insmod bdm.o
```

6.7 CONCLUSIONES

Partiendo de las herramientas proporcionadas con el SoccerBot se ha desarrollado un entorno para MaRTE OS. Un entorno de desarrollo capaz de soportar aplicaciones en Ada y C, entorno que dispone de herramientas como el compilador(GCC), el enlazador(GLD), ensamblador(GAS), depurador(GDB), generando un código binario para la máquina objetivo (MC68332). El entorno queda preparado para utilizar MaRTE OS (en la orden de enlazado se incluyen las librerías de MaRTE OS).

Se ha demostrado la validez de las herramientas elegidas para generar el entorno de desarrollo.

7 CONCLUSIONES Y TRABAJO FUTURO

7.1 CONCLUSIONES

Se ha desarrollado un entorno de desarrollo cruzado para el MC68332, indispensable para realizar el portado de MaRTE OS a esta plataforma. Por un lado tenemos el compilador cruzado: hemos particularizado el compilador, y el conjunto de herramientas que conforman el entorno de desarrollo, para que codifique MaRTE OS correctamente para MC68332 y además compile, ensamble y enlace aplicaciones y programas escritos en Ada y en C.

Debido a la gran cantidad de archivos y para facilitar la búsqueda y depuración de errores, como complemento al compilador cruzado se desarrolló un depurador cruzado, siempre basándonos en herramientas de código libre.

Se ha adaptado el depurador cruzado GDB, para que a través del cable de depuración y gracias al módulo de depuración en segundo plano del EyeBot, se haya podido establecer una comunicación entre el PC 'host' la plataforma de desarrollo,'target', que nos ha facilitado en gran medida la detección de errores, tanto en la compilación de MaRTE OS (de todos sus librerías y paquetes), como en los ejemplos de aplicaciones, permitiendo una velocidad de transferencia de archivos hacia la plataforma superior a la velocidad máxima que nos permite el EyeBot por el cable serie RS-232 (115200 bauds).

En cuanto a la portabilidad de MaRTE OS al MC68332, se han cumplido todos los requisitos mínimos exigidos, teniendo en cuenta que para la generación de un reloj para el sistema por no contar la plataforma con un dispositivo hardware para esta causa, se ha tenido que implementar por medio de software, manteniendo dentro de unos márgenes razonables para MaRTE OS tanto el tiempo de reprogramación como el periodo máximo de programación del temporizador hardware. También se ha diseñado un sistema manejador de interrupciones, que permitirá instalar rutinas de interrupción al propio usuario. Cabe destacar la función que permitirá el cambio de contexto entre tareas, escrita en el propio lenguaje del MC68332 con lo que reducimos su código a unas pocas líneas.

Se ha podido portar MaRTE OS salvo una pequeña parte de la librería GNULL (librería de bajo nivel GNU), la cual exige ligeras modificaciones para cada plataforma. Éste será el impedimento para que la aplicación de demostración no se encuentre escrita en Ada..

La ventaja más evidente al utilizar herramientas con el código fuente accesible es del tipo económica: nos evitamos los gastos de software que en un proyecto que trabaja con prototipos suele llevar una parte importante del

presupuesto del proyecto. Una segunda ventaja es paradójicamente la fiabilidad, debido a la accesibilidad que tenemos sobre el código fuente y la enorme aceptación de estos productos en los que se añaden mejoras continuamente.

Con la aplicación de demostración no sólo queda demostrado que es posible portar MaRTE OS a otras arquitecturas, sino que además hemos sido capaces de amoldarlo a una plataforma que mantiene un software base, evitando conflictos entre ambos, y aunque se utilizan las funciones (de la librería de funciones RoBIOS) que controlan los periféricos, todos los procesos que corresponden a un sistema operativo los desempeña MaRTE OS mientras se ejecuta la aplicación.

