ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

ENTORNO PARA LA INSTALACIÓN Y UTILIZACIÓN DE MANEJADORES DE DISPOSITIVOS EN MARTE OS

Para acceder al Título de

INGENIERO DE TELECOMUNICACIÓN

Francisco Guerreira Payo

Marzo 2003

INDICE

Capítulo 1
Introducción

1. 1. - Origen del sistema operativo MaRTE OS. (MaRTE y el estándar POSIX) 1

1. 2. - Características y situación actual de MaRTE OS. 4

1. 3. - Uso de 'drivers' dentro de los sistemas operativos. 7

1. 4. - Objetivos del proyecto. 8

1. 5. - Organización de la memoria del proyecto 10

Capítulo 2
Diseño del sistema de archivos para dispositivos de entrada/salida en MaRTE.

2. 1. - Idea general sobre el nuevo sistema de archivos de dispositivos de MaRTE. 11

2. 2. - Comparación con sistemas de archivos de dispositivos en otros sistemas operativos: Linux, Lynx, VxWorks.

2.2.1. Características de Lynx OS 13
2.2.2. Características de VxWorks 14
2.2.3. Características de Linux 15

2. 3. - Diseño y características del sistema de archivos a desarrollar en MaRTE.

2.3.1. Características básicas en el diseño del sistema de archivos de dispositivos de MaRTE OS 16
2.3.2. Explicación y diseño del sistema de archivos de dispositivos de MaRTE OS 17

2. 4. - Implementación del sistema de archivos de MaRTE.

2.4.1. Tipos de datos internos al sistema de archivos de dispositivos 20
2.4.2. Diseño de las tablas usadas para el manejo de información referida al sistema de archivos de dispositivos 23
2.4.3. Núcleo central del sistema de archivos de dispositivos de MaRTE (paquete 'Kernel.File_System') 26
2.4.4. Interfaz POSIX para el paso desde el 'kernel' hacia la aplicación 29
2.4.5. Funciones genéricas (uso de la función 'Ioctl') 33
2.4.6. Variaciones sobre el código inicial de MaRTE 34

2. 5. - Resumen y comentarios acerca del diseño del sistema de archivos de dispositivos. 40

Capítulo 3
Modo de uso de 'drivers' de dispositivos, a través del sistema de archivos de MaRTE.

3. 1. - Instalación de 'drivers' en la tabla de dispositivos. 42
3.1.1. El paquete 'Devices_Table'. 46
3.1.2. Modo de introducir un nuevo 'driver' en la tabla 48
3. 2. - Interfaz de un 'driver' de prueba realizado en Ada. 51
3. 3. - Interfaz de un 'driver' de prueba realizado en C. 55
3. 4. - Resumen acerca del primer uso de 'drivers' de dispositivos.

Capítulo 4
Ejemplos de 'drivers' de dispositivos y aplicaciones desarrolladas para su uso.

4. 1. - Primer contacto con las funciones de entrada/salida. 56
4.1.1. 'Driver' con cola circular para almacenamiento de datos con tamaño configurable desde la aplicación 58
4.1.2. Aplicación para un primer uso de funciones de entrada/salida, utilizando el 'driver' con la cola circular de tamaño configurable
4. 2. - Puerto paralelo. 61
4.2.1. 'Driver' para el puerto paralelo. 64
4.2.2. Aplicaciones para el uso del puerto paralelo.
4. 3. - Puerto serie. 71
4.3.1. 'Driver' para el puerto serie. 77
4.3.2. Aplicaciones para el uso del puerto serie.
4. 4. - Sincronización entre tareas, con uso de funciones de entrada/salida bloqueantes. 79
4.4.1. 'Driver' con cola circular de tamaño fijo, con funciones bloqueantes controladas con variables condicionales. 82
4.4.2. Aplicación para la sincronización entre 'threads'. 85
4. 5. - Resumen de la implementación de los 'drivers' de dispositivos.

Capítulo 5
Conclusiones del proyecto. 86
5. 1. - Comentarios y conclusiones. 87
5. 2. - Línea futura de trabajo.

Referencias

Capítulo 1

Introducción

1. 1. - Origen del sistema operativo MaRTE OS. (MaRTE y el estándar POSIX)

El origen y desarrollo del sistema MaRTE provienen del resultado de una tesis doctoral que se desarrolló recientemente dentro del Grupo de Computadores y Tiempo Real, adscrito al Departamento de Electrónica y Computadores de la Universidad de Cantabria. Esta tesis doctoral *"Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas"* [1] propone una interfaz para sistemas empotrados bajo la cual sea posible desarrollar aplicaciones en las que se puedan definir internamente algoritmos de planificación de tareas, utilizando para su implementación las bases establecidas en el estándar POSIX de sistemas operativos de tiempo real.

El hecho de cumplir la integración y compatibilidad con este estándar POSIX hace que se facilite el uso de la interfaz para programadores familiarizados con este estándar, en gran crecimiento debido a su alto grado de aceptación entre los fabricantes de sistemas operativos. Además, incluso se podría tratar de realizar su incorporación a futura revisiones del estándar POSIX, partiendo desde una base inicial bien establecida.

Para situar el estándar POSIX, comentemos sus orígenes. Se desarrolló en un principio para unificar criterios en cuanto a los diferentes sistemas operativos, y con ello lograr la portabilidad de las aplicaciones a nivel de código fuente entre los distintos sistemas operativos, facilitándose así la adaptación de los programadores a nuevos entornos de trabajo. La sucesiva aprobación, primero del estándar POSIX 1003.1b [2] en el año 1993 (extensiones de tiempo real), también conocido como POSIX.4, y posteriormente del estándar POSIX 1003.1c [3] en el año 1995 (extensión de '*threads*'), también llamado POSIX.4a, supusieron que ya se pudiera desarrollar aplicaciones con requisitos de tiempo real sobre sistemas operativos POSIX.

El POSIX 1003.1b (extensiones de tiempo real) aportó al estándar dos tipos de servicios con los cuales ya se podían ejecutar en un sistema POSIX aplicaciones que fueran capaces de cumplir requisitos temporales:

- Servicios para facilitar la programación concurrente.
- Servicios necesarios para obtener un tiempo de respuesta predecible.

El POSIX 1003.1c (extensiones de '*threads*') supuso el comienzo del uso de '*threads*', a los que también se les podría calificar con el término '*procesos ligeros*', como unidades de concurrencia del POSIX, además de los ya existentes procesos. Con ello se mejoró la eficacia, ya que los '*threads*' suponen un uso de espacio de direcciones compartido, teniendo a la vez diversos flujos de control concurrentes dentro de un único proceso.

Pero un sistema que implemente el estándar POSIX de manera completa es un sistema demasiado grande para su uso en pequeños sistemas empotrados y por ello se aprobó en el año 1997 otro estándar, POSIX.13 [4], en el cual se definieron cuatro subconjuntos de servicios del sistema operativo (*"perfiles de entornos de aplicación"*) para que distintas plataformas pudieran usar distintos perfiles. Sus diferencias se encuentran en la presencia o no presencia de un sistema de ficheros complejo y jerárquico, y en la utilización o no utilización de múltiples procesos.

Perfil de aplicación	***Sistema de ficheros***	***Múltiples procesos***	***Uso de threads***
Sistema de Tiempo Real Mínimo	✗	✗	✓
Controlador de Tiempo Real	✓	✗	✓
Sistema de Tiempo Real Dedicado	✗	✓	✓
Sistema de Tiempo Real Multi-Propósito	✓	✓	✓

Figura 1 . Características propias de los '*perfiles de aplicación*' POSIX.13

De todos ellos, el perfil que interesa para la intenciones de desarrollo de MaRTE OS es el *"Sistema de Tiempo Real Mínimo"*, usado en aplicaciones empotradas pequeñas, que soportan unos servicios POSIX mínimos para reducir el tamaño de las aplicaciones desarrolladas bajo este perfil. Las características que lo diferencian del resto de los perfiles se encuentran en el hecho de que no requiere el soporte para múltiples procesos ni tampoco un complejo sistema de ficheros jerárquico, eliminándose así la mayor parte del tamaño y la complejidad del POSIX completo.

En el desarrollo de la citada tesis doctoral [1], una vez situada la base que conforma el estándar POSIX para la implementación de esta interfaz para definir los algoritmos de planificación, se llega a la conclusión de que no se encuentra ningún sistema operativo POSIX apropiado para poder realizar la necesaria implementación de la interfaz mencionada, de modo que se diseña e implementa un nuevo sistema operativo de tiempo real bautizado como MaRTE [5] (*"**M**inim**a**l **R**eal-**T**ime Operating System for **E**mbedded Applications"*), sobre el cual se va a desarrollar toda la base de este proyecto.

El código de MaRTE se ha realizado bajo las condiciones de libre distribución de GPL ("***G****NU's* ***P****ublic* ***L****icense*"). Este uso de las condiciones GPL de libre *software* [6] hace del MaRTE OS un sistema operativo de código abierto, de gran utilidad para docencia y futuras investigaciones en sistemas de tiempo real.

Esas condiciones establecidas para el proyecto GNU suponen que el propósito de la distribución sea la utilidad general, pero este hecho de ser *software* libre implica que no se puede ofrecer ninguna garantía de su correcto funcionamiento, incluso las implícitas '*garantía mercantil*' o la '*garantía de ajuste a un propósito particular*'. La libre distribución hace que se rompa con la tendencia de los sistemas propietarios que realmente siguen el estándar POSIX de manera completa, pero que no ofrecen la posibilidad de tener acceso al código del sistema operativo.

Algunos otros sistemas operativos que ofrecen código de libre distribución que se podrían considerar como válidos para la interfaz POSIX, como pueden ser RTEMS [7] , o RT-Linux [8] , no siguen en su diseño el modelo de '*threads*' concreto que usa el estándar POSIX, con lo cual no se pueden considerar como sistemas operativos válidos para la necesidad impuesta en la tesis doctoral [1], de la que se parte como origen.

Este nuevo sistema operativo MaRTE, desarrollado bajo el perfil *"Sistemas de Tiempo Real Mínimo"* del POSIX.13, se ha implementado utilizando el lenguaje Ada95, lenguaje que se caracteriza por proporcionar una gran fiabilidad y además, una sencilla legibilidad y capacidad de generar código reutilizable.

La fiabilidad que aporta el lenguaje Ada viene dada por la posibilidad de corregir numerosos errores en tiempo de compilación, debido al estricto uso de los tipos de datos que se debe hacer, e incluso en tiempo de ejecución, con el uso de excepciones, que tienen la capacidad de ejecutar acciones correctoras para subsanar el error cometido.

La sencilla legibilidad y la posibilidad de generar código reutilizable de manera cómoda vienen dadas por el uso de los paquetes, como módulos compactos de código que permiten un alto nivel de abstracción y el ocultamiento de la información a usuarios que no necesiten saber del contenido del código.

1. 2. - Características y situación actual de MaRTE OS.

El sistema operativo MaRTE se caracteriza principalmente por ser un sistema operativo diseñado bajo las normas del subconjunto mínimo definido en el POSIX.13, y a raíz de esta característica primordial, se tienen otra serie de características asociadas, como:

- Está pensado para aplicaciones principalmente estáticas
- Presenta tiempos de respuesta acotados en todos sus servicios
- Existe un sólo espacio de direcciones de memoria, compartido tanto por el núcleo como por la aplicación
- Se permite la ejecución de aplicaciones desarrolladas tanto en lenguaje Ada como en lenguaje C
- Incluye gestión de memoria dinámica simplificada
- Es portable para el uso en distintas arquitecturas
- Presenta un '*núcleo monolítico*', con la mayor parte de las llamadas al sistema utilizando accesos a secciones críticas
- Toma la forma de una librería para ser enlazada con la aplicación

Como se comentó en el apartado anterior, MaRTE sigue las directrices marcadas por el estándar POSIX.13 para *"Sistemas de Tiempo Real Mínimo"*, referido a pequeños sistemas empotrados, y es por ello que se tienen que soportar una serie de servicios para cumplir con las necesidades impuestas por el estándar. Algunos de ellos son los siguientes:

- Soporte para ejecución concurrente.
- Planificación de threads.
- Sincronización entre threads.
- Señales.
- Acceso a dispositivos.
- Servicios de temporización.
- Paso de mensajes.
- Servicios de configuración.
- Gestión de memoria dinámica.

Además también se soportan algunos servicios no necesarios para pequeños sistemas empotrados, pero que deben ser incluidos en el sistema MaRTE para mantener la posible compatibilidad con el resto de los perfiles:

- Bloqueo de memoria.
- Entrada/Salida sincronizada y no sincronizada.
- Objetos de memoria compartida.

El sistema así engendrado, bajo el desarrollo propio de la tesis, nos ofrece un núcleo (*"kernel"*) que da soporte a dos interfaces POSIX, la original, para el POSIX-C y también para POSIX-Ada, pudiéndose ejecutar aplicaciones empotradas desarrolladas tanto con '*threads*' en C como con tareas en Ada.

A continuación, se puede observar la arquitectura con la cual se trabaja cuando se ejecuta una aplicación en el sistema operativo MaRTE, tanto para las aplicaciones en Ada como para las desarrolladas en lenguaje C. Entre ambas se tiene la diferencia de la capa intermedia que hace de interfaz de unión entre la aplicación y la parte más interna del núcleo del sistema operativo.

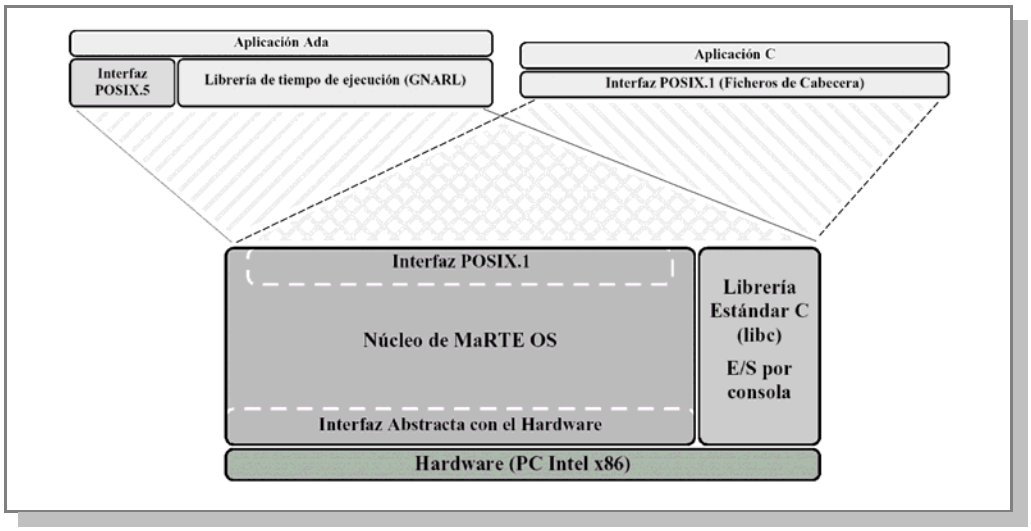

Figura 2 . Arquitectura de las aplicaciones ejecutándose sobre MaRTE OS

Para realizar el entorno de creación, carga y desarrollo de las aplicaciones del sistema MaRTE OS se utiliza un equipo con Linux bajo un entorno cruzado especifico, con los compiladores GNAT y GCC y una serie de archivos de órdenes (*scripts*) desarrollados en lenguaje Perl que facilitan la generación y ejecución de dichas aplicaciones.

En este equipo con Linux se encuentra el núcleo del sistema MaRTE además de una serie de librerías, que son enlazadas de manera conjunta con el código de la aplicación que pretende crearse, tras haberse hecho la compilación correcta del código.

Las aplicaciones se generan en el equipo Linux, pero han de ser cargadas de modo remoto desde el sistema empotrado, y para ello se usa una utilidad especial que carga la aplicación a través de la red de comunicaciones. Esta utilidad se conoce como ***'NetBoot'***, y se caracteriza por realizar un arranque remoto de una aplicación dentro de una red Ethernet, configurándose para ello diferentes parámetros de manera inicial.

Para intentar comprender mejor cómo se realiza esta carga y ejecución de las aplicaciones, veamos a través de un gráfico cómo se comunican los equipos.

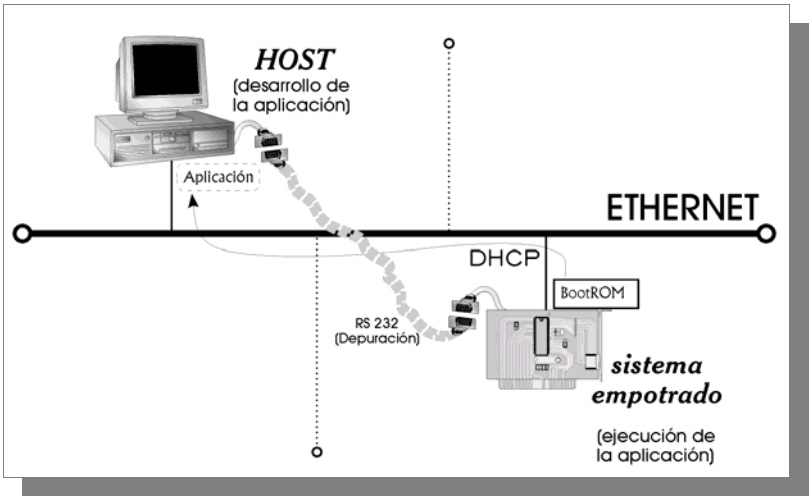

Figura 3 . Arranque remoto de la aplicación MaRTE desde el sistema empotrado

Se observa que la máquina empotrada recibe la aplicación a través de la red Ethernet, a la cual están unidos tanto dicho terminal remoto como el sistema que da el soporte para permitir el desarrollo de la aplicación. Con el tiempo, cuando la aplicación quede completamente desarrollada y probada, el sistema empotrado puede aislarse, y realizar la carga de la aplicación a través de un arranque propio (por memoria Flash, mediante diskette, ...)

Aparte también se tiene una comunicación entre ambos equipos establecida a través del puerto serie, utilizada para realizar la depuración de la aplicación en tiempo de ejecución a través de la utilidad GDB, cargada desde el equipo Linux sobre el que se desarrollado la aplicación en primera instancia.

Actualmente, el sistema MaRTE es un producto perfectamente útil para el desarrollo de aplicaciones industriales empotradas, y también es una gran herramienta para el estudio e investigación de los sistemas operativos de tiempo real.

De todas maneras, MaRTE OS es una utilidad en constante mejora, y algunos otros aspectos sobre los que se pretende realizar un mayor estudio son los siguientes:

- Desarrollo de un entorno para uso de manejadores de dispositivos.
- Portado del núcleo de MaRTE a otra arquitectura basada en el controlador MC68332.
- Desarrollo y uso de librerías gráficas para posible utilización en consolas.
- Extensión a sistemas multiprocesadores.
- Utilización en sistemas reales de control industrial.

1. 3. - Uso de '*drivers*' dentro de los sistemas operativos.

Los '*drivers*' (cuya traducción aproximada podría ser 'manejadores de dispositivos') son módulos software especiales utilizados por los sistemas operativos para hacer de interfaz entre el propio sistema operativo y los posibles dispositivos asociados al sistema que se pueden tener. En general, estos dispositivos suelen ser dispositivos '*hardware*', es decir dispositivos físicos externos al sistema, que se utilizan para aumentar la funcionalidad del propio sistema, como pueden ser los puertos de comunicaciones, el teclado, o nuevas tarjetas con una utilidad específica; pero también se pueden tener dispositivos '*virtuales*', que a través del propio '*software*' del sistema obtienen el soporte para trabajar como nuevos dispositivos ajenos a la estructura inicial del sistema, como pueden ser las '*tuberías*', (a las que se suele conocer como '*pipes*') para comunicaciones entre procesos; o las colas (también llamadas '*buffers*'), para el almacenamiento de datos.

En muchos otros sistemas operativos, como pueden ser el sistema de tiempo real LynxOS [9] o también el sistema operativo Linux, un sistema operativo más de propósito general, los '*drivers*' se suelen escribir diferenciándose en dos clases distintas:

- *'drivers' de caracteres* : para dispositivos más simples, que no necesitan mucha complejidad de diseño.
- *'drivers' de bloques* : para dispositivos de almacenamiento masivo, manejándose la información por bloques de tamaño fijo.

Otro aspecto a destacar es cómo se realiza la identificación de los '*drivers*' y de los dispositivos que se utilizan en el sistema operativo. Siguiendo con los ejemplos de sistemas operativos mencionados antes, es decir, en estos sistemas LynxOS y Linux se hace la identificación a través de una serie de números con los que se consigue diferenciarlos, conocidos como *números mayores y menores*.

Los números mayores se suelen asociar a unidades funcionales separadas, que la mayor parte de las veces se corresponden con dispositivos físicos independientes, aunque no es estrictamente necesario. Esta unidad funcional independiente normalmente dispone de acceso a registros de control propios, direcciones en memoria para la entrada/salida y también de vector de interrupción propio.

Los números menores son utilizados para diferenciar subunidades internas a un dispositivo mayor, de los comentados anteriormente. Normalmente todos los dispositivos menores identificados con el mismo número mayor suelen compartir recursos entre ellos, como los registros de control o el vector de interrupción. También se puede utilizar la diferenciación por números menores para identificar diferentes usos o funciones del dispositivo mayor.

Para realizar las operaciones internas dentro de un '*driver*' se establecen unas funciones específicas, con una especificación general para todos los dispositivos, pero cada una de ellas con un comportamiento interno distinto para cada dispositivo, adecuandose así a las necesidades que deben implementar cada uno de ellos.

En general, las principales funciones de entrada/salida que se ofrecen en casi todos los sistemas operativos para utilizarse dentro de los '*drivers*' son las siguientes:

- '`open`' : para la apertura de un nuevo fichero de dispositivo.
- '`close`' : para el cierre del fichero de dispositivo, que debe encontrarse '*abierto*', con la consiguiente liberación de recursos necesaria.
- '`read`' : para la lectura de los datos que ofrece el dispositivo, para poder ser pasados al sistema, y utilizados por el mismo.
- '`write`' : para la escritura de datos del sistema sobre el dispositivo, en muchos casos en los registros habilitados para el almacenamiento de esta información.
- '`ioctl`' : para establecer parámetros de configuración y control del dispositivo, o también para obtener información del estado en el que se encuentra.

1. 4. - Objetivos del proyecto.

El sistema operativo MaRTE, tal y como se ha dicho, es adecuado para pequeñas aplicaciones empotradas, con lo cual entre sus características claramente bien definidas se encuentra el hecho de que no necesita soportar un sistema de archivos jerárquico y complejo.

Pero los servicios propios del perfil mínimo incluyen una parte de acceso a dispositivos. Aunque las operaciones POSIX no requieren de drivers de dispositivos, en sí mismos, resulta de gran utilidad ofrecer una forma uniforme para que cualquier usuario pueda utilizar los dispositivos y hacer uso de ello a través de los drivers, sin tener que entender o modificar los detalles internos del núcleo del sistema operativo.

Al tratarse de sistemas empotrados pequeños no se utilizará un sistema de ficheros, tal y como se conoce normalmente, con accesos a ficheros de disco y directorios, sino que se realizará un sistema de ficheros almacenado en memoria que de acceso a los dispositivos, perfectamente conocidos, y que sólo será cargado en el inicio de la aplicación, sin poderse generar en ningún caso nuevos ficheros de dispositivo de manera dinámica desde la aplicación.

Estos ficheros almacenados en memoria representarán cada uno de ellos a cada dispositivo que se desee utilizar desde el sistema, y cada uno de ellos tendrá sus propias funciones de entrada/salida; pero todas ellas deben seguir una única especificación, para poder ser ofrecidas al sistema.

Este va a ser el principal cometido de este proyecto: realizar todo el diseño necesario para tener un acceso completo a todos los posibles dispositivos, a través de un sistema de ficheros simplificado, propio del sistema operativo MaRTE. Además se realizarán una serie de *'drivers'* (manejadores específicos de cada uno de los dispositivos), preparados para interaccionar con el núcleo del sistema a través de unas funciones de entrada/salida.

Con todo esto, los objetivos claramente definidos del proyecto se pueden situar en tres niveles bien diferenciables:

- Realizar un diseño completo del sistema de archivos de dispositivo, que se incorpore al núcleo del sistema operativo MaRTE, con el cual se puedan utilizar las primitivas de entrada/salida definidas en el estándar POSIX, tanto para el POSIX-C como el POSIX-Ada, situándonos a nivel de desarrollo del núcleo del sistema operativo MaRTE.

- Diseñar una interfaz que realice la interconexión entre las funciones de entrada/salida propias de cada dispositivo y las funciones de entrada/salida propias del sistema operativo, trabajando a nivel de enlace entre el *'kernel'* del sistema MaRTE y el usuario que debe programar '*drivers*' en MaRTE.

- Desarrollar una serie de '*drivers*' para los dispositivos más comunes con los cuales comprobar el correcto funcionamiento de los dos niveles previamente comentados, y dar una primera base para un desarrollo futuro más amplio de distintos '*drivers*' de dispositivos, situando este desarrollo a un nivel de usuario que actúe como programador de '*drivers*'.

1. 5. - Organización de la memoria del proyecto

La clara separación en tres niveles del trabajo a desarrollar hace que la organización de la memoria se pueda separar en tres capítulos bastante independientes entre sí. Tras estos capítulos se concluirá con un resumen de objetivos cumplidos, y futuras perspectivas de trabajo que quedan abiertas.

Para comenzar, en el capítulo 2 se comentará en profundidad el diseño realizado para implementar el sistema de archivos de MaRTE que se requiere. En principio se comentan los cambios de código de la estructura original de los paquetes que se tienen de la versión inicial, para ya luego empezar con el diseño y la implementación de los paquetes desarrollados propiamente para este uso. Se recogerán después una serie de diferencias que puede tener este nuevo sistema de archivos de MaRTE con los sistemas de archivos de otros sistemas operativos comunes y conocidos, para terminar con un breve resumen del capítulo en el que se comentan pequeñas conclusiones obtenidas y comentarios a este diseño.

En el capítulo 3 se va a dar una perspectiva de cómo se pueden utilizar los *'drivers'* a través del sistema de ficheros de MaRTE. Esta forma de uso de los *'drivers'* se puede simplificar al hecho de introducir una serie de datos dentro de unas tablas de datos, que luego el sistema utilizará para obtener toda la información necesaria de cada dispositivo. Para poder entender cual es la forma práctica de realizar este manejo de los datos necesarios para el uso de *'drivers'* de dispositivos, se realizarán dos sencillos ejemplos prácticos, mediante los cuales se podrá comprender de manera simple cómo se puede utilizar un *'driver'* realizado en lenguaje Ada, y también cómo hacerlo para un *'driver'* realizado en C, ya que para ambos lenguajes se tienen distintas formas de utilización.

Como última parte a comentar, en el capítulo 4 se recogerá toda la información acerca de diferentes *'drivers'* de dispositivos desarrollados en el proyecto, que nos servirán como comprobación práctica de que el sistema de archivos tiene un correcto funcionamiento, y que la interfaz de interconexión entre los distintos niveles se comunica de manera adecuada. Para ello se realizarán *'drivers'* para el puerto paralelo y puerto serie, como ejemplo de dispositivos hardware de uso más común; y además se realizarán un par de *'drivers'* con dispositivos "virtuales", que utilizarán unas colas circulares, internas al *'driver'*, para el almacenamiento de datos a semejanza de ficheros de almacenamiento. Uno de los *'drivers'* se realizará con una cola dinámica de tamaño variable, y otro de ellos realizado para la comprobación del uso de variables condicionales, elemento clave en la sincronización entre '*threads*', dentro del estándar POSIX.

Al final de la memoria, como se comentó antes, en el capítulo 5 se recogen una serie de conclusiones y comentarios, y se dan unas pequeñas ideas acerca de posibles líneas de trabajo futuro que quedan abiertas con este proyecto. También se incluye al final la relación de las distintas referencias de las que se ha obtenido información valiosísima tanto para el desarrollo del proyecto, como para la realización de esta memoria.

Capítulo 2

Diseño del sistema de archivos para dispositivos de entrada/salida en MaRTE.

2. 1. - Idea general sobre el nuevo sistema de archivos de dispositivos de MaRTE.

Este capítulo pretende ofrecer las primeras nociones sobre el uso de los archivos de dispositivos en MaRTE OS, debiéndose para ello partir de la idea inicial de crear un entorno apropiado para el uso por parte del sistema MaRTE de los referidos archivos de dispositivo. Este entorno no tiene ninguna forma concreta establecida, de manera que su diseño no se debe basar sobre ninguna idea ya implementada. Pero como bien se ha comentado, MaRTE OS es un sistema POSIX mínimo, por lo que se debe seguir un diseño basado en las interfaces que ofrece el estándar POSIX.

La idea principal a tener en cuenta en este nuevo sistema de archivos para dispositivos es que se desea tomar la información que puede haber en cualquiera de los posibles dispositivos que se pueden encontrar dentro de un sistema operativo, y esta información debe poder pasarse correctamente a la aplicación, que va a ser donde finalmente la información ha de ser tratada y ofrecida al usuario final.

Por ello, se debe diseñar un sistema que pase toda la información desde un nivel, la aplicación, hacia el otro nivel, los dispositivos, gestionandose su funcionamiento desde el núcleo del sistema operativo, para hacerlo invisible a los usuarios finales. Para ello se tienen que ofrecer una serie de interfaces que simplifiquen el paso por cada uno de los niveles, siendo un paso de información totalmente transparente, sin pérdida de integridad.

El estándar POSIX, del que se ha hablado anteriormente, nos da las especificaciones a seguir para el uso de funciones de entrada/salida de los dispositivos por parte de las aplicaciones, de manera que esta interfaz que se nos ofrece entre la aplicación y la zona interna del sistema operativo (también conocida como '*kernel*') ha de mantenerse íntegramente. Además de esto, como se comentó en el apartado 1.2, el sistema MaRTE OS tiene la característica propia de ofrecer la capacidad de diseñar aplicaciones tanto para lenguaje Ada como para lenguaje C, de manera que se tiene que adaptar la interfaz para dar la posibilidad de ofrecer la doble información de forma correcta.

Por otro lado, los dispositivos deben disponer de una interfaz con el '*kernel*' pero ésta no se encuentra especificada de ninguna manera concreta en el estándar POSIX. En nuestro caso, se ha elegido construir esta interfaz a través de una '*tabla de dispositivos*', en la cual se debe guardar toda la información necesaria acerca de los parámetros de '*instalación*' de todos los dispositivos agregados al sistema operativo, y donde se debe diferenciar cada uno de ellos mediante alguna identificación propia para que cada dispositivo sea único.

Con esta descripción abstracta de cómo se quiere integrar este nivel de dispositivos dentro del sistema operativo MaRTE, podemos ahora comparar la arquitectura que se vio en el anterior capítulo (véase figura 2), en el apartado 1.2. En esa arquitectura de ambos tipos de aplicaciones no se incluía este nivel, y ahora vamos a situar este nuevo nivel de '*drivers*' de dispositivos únicamente dentro de las aplicaciones Ada. Para las aplicaciones C se tendría una disposición similar a la que tienen las aplicaciones Ada.

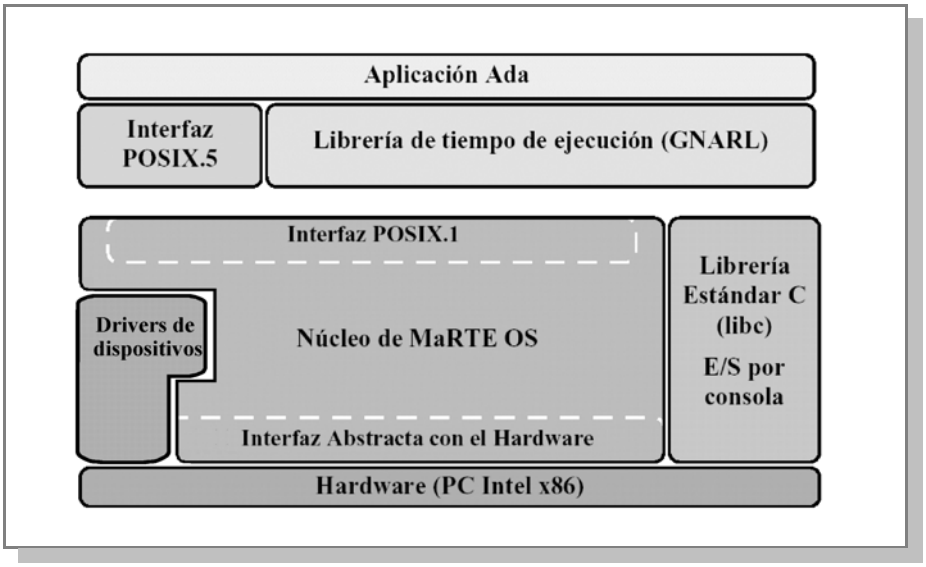

Figura 4 . Arquitectura de las aplicaciones Ada, incluyéndose el nivel de '*drivers*'

2. 2. - Comparación con sistemas de archivos de dispositivos en otros sistemas operativos: Linux, Lynx, VxWorks.

En este apartado se pretenden recoger algunas de las características más concretas que pueden diferenciar los distintos sistemas de archivos usados en diversos sistemas operativos, y con ello situar las principales características sobre las que se puede sustentar el nuevo sistema de archivos para dispositivos que se pretende diseñar en MaRTE.

Dentro de los distintos sistemas operativos por analizar, se van a estudiar sistemas de tiempo real, como son LynxOS [9], un sistema de tiempo real bastante extendido hace tiempo y el sistema VxWorks [10], desarrollado por Wind River como nuevo sistema de tiempo real que se intenta integrar con la plataforma Win32, y que actualmente es el líder del mercado en este campo; y además también se estudia el sistema operativo Linux [11], un sistema bastante más general y muy extendido, que presenta un código abierto pero no cumple requisitos de sistema operativo de tiempo real.

2.2.1. Características de Lynx OS

- **Tipos de '*drivers*'**

Se tienen los tipos de '*drivers*' básicos y más extendidos, esto es, de caracteres y de bloques, y también '*drivers*' de red, que también pueden considerarse como un tipo de '*driver*' de bloques.

- **Identificación de ficheros de dispositivo**

En este caso se utiliza la identificación mediante números mayores y menores. El número mayor sólo puede ser asignado por el '*kernel*', mientras que el número menor es establecido por parte del usuario. Se pueden comprobar qué archivos de dispositivo se encuentran definidos en el sistema con `$ls -l /dev`.

- **Instalación de '*drivers*'**

Se realiza mediante una secuencia de ordenes consecutivas: '`drinstall`', para la instalación del '*driver*', a través de su código objeto; '`devinstall`', para la instalación del dispositivo, con la asignación de su número mayor; y posteriomente '`mknod`', creando un nodo, con el número menor asociado.

- **Funciones en '*drivers*' de caracteres**

Aparte de las funciones ya vistas como básicas ('`open`', '`close`', '`read`', '`write`'), se tiene el '`ioctl`' para implementar códigos propios de cada dispositivo; '`select`' para el chequeo de la disponibilidad del dispositivo; y además dos funciones propias '`install`' para reservar recursos a la hora de utilizarse un dispositivo mayor y '`uninstall`' para liberarlos una vez usados.

- **Acceso a '*hardware*'**

Este apartado es dependiente de la plataforma que se utilice, ya sea IBM para 486 o Pentium, o bien Motorola para MVME compatibles.

- **Sincronización**

Para sincronizarse se tienen tres modos: mediante semáforos del '*kernel*' (con las órdenes '`swait`', para dormir, y '`ssignal`', para despertar); con la inhabilitación de las expulsiones (con '`sdisable`' y '`srestore`') ; o con la inhabilitación de todas la interrupciones (con las órdenes '`disable`' y '`restore`').

2.2.2. Características de VxWorks

- **Tipos de '*drivers*'**

En este sistema se soportan todo tipo de '*drivers*' , tanto los más sencillos y comunes, como son los '*drivers*' de caracteres y de bloques, además de dispositivos virtuales como '*pipes*' o dispositivos de control como tarjetas de entrada/salida analógicas o digitales; y también '*drivers*' para dispositivos de red.

- **Identificación de ficheros de dispositivo**

A diferencia de todo el resto de sistemas, en VxWorks sólo se realiza la identificación a través de un único campo llamado '`drvnum`', asignado por el '*kernel*' y que se asemeja al número mayor, ya visto. No se tiene ningún equivalente con números menores, de modo que todas las funciones de entrada/salida van a ser las mismas, y se han de diferenciar los comportamientos de posibles distintos archivos de dispositivo a través de campos de información añadida.

- **Instalación de '*drivers*'**

La instalación se realiza de manera interna, a través de subrutinas que son llamadas desde el código del '*driver*' a desarrollar. De estas subrutinas la que realiza la tarea principal de inicialización es '`iosDrvInstall`', a la cual se le pasan como argumentos las funciones de entrada/salida que desean asignarse al dispositivo, y devuelve el valor del '`drvnum`', que tal y como vimos, es el argumento utilizado por el sistema para la identificación de los distintos ficheros de dispositivos.

- **Funciones en '*drivers*' de caracteres**

Incluye funciones que no son necesariamente POSIX, como son '`creat`', '`remove`', y también '`ioctl`', y además las funciones especificadas en la interfaz POSIX, como son '`open`', '`close`', '`read`', '`write`'.

- **Acceso a '*hardware*'**

Para este sistema tan complejo, el acceso a hardware es dependiente de unos paquetes específicos denominados BSPs ('***B**oard **S**upport **P**ackage**s***'), que son la base de la portabilidad de VxWorks con todas la platoformas que se pueden utilizar, a través de una interfaz software única.

- **Sincronización**

Como sistema POSIX que se considera a VxWorks, tiene establecida una variada serie de elementos de sincronización, como pueden ser semáforos para tener mútua exclusión, memoria compartida para compartir datos, colas de mensajes y '*pipes*', '*sockets*' y RPC's para tareas remotas, y señales, para el manejo de interrupciones

2.2.3. Características de Linux

- **Tipos de '*drivers*'**

Para este sistema, se pueden establecer muchos tipos diferentes de '*drivers*' . Desde los más comunes, de caracteres; pasando por los '*drivers*' de bloques; y además también '*drivers*' de red, y para dispositivos SCSI.

- **Identificación de ficheros de dispositivo**

Como en muchos otros sistemas, se realiza la identificación a través de números mayores y menores, cuyos valores pueden asignarse por parte del usuario en la inicialización. Además pueden ser consultados los archivos de dispositivo a través de la '*shell*', mediante la orden `$ls -l /dev`

- **Instalación de '*drivers*'**

A través de los módulos del sistema Linux, se consigue tratar a los dispositivos como ficheros de la estructura de archivos, que son tratados con las órdenes '`insmod`' y '`rmmod`'. Después se puede realizar la asignación de los números de identificación, usando '`register_chrdev`' para el número mayor, y '`mknod`' para el número menor.

- **Funciones en '*drivers*' de caracteres**

Se tiene una mayor variedad de funciones que pueden ser utilizadas por los drivers, las más básicas, similares a las utilizadas en MaRTE:'`open`', '`release`' (similar a '`close`'), '`read`', '`write`', '`ioctl`'; y aparte otra serie de funciones para otros usos, como '`select`' para encuestar sobre la disponibilidad del dispositivo, '`lseek`' para variar la posición en un fichero, '`mmap`' para el mapeo en memoria de la información de un dispositivo, '`fsync`' para realizar una espera del dispositivo o '`fasync`', para notificaciones asíncronas.

- **Acceso a '*hardware*'**

Utilizando las funciones de las cabeceras definidas en `<linux/ioport.h>`, y también en `<asm/io.h>`.

- **Sincronización**

Se puede realizar de tres maneras distintas: con funciones de espera sobre colas, durmiendo y despertando ('`sleep_on`', '`wake_up`', ...); mediante la inhabilitación de interrupciones ('`disable_irq`', '`enable_irq`', ...); y con la utilización de variables atómicas ('`set_bit`', '`clear_bit`', '`change_bit`', '`atomic_add`', ...)

2. 3. - Diseño y características del sistema de archivos a desarrollar en MaRTE.

2.3.1. *Características básicas en el diseño del sistema de archivos de dispositivos de MaRTE OS*

En el anterior apartado se ofrecieron las características más significativas de unas serie de sistemas operativos relativas a los archivos de dispositivos que manejan, comentando diferentes aspectos bastante relacionados. Ahora se comentarán estos mismos aspectos para el nuevo sistema de archivos que se va a diseñar para el sistema MaRTE OS, para poder con ello compararse con los otros sistemas vistos, y servir estas características como base del nuevo diseño.

- **Tipos de '*drivers*'**

Por ahora, sólo se utilizarán '*drivers*' de caracteres, puesto que para los drivers de bloques sería necesario un soporte físico de almacenamiento (discos, cintas, ..) que el sistema MaRTE no presenta, al considerarse un sistema POSIX mínimo.

- **Identificación de ficheros de dispositivo**

Se realiza a través de números mayores y menores, siendo la '*tabla de dispositivos*' donde se asignan ambos identificadores, y donde se ofrece la interfaz de paso desde el nivel de usuario programador de '*drivers*' hacia el nivel de '*kernel*' del sistema operativo donde se debe gestionar la información de los dispositivos. Cualquier variación de esta tabla de información de los dispositivos ha de suponer una necesaria recompilación del '*kernel*'.

- **Instalación de '*drivers*'**

La instalación de los drivers se puede considerar '*estática*' en el sentido de que sólo se va a poder hacer a través de la variación de código de la '*tabla de dispositivos*', con la consabida recompilación del '*kernel*', y no se podrá hacer a través de la propia aplicación, de una manera dinámica o a través de una consola del sistema.

- **Funciones en '*drivers*' de caracteres**

En total se tienen siete funciones de entrada/salida a utilizar:

- dos funciones externas al sistema de archivos, ya que no necesitan ninguna gestión de información, incluidas por semejanza con el VxWorks [10], que son '`Create`', '`Remove`'
- las cuatro funciones de entrada/salida POSIX básicas, gestionadas internamente por el sistema de archivos, y estas son '`Open`', '`Close`', '`Read`', '`Write`'
- la función no-POSIX, '`Ioctl`', necesaria para poder dar una generalidad a todos los dispositivos y que a través de ella se pueda implementar cualquier nueva función que necesite cualquier dispositivo.

- **Acceso a '*hardware*'**

El acceso a la información requerida por cualquier hardware se va a realizar a través de las funciones establecidas en la librería `<oskit/x86/pio.h>`. Con ellas se consiguen manejar los registros apropiados para cada uno de los dispositivos.

- **Sincronización**

Para realizar la sincronización entre los archivos de dispositivos, se dispone en los descriptores de fichero de una serie de '*mutexes*', que posteriormente podrán utilizarse siguiendo los procedimientos POSIX de sincronización mediante variables condicionales que hacen uso del '*mutex*'. Además de ello, para la sincronización entre procesos y '*threads*' fuera del sistema de archivos de dispositivo, también se tiene en el sistema MaRTE el uso de semáforos contadores y también de señales e interrupciones, para cumplir con los criterios establecidos en el estándar POSIX.

2.3.2. Explicación y diseño del sistema de archivos de dispositivos de MaRTE OS

En el apartado 2.1 se vio una noción general sobre la forma general que debía seguir el nuevo sistema de archivos de dispositivos que pretende implementarse en MaRTE OS. En este punto, ya se entra al diseño concreto del sistema de archivos, a través de unas especificaciones más precisas, y dando forma a todo el conjunto de paquetes que se debe utilizar.

Como se pudo concluir en la introducción del capítulo, se tiene que trabajar en tres niveles de trabajo, teniéndose que utilizarse unas interfaces bien especificadas para hacer el paso entre los niveles. El núcleo del sistema operativo ('*kernel*') será la zona intermedia que va a servir de paso desde los dispositivos hasta la aplicación, con lo cual se van a establecer dos interfaces diferentes:

- una interfaz entre la aplicación y el '*kernel*', en este caso una interfaz POSIX recogida y bien especificada en el estándar.
- una interfaz entre los dispositivos y el '*kernel*', que no está especificada como tal en el estándar POSIX, con lo que se implementará de la manera que mejor se pueda acercar a las necesidades que se requieran en el diseño.

Ya se comentó en la introducción que la interfaz para el paso de información de los dispositivos hacia el '*kernel*' se implementará a través de una tabla en la que se guarden todos los datos necesarios que todos los posibles dispositivos que puede utilizar el sistema operativo requieran almacenar, como pueden ser identificadores únicos que los diferencien entre sí, o bien funciones de entrada/salida propias, o también etiquetas usadas para la identificación de los dispositivos por parte de los usuarios.

Para tener una noción más visual de la forma global del sistema de archivos de dispositivos, veamos a continuación la figura, en la cual se diferencian cada uno de los diferentes niveles en los que nos vamos a mover.

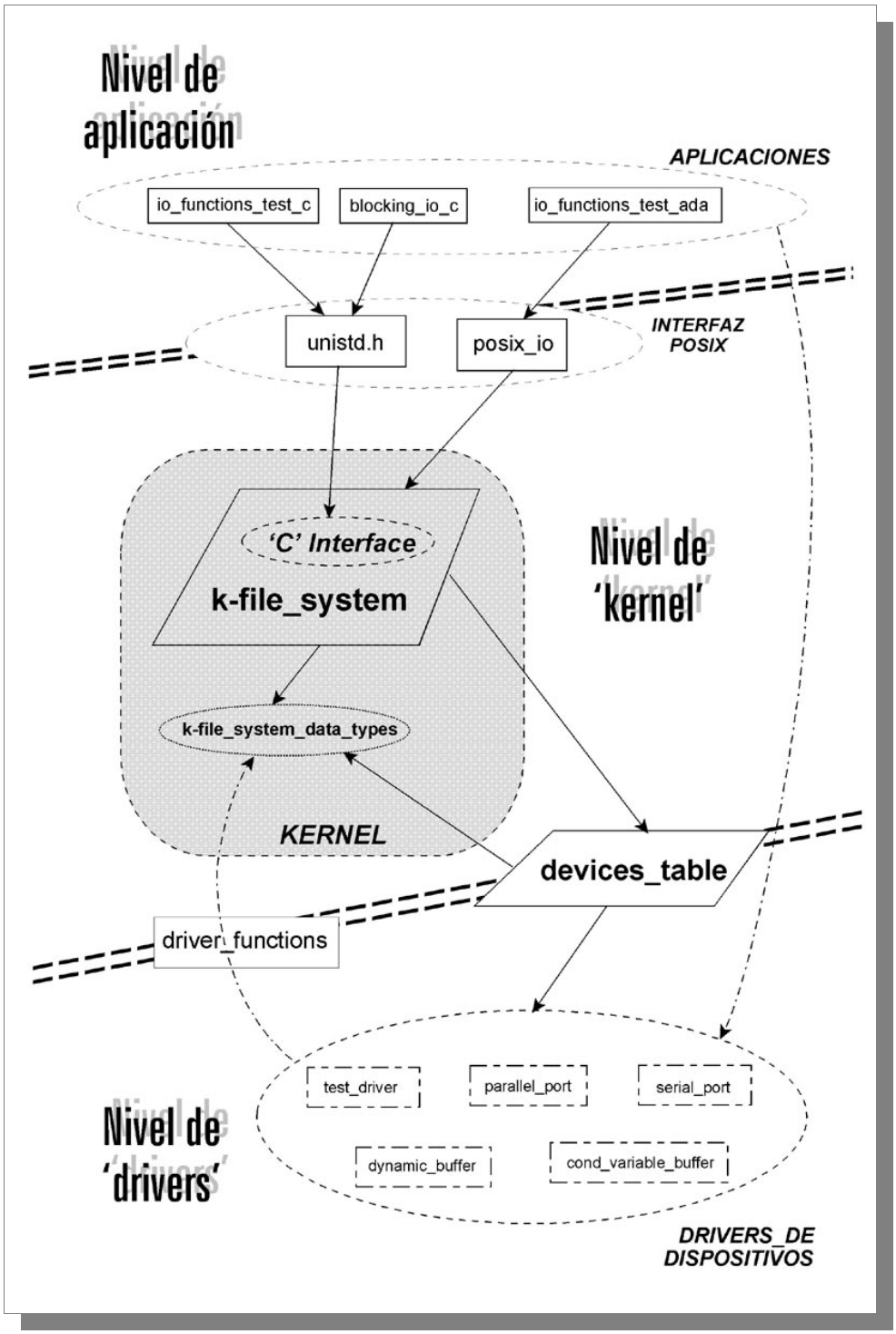

Figura 5 . Estructura completa del sistema de archivos de dispositivos de MaRTE OS

La parte central del diseño se encuentra en los paquetes propios del *"kernel"* (núcleo del sistema operativo). En ellos es donde el sistema MaRTE OS va a realizar todas la acciones necesarias para el control de la primitivas de entrada/salida, que podrán ser usadas desde la aplicación a través de la *"interfaz de funciones de entrada/salida"*. En esta interfaz es donde se van a encontrar las llamadas a las funciones de entrada/salida, es decir, '`open`', '`close`', '`read`', '`write`', '`ioctl`' y es la única parte visible desde el *"nivel de aplicación"*, de manera que la aplicación sólo debe tener en cuenta las llamadas a esta interfaz intermedio, sin tener que entrar en ningún momento en el *"kernel"*. Esta interfaz para el paso hacia la aplicación será ampliada posteriormente, comentando su implementación, en este mismo capítulo, en la sección 2.4.4.

Además de esto, también nos encontramos con el*"nivel de 'drivers' de dispositivo"*. Aquí se van a situar los paquetes que se refieren a los distintos '*drivers*' que pueden existir en cualquier sistema operativo (puerto serie, paralelo; tarjetas de red, ...). Estos paquetes van a poder ser utilizados por el *"kernel"* gracias a la *"tabla de dispositivos"* (que se observa en el gráfico con el nombre *"devices_table"*), para conformar el enlace entre las funciones de entrada/salida propias de cada dispositivo y las funciones de entrada/salida que en realidad son ofrecidas a la aplicación, por medio de la interfaz antes comentada, *"interfaz de funciones de entrada/salida"*. Toda la información referente a la interfaz de la '*tabla de dispositivos*' es ampliamente comentada a lo largo del capítulo 3, de manera que en este momento no se comentará más en profundidad.

Es posible que algún *"driver de dispositivo"* posea una especificación propia, que deba ser utilizada por la aplicación, de ahí que en alguna ocasión se puede dar un enlace directo entre el *"nivel de aplicación"* y la parte de *"drivers de dispositivo"*, a través de las cabeceras específicas establecidas para tal efecto en el diseño del '*driver*'. Sin embargo, para facilitar la portabilidad de la aplicación, debe intentarse hacer todas las operaciones de entrada/salida mediante esta interfaz estándar que hemos estado comentando hasta ahora, y que utiliza las llamadas a dispositivos a través del '*kernel*' del sistema operativo.

2. 4. - Implementación del sistema de archivos de MaRTE.

Vista ya la parte más externa del diseño, es decir, la parte visible del mismo con sus distintas interfaces brevemente descritas y los enlaces entre los paquetes, entremos ya en la parte interna de la especificación, la parte del diseño de los diferentes paquetes.

Para ello, en primer lugar se comentarán los tipos de datos creados específicamente para todo el manejo de la información, y después se incidirá de manera más especial en las tablas internas que se van a utilizar por parte del sistema operativo MaRTE para poder facilitar todos los tránsitos desde el nivel de "*drivers de dispositivos*", pasando por los "*archivos de dispositivos*", hasta llegar al nivel de "*descriptores de fichero*". Un aspecto a destacar es el hecho de que todas la interfaces se han especificado en inglés, para así facilitar su uso por parte de la comunidad internacional.

2.4.1. Tipos de datos internos al sistema de archivos de dispositivos

Observando el gráfico adjuntado del diseño, en la figura 5, se puede comprobar que el paquete '`k-file_system_data_types`' se puede considerar el más inferior a todos, en cuanto al concepto de no tener que hacer referencia a ningún otro y éste va a ser punto de partida del resto. En este paquete se tienen recogidos todos los nuevos tipos de datos que son utilizados por el sistema de archivos. Los nuevos tipos de datos que se encuentran en este paquete, definidos para el uso en el sistema de archivos de dispositivos, son:

- '*File_Access_Mode*' : utilizado para los modos de acceso que puede tomar un nuevo fichero que se abra con la función '*open*', a través de una serie de constantes. Estos modos son de lectura ('`Read`'), escritura ('`Write`'), y por último, también la lectura-escritura ('`Read-Write`').
- '*File_Descriptor*' : utilizado para poder ir numerando cada uno de los sucesivos descriptores de fichero que se van creando de forma sucesiva cada vez que se llama a la función '*open*'.
- '*Major*' : sirve para identificar los dispositivos mayores, es decir, cada uno de los dispositivos que presentan un código del '*driver*' independiente de cualquiera de los otros dispositivos del sistema operativo.
- '*Minor*' : sirve para diferenciar los archivos de dispositivo que pueden generarse a través de una referencia al mismo dispositivo mayor, y que a través del propio código del '*driver*' asociado al dispositivo mayor consiguen hacer la diferenciación interna del comportamiento del archivo de dispositivo respecto de otros archivos de dispositivo que sean generados a raíz del mismo dispositivo mayor.
- '*Device_File_Number*' : se utiliza por parte del sistema operativo para gestionar y diferenciar los diferentes archivos de dispositivo que se van generando, en los que el par de números mayor-menor constituyen un identificador único.
- '*Path*' : string de caracteres que sirve para generar una etiqueta con la cual el usuario de la aplicación puede identificar cada uno de los archivos de dispositivo que se generen en la tabla de dispositivos.
- '*Buffer_Length*' : representa un valor interno del sistema de archivos a través de la cual se establece el control de orden dentro del conjunto de '*bytes*' ("*Buffer*") que se va a utilizar para el almacenamiento de información.
- '*Buffer*' : conjunto de '*bytes*' que van a conformar una unidad de almacenamiento de datos interna al sistema operativo, y a través de la cual se va a pasar toda la información de datos que la aplicación necesite del dispositivo y viceversa.
- '*Buffer_Ac*' : puntero a la variable "*Buffer*", que va a ser utilizado para no crear unidades completas de esta unidad de almacenamiento, que supondrían mucho tamaño en memoria, y así sólo se ocuparía en memoria el tamaño de un puntero.

- '*Dynamic_Buffer*' : es también una unidad de almacenamiento, con la misma utilidad que presenta el "*Buffer*", pero en su definición no está concretado su tamaño, para así poderse configurar desde la aplicación y dar más capacidad de configuración al usuario de la aplicación, sin tener que entrar dentro del núcleo del sistema.
- '*Dynamic_Buffer_Ac*' : puntero a la variable no definida "*Dynamic_Buffer*", para poder igualar los comportamientos de ambas unidades de almacenamiento, y siempre trabajar con los punteros, en lugar de las variables completas.
- '*Ioctl_Option_Value*' : sirve para diferenciar cada una de las posibles opciones que cada dispositivo puede incluir dentro de la función de entrada/salida '`ioctl`', de modo que se puedan gestionar las funciones correspondiente a la función que se desee ejecutar desde dentro del propio '`ioctl`'
- '*(function_id)_Function_Ac*' : indica los distintos punteros a las funciones de entrada/salida que serán utilizados por los dispositivos para poder implementar cada dispositivo su propia función entrada/salida; todas ellas sigan la especificación que el puntero a la función impone. Para no extenderse en exceso, la etiqueta '*(function_id)*' debe irse sustituyendo por cada uno de los nombres de las funciones de entrada salida, de manera que se tienen los siguientes punteros a funciones:
  '`Create_Function_Ac`', '`Remove_Function_Ac`', '`Open_Function_Ac`', '`Close_Function_Ac`', '`Read_Function_Ac`', '`Write_Function_Ac`', '`Ioctl_Function_Ac`'

Profundizando de mayor forma dento de estos tipos de datos definidos, comencemos con los punteros de las funciones de entrada/salida, mostrando cual es el perfil de cada uno de los punteros a función:

```
type Create_Function_Ac is access function  return Int;

type Remove_Function_Ac is access function  return Int;

type Open_Function_Ac is access function
  (Fd   : in File_Descriptor;    Mode : in File_Access_Mode)
return Int;

type Close_Function_Ac is access function
  (Fd : in File_Descriptor)
return Int;

type Read_Function_Ac is access function
  (Fd         : in File_Descriptor;    Buffer_Ptr : in Buffer_Ac;
   Bytes      : in Buffer_Length)
return Int;

type Write_Function_Ac is access function
  (Fd         : in File_Descriptor;    Buffer_Ptr : in Buffer_Ac;
   Bytes      : in Buffer_Length)
return Int;

type Ioctl_Function_Ac is access function
  (Fd             : in File_Descriptor;   Request : in Ioctl_Option_Value;
   Ioctl_Data_Ptr : in Buffer_Ac)
return Int;
```

Para definir completamente todo el resto de tipos de variables que se definen, sólo nos queda por ver los tipos discretos definidos, sobre los que habría que ver cual es su tipo de dato de origen (tipo base) y los límites sobre los que se trabaja, así que en esta tabla se recoge toda esta información para tener una completa explicación de estos tipos de datos:

		Limites	
Nombre	**Tipo base**	**Inferior**	**Superior**
File_Access_Mode	Int	1	3
File_Descriptor	Int	0	Configuration_Parameters. Open_Files_Mx - 1
Major	Int	1	Configuration_Parameters. Devices_Mx
Minor	Int	1	Configuration_Parameters. Devices_Mx
Device_File_Number	Int	1	Configuration_Parameters. Devices_Files_Mx
Path	String	1	Configuration_Parameters. Path_Mx
Buffer_Length	Int	0	Configuration_Parameters. Buffer_Mx
Buffer	array of Unsigned_8	1	Configuration_Parameters. Buffer_Mx
Dynamic_Buffer	array of Unsigned_8	?	?
Ioctl_Option_Value	Int	0	Configuration_Parameters. Ioctl_Functions_Mx - 1

Figura 6 . Límites de los tipos de datos discretos utilizados en la implementación del sistema de archivos de dispositivos

2.4.2. Diseño de las tablas usadas para el manejo de información referida al sistema de archivos de dispositivos

Además de los tipos de datos discretos comentados anteriormente, también se tienen definidos en el paquete '`k-file_system_data_types`' una serie de nuevos tipos formados por registros, a través de los cuales se van a implementar una idea de tablas enlazadas, en las cuales el sistema operativo va a ir guardando una serie de informaciones y datos que van a facilitar las gestión del sistema de archivos de dispositivo.

Esta idea de realizar esta implementación del diseño del sistema de archivos a través de estas tablas donde se recoge toda la información de los dispositivos en uso, con cursores internos para el entralazado de las tablas, se realiza siguiendo la forma original establecida en el diseño de VxWorks [10], otro SO de tiempo real del que ya se han comentado sus características en la sección 2.2.2.

El significado y la necesidad de esta implementación en forma tablas se debe al hecho de separar tres claros conceptos bien diferenciados: los *"descriptores de fichero"*, que se concentran en la tabla ***'The_Fd_Table'***; los *"archivos de dispositivos"* que se recogen en la tabla ***'The_Device_Files_Table'***; y por último los *"punteros a funciones de entrada/salida de drivers"*, que se encuentran en ***'The_Drivers_Table'***.

Para hacer de nexo de unión entre las tablas, se utilizan varios campos internos de cada una de ellas que funcionan en modo de 'cursor'. Esta forma de implementación de las tablas con entrelazado entre ellas, hace que cualquier descriptor de fichero sepa siempre de manera correcta a qué funciones de entrada/salida de qué dispositivo debe referirse.

a) Tabla de descriptores de fichero

Analizando más en profundidad la composición interna de cada tabla, comencemos con la tabla *'The_Fd_Table'*. En ella se guarda el modo de acceso al fichero ('`Read`', '`Write`', '`Read-Write`'); el archivo de dispositivo al cual está asociado el descriptor de fichero; un '*mutex*' que será utilizado por cada descriptor de fichero para realizar la sincronización de procesos que utilicen simultáneamente las funciones de entrada/salida; y por último, una variable de control que nos indica si el descriptor de fichero está abierto, con un uso válido, o si por el contrario su información no es válida.

El hecho de poner el '*mutex*' para la sincronización en esta tabla de descriptores de ficheros, y no colocarlo en la tabla de archivos de dispositivos, y menos aún en la tabla de punteros a funciones, se hace para evitar que un mismo '*mutex*' pueda ser compartido por diferentes descriptores, ya que si esto sucede se puede tener una desincronización fatal, como sería una lectura sobre un archivo que está en medio de una escritura. Como no existe ningún impedimento para que dos descriptores de fichero puedan abrirse compartiendo el mismo archivo de dispositivo, si se desea que el '*mutex*' no pueda ser compartido por estos dos descriptores de fichero no se puede colocar el '*mutex*' en la tabla de archivos de dispositivo porque ambos descriptores de fichero tendrían asignado el mismo '*mutex*'.

b) Tabla de archivos de dispositivo

Pasando a comentar aspectos de la tabla intermedia de archivos de dispositivos, *'The_Device_Files_Table'*, en primera instancia nos puede parecer que su presencia no es necesaria, que se podría hacer directamente el paso entre los descriptores de fichero y los punteros asociados a las funciones de entrada/salida de cada '*driver*', pero esto haría que no se pudiera tener ninguna diferenciación interna dentro de la implementación en el '*driver*' de las funciones de entrada/salida.

Pero en muchos casos se tiene la necesidad de diferenciar comportamientos del mismo dispositivo, y para ello se tienen que poder hacer diferencias internas, y es por ello que se introduce el concepto de los números menores (*'minor'*), que de modo conjunto con los números mayores hacen la identificación completa del nuevo 'archivo de dispositivo'.

Con estos números menores ya se puede diferenciar dos archivos de dispositivo que apunten al mismo conjunto de funciones de entrada/salida (identificadas con el mismo *'major'*), tomando cada uno de los archivos de dispositivo distintos valores de *'minor'*.

Este caso de variedad de comportamientos del dispositivo a través del uso de *'minor'* se dará posteriormente en la implementación realizada para los archivos de dispositivo del puerto serie, en el capítulo 4, de manera que a través de su número menor se consigue obtener la información del puerto COM al cual se refiere (COM1, COM2, ...).

c) Tabla de punteros a funciones de entrada/salida

Para concluir con este estudio de los tipos de datos utilizados, sólo queda hacer un comentario de la tabla de funciones de entrada/salida propias de cada dispositivo, es decir, *'The_Drivers_Table'*. Cada dispositivo posee unas únicas funciones de entrada/salida, y de esta manera cada '*driver*' realizado aporta sólo unas funciones de entrada/salida propias y cada fila de esta tabla de funciones se refiere a cada uno de los '*drivers*' utilizables por parte del sistema operativo. Ya posteriormente, en el desarrollo completo del capítulo 3, se verá mucho más en profundidad de qué manera se realiza un uso correcto de las funciones de entrada/salida propias de cada '*driver*'.

Para intentar entender el significado de estas tres tablas con sus relaciones internas entre ellas, se recoge un ejemplo de asignación de tablas, en el que también se pueden observar cómo son estos enlaces entre las tablas.

- **The_Fd_Table**

File_Descriptor	***File_Open_ Status*** File_Access_Mode	***Device_File_ Assigned*** Device_File_Number	***Mutex*** Kernel.Mutexes.Mutex	***Fd_Used*** Boolean
0	?	?	?	False
3	Read_Write	4	(access)	True
Configuration_Parameters. Open_Files_Mx - 1	?	?	?	False

- **The_Device_Files_Table**

Device_File_ Number	***File_Name*** Path	***Major_ Number*** Major	***Minor_ Number*** Minor	***Mutex_ Ceiling_Prio*** Kernel.Mutexes. Ceiling_Priority	***Device_ Used*** Boolean
1	?	1	1	Ceiling_Mx	False
4	test_driver_1	5	1	Ceiling_Mx	True
Configuration_Parameters. Devices_Files_Mx	?	1	1	Ceiling_Mx	False

- **The_Drivers_Table**

Major	***Create*** Create_ Function_Ac	***Remove*** Remove_ Function_Ac	***Open*** Open_ Function_Ac	***Close*** Close_ Function_Ac	***Read*** Read_ Function_Ac	***Write*** Write_ Function_Ac	***Ioctl*** Ioctl_ Function_Ac
1	null	null	null	null	null	null	null
5	Test_ Create	Test_ Remove	Test_ Open	Test_ Close	Test_ Read	Test_ Write	Test_ Ioctl
Configuration _Parameters. Devices_Mx	null	null	null	null	null	null	null

Figura 7 . Interrelaciones entre las tablas utilizadas en la implementación para la gestión de la información de los archivos de dispositivos del sistema MaRTE

2.4.3. Núcleo central del sistema de archivos de dispositivos de MaRTE (paquete 'Kernel.File_System')

Ya con toda la estructura de tipos de datos comentada, centremos la situación en el núcleo central del diseño, que se sitúa de modo interno al *'kernel'* del sistema, con el paquete '`Kernel.File_System`' como parte principal (el cual se corresponde con los archivos `$~/kernel/k-file_system.ad*`)

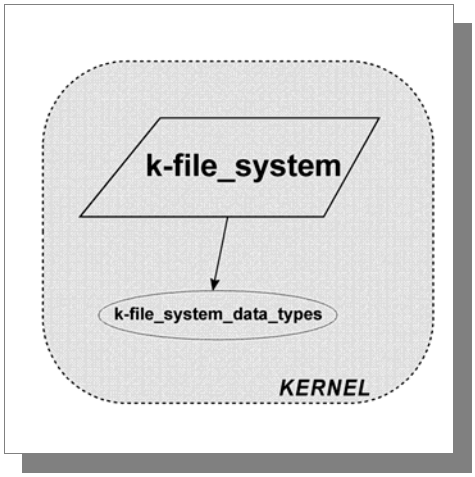

Figura 8 . Zona de 'kernel' correspondiente al diseño interno del sistema de archivos

Veamos la especificación completa de este paquete `k-file_system`, para así ya poder situar la explicación de una manera más concreta:

```
------------------------------------------------------------------------------
-- --------------------       M a R T E   O S         ------------------- --
------------------------------------------------------------------------------
--
--                    'K e r n e l . F i l e _ S y s t e m'
--
--                                  Spec
--
--
--  File 'k-file_system.ads'                                    By Fguerreira
--
--
--  File system I/O functions
--
------------------------------------------------------------------------------

with Kernel.Devices_Types;  use Kernel.Devices_Types;

package Kernel.File_System is

   ----------
   -- Open --
   ----------
   function Open
      (Path_Name : access Path;
       Mode : in File_Access_Mode)
     return Integer;
   pragma Export (C, Open, "open");
```

```
   --  **********************  --
   --  File_Descriptor_Correct  --
   --  **********************  --
   function File_Descriptor_Correct (File_To_Check : in File_Descriptor)
                                    return Boolean;

   -----------
   -- Close --
   -----------
   function Close (File_Closed : in File_Descriptor)
   return Integer;
   pragma Export (C, Close, "close");


   ----------
   -- Read --
   ----------
   function Read
      (File_Read     : in File_Descriptor;
       Buffer_Ptr    : in Buffer_Ac;
       Bytes_To_Read : in Unsigned_32)
     return Integer;
   pragma Export (C, Read, "read");


   -----------
   -- Write --
   -----------
   function Write
      (File_Written   : in File_Descriptor;
       Buffer_Ptr     : in Buffer_Ac;
       Bytes_To_Write : in Unsigned_32)
     return Integer;
   pragma Export (C, Write, "write");


   -----------
   -- Ioctl --
   -----------
   function Ioctl
     (File           : in File_Descriptor;
      Request        : in Ioctl_Option_Value;
      Ioctl_Data_Ptr : in Buffer_Ac)
     return Integer;
   pragma Export (C, Ioctl, "ioctl");

end Kernel.File_System;
```

Las aplicaciones han de utilizar una única función de entrada/salida que es la que se les ofrece a través de la interfaz POSIX, tal y como se observó en la figura 5, pero estas funciones han de estar asociadas con las funciones de entrada/salida propias de cada dispositivo, y para ello se tiene que hacer que la función única de entrada/salida sepa diferenciar cuál es el dispositivo al cual se tiene que acceder.

Este cometido es el que se realiza en la implementación interna, dentro del cuerpo del paquete `Kernel.File_System`. A través de la información que se guarda en el descriptor de fichero, se consigue acceder al dispositivo correspondiente, que ha sido escogido por la aplicación a la hora de abrir el fichero de dispositivo, guardándose la información necesaria para asociar los descriptores de fichero con los archivos de dispositivo y también con las propias funciones de entrada/salida de cada dispositivo.

De todas las funciones de entrada/salida que se recogen en la especificación, la función '`Open`' es la más importante de todas las utilizadas en el sistema de archivos. Esta función tiene la tarea de asociar el dispositivo que se desea utilizar, a través del nombre de archivo entrado, con un descriptor de fichero. A través de este descriptor de fichero, ya posteriormente se podrá acceder a todas las funciones asociadas a dicho dispositivo y a otras características propias de cada archivo de dispositivo.

Sin entrar a estudiar el código recogido en la implementación realizada, veamos a través de un esquema detallado de qué manera trabaja internamente esta función '`Open`'.

- Se entra en la `Critic_Section` del '*kernel*', para trabajar en modo protegido.
- Se limpia el indicador de posibles errores POSIX que pueda haber.
- Se chequea el parámetro `Mode` para saber si está en el rango apropiado. Para un caso de `Mode` no válido, se establece el error *'INVALID_ARGUMENT'*, retornándose la función con valor -1.
- Se hace el chequeo del parámetro `Path_Name`, a través del procedimiento interno `Correct_Path`, y con él se obtiene el número de archivo de dispositivo asociado a la ruta escogida en la etiqueta.
- En caso de no encontrar ningún dispositivo asociado a ese `Path_Name`, se establece el error *'NO_SUCH_FILE_OR_DIRECTORY'*, y se sale de la función con valor -1.
- Se busca en la tabla de descriptores de ficheros, `The_Fd_Table`, alguna posición libre y para ello se chequea el campo `Fd_Used` en cada una de las filas de la tabla.
- En caso de encontrar alguna posición libre se realizan las asignaciones oportunas, correspondientes a los respectivos campos de la tabla `The_Fd_Table`, y se guarda el valor de la fila que se ocupa en la correspondiente tabla, para su posterior uso.
- En caso de no encontrarse ninguna fila disponible, por el hecho de encontrarse toda la tabla ocupada, se establece el error POSIX *'TOO_MANY_OPEN_FILES'*, y se sale de la función con valor -1.
- Se inicializa el '*mutex*' que está asociado al nuevo archivo de dispositivo de la tabla `The_Fd_Table`.
- Se sale de la `Critic_Section` del '*kernel*', antes de ejecutar la función '`Open`' propia del '*driver*', para permitir que se ejecute la función sin estar trabajando en el modo protegido del '*kernel*'.
- Se ejecuta la función '`Open`' asociada al '*driver*' del dispositivo seleccionado, siempre que exista dicha función, con su puntero a función no nulo, para evitar un error de ejecución *'CONSTRAINT_ERROR'*
- Se devuelve el valor `The_Fd` convertido al tipo `Integer`, correspondiente a la fila ocupada en la tabla `The_Fd_Table`.

El resto de las funciones de entrada/salida de esta implementación, todas más o menos siguen un patrón bastante parecido a esta función '`Open`', analizada anteriormente más en detalle. Cada una de las funciones tiene sus pequeños distintos matices, pero en conjunto los principales pasos que se siguen son:

- Se entra en la *'Sección Crítica'* del '*kernel*', limpiando el indicador de error POSIX.
- Se chequea el parámetro de entrada correspondiente al descriptor de fichero, para salir de la función en caso de error.
- Se toma el '*mutex*' asociado a dicho descriptor de fichero, ya correcto.
- En este paso, se establecen las diferencias entre funciones:
  p.ej. en el '`Close`' eliminando de la tabla la fila correspondiente; en '`Read`' y '`Write`', chequeándose el numero de bytes y los modos de acceso.
- Se sale de la *'Sección Crítica'* del '*kernel*', para ejecutar libre la función de entrada/salida propia del '*driver*'.
- Se ejecuta la función de entrada/salida correspondiente, asociada al '*driver*' del dispositivo seleccionado.
- Se libera el '*mutex*' asociado al descriptor de fichero.
- Se devuelve el valor correspondiente, p.ej. para '`Read`' y '`Write`', los bytes leídos o escritos; para el resto, devuelve 0.

Por último se debe comentar que en la ejecución inicial asociada al propio paquete `Kernel.File_System`, se realiza la llamada a las funciones '`Create`' de cada uno de los dispositivos que disponen de una entrada establecida en la tabla de dispositivos, de manera que esto podría considerarse como una '*inicialización de los dispositivos*'. Además también se hace una inicialización de la tabla de descriptores de ficheros, para que siempre se tenga que las tres primeras filas correspondan a los tres dispositivos estándar: '`Standard_Input`', '`Standard_Output`', '`Standard_Error`'

2.4.4. Interfaz POSIX para el paso desde el 'kernel' hacia la aplicación

Tal y como se comentó anteriormente, estas funciones de entrada/salida que son generadas por el sistema de archivos son únicas, en el sentido de ser capaces de congregar en una llamada única todas las posibles funciones de entrada/salida equivalentes que puede ofrecer cada uno de los diferente dispositivos. Pero estas funciones de entrada/salida del sistema de archivos no son las que utiliza la aplicación, porque en realidad la especificación interna del sistema de archivos ha sido diseñado siguiendo ideas propias, sin seguir ningún patrón concreto. Para seguir este patrón único se establece la interfaz POSIX, de tal modo que ésta sea la única interfaz vista por la aplicación, sin entrar en el nivel de '*kernel*' en ningún momento.

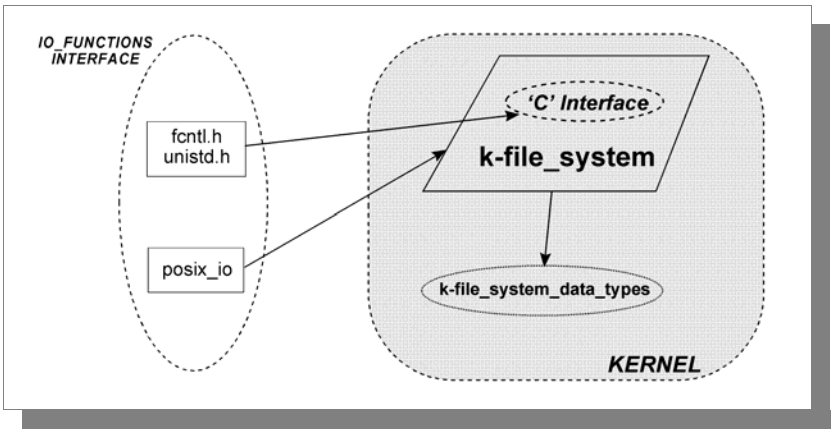

Figura 9 . Zona de '*kernel*' con la conexión con las interfaces POSIX

Como se ve en el gráfico, el paquete `k-file_system` es el que internamente ofrece la posibilidad de desglosar la doble interfaz C-Ada, característica propia del sistema MaRTE, de modo que siempre se tienen que ofrecer las funciones POSIX tanto para el estándar C-ANSI como para el lenguaje Ada 95. Esta característica tan concreta supone que en la implementación siempre se tenga que respetar al máximo los tipos de datos propios de cada lenguaje, haciendo que las funciones utilizadas internamente en el '*kernel*' siempre trabajen de manera única, pero con la posibilidad de recibir la información de manera doble ya que le puede llegar por cada una de las dos partes, debiéndose hacer, por tanto, las conversiones necesarias para mantener la integridad de la información.

Para comprobar de qué manera se mantiene esta integridad, veamos como ejemplo cómo se pasa la primitiva '`Read`' a cada una de la dos interfaces POSIX.

- Para la interfaz POSIX-C, el paso es inmediato, ya que antes de ejecutar la orden `pragma Export (C, Read, "read");` que realiza la exportación de la función '`read`' del sistema de archivos de MaRTE, ya se tienen que tener los tipos Ada completamente equivalentes, para que los tipos de la función exportada utilizada a través de la cabecera C ya sean los adecuados.

  Esta cabecera se encuentra en el archivo `$~/marte/include/unistd.h` , y en él se tiene la forma que debe seguir esta función '`read`' para el POSIX-C:

  ```
  ssize_t read (int fd, void *buf, size_t _bytes);
  ```

  Los parámetros utilizados en la especificación del '`k-file_system.ads`' mantienen perfectamente la integridad, puesto que la función devuelve un tipo `Integer` equivalente al tipo `ssize_t` definido como `int` en el archivo de cabeceras.

  Se le pasa en primer lugar el tipo predefinido `File_Descriptor`, pero que en sí mismo es un subtipo `Integer`, lo cual supone equivalencia completa con el tipo C `int` del parámetro `fd`.

Respecto al parámetro del tipo `Buffer_Ac`, en sí mismo es en realidad un tipo `Access` con lo cual la equivalencia con el tipo `void *` está garantizada, ya que para el compilador utilizado se establece que los tipos `Access` tienen una correspondencia directa en la exportación con los punteros C (cualquier tipo `*`).

Por último para el número de bytes se utiliza el parámetro `Bytes_To_Read`, que está definido como `Unsigned_32`, lo cual supone que para el compilador va a tener un comportamiento equivalente con el tipo `size_t`, utilizado en la función C.

- Para la interfaz POSIX-Ada, tal y como se aprecia en la figura 9, se utiliza el paquete '`POSIX_IO`', del cual se tiene perfectamente recogida su especificación en el estándar POSIX Ada [12], pero quedando libre la implementación del mismo. Esta libertad permite que se realicen todas las transformaciones y conversiones de tipos, para conseguir un perfecto paso de parámetros entre las funciones de entrada/salida llamadas por la aplicación a través de la interfaz POSIX y las funciones de entrada/salida internas, que utiliza el MaRTE dentro de su sistema de archivos de dispositivos.

  Como explicación práctica de cómo se hace esta conversión interna, veamos de qué manera se utiliza la primitiva '`Read`' a través de la interfaz POSIX-Ada, en la cual se realiza internamente la llamada a la función'*Read*' asociada al sistema de archivos de MaRTE.

  Este es el código utilizado para la implementación de la primitiva '*Read*' del POSIX

```
procedure Read
  (File           : in  File_Descriptor;
   Buffer         : out Ada.Streams.Stream_Element_Array;
   Last           : out Ada.Streams.Stream_Element_Offset;
   Masked_Signals : in  Signal_Masking := RTS_Signals) is
   --  'Masked_Signals' is not useful

   function Address_To_Buffer_Ac is new
     Ada.Unchecked_Conversion
     (System.Address, Kernel.File_System_Data_Types.Buffer_Ac);

   Bytes_Read : Integer;
   Buffer_Ptr : Kernel.File_System_Data_Types.Buffer_Ac;

begin
   Buffer_Ptr := Address_To_Buffer_Ac (Buffer (1)'Address);
   Bytes_Read := Kernel.File_System.Read
     (Kernel.File_System_Data_Types.File_Descriptor (File), Buffer_Ptr,
      Buffer'Length * (Ada.Streams.Stream_Element'Size / 8));
   Last := Ada.Streams.Stream_Element_Offset
     (Bytes_Read / (Ada.Streams.Stream_Element'Size / 8));
   PI.Raise_POSIX_Error_On_Error;
end Read;
```

Como se observa claramente, la especificación de este '`Read`' del POSIX Ada y la especificación de la función '`Read`' que se utiliza en el paquete `Kernel.File_System`, difieren notablemente, y para mantener fielmente la integridad de la información de los parámetros se debe hacer un juego de conversiones bastante variado.

En principio, el primer parámetro, que se refiere a los descriptores de fichero, no supone ningún problema ya que con hacer una conversión al tipo interno que se utiliza en el paquete del *'kernel'* es suficiente.

El problema más serio se tiene con el paso de la información de dónde se quieren guardar los datos leídos. El estándar POSIX-Ada utiliza para ello la estructura de datos de '`Ada.Streams`', con unas formas bastante particulares, mientras que, siguiendo una forma similar al POSIX-C, el sistema de archivos utiliza un puntero a una estructura de almacenamiento de datos interna.

La conversión directa entre los punteros de ambos tipos no es posible, debido a la manera dinámica de generarse la estructura de los '`Ada.Streams`', con lo cual se tiene que hacer una conversión intermedia usando el tipo '`System.Address`' para sacar la dirección de memoria del inicio del '`Stream_Array`' y utilizar esta dirección de memoria para obtener el puntero a la estructura de datos que necesita el '`Read`' del sistema de archivos.

Tampoco el parámetro del número de bytes que se desean leer se obtiene de una manera muy directa, y se tiene que hacer de una manera bastante implícita. Para ello se utiliza la longitud del '`Stream_Array`', establecida por la aplicación en su definición, de modo que esta es la manera indirecta que tiene la aplicación de establecer el número de bytes que se desean leer, o en su caso, escribir para la función '`Write`'. Además de esto, se tiene que multiplicar este valor por la constante '`Ada.Streams.Stream_Element'Size/8`' debido a que si se cambia la implementación de los '`Ada.Streams`' es posible que en lugar de aglutinarse la información byte a byte, como se hace en el caso visto, se pueda guardar en grupos de bytes, con lo que se cambiaría el tamaño de la longitud del '`Stream_Element`', y con ello la longitud de la cola.

Por último, hacer notar que en este paquete intermedio es donde se realiza la llamada a la función que eleva las excepciones '`Raise_POSIX_Error_On_Error`' ; luego aquí es donde se va a producir el final de la ejecución del procedimiento, en caso de producirse algún error, puesto que se elevaría la excepción que haya sido generada internamente por cualquiera de las funciones de entrada/salida.

2.4.5. Funciones genéricas (uso de la función 'Ioctl')

Dentro de la interfaz POSIX-Ada [12], hay que recalcar especialmente la definición de '*funciones genéricas*'. Esta característica propia del lenguaje Ada, permite que se puedan definir distintas funciones dejando sin especificar de manera concreta algún tipo de dato, que se conocen como '*tipos de datos genéricos*'.

De este modo, la especificación de la interfaz impone que se definan funciones genéricas para las funciones '`Read`' y '`Write`', teniéndose con ello las funciones genéricas '`Generic_Read`' y '`Generic_Write`'. Estas funciones quedan abiertas, y son instanciadas posteriormente en la aplicación correspondiente que quiera utilizarlas, en la cual ya se va a poder saber exactamente el tipo de dato que quiere ser usado, para cubrir todos los posibles datos genéricos que hayan quedado sin definir en la función.

En la especificación del POSIX ofrecida en el estándar, no se habla en ningún momento de la función '`Ioctl`', ya que por sí misma no es una función POSIX que deba incluirse en todos los sistemas de tiempo real que sigan este estándar, pero realmente sí que es una función de entrada/salida bastante extendida en general, y que da un gran soporte para todo el manejo de dispositivos por su carácter abierto, que permite manejar cualquier tipo de información que necesite cualquiera de todos los posibles dispositivos a utilizar.

Este carácter tan abierto y general que se le supone a la función '`Ioctl`' hace que el paso de la información no puede ser concretado de manera fija, puesto que debe ser posible que cualquier dispositivo pueda pasar cualquier información, lo cual lleva a que en su especificación se tengan que quedar tipos abiertos que permitan que se mantenga esta característica.

Para el caso de la interfaz POSIX-C, el parámetro que permite el paso de esta información general es bastante directo, puesto que para ello se puede hacer uso del '*puntero a void*' (`void *`), de manera que la especificación concreta del '`ioctl`', que se recoge en `<unistd.h>` es:

```
int ioctl (int __fd, int __request, void *__arg);
```

El hecho de que la información sea pasada correctamente a la aplicación a través del contenido del puntero entra dentro de la responsabilidad del diseñador de la aplicación, que debe seguir la especificación ofrecida para el '`ioctl`' propio del dispositivo que se desea utilizar. Este caso es perfectamente posible, puesto que el compilador no tiene por qué generar ningún error, puesto que con el hecho de pasarle cualquier puntero ya cumple las premisas requeridas por la especificación de la función, y después la información apuntada por el puntero puede ser cualquiera, de ahí la responsabilidad de la aplicación.

La interfaz POSIX-Ada ofrece mucha mayor dificultad para poder realizar este paso tan '*abierto*' de información, puesto que la misma naturaleza del lenguaje nos lo impide, ya que todos los tipos, funciones, ... deben estar perfectamente establecidos y concretos, y no hay posibilidad alguna de darle un paso de punteros tan general, como sucede en POSIX-C.

De esta manera, la función '`Ioctl`' necesita conocer perfectamente cual es el tipo de información requerida, pero eso no se puede saber hasta que no se sepa cual es el dispositivo que se desea usar, y cual es la información que su propio '`Ioctl`' ofrece. Por ello, la manera única que se tiene de utilizar esta función es a través de un *'`Ioctl` genérico'* que sea instanciado por la aplicación, cuando se sepa cuál es el dispositivo que se utiliza.

```
generic
   type Ioctl_Options_Type is (<>);
   type Ioctl_Data_Type is private;
procedure Generic_Ioctl
  (File     : in File_Descriptor;
   Request  : in Ioctl_Options_Type;
   Data     : in Ioctl_Data_Type);
```

Como para cada dispositivo se pueden dar distintas opciones y distintos tipos de información, se dejan ambos tipos como '*genéricos*', debiéndose instanciar ambos en la aplicación que deba usar el '`Ioctl`'.

En el capítulo 4, donde se recogen todos los ejemplos realizados, se profundizará más sobre la manera de utilizar esta función '`Ioctl`' genérica, mostrándose cómo se realiza la instanciación para una cola circular de tamaño configurable, que a través de su '`Ioctl`' permite que se escoja su tamaño dentro de la aplicación.

2.4.6. Variaciones sobre el código inicial de MaRTE

Ahora que ya se ha explicado detalladamente todo el diseño del nuevo sistema de archivos, en el cual se han ido incorporando toda una serie de paquetes a todo el conjunto del sistema operativo MaRTE, falta por comentar los cambios realizados sobre el código original de la versión inicial de MaRTE, cambios que afectan a la configuración del sistema operativo, que deben variarse para poder ajustarse a las nuevas partes agregadas.

Inicialmente se comienza trabajando con la versión de código de MaRTE v1.0 [13], distribuida a mediados de 2001, de manera que todos estos cambios han sido hechos sobre esta versión. Estos cambios realizados sobre el código de MaRTE, junto con todo el nuevo '*software*' desarrollado en este proyecto ya se encuentran integrados de manera completa en las nuevas versiones posteriores de MaRTE, como sucede con la nueva versión de MaRTE v1.2 [14] , lanzada durante las fechas de la escritura de esta memoria.

En un principio, para los códigos de los '*drivers*', tanto para los que se van a desarrollar en este proyecto, como para los que se vayan a ir agregando con el tiempo al sistema MaRTE OS, se va a añadir un nuevo directorio dentro de la estructura jerárquica de directorios sobre la que se organiza MaRTE, que será: `$~/marte/drivers/`

Para la correcta compilación final de un programa que utilice los '*drivers*' (tanto en Ada como en C), se debe añadir esta nueva localización dentro de los diferentes archivos de órdenes ('*scripts*') de instalación, añadiendo esta ruta de búsqueda ('*path*') a la variable *$CODE_LOCATIONS*.

Esta variable *$CODE_LOCATIONS* es utilizada por el sistema para comprobar en qué rutas de búsqueda se encuentran los paquetes que contienen código propio para la generación del sistema operativo, y que por tanto son susceptibles de ser compilados cada vez que se quiera generar una nueva aplicación MaRTE, que se apoya en el propio sistema operativo MaRTE generado a través de estos paquetes comentados. Este nuevo directorio `~/marte/drivers/` ha de ser agregado a la variable *$CODE_LOCATIONS* en dos archivos `$~/marte/utils/mkkernel` y `$~/marte/utils/mgnatmake`.

Ademas también se tienen que añadir en el '*script*' `$~/marte/utils/global.pl` los archivos objeto (*.*o*) que se tengan en este directorio `~/marte/drivers/` dentro de la variable *$KERNEL_OBJS*, para que así el enlazador pueda encontrar los archivos objeto de este nuevo directorio, que necesita el compilador GCC, ya que GNAT solo reconoce a la variable anteriormente vista, *$CODE_LOCATIONS*.

```
$KERNEL_OBJS = "...  $MPATH/drivers/*.o ";
```

Para la exportación de las primitivas de entrada/salida a la interfaz C, se tienen que seguir la normativas de POSIX.1 en cuanto a la especificación de las mismas, y se tiene:

- *'open'* se encuentra en : `<fcntl.h>` [15]
- *'close'*, *'read'*, *'write'*, se encuentran en : `<unistd.h>` [16]

La cabecera <*fcntl.h*> se genera completamente nueva utilizando el código que se nos ofrece en las referencia comentada, mientras que la cabecera <*unistd.h*> ya está presente en la versión 1.0 de MaRTE OS y posteriores, con lo cual lo que hay que hacer es habilitar el uso de estas tres primitivas, puesto que en la cabecera se encuentra el código como si fuese un comentario, de manera que el compilador no las reconozca, de manera que hay que dejar estas tres funciones *"sin comentar"*.

Ya con todos estos cambios iniciales, se tienen solucionados los problemas referidos a la compilación de cualquier aplicación, pero en la creación del archivo ejecutable '*mprogram*', que debe ser exportado, se tienen errores de enlazado ("*linker*"), debido a que en la librería `$~/marte/libmc/non_implemented.c` están definidas las tres funciones `'open'`, `'close'`, `'read'`, de tal manera que hacen saltar este error de *"linkado"* al intentar utilizar alguna de estas tres funciones.

Para solucionar este problema, se deben eliminar estas referencias de error por uso de las primitivas, que se ejecutan en esta librería `$~/marte/libmc/non_implemented.c` , y para ello hay que "*comentar*" estas líneas de error habilitadas (haciéndolo de forma similar a lo que sucedía en la cabecera <*fcntl.h*>, pero a la inversa). Ya posteriormente se recompila la librería `$~/marte/libmc/non_implemented.c` para que ya se tengan en cuenta las líneas'*comentadas*', utilizando para la compilación el archivo de órdenes `$~/marte/utils/mklibmc`

Una vez cambiados estos datos en los '*scripts*' que son utilizados en la instalación inicial, la cual es ejecutada por el archivo de órdenes principal de la instalación, es decir, `$~/marte/minstall`, a partir de ahora queda ir añadiendo en una serie de paquetes ya existentes en el sistema MaRTE unos determinados parámetros que son necesarios debido a la inclusión del nuevo sistema de archivos de dispositivos, generado con este proyecto.

Los datos que se deben variar se refieren a:

- nuevos parámetros de configuración
  (en `$~/marte/kernel/configuration_parameters.ads`)
- nuevas constantes generales
  (en `$~/marte/kernel/general_constants.ads`)
- nuevos códigos de error
  (en `$~/marte/kernel/error_codes.ads`)

a) Nuevos parámetros de configuración

En primer lugar, los valores que se añaden a los que ya se tienen en el paquete `$~/marte/kernel/configuration_parameters.ads` son los siguientes:

```
Open_Files_Mx : constant := 16;
--  The number of files that one process can have open at one time.
--  (El número de archivos que un proceso puede tener abiertos al tiempo)

Devices_Files_Mx : constant := 50;
--  The number of device files that the record can support.
--  (El número de archivos de dispositivo que el registro puede soportar)

Path_Mx : constant := 32;
--  The number of bytes in a pathname.
--  (El número de bytes en una etiqueta de ruta de búsqueda)

Buffer_Mx : constant := 32;
--  The size that a buffer can use for read or write.
--  (El tamaño que una cola puede usar para lectura o escritura)

Num_Devices_Mx : constant := 16;
--  Maximun number of devices that can be supported.
--  (El máximo número de dispositivos que pueden ser soportados)
```

Los valores numéricos son establecidos en principio sin seguir ninguna norma concreta y perfectamente pueden ser variados posteriormente, de manera que se debe hacer una nueva recompilación completa del código del paquete, aunque en realidad se debería hacer del '*kernel*' completo del MaRTE y así estos parámetros ya actuarán de acuerdo a la información variada.

b) Nuevas constantes generales

Con respecto a las nuevas constantes generales, los nuevos valores se introducen en el archivo `$~/marte/kernel/general_constants.ads`

```
--  File access mode
--  (Modo de acceso al fichero)

READ_ONLY   : constant := 0;
WRITE_ONLY  : constant := 1;
READ_WRITE  : constant := 2;
```

Estos tres modos de acceso suponen que el fichero abierto con cada determinado modo tenga capacidad de realizar ciertas funciones o no. Así, con el modo '`READ_ONLY`', sólo se permite lecturas de información y ninguna escritura en el dispositivo; con el modo '`WRITE_ONLY`', justamente lo contrario, sólo pudiéndose escribir datos sobre el dispositivo sin consultar la información a través de la lectura; y por último, el modo más común, el modo '`READ_WRITE`' que permite poder realizar cualquiera de las dos funciones sobre el dispositivo, sin restricción en sus accesos.

Estas nuevas constantes generales son añadidos al código del paquete mencionado, es decir, `$~/marte/kernel/general_constants.ads` , pero también se tiene que añadir en otro paquete llamado `~/marte/kernel/general_constants_info.ads` una información adicional que será utilizada para generar las constantes que serán usadas por las cabeceras de la interfaz C.

En este paquete se genera un registro con todas las posibles constantes utilizables por el sistema MaRTE que se encuentran en `$~/marte/kernel/general_constants.ads` y se le añaden dos campos más en el que se pone el nombre con el se quiere identificar a la constante en la interfaz C, y un pequeño comentario sobre el uso de esta constante, ya que en la interfaz C normalmente los nombres de las constantes no dejan entrever fielmente cuál es su significado real.

Esta es la nueva información que se debe colocar en el código del mencionado paquete `~/marte/kernel/general_constants_info.ads`.

```
READ_ONLY      => (General_Constants.READ_ONLY,
                   "O_RDONLY                ",
                   "Open for reading only             "),
                --  (Abierto sólo para lectura)

WRITE_ONLY     => (General_Constants.WRITE_ONLY,
                   "O_WRONLY                ",
                   "Open for writing only             "),
                --  (Abierto sólo para escritura)

READ_WRITE     => (General_Constants.READ_WRITE,
                   "O_RDWR                  ",
                   "Open for reading and writing      "),
                --  (Abierto para lectura y escritura)
```

Además de incluir en `$~/marte/kernel/general_constants_info.ads` también se tiene que incluir estos cambios sobre la información de las constantes en el archivo `$~/marte/kernel/write_marte_c_headers/general_constants_info.ads`, que será posteriormente utilizado por el archivo de órdenes '*`write_marte_c_headers`*'. para que pueda ser actualizada correctamente la parte correspondiente a la interfaz C con la misma variación de información

Estas cabeceras C son generadas internamente con la ejecución durante la instalación de este archivo de órdenes mencionado, '*`write_marte_c_headers`*', pero no se va profundizar en exceso en este aspecto porque el cometido principal del proyecto no se centra en dar una explicación exhaustiva del diseño completo del código del MaRTE OS.

La mejor manera de asegurar que esta información variada sea idéntica, y no haya problemas por tener información no actualizada errónea, sería haciendo un *"link"* entre los archivos, es decir un 'enlace lógico' entre los dos archivos distintos, acción que es posible hacerse dentro del entorno Linux sobre el que se está desarrollando el código. De este modo los dos ficheros llamados `'general_constants_info.ads'` que están en los directorios `$~/marte/kernel` y en `$~/marte/kernel/write_marte_c_headers` siempre presentarán un código idéntico, sea cual sea el paquete sobre el que se varía el código con la información que se quiere añadir.

c) Nuevos códigos de error

Por último, nos queda comentar los nuevos códigos de error que se añadieron. Estos valores se encuentran en el paquete `$~/marte/kernel/error_codes.ads`

```
PERMISSION_DENIED  : constant := 13;
--  (Permiso de acceso denegado)

BAD_FILE_DESCRIPTOR  : constant := 9;
--  (Descriptor de fichero erróneo)

TOO_MANY_OPEN_FILES  : constant := 24;
--  (Demasiados archivos abiertos en el sistema)

FILENAME_TOO_LONG  : constant := 63;
--  (Nombre de archivo demasiado largo)

NO_SUCH_FILE_OR_DIRECTORY  : constant := 2;
--  (No se encuentra este determinado archivo o directorio)
```

Los valores numéricos de las constantes no tienen ningún sentido concreto dentro del diseño del código del sistema operativo, sólo tiene que evitarse repetir los valores que ya están establecidos en el paquete anteriormente. Pero para la interfaz C sí que pueden tener un significado, ya que se podría intentar identificar a través del número además de la etiqueta. Por ello se establecen los valores numéricos buscando su equivalente UNIX [17]

Al igual que en el caso anterior, para mantener la doble compatibilidad con la interfaz C, se tiene que añadir una información extra en otro paquete, que en este caso es `~/marte/kernel/error_codes_info.ads`, y que será utilizada para que las cabeceras C generadas en la instalación incluyan estos códigos de error.

De una manera muy similar a como se hacía en el paquete de la información de las constantes del sistema, para este caso de los códigos de error, también en este paquete `~/marte/kernel/error_codes_info.ads` se genera un registro completo en el que se van incluyendo todos los posibles errores, y luego en otros campos, primero se pone el nombre con el que se identifica al error dentro de la interfaz C, y después se añade un comentario que será el texto ofrecido al usuario cuando se genere dicho error en la ejecución de una aplicación desarrollada en lenguaje C:

En este caso, la nueva información incluida concretamente dentro de este paquete `~/marte/kernel/error_codes_info.ads` es la siguiente:

```
PERMISSION_DENIED =>
  (Error_Codes.PERMISSION_DENIED,
   "EACCES        ",
   "Wrong attempt to access a file     "),
--  (Intento erróneo de acceder al fichero)


BAD_FILE_DESCRIPTOR =>
  (Error_Codes.BAD_FILE_DESCRIPTOR,
   "EBADF         ",
   "Not a valid open file descriptor   "),
--  (Descriptor de fichero abierto no es válido)

TOO_MANY_OPEN_FILES =>
  (Error_Codes.TOO_MANY_OPEN_FILES,
   "EMFILE        ",
   "Too many files open in the system  "),
--  (Demasiados ficheros abiertos en el sistema)

FILENAME_TOO_LONG =>
  (Error_Codes.FILENAME_TOO_LONG,
   "ENAMETOOLONG  ",
   "Lenght of path exceeds             "),
--  (Longitud de la ruta de búsqueda excesiva)

NO_SUCH_FILE_OR_DIRECTORY =>
  (Error_Codes.NO_SUCH_FILE_OR_DIRECTORY,
   "ENOENT        ",
   "Specified pathname does not exist  "),
--  (La ruta de búsqueda especificada no existe)
```

Además en este caso de los códigos de error, para mantener la integridad con el diseño original de MaRTE, se deben añadir también estos nuevos errores al paquete principal del '*kernel*' del sistema operativo, que se encuentra en el paquete `$~/marte/kernel/kernel.ads`, debiéndose hacerse una copia de los nuevos valores añadidos al mencionado paquete de códigos de error `$~/marte/kernel/error_codes.ads`

```
TOO_MANY_OPEN_FILES        : constant Error_Code :=
  Error_Codes.TOO_MANY_OPEN_FILES;

FILENAME_TOO_LONG          : constant Error_Code :=
  Error_Codes.FILENAME_TOO_LONG;

BAD_FILE_DESCRIPTOR        : constant Error_Code :=
  Error_Codes.BAD_FILE_DESCRIPTOR;

NO_SUCH_FILE_OR_DIRECTORY  : constant Error_Code :=
  Error_Codes.NO_SUCH_FILE_OR_DIRECTORY;

PERMISSION_DENIED          : constant Error_Code :=
  Error_Codes.PERMISSION_DENIED;
```

2. 5. - Resumen y comentarios acerca del diseño del sistema de archivos de dispositivos.

Para realizar el diseño del nuevo sistema de archivos de dispositivos de MaRTE OS, se comienza con un estudio inicial de la idea global que se quiere realizar, dando un esquema abstracto de cómo situar este nuevo nivel de dispositivos que se quiere añadir dentro de la estructura global del sistema MaRTE OS, y como hacer las comunicaciones entre los niveles de la arquitectura mediante interfaces.

Se estudian toda una serie de distintos sistemas operativos (Lynx, Linux, VxWorks) observando las características que ofrecen cada uno de ellos con respecto a los dispositivos, y de todos ellos se van tomando ideas y matices para ir formando la base del sistema de archivos de dispositivo de MaRTE OS.

Ya con todas las ideas y características recogidas sobre las que se pretende basar el diseño del sistema de archivos, entonces se puede ofrecer una idea de cómo se quiere organizar este sistema de archivos, situando cada uno de los niveles sobre los que se puede trabajar dentro de todo el diseño, para ya posteriormente entrar de lleno en la implementación.

En el comienzo de la implementación, en principio se establecen cuáles son los tipos de datos básicos que van a ser utilizados durante toda la implementación del sistema de archivos, haciendo especial hincapié sobre las tablas enlazadas, que van a guardar toda la información relevante del sistema de archivos, y que van a servir para facilitar notablemente toda la gestión de información que el sistema operativo tiene que hacer respecto al paso desde el nivel de aplicación hasta los dispositivos.

Ya se entra de lleno en la implementación, realizando la gestión interna de las funciones de entrada/salida por parte del sistema operativo, tomando la información de las funciones de entrada/salida propias de cada dispositivo, para poder diferenciar en todo momento sobre el dispositivo que se quiere trabajar.

De todas las funciones de entrada/salida sobre las que puede trabajar el sistema operativo, la función '`Open`' es la principal dentro de la gestión interna de todo el sistema de archivos de dispositivo, y como tal sobre ella hay que hay un especial hincapié, y el resto de las funciones de entrada/salida presentan ya un comportamiento bastante más similar en muchos de sus pasos, con lo cual su desarrollo se hace bastante parejo entre todas ellas.

Una vez completada toda la gestión interna de las funciones de entrada/salida, realizada de forma interna al núcleo del sistema operativo, se deben situar las interfaces que se van a necesitar usar por parte de las aplicaciones para el uso de estas funciones de entrada/salida, dos interfaces bien distintas, una para el lenguaje Ada y otra para lenguaje C

Se debe recalcar especialmente la necesidad de mantener de forma estricta en el diseño de estas interfaces la integridad de la información que se pasa por cada una de las interfaces, puesto que cada una funciona con unas especificaciones de las funciones de entrada/salida muy distintas entre sí.

Dentro de la interfaz POSIX-Ada, una de las dos que se deben realizar, hay que tener en cuenta la imposición por parte del estándar POSIX del uso de funciones genéricas, definiéndose funciones genéricas para las funciones '`Read`' y '`Write`'.

Por último también hay que tener en cuenta la inclusión de la función de entrada/salida '`ioctl`', bastante especial en su esencia, pero que es clave para el uso de muchos dispositivo por el carácter de generalidad que le da al paso de cualquier información. Por este carácter de generalidad, también en el POSIX-Ada se debe definir la función '`Ioctl`' como función genérica, y que su uso sea instanciando desde la aplicación.

Para el próximo capítulo, subiendo de nivel dentro del diseño del sistema de archivos de dispositivo, nos queda ver de qué manera se realiza la '*instalación*' concreta de un '*driver*', utilizándose para ello el paquete denominado '**Devices_Table**' ('*tabla de dispositivos*'), en el cual se guarda la información de todas las funciones de entrada/salida propias de cada uno de los dispositivos. El sistema de archivos de dispositivo toma de este paquete toda la información para gestionar las funciones de entrada/salida internas al '*kernel*', que es la parte que ha sido desarrollada más en profundidad en el capítulo que ahora termina.

Capítulo 3

Modo de uso de '*drivers*' de dispositivos,
a través del sistema de archivos de MaRTE.

3. 1. - Instalación de '*drivers*' en la tabla de dispositivos.

En el capítulo anterior se habló en profundidad del nuevo sistema de archivos de dispositivo creado para ser utilizado por MaRTE OS, pero todo lo que se comentó se quedó siempre en niveles internos del *'kernel'*, sin llegar a poder situarnos en ningún momento en el papel de diseñador de '*drivers*' útiles para el sistema MaRTE.

En este capítulo se va a explicar de qué manera se tiene que actuar en el momento de realizar un '*driver*' de dispositivo, cómo se tiene que diseñar, qué pasos hay que seguir a la hora de la instalación,.. y todo esto se documentará con un par de ejemplos de '*drivers*' sencillos, uno realizado en Ada, y otro en lenguaje C.

3.1.1. El paquete 'Devices_Table'.

En el diseño del sistema de archivos de dispositivos, se habló en numerosas ocasiones acerca de las tablas enlazadas (véase figura 5, en la sección 2.3.2.) utilizadas en el diseño preliminar del sistema de archivos de dispositivo, que van a ser las que contengan toda la información acerca de los descriptores de ficheros (***'The_Fd_Table'***), acerca de los archivos de dispositivos (***'The_Device_Files_Table'***), o bien de las funciones de entrada/salida de cada dispositivo (***'The_Drivers_Table'***).

Estas tablas han sido definidas como tipos de datos en el paquete más interno de todos, el '`Kernel.File_System_Data_Types`', en el cual sólo se definen los tipos de variables que el sistema operativo utiliza durante toda la implementación del sistema de archivos. Pero las tablas enlazadas concretas se declaran en este otro paquete externo al '*kernel*', '`Devices_Table`', del cual vamos a hablar.

Este paquete '`Devices_Table`' tiene como finalidad recoger toda la información que necesita ser definida para todos los dispositivos en una única interfaz. De este modo, cualquiera que necesite incorporar algún '*driver*' al sistema operativo o bien variar algún parámetro relativo a otro '*driver*' ya incluido previamente en el sistema, tiene todo la información necesaria en un único paquete, y no tiene que conocer mucho más de la estructura del sistema operativo para manejar '*drivers*'.

En el paquete se va a trabajar con dos conceptos bien distintos. Por un lado, se encuentran todos los punteros a las funciones de entrada/salida con los que se puede trabajar en el sistema, que son tomados de las propias funciones declaradas en las distintas especificaciones de cada uno de los '*drivers*' que se incluyen en el sistema, información de los punteros que se recoge en la tabla *'The_Driver_Table'*. Por otro lado, se tienen los archivos de dispositivos, que se ofrecen a los usuarios de las aplicaciones como modo intermedio de utilizar los dispositivos, y que cuya información se guarda en la tabla *'The_Device_Files_Table'*

Destacar, por último, que toda la información guardada por los ficheros de dispositivo, a diferencia de otros sistemas, se representan como constantes en memoria. De ahí que se tenga un carácter 'estático' en el uso de estos ficheros de dispositivo.

Para poder situar todas las ideas que surgen acerca del paquete '`Devices_Table`', veamos con una figura la situación de los paquetes del sistema de archivos de dispositivo sobre los que se va a trabajar.

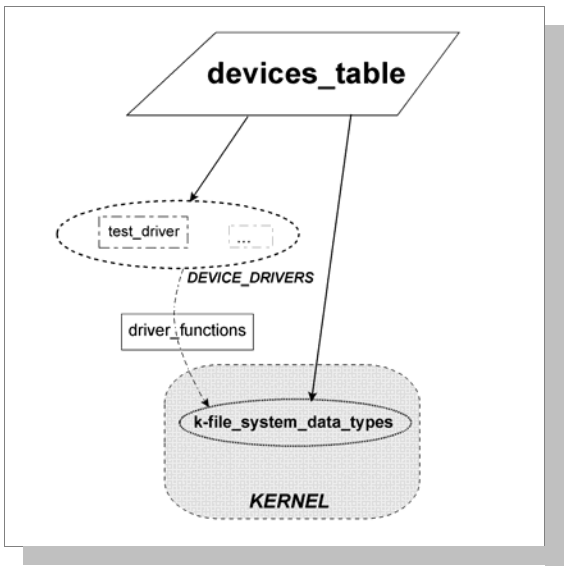

Figura 10 . Enlace entre la tabla de dispositivos y el '*kernel*' de MaRTE OS

Para entender mejor los modos de uso del paquete '`Devices_Table`', mostraremos primeramente su especificación completa, a través del código desarrollado. En esta especificación se hace uso de una serie de '*drivers*' que posteriormente serán comentados en profundidad en el capítulo 4, como son el '*driver*' para el puerto paralelo, para el puerto serie; los '*drivers*' de las colas de almacenamiento que funcionan como dispositivos virtuales ; y además también una serie de '*drivers*' de prueba que serán comentados en este mismo capítulo en el apartado 3.2.

```
------------------------------------------------------------------------------
-- --------------------       M a R T E   O S         ------------------- --
------------------------------------------------------------------------------
--
--                     ' D e v i c e s _ T a b l e'
--
--                                  Spec
--
--  File 'devices_table.ads                                  By Fguerreira
--
--
--  Table definition where all the drivers are loaded
------------------------------------------------------------------------------
```

```
--  with {your_driver}
with Parallel_Port_Driver.Functions;   use Parallel_Port_Driver.Functions;
with Serial_Port_Driver_Import;        use Serial_Port_Driver_Import;
with Cond_Variable_Buffer_Driver;      use Cond_Variable_Buffer_Driver;
with Dynamic_Buffer_Driver.Functions;  use Dynamic_Buffer_Driver.Functions;
with Test_Ada_Driver;                  use Test_Ada_Driver;
with Test_C_Driver_Import;             use Test_C_Driver_Import;

with Kernel.File_System_Data_Types;    use Kernel.File_System_Data_Types;
with Kernel.Mutexes;


package Devices_Table is

   --  any number available =>
   --  (cualquier número disponible) =>
   --
   --  ( (your_driver}_create_ac, {your_driver}_remove_ac,
   --    {your_driver}_open_ac, {your_driver}_close_ac,
   --    {your_driver}_read_ac, {your_driver}_write_ac,
   --    {your_driver}_ioctl_ac )

   The_Driver_Table : constant Kernel.File_System_Data_Types.Driver_Table_Type :=
     (
      1 => (Standard_Input_Create_Ac, Standard_Input_Remove_Ac,
            Standard_Input_Open_Ac,   Standard_Input_Close_Ac,
            Standard_Input_Read_Ac,   Standard_Input_Write_Ac,
            Standard_Input_Ioctl_Ac),
      2 => (Standard_Output_Create_Ac, Standard_Output_Remove_Ac,
            Standard_Output_Open_Ac,   Standard_Output_Close_Ac,
            Standard_Output_Read_Ac,   Standard_Output_Write_Ac,
            Standard_Output_Ioctl_Ac),
      3 => (Standard_Error_Create_Ac, Standard_Error_Remove_Ac,
            Standard_Error_Open_Ac,   Standard_Error_Close_Ac,
            Standard_Error_Read_Ac,   Standard_Error_Write_Ac,
            Standard_Error_Ioctl_Ac),
      4 => (Serial_Port_Create_Ac, Serial_Port_Remove_Ac,
            Serial_Port_Open_Ac,   Serial_Port_Close_Ac,
            Serial_Port_Read_Ac,   Serial_Port_Write_Ac,
            Serial_Port_Ioctl_Ac),
      5 => (Parallel_Port_Create_Ac, Parallel_Port_Remove_Ac,
            Parallel_Port_Open_Ac,   Parallel_Port_Close_Ac,
            Parallel_Port_Read_Ac,   Parallel_Port_Write_Ac,
            Parallel_Port_Ioctl_Ac),
      6 => (Cond_Variable_Buffer_Create_Ac, Cond_Variable_Buffer_Remove_Ac,
            Cond_Variable_Buffer_Open_Ac,   Cond_Variable_Buffer_Close_Ac,
            Cond_Variable_Buffer_Read_Ac,   Cond_Variable_Buffer_Write_Ac,
            Cond_Variable_Buffer_Ioctl_Ac),
      7 => (Dynamic_Buffer_Create_Ac, Dynamic_Buffer_Remove_Ac,
            Dynamic_Buffer_Open_Ac,   Dynamic_Buffer_Close_Ac,
            Dynamic_Buffer_Read_Ac,   Dynamic_Buffer_Write_Ac,
            Dynamic_Buffer_Ioctl_Ac),
      8 => (Test_Ada_Create_Ac, Test_Ada_Remove_Ac,
            Test_Ada_Open_Ac,   Test_Ada_Close_Ac,
            Test_Ada_Read_Ac,   Test_Ada_Write_Ac,
            Test_Ada_Ioctl_Ac),
      9 => (Test_C_Create_Ac, Test_C_Remove_Ac,
            Test_C_Open_Ac,   Test_C_Close_Ac,
            Test_C_Read_Ac,   Test_C_Write_Ac,
            Test_C_Ioctl_Ac),
      others => (null, null, null, null, null, null, null)
  );


   Ceiling_Mx : constant Kernel.Mutexes.Ceiling_Priority :=
                  Kernel.Mutexes.Ceiling_Priority'Last;
```

```
   The_Device_Files_Table :
     constant Kernel.File_System_Data_Types.Device_Files_Table_Type :=
     (
       1  => (To_Path ("standard_input"),              1, 1, Ceiling_Mx, True),
       2  => (To_Path ("standard_output"),             2, 1, Ceiling_Mx, True),
       3  => (To_Path ("standard_error"),              3, 1, Ceiling_Mx, True),
       4  => (To_Path ("serial_port_driver_com1"),     4, 1, Ceiling_Mx, True),
       5  => (To_Path ("serial_port_driver_com2"),     4, 2, Ceiling_Mx, True),
       6  => (To_Path ("serial_port_driver_com3"),     4, 3, Ceiling_Mx, True),
       7  => (To_Path ("serial_port_driver_com4"),     4, 4, Ceiling_Mx, True),
       8  => (To_Path ("parallel_port_driver"),        5, 1, Ceiling_Mx, True),
       9  => (To_Path ("cond_variable_buffer_driver"), 6, 1, Ceiling_Mx, True),
       10 => (To_Path ("dynamic_buffer_driver"),       7, 1, Ceiling_Mx, True),
       11 => (To_Path ("test_ada_driver"),             8, 1, Ceiling_Mx, True),
       12 => (To_Path ("test_c_driver"),               9, 1, Ceiling_Mx, True),
       others => (To_Path (""), Major'Last, Minor'Last, Ceiling_Mx, False)
       );

   --  the first number in the assigment (major_number) must be the one
   --  that points to the driver in The_Driver_Table rows
   --
   --  (el primer número en la asignación [Número_Mayor] debe ser aquel
   --   que apunte al manejador en la fila de la tabla The_Driver_Table)
   --
   --
   --  the second number in the assigment (minor_number) may be anyone,
   --  as it's used only as a number to differentiate assigments
   --  of the same Major_Number
   --
   --  (el segundo número en la asignación [Número_Menor] puede ser cuaquiera
   --   ya que es usado únicamente como un número para diferenciar asignaciones
   --   del mismo Número_Mayor)
   --
   --
   --  the third in the assigment (mutex_ceiling_prio) matches with
   --  the value that all the mutex created from this device file
   --  will use as ceiling priority
   --
   --  (el tercero en la asignación [Techo_Prioridad_Mutex] concuerda con
   --   el valor que todos los mutex creados del mismo archivo de dispositivo
   --   usarán como techo de prioridad)
   --
   --
   --  the Boolean indicates whether the assigment is in correct use
   --  or it has no meaningful use
   --
   --  (el Booleano indica si la asignación está dentro de un uso correcto
   --   o si tiene un uso sin ningún significado)


   The_Fd_Table :
       constant Kernel.File_System_Data_Types.File_Descriptor_Table_Type;


end Devices_Table;
```

3.1.2. Modo de introducir un nuevo 'driver' en la tabla

El modo de introducir el '*driver*' de un nuevo dispositivo del sistema, utilizándose el paquete '`Devices_Table`', implica cubrir tres aspectos bien claros y diferenciados.

- Se debe incluir el paquete asociado a la especificación propia del nuevo '*driver*' generado, lo que en el código se quiere dar a entender con el comentario inicial `-- with {your_driver}`. En este paquete se deben tener disponibles todos los punteros a las funciones de entrada/salida propias del '*driver*'. No es necesario incluir los paquetes del '*driver*' donde se ofrezca una especificación de tipos y constantes propias del '*driver*'; esto sólo será necesario a nivel de la aplicación, tal y como se observó en el capítulo anterior en el diseño completo (véase figura 5, sección 2.3.2.) .

- Se debe realizar la asignación de los punteros a las funciones de entrada/salida propias del nuevo '*driver*' en la tabla '`The_Driver_Table`'. En la asignación de esta tabla se debe escoger un número de una fila que se encuentre disponible. Lo más común es hacerlo con la siguiente fila a la última ocupada, aunque esto no es estrictamente necesario.

  En esta fila se deben ir colocando sucesivamente los punteros de las funciones de entrada/salida propias del nuevo '*driver*', que son accesibles gracias al hecho de haber incluido la especificación del '*driver*' con la orden '`with`'. Esta colocación de las funciones de manera sucesiva ha de seguir un orden preestablecido por la definición realizada previamente del tipo '`Driver_Table_Type`', y para poder cumplir este orden se ofrece en el código la guía para seguir el correcto orden:

```
--  any number available => (
--  {your_driver}_create_ac, {your_driver}_remove_ac,
--  {your_driver}_open_ac,   {your_driver}_close_ac,
--  {your_driver}_read_ac,   {your_driver}_write_ac,
--  {your_driver}_ioctl_ac )
```

- Se debe generar un nuevo archivo de dispositivo en la tabla intermedia asociada '`The_Device_Files_Table`'. De manera similar a la asignación en la anterior tabla, se debe escoger un número de fila disponible, y así ya se pueden ir rellenando de forma sucesiva los distintos campos que se corresponden con el tipo ya predefinido '`Device_Files_Table_Type`'.

  Al igual que para el caso de los punteros a funciones, también en el código del paquete se aportan unos comentarios de guía acerca de cómo van ordenados los campos y qué significado tienen.

  En primer lugar se coloca el nombre con el cual se quiere identificar al archivo de dispositivo. Después se deben situar el número mayor y el numero menor. El número mayor debe corresponderse con la fila de la tabla '`The_Driver_Table`', utilizada anteriormente, escogiendo el número de la fila del dispositivo al cual se correspondan las funciones de entrada/salida que se desean asociar con el nuevo archivo de dispositivo.

El número menor puede ser cualquiera que entre dentro del rango del tipo interno. Sólo se restringe su uso para archivos de dispositivos que quieran utilizar el mismo dispositivo mayor, teniéndose que utilizar diferentes números menores, cualesquiera que sea con tal de que no haya repetición de valores.

El valor siguiente es utilizado para dar el valor del techo de prioridad que se le desea asignar a todos los '*mutex*' asociados a descriptores de ficheros creados a partir de un mismo archivo de dispositivo, es decir, '*mutexes*' cuyo cursor apunte al mismo archivo de dispositivo van a tener el mismo techo de prioridad, aunque tengan distinto valor de descriptor de fichero.

Por último todas aquellas filas que hayan sido creadas como nuevos archivos de dispositivo deben establecer el valor booleano '`True`', ya que éste será el campo utilizado internamente en el '*kernel*' para saber si el archivo de dispositivo es correcto y si está en uso, porque sólo con colocar este campo a '`False`' ya se anula cualquier tipo de significado del resto de los campos anteriores.

Además también se puede observar que en este paquete '`Devices_Table`' se tiene declarada la tabla de descriptores de fichero que van a ser utilizados ('`The_Fd_Table`'). Pero en esta interfaz no tiene que ser introducido ningún valor concreto en esta tabla, ya que el usuario programador de '*drivers*' no tiene que realizar ningún uso sobre ella, puesto que toda la información correspondiente a descriptores de ficheros es manipulada internamente al '*kernel*' a través del paquete del conjunto de funciones internas del sistema de archivos, que se encuentran en '`Kernel.File_System`'.

La idea de realizar esta declaración de las tablas donde se recoge toda la información de los '*drivers*' y dispositivos del sistema en un paquete separado y visible en el sentido de que sea ofrecido al usuario programador de '*drivers*', es para unificar en un único lugar la idea '*inicialización de drivers*'. Esta inicialización, como tal, se puede considerar que es "*estrictamente estática*" porque se obliga al usuario a introducir el nuevo '*driver*' en este paquete '`Devices_Table`', a través de los punteros a las funciones de entrada/salida. Luego ya posteriormente, se debe recompilar el '*kernel*' para que el nuevo '*driver*' que haya sido introducido esté ya disponible, y para ello, bien puede ejecutarse el archivo de órdenes `$~/marte/utils/mkkernel`, o bien hacer una recompilación completa de todos los niveles, con otro archivo de órdenes `$~/marte/utils/mkall`.

Queda claro comentar que esta '*inicialización estática*' supone que en tiempo de ejecución no puede ser añadido ningún '*driver*' de dispositivo nuevo al sistema, condición que queda bastante ligada al hecho de que el sistema MaRTE siga el estándar POSIX para pequeños sistemas empotrados, en el cual se establece que no existe una estructura compleja de ficheros jerárquica que pueda ser tratada en tiempo de ejecución. Realmente este hecho de tener muy claramente definida la estructura que tienen que seguir los archivos de dispositivos hace que su manejo sea tan simple, como ha podido ser visto en esta explicación de su utilización.

3. 2. - Interfaz de un '*driver*' de prueba realizado en Ada.

Para comenzar a utilizar '*drivers*' dentro del sistema operativo MaRTE, lo mejor es comenzar diseñando un '*driver*' sencillo y simple que sirva como base para poderse utilizar como plantilla para futuros nuevos '*drivers*'.

Este '*driver*' inicial va a ser desarrollado en Ada, puesto que el paso de los punteros de sus funciones propias hacia la tabla de dispositivos, es prácticamente inmediata, y como primer contacto con el uso de la tabla de dispositivos su forma es bastante intuitiva. En el '*driver*' se van a definir todas las funciones de entrada/salida que son susceptibles de ser usadas por parte del sistema operativo, y posteriormente para el paso hacia la tabla se tendrán que utilizar los punteros de estas propias funciones, aquí definidas.

Estos punteros que se introducen en la tabla se definen a partir de unos tipos establecidos en un paquete externo, '`Driver_Functions`', que va a ser de uso obligado para todos los '*drivers*'. Como podrá comprobarse posteriormente, al inicio del código de la especificación del '*driver*', este paquete '`Driver_Functions`' se incluye en la especifición con un '`with`'. El paquete es utilizado como una interfaz intermedia, para hacer de transición hacia el paquete interno '`Kernel.File_System_Data_Types`', de manera que el usuario quede siempre por encima del nivel de '*kernel*'. En el paquete '`Driver_Functions`' se hace la inclusión de todos los tipos de variables puntero a las distintas funciones de entrada/salida que se van a usar en todas las especificaciones de los '*drivers*' ('`Create_Function_Ac`', '`Remove_Function_Ac`', ...)

Otros usos que se pueden hacer del paquete '`Driver_Functions`' consisten en la inclusión de otras funciones distintas que puedan ser accesibles para todos los '*drivers*' en general, como puede ser obtener el número mayor o menor de un dispositivo a través de la información ofrecida por un descriptor de fichero ('`Get_Major`', '`Get_Minor`', ...) , como se verá en el capítulo 4 para el '*driver*' del puerto serie; o también puede utilizarse para hacer visibles operaciones sobre tipos internos del paquete `Kernel.File_System_Data_Types`, como pueden ser sumas internas de los tipos allí definidos, o también operaciones de orden.

Veamos entonces, cuál es el código que se utiliza para generar esta especificación del '*driver*' básico, primero desarrollado en Ada:

```
------------------------------------------------------------------------------
-- --------------------        M a R T E   O S         ------------------- --
------------------------------------------------------------------------------
--
--                     'T e s t _ A d a _ D r i v e r'
--
--                                 Spec
--
--
--  File 'test_ada_driver.ads'                                  By Fguerreira
--
--  A driver with all the I/O functions declared, in order to test
--
------------------------------------------------------------------------------

with Driver_Functions;  use Driver_Functions;


package Test_Ada_Driver is

   function Test_Ada_Create
     return Int;
   Test_Ada_Create_Ac : Create_Function_Ac := Test_Ada_Create'Access;


   function Test_Ada_Remove
     return Int;
   Test_Ada_Remove_Ac : Remove_Function_Ac := Test_Ada_Remove'Access;


   function Test_Ada_Open
     (Fd   : in File_Descriptor;
      Mode : in File_Access_Mode)
     return Int;
   Test_Ada_Open_Ac : Open_Function_Ac := Test_Ada_Open'Access;


   function Test_Ada_Close
     (Fd : in File_Descriptor)
     return Int;
   Test_Ada_Close_Ac : Close_Function_Ac := Test_Ada_Close'Access;


   function Test_Ada_Read
     (Fd         : in File_Descriptor;
      Buffer_Ptr : in Buffer_Ac;
      Bytes      : in Buffer_Length)
     return Int;
   Test_Ada_Read_Ac : Read_Function_Ac := Test_Ada_Read'Access;


   function Test_Ada_Write
     (Fd         : in File_Descriptor;
      Buffer_Ptr : in Buffer_Ac;
      Bytes      : in Buffer_Length)
     return Int;
   Test_Ada_Write_Ac : Write_Function_Ac := Test_Ada_Write'Access;


   function Test_Ada_Ioctl
     (Fd             : in File_Descriptor;
      Request        : in Ioctl_Option_Value;
      Ioctl_Data_Ptr : in Buffer_Ac)
     return Int;
   Test_Ada_Ioctl_Ac : Ioctl_Function_Ac := Test_Ada_Ioctl'Access;


end Test_Ada_Driver;
```

Como se ha dicho en la introducción de este capítulo, esta interfaz donde están declarados todos los punteros a funciones va a ser el prototipo de interfaz que debe ser utilizada siempre que se desarrolle un '*driver*' en Ada. De este modo, todos los '*drivers*' podrán ser usados y tratados de la misma manera desde el sistema de archivos, y ya sólo se diferenciarán internamente en su implementación.

Para que esta interfaz sea válida para cualquier dispositivo solamente hay que cambiar la etiqueta '`Test_Ada_`', que precede a todos los nombres de funciones y variables que se usan en esta interfaz por otra etiqueta con la que se identifique al dispositivo. p.ej. para el puerto paralelo, que será comentado posteriormente, se utilizará la etiqueta '`Parallel_Port_`' en lugar de la vista '`Test_Ada_`'.

Ya internamente al '*driver*', no hay ninguna restricción que aplicar en la implementación que se haga del mismo, sólo cumplir las características de las funciones que se han expuesto anteriormente en la especificación típica de un '*driver*' en Ada. La manera más sencilla de comprobar el funcionamiento de la implementación del '*driver*'es haciendo que las funciones de entrada/salida en su ejecución interna escriban unos mensajes en pantalla, de manera que cada vez que se haga una llamada a la respectiva función de entrada/salida que corresponda se sepa que se ha alcanzado dicho punto en la ejecución del programa.

Es posible que algunos '*drivers*' necesiten una especificación externa destinada a usuarios que quieran utilizar este '*driver*', con tipos de datos concretos del dispositivo o constantes que simplifiquen el manejo del mismo, pero sin tener por qué dar la información de las funciones de entrada/salida utilizadas. Para poder realizar esto, se puede hacer que esta especificación más externa sea visible en un paquete único, que tome el nombre del dispositivo que va a utilizar el '*driver*' ['`(any_device)`'] ; y ya en otro paquete interno ofrecer únicamente la especificación de las funciones de entrada/salida, que va a ser el único paquete que se necesita incluir en '`Devices_Table`'.

A este nuevo paquete interno se le puede renombrar '`(any_device).Functions`' desempeñando el mismo papel que en la especificación ya vista hacía el paquete '`Test_Ada_Driver`'. El hecho de poner el sufijo '`.Functions`' es como un nombre de ejemplo, bastante intuitivo por el hecho de encontrarse en este paquete las funciones de entrada/salida, pero se puede poner cualquier sufijo, con tal de que se cumpla que delante de éste se tenga el nombre del dispositivo seguido de '.', teniéndose de esta manera que cumplirse la expresión '`(any_device).(any_sufix)`' como nombre válido para este paquete.

3. 3. - Interfaz de un '*driver*' de prueba realizado en C.

En este otro caso de desarrollo de '*drivers*', para lenguaje C, no se va a tener una interfaz tan simplificada para el usuario programador de '*drivers*', puesto que se tienen que realizar varios pasos intermedios para poder tener un perfecto paso de las funciones de entrada/salida que utilizará el '*driver*' en C a los punteros de las funciones de entrada/salida que se utilizan de manera interna en el sistema de archivos, desarrollado en lenguaje Ada.

Como se comentó para el caso de la exportación en la interfaz POSIX-C de las funciones de entrada/salida internas al sistema de archivos, para que se tenga un paso correcto de la información de cada uno de los parámetros, las funciones deben tener los tipos equivalentes en cada una de las interfaces. Por ello, las funciones de entrada/salida propias del '*driver*' en C tienen que tomar unas formas preestablecidas, tal y como se tiene en la especificación de los '*drivers*' en lenguaje Ada.

De manera similar a las especificaciones que se utilizan para los paquete Ada, en el caso del lenguaje se generan unos archivos de cabecera ('*header*') en los cuales se definen las funciones que serán utilizadas en la implementación interna, llevada a cabo por el archivo C.

Para ver cuáles son estas formas que deben tomar las funciones de entrada/salida de cualquier '*driver*' que se vaya a escribir en C, tomemos las declaraciones de las funciones de entrada/salida del '*driver*' de prueba realizado, que se encuentran en el archivo de cabeceras (archivo '**.h*') creado para su definición:

```
int test_c_create (int arg);

int test_c_remove ();

int test_c_open (int file_descriptor, int file_access_mode);

int test_c_close (int file_descriptor);

unsigned int test_c_read
   (int file_descriptor, void *buffer, unsigned int bytes);

unsigned int test_c_write
   (int file_descriptor, void *buffer, unsigned int bytes);

int test_c_ioctl (int file_descriptor, int request, void* argp);
```

Al igual que en el caso del '*driver*' en Ada, para cualquier nuevo '*driver*' en C que se quiera desarrollar, únicamente habría que cambiar la etiqueta '`test_c_`' por la etiqueta con la que se distingue al nuevo dispositivo '`(any_device)`', para que así el compilador pueda tener capacidad de diferenciar las funciones de entrada/salida propias de cada dispositivo, y por supuesto, aportar la nueva implementación que tendrían las funciones de entrada/salida del nuevo '*driver*' a desarrollar.

Como se comentó brevemente en la introducción del apartado, para este caso de los '*drivers*' realizados en lenguaje C, se debe establecer una interfaz intermedia en la cual se haga el paso desde las funciones en el lenguaje C, en el que se desarrolla el '*driver*', al lenguaje Ada, utilizado por el sistema de archivos. Esta interfaz intermedia, ya escrita en lenguaje Ada, va a ser la que realice la importación hacia los punteros de funciones Ada de las funciones de entrada/salida del '*driver*', hechas en C, de manera complementaria a como se hacía la exportación de funciones Ada hacia las funciones C en la interfaz POSIX. A continuación se muestra esta interfaz:

```
------------------------------------------------------------------------------
-- --------------------        M a R T E   O S         ------------------- --
------------------------------------------------------------------------------
--
--                  'T e s t _ C _ D r i v e r _ I m p o r t'
--
--                                    Spec
--
--  File 'test_c_driver_import.ads'                            By Fguerreira
--
--  Importing the functions of the driver in C used to test
--
------------------------------------------------------------------------------

with Driver_Functions;  use Driver_Functions;
with System, Ada.Unchecked_Conversion;


package Test_C_Driver_Import is

   pragma Linker_Options
     ("/home/paco/aplicaciones/marte/drivers/test_c_driver.o");


   function Test_C_Create
     (Create_Arg : in Int)
     return Int;
   pragma Import (C, Test_C_Create, "test_c_create");
   function Address_To_Create_Ac is new
     Ada.Unchecked_Conversion (System.Address, Create_Function_Ac);
   Test_C_Create_Ac : constant Create_Function_Ac :=
                       Address_To_Create_Ac (Test_C_Create'Address);


   function Test_C_Remove
     return Int;
   pragma Import (C, Test_C_Remove, "test_c_remove");
   function Address_To_Remove_Ac is new
     Ada.Unchecked_Conversion (System.Address, Remove_Function_Ac);
   Test_C_Remove_Ac : constant Remove_Function_Ac :=
                       Address_To_Remove_Ac (Test_C_Remove'Address);


   function Test_C_Open
     (Fd   : in File_Descriptor;
      Mode : in File_Access_Mode)
     return Int;
   pragma Import (C, Test_C_Open, "test_c_open");
   function Address_To_Open_Ac is new
     Ada.Unchecked_Conversion (System.Address, Open_Function_Ac);
   Test_C_Open_Ac : constant Open_Function_Ac :=
                     Address_To_Open_Ac (Test_C_Open'Address);
```

```
   function Test_C_Close
     (Fd : in File_Descriptor)
     return Int;
   pragma Import (C, Test_C_Close, "test_c_close");
   function Address_To_Close_Ac is new
     Ada.Unchecked_Conversion (System.Address, Close_Function_Ac);
   Test_C_Close_Ac : constant Close_Function_Ac :=
                      Address_To_Close_Ac (Test_C_Close'Address);


   function Test_C_Read
     (Fd         : in File_Descriptor;
      Buffer_Ptr : in Buffer_Ac;
      Bytes      : in Buffer_Length)
     return Int;
   pragma Import (C, Test_C_Read, "test_c_read");
   function Address_To_Read_Ac is new
     Ada.Unchecked_Conversion (System.Address, Read_Function_Ac);
   Test_C_Read_Ac : constant Read_Function_Ac :=
                     Address_To_Read_Ac (Test_C_Read'Address);


   function Test_C_Write
     (Fd         : in File_Descriptor;
      Buffer_Ptr : in Buffer_Ac;
      Bytes      : in Buffer_Length)
     return Int;
   pragma Import (C, Test_C_Write, "test_c_write");
   function Address_To_Write_Ac is new
     Ada.Unchecked_Conversion (System.Address, Write_Function_Ac);
   Test_C_Write_Ac : constant Write_Function_Ac :=
                      Address_To_Write_Ac (Test_C_Write'Address);


   function Test_C_Ioctl
     (Fd             : in File_Descriptor;
      Request        : in Ioctl_Option_Value;
      Ioctl_Data_Ptr : in Buffer_Ac)
     return Int;
   pragma Import (C, Test_C_Ioctl, "test_c_ioctl");
   function Address_To_Ioctl_Ac is new
     Ada.Unchecked_Conversion (System.Address, Ioctl_Function_Ac);
   Test_C_Ioctl_Ac : constant Ioctl_Function_Ac :=
                      Address_To_Ioctl_Ac (Test_C_Ioctl'Address);

end Test_C_Driver_Import;
```

Lo primero a señalar para esta especificación intermedia es la inclusión que se debe hacer del archivo de '*código objeto*' (`test_c_driver.o`) generado a partir de la compilación en C del '*driver*'. Esta es la manera que va a tener el compilador de Ada de conocer las funciones importadas, y como tantas veces se ha reiterado, los tipos de los parámetros utilizados para las funciones han de mantener la integridad completa para que el compilador pueda hace un paso de información completamente transparente.

Esta inclusión del código objeto del '*driver*' escrito en C, y por consiguiente de sus funciones de entrada/salida, se realiza a través de la orden "`pragma Linker_Options`" [18] Este '*pragma*' utilizado por el compilador se usa para especificar al enlazador del sistema ('*linker*') algunos parámetros que se necesitan cuando una unidad de compilación dada se incluye en una partición externa.

El hecho de tener que incluir este archivo objeto '`test_c_driver.o`' hace que se tenga que renombrar a este paquete intermedio con otro identificador que no sea '`Test_C_Driver`' ya que el compilador Ada generaría otro archivo objeto del mismo nombre, lo cual produciría una confusión al '*enlazador*', generando un error en tiempo de enlazado. Por ello, el paquete Ada intermedio para la importación de función se denomina con el nombre '`Test_C_Driver_Import`', aunque como sucede para el caso ya visto del '*driver*' de Ada que generaba un paquete interno de nombre '`(any_device).Functions`', se puede utilizar cualquier nombre '`(any_device)_(any_sufix)`', en este caso con una variación, ya que debe estar separado por '_' en lugar de '.', ya que el separador '.' se utiliza para expresar la idea de jerarquía y herencia de paquetes, que para este caso no es necesaria.

La conversión de los punteros que se debe llevar a cabo para que los punteros referidos a las funciones importadas del '*driver*' C a través del código objeto se conviertan en punteros apropiados que puedan ser pasados a la tabla de dispositivos, no es completamente directa, ni mucho menos, puesto que el compilador no permite hacer una asignación directa con la que la variable puntero tome el valor del puntero de la función, ya que los tipos sobre los que trabaja son distintos.

Esto supone pensar que se podría hacer una conversión de tipos 'puntero', pero tampoco se puede realizar ya que se tienen problemas en la compilación del paquete con la convención utilizada para la conversión del puntero ('*convention*') que se tiene de la función importada hacia un puntero Ada. De esta manera, se llega a la conclusión de que sólo se puede hacer el proceso de transformación a través del '*Unchecked_Conversion*', que realiza la conversión sin chequear los tipos, dejando al programador la responsabilidad de comprobar que los contenidos de los punteros sean correctos. Así, se saca la información de la dirección de memoria correspondiente a la función importada, mediante el uso del tipo '*System.Address*', y posteriormente se convierte este tipo de dirección de memoria al tipo de puntero correspondiente a la función de entrada/salida, para poder ser pasado correctamente a su lugar correspondiente dentro de la tabla de dispositivos.

Al igual que sucedía en el caso visto del '*driver*' en lenguaje Ada, los punteros a las funciones de entrada/salida que van a ser entregados a la tabla de dispositivos son renombrados de acuerdo al dispositivo que se trate. De este modo, para cualquier otro dispositivo, se debería sustituir la etiqueta '`Test_C_`' de las variables puntero, y colocar la etiqueta con el nombre apropiado al dispositivo '`(any_device)_`' . También, de igual manera que sucedía como el anterior apartado 3.2, la implementación interna del '*driver*' es realizada de tal manera que sean presentados unos mensajes en pantalla que supongan una manera de comprobar que las llamadas a las funciones de entrada/salida se realizan de forma correcta.

3. 4. - Resumen acerca del primer uso de '*drivers*' de dispositivos.

En este capítulo se han comenzado a utilizar los '*drivers*' dentro del MaRTE OS, mediante el uso de los punteros a las funciones de entrada/salida de cada dispositivo, recogida en la tabla de dispositivos del sistema.

El primer uso de '*drivers*' obliga a comenzar por saber cuál es el modo que se tiene que seguir para agregar un nuevo '*driver*' dentro del sistema operativo, y para eso, se tiene que escribir sobre la tabla de dispositivos la información apropiada y con su orden bien establecido. Hay que hacer notar el hecho de que dicha tabla, una vez que se ha variado la información, después ha de ser recompilada, dado el carácter estático que tiene la inicialización de '*drivers*'.

Para familiarizarnos realmente con el uso de '*drivers*' reales, se desarrollan dos sencillos ejemplos de '*drivers*' en la tabla de dispositivos, uno de ellos realizado en Ada, de una manera más simple; y otro desarrollado en C, éste ya con una mayor complejidad. El '*driver*' en C es más complicado debido a que su transformación no es directa y se tiene que utilizar un paquete intermedio, éste desarrollado en Ada, en el que se van a importar las funciones de entrada/salida del '*driver*', éstas escritas en C. Para poder utilizar estas funciones importadas dentro de la tabla de dispositivos se tiene que realizar un enlazado con el archivo de 'código objeto' generado a través de una compilación C del '*driver*', y posteriormente convertir los punteros de estas funciones para poder agregarlos correctamente dentro de la tabla de dispositivos.

Ahora nos queda entrar mucho más en profundidad en toda una serie de '*drivers*' de dispositivos propios de cualquier sistema, que tienen bastante mayor complejidad que los dos ejemplos desarrollados, pero estos dos simples ejemplos nos sirven perfectamente para poder comprobar fielmente el correcto paso de la información ofrecida por las funciones de entrada/salida propias de los dispositivos ('virtuales' en estos simples casos vistos), constatando que a través de la gestión del sistema operativo llega esta información hasta las funciones de entrada/salida POSIX, que son las que realmente van a ser usadas por el nivel de aplicación.

Capítulo 4

Ejemplos de ‘*drivers*’ de dispositivos y aplicaciones desarrolladas para su uso.

4. 1. - Primer contacto con las funciones de entrada/salida.

4.1.1. 'Driver' con cola circular para almacenamiento de datos con tamaño configurable desde la aplicación

El primer '*driver*' que se desarrolla dentro del sistema de archivos de MaRTE es un '*driver*' que sólo realiza una copia de los datos que utiliza la aplicación en una cola interna, que se va rellenando y vaciando según las peticiones que se hagan por parte de la aplicación, con un comportamiento similar al que puede presentar una cola FIFO (***F**irst **I**n, **F**irst **O**ut*), de la cual se extraen siempre los primeros elementos introducidos . Para su identificación dentro del conjunto de nuevos paquetes de '*drivers*' se le asignó el nombre '`Dynamic_Buffer_Driver`'.

En principio se podía realizar el '*driver*' con una cola de un tamaño fijo máximo en la cual se pudiera guardar toda la información necesaria, pero para darle un ligero mayor grado de complejidad y hacerlo más extensible y útil, se estableció que se diese el valor del tamaño de la cola circular desde la aplicación, para así permitir que cada aplicación ajustase al máximo sus necesidades de tamaño, y no tener que obligar a la recompilación del código del '*driver*' para hacer un posible cambio de tamaño.

Esta posibilidad de darle un tamaño desde la aplicación se hace factible con el uso de las opciones que ofrece la función '`ioctl`', a través de la cual se pasa el valor de tamaño deseado desde la aplicación al '*driver*' en sí mismo, para que éste ya pueda darle formato real a la cola, que en principio está declarada sin ningún rango específico.

Además, al '*driver*' se le confiere la característica de que el uso de la cola sea circular, utilizando el espacio de almacenamiento de modo que no se restrinja a una sola pasada de la misma. Para ello se va '*machacando*' la información ya leída, a medida que se va esta es extraída de la cola, con el control que se lleva a cabo por parte de una serie de contadores. Esto hace que pueda haber una continuidad en la escritura siempre que el ritmo de lectura sea tal que se vaya vaciando espacio, para nunca llegar a la cola llena, puesto que un intento de escritura sobre ella llena nos supondría hacer saltar un error de ejecución, no gestionado en un principio en la implementación ofrecida del '*driver*'.

Otra alternativa que se podría dar en el diseño del '*driver*' sería suspender a la tarea que intentase escribir sobre la cola llena hasta que volviera a quedar espacio vacío en la cola, utilizándose para ello funciones bloqueantes. Esta alternativa será ofrecida y comentada más adelante, en este mismo capítulo, en el apartado 4.4.

Para ver de qué manera funciona este '*driver*' diseñado para el almacenamiento en una cola circular, se ofrece a continuación un breve esquema del funcionamiento de las principales funciones de entrada/salida que maneja.

Función '***Read***'

- Se calculan los valores de unos contadores internos (de principio y final) a partir del valor global del contador general de lectura.
- Si el contador de final de lectura excede el valor del contador global de escritura se hace la lectura hasta este punto y se devuelve como valor de bytes leídos la diferencia entre este contador de escritura y el inicio de la lectura, sin llegar a ser todos los solicitados por el parámetro; en el caso contrario es porque se pueden leer todos los bytes pedidos.
- Se hace la lectura de los bytes del dispositivo de tal modo que la porción de la cola interna del '*driver*', partiendo desde el punto que indica el contador de inicio de lectura y con el tamaño correspondiente al número de bytes que se pueden leer, es copiada como contenido del puntero que se pasa como parámetro para que se almacene la información que ofrece el '*driver*'.

Función '***Write***'

- De manera muy similar a la lectura, se calculan los valores de unos contadores internos (de principio y final) a partir del valor global de un contador general, en este caso el que controla la escritura.
- Si el contador de final de escritura excede el valor del contador global de lectura se hace la escritura únicamente hasta este punto y se devuelve la diferencia entre el contador de lectura y el inicio de escritura, sin llegar a ser todos los solicitados por el parámetro; en el caso contrario es porque se pueden escribir todos los bytes pedidos.
- Se hace la escritura de los bytes sobre el dispositivo, de tal modo que la información que ofrece el puntero pasado como parámetro es copiado en la porción de la cola interna del '*driver*' correspondiente, es decir partiendo desde el punto que indica el contador de inicio de escritura y con el tamaño correspondiente al número de bytes que se pueden escribir.

Función '***Ioctl***'

- Dentro de las opciones de la función, la principal de ellas es la que establece el tamaño de la cola, y que es una opción que debe ser siempre realizada al inicio de cualquier aplicación, puesto que de otra manera la cola no estaría definida perfectamente para su uso.
- Además de opción de establecer el tamaño de la cola, que es completamente necesaria, se ofrecen una serie de opciones que sirven para borrar los valores de los distintos contadores de la cola, para que se pueda limpiar la información de la cola desde la aplicación.

4.1.2. Aplicación para un primer uso de funciones de entrada/salida, utilizando el 'driver' con la cola circular de tamaño configurable

Para realizar una aplicación que nos introduzca por primera vez cómo sería el uso de las funciones de entrada/salida, se debe destacar como primordial el hecho de que se pueden tener aplicaciones para cada uno de las dos interfaces posibles que nos ofrece el MaRTE OS, es decir, tanto para Ada como para C. Por eso, en esta primera aplicación desarrollada para comprobar el uso correcto de las funciones, nos detendremos en ambas interfaces, para ver cómo se usarían, pero posteriormente nos centraremos únicamente en el uso del POSIX-Ada, que es al que mayor importancia debemos darle, porque va a ser mucho mayor su utilización en desarrollos posteriores a este proyecto.

Para la aplicación desarrollada en Ada, se debe destacar el hecho de que la interfaz POSIX nos impone el hecho de tener que usar las funciones '`Read`' y '`Write`' con el tipo de datos `Ada.Streams`, de manera que se tiene que tratar de explicar su uso.

Para su utilización, hay que hacer notar que en las llamadas a las funciones de entrada/salida no se puede establecer el número de bytes que se desean utilizar de forma explícita, sino que su valor está intrínseco asociado a la definición que se haga en la aplicación del tipo `Ada.Streams`. De este modo, en esta aplicación concreta aquí desarrollada, para la escritura se establece el número de bytes deseados a través del uso de un '`String`', que se genera mediante una función de conversión del `Ada.Streams.Stream_Element_Array` al tipo '`String`'. Para la lectura, directamente se toma el número de bytes con los que se define la variable `Ada.Streams.Stream_Element_Array` creada al inicio de la aplicación, sin hacerse ninguna conversión.

También hay que hacer notar el hecho de tener que realizar la instanciación de la función '`Ioctl`' genérica, de la cual ya se habló en el capítulo 2 (sección 2.4.5), y aquí ya se puede entender su comportamiento y de qué manera se tiene que actuar con ella (instanciando la función '`Ioctl`' definida en la especificación del '*driver*' del dispositivo se define una nueva función y posteriormente ésta sera la función que realmente se use en la aplicación para hacer la llamada a la función entrada/salida '`Ioctl`' del dispositivo que se utiliza en concreto)

A través de un esquema, describamos cómo va funcionando la aplicación como tal, de una manera lineal, desde las definiciones hasta el final del programa:

- Se definen las funciones y procedimientos internos que van a ser usado en la aplicación, esto es, '`Put_SEA`' y '`String_To_SEA`'; y además también las variables internas.
- Se instancia la función '`Ioctl`' del dispositivo utilizado, que es este caso es el que se encuentra en el '*driver*' de la cola circular con tamaño configurable desde la aplicación. Además también se define la variable de datos que ha de ser usada por la función '`Ioctl`' del dispositivo, que es del tipo definido en la especificación del '*driver*'.

- Se abre el fichero de dispositivo, a través de la función de entrada/salida '`Open`', con modo de lectura/escritura, creándose con ello el descriptor de fichero asociado.
- Se establece el tamaño con el que se quiere crear la cola, dándole valor a la variable de datos que se le va a pasar al '`Ioctl`' instanciado.
- Se llama a la función '`Ioctl`' instanciada, que ha sido definida previamente, con la opción de '`Set_Buffer_Length`' (para establecer la longitud de la cola) y se le pasa el valor previamente establecido a través de la variable de datos del '`Ioctl`' .
- Se escribe en la variable de tipo '`String`' la información que se desea que sea guardada en el dispositivo posteriormente a través de la función '`Write`', función de entrada/salida.
- Se hace la llamada a la función '`Write`', utilizandose la función interna de conversión '`String_To_SEA`' para que en el parámetro de tipo `Ada.Streams.Stream_Element_Array` sea pasada la información que contiene el '`String`' anteriormente establecido.
- Se lee la información que se tiene en el dispositivo a través de la función de entrada/salida '`Read`', guardándose los datos en una variable de tipo `Ada.Streams.Stream_Element_Array` .
- Se muestra en la consola la información leída del dispositivo que se encuentra en la variable `Ada.Streams.Stream_Element_Array`, utilizando para ello la función interna '`Put_SEA`', definida al inicio.
- Se cierra el descriptor de fichero creado, usando la función de entrada/salida '`Close`'.

Veamos un resumen del código que se ha implementado para la realización de esta aplicación para el uso de las funciones entrada/salida:

```
procedure Io_Functions_Test_Ada is

   procedure Put_SEA (SEA : in Ada.Streams.Stream_Element_Array) ...
   function  String_To_SEA (Str : in String)
     return Ada.Streams.Stream_Element_Array) ...

   The_Fd      : File_Descriptor;
   Written_Str : String (1 ..10);
   SEA_Read    : Ada.Streams.Stream_Element_Array (1 .. 15);;
   Position    : Ada.Streams.Stream_Element_Offset;


   procedure Dyn_Buffer_Port_Ioctl is new
     POSIX_IO.Generic_Ioctl (Dynamic_Buffer_Driver.Ioctl_Options,
                             Dynamic_Buffer_Driver.Ioctl_Data);

   Ioctl_Data_Used : Dynamic_Buffer_Driver.Ioctl_Data;


begin

   The_Fd := Open ("dynamic_buffer_driver", READ_WRITE);

   Ioctl_Data_Used := 10;
   Dyn_Buffer_Port_Ioctl
     (The_Fd, Dynamic_Buffer_Driver.Set_Buffer_Length, Ioctl_Data_Used);
```

```
   Written_Str := "0123456789";
   Write (The_Fd, String_To_SEA (Written_Str), Position);

   Read (The_Fd, SEA_Read, Position);
   Put_SEA (SEA_Read);

   Close (The_Fd);

end Io_Functions_Test_Ada;
```

Pasando ya al otro tipo de aplicaciones que se pueden generar en MaRTE OS, es decir, la aplicación realizada en C, para almacenar la información que se ofrece por parte del dispositivo dentro de la aplicación se utilizan punteros a variable '`char`', cuyo modo de almacenamiento se aproxima bastante al que se utiliza para guardar la información ofrecida por la cola circular del '*driver*'.

Además de este detalle de cómo se almacena la información, también se puede comentar que las funciones C de entrada/salida '`read`' y '`write`' ofrecen la posibilidad de poder establecer los valores de entrada para el número de bytes de una forma explícita en la propia llamada a la función, es decir, la circunstancia más normal, no como se daba en el caso del POSIX-Ada, en el que tenía que hacer de una manera más implícita, a través de los tamaños de las variables definidas para el almacenamiento de la información, esto es, las variables del tipo `Ada.Streams.Stream_Element_Array`, en las cuales se impone que las lecturas/escrituras se realicen para el tamaño completo de la variable.

Por último, comentar que en este caso de la aplicación escrita en C también se utiliza una cola con tamaño configurable desde la aplicación, a la que se le debe pasar obligatoriamente el valor del tamaño que se desea, pero en este caso no se tiene la necesidad de instanciar la función '`ioctl`' ya que para la interfaz C se tiene una única definición de esta función de entrada/salida, a la cual sólo es necesario pasar cualquier tipo de puntero, en este caso se hace con un puntero de una variable `int`. A cambio, con la interfaz Ada se mantiene un mayor grado de seguridad gracias a la tipificación estricta que se necesita seguir en la especificación del '*driver*'.

Para explicar el modo de funcionamiento, comentar que su diseño es prácticamente idéntico al visto previamente para la aplicación Ada, excepto el hecho comentado de no necesitar hacer una instanciación de la función '`ioctl`'. Veámoslo a través de un esquema:

- Definición de las variables internas necesarias.
- Apertura del fichero de dispositivo, con la creación del descriptor de fichero asociado
- Establecer el valor del tamaño con el que se quiere crear la cola del '*driver*'
- Crear esta cola del '*driver*', a través del '`ioctl`', con el tamaño previamente asignado
- Establecer cuál es la información que se va a escribir posteriormente en el dispositivo
- Escribir esta información dentro de la cola del dispositivo, a través del '`write`'
- Leer la información que está disponible en el dispositivo, a través de '`read`'
- Mostrar en pantalla cual es la información leída del dispositivo
- Cerrar el descriptor de fichero abierto

Recojamos el código C completo de la aplicación simple desarrollada para la comenzar la iniciación al uso de funciones de entrada/salida :

```
#include <fcntl.h>
#include <unistd.h>
#include <drivers/dynamic_buffer_driver.h>

int main()
{
  int file_desc, i, buffer_length;
  ssize_t bytes_read, bytes_written;
  char *read_buffer = "\0";
  char *written_buffer = "\0";


  if ((file_desc = open ("dynamic_buffer_driver", O_RDWR)) == -1)
   {perror ("error in 'open'");
    exit (1);
   }

  buffer_length = 28;
  ioctl (file_desc, SETLENGTH, &buffer_length);


  written_buffer = "0123456789";

  if ((bytes_written = write (file_desc, written_buffer, 10)) == -1)
   {perror ("error in 'write' #2");
    exit (1);
   }


  if ((bytes_read = read (file_desc, read_buffer, 8)) == -1)
   {perror ("error in 'read'");
    exit (1);
   }

  printf ("%s", read_buffer);

  if (close (file_desc) == -1)
   {perror ("error in 'close'");
    exit (1);
   }

  return 0;
}
```

4. 2. - Puerto paralelo.

4.2.1. 'Driver' para el puerto paralelo.

Este '*driver*' que va a realizar un control sobre el puerto paralelo, es un sencillo '*driver*' que únicamente tiene por finalidad comprobar que todos los datos utilizados por los registros asociados con el dispositivo son tomados de manera correcta por una aplicación remota. En ningún momento se pretende diseñar un '*driver*' extenso que contemple todas las amplias posibilidades que ofrece el puerto paralelo, tarea que queda como susceptible de ser realizada en un posterior trabajo.

Como característica propia de este '*driver*', que será muy útil en muchos otros '*drivers*' y que se comentó en el ejemplo inicial de '*driver*' desarrollado en Ada que se vio al final del apartado 3.2. , es que se diseña el conjunto de paquetes del '*driver*' con un paquete de especificación de nivel superior '`Parallel_Port_Driver`', que es el que ofrece a los usuarios del '*driver*' para conocer todas las opciones que se pueden utilizar, principalmente de la función '`ioctl`', y luego un paquete más interno donde se encuentra la especificación de las funciones de entrada/salida, '`Parallel_Port_Driver.Functions`', del que no se necesita tener ningún conocimiento por parte del usuario del '*driver*'.

En la especificación externa se recogen toda una serie de tipos de datos y constantes que pueden ser utilizados en el manejo del '*driver*' como puede ser el tipo '`Byte`' que representa el único tipo de dato que puede manejar el puerto paralelo; la constante '`PP_Interrupt`' que sirve para fijar el valor de la interrupción que maneja el puerto paralelo; unos tipos de datos enumerados para las diferentes opciones, como son '`Logic_Status`' (*estado lógico*), '`Control_Lines`' (*líneas de control*) y '`Status_Lines`' (*líneas de estado*); y por último, se ofrecen los tipos que ha de manejar el '`Ioctl`': en primer lugar las opciones a través del '`Ioctl_Options`' y después los datos que se le pueden pasar al '`Ioctl`', utilizándose para ello un registro del tipo '`Ioctl_Data`'

Para tener un ejemplo práctico y concreto del uso de este diseño comentado anteriormente que utiliza un paquete de especificación externo, recojamos el código de la especificación real ofrecida por el '*driver*' del puerto paralelo :

```
package Parallel_Port_Driver is

   subtype Byte is Basic_Integer_Types.Unsigned_8;

   --  Parallel Port Interrupt
   PP_Interrupt : constant Hardware_Interrupts.Hardware_Interrupt :=
     Hardware_Interrupts.PARALLEL1_INTERRUPT;

   --  Logic Status
   type Logic_Status is (Low, High);
   for  Logic_Status'Size use Basic_Integer_Types.Int'Size;

   --  Control lines
   type Control_Lines is (Strobe, Auto_Feed, Init_Printer, Select_Input);
   for  Control_Lines'Size use Basic_Integer_Types.Int'Size;

   --  Status lines
   type Status_Lines is (Busy, Acknowledge, Paper_End, Device_Select, Error);
   for  Status_Lines'Size use Basic_Integer_Types.Int'Size;


   --  Parallel Port 'Ioctl' Options
   type Ioctl_Options is
     (Set_Control_Line,      -- Ioctl_Options => 0
      Read_Status_Line,      -- Ioctl_Options => 1
      Enable