7.2 TRABAJO FUTURO

Como trabajo futuro se plantea la actualización del entorno de desarrollo, esto es, trabajar con versiones actualizadas de las herramientas utilizadas (p.e. gnat 3.13 o 3.14) ya que existen versiones de MaRTE OS que lo soportan. Sería una notable mejoría la adaptación del la librería de runtime del GNAT elegido y la librería estándar del GNAT para que así MaRTE OS pueda ver cumplidos todos sus objetivos.

También como trabajo futuro se podría llegar a desarrollar *drivers* para controlar los periféricos del SoccerBot.

Con el desarrollo de MaRTE OS en el microcontrolador MC68332, y la rápida expansión de los sistemas empotrados en gran variedad de campos, no sería descabellado pensar que MaRTE OS fuera el sistema operativo a utilizar en juguetes, electrodomésticos, sistemas empotrados en automóviles, etc... Que aunque varíen el microcontrolador del sistema, no sería difícil ahora desarrollar una versión de MaRTE OS que se adapte al microcontrolador elegido (la versión de MaRTE OS que se obtiene de este proyecto serviría en todos los microcontroladores de la familia motorola mc68300).

Como ya se comentó anteriormente, no era el propósito de este proyecto el utilizar todos los sensores y actuadores que contenía el SoccerBot, pero parece atractiva la opción de poder desarrollar aplicaciones con algunos de ellos, como son el puerto de infrarrojos, el sensor de infrarrojos junto con el control remoto y la cámara digital a color. Con el cable serie RS-232 se consiguen velocidades de transmisión de datos por el puerto de 115200 bauds, sería interesante utilizar el puerto de infrarrojos del SoccerBot con el dispositivo conectado al puerto del PC de desarrollo. Sería útil no sólo para la descarga de programas de aplicación en el SoccerBot, sino que se podrían recibir imágenes en el PC tomadas con la cámara del SoccerBot.

I BIBLIOGRAFÍA

[MAR02] Mario Aldea Rivas, tesis doctoral, "planificación de tareas en sistemas operativos de tiempo real estricto para aplicaciones empotradas" noviembre de 2002,

[VDC00] Venture Development Corporation, "http://www.vdc-corp.com/embedded/press/archives/00/pr00-06".

[WSTS] World Semiconductor Trade Statics. "http://www.wsts.org/."

[DSECUPM] Departamento de Sistemas Electrónicos y de Control, Universidad Politécnica de Madrid, "Introducción a los microcontroladores"

[TB03] Thomas Bräunl, Univ. of Western Australia, "http://robotics.ee.uwa.edu.au/eyebot/".

[GNU02] GNU-project, http://www.gnu.org/

[MaR03] MaRTE OS, "http://marte.unican.es/"

[IMM03] Informatics And Mathematical Modelling, Technical University of Denmark," http://www.imm.dtu.dk/cs/systemonchip.htm."

[CSIC00] Consejo Superior de Investigaciones Científicas**,** "http://www.dicat.csic.es/epsonesp.html"

[MCPU] Reference Manual MC68332 and CPU32, "http://robotics.ee.uwa.edu.au/eyebot/".

[TPURM] Motorola TPU Reference Manual, "http://robotics.ee.uwa.edu.au/eyebot/".

[SIMRM] Motorola SIM Reference Manual, "http://robotics.ee.uwa.edu.au/eyebot/".

[RTE03] RTEMS, "http://www.rtems.com/onlinedocs/releases/4.5.1-pre3/rtemsdoc/html/started_ada/started_ada00013.html"

[PSX96] ISO/IEC Standard 9945-1:1996, "Information Technology – Portable Operating System Interface (API) [C Languaje]". Institute of Electrical and Electronic Engineers, 1996

[PSX96] ISO/IEC Standard 1003.1:2001, “Standard for Information Technology – Portable Operating System Interface (POSIX) - PART 1: System Application Program Interface (API) [C Languaje]”. Institute of Electrical and Electronic Engineers, 2001

[C++98] International Standard ISO/IEC 14882:1998(E). “Programming languages – C++”, ISO/IEC, 1998

[ADA95] International Standard ISO/IEC 8652:1995 (E): ”Information Technology – Programming Languajes – Ada. Ada Reference Manual”.1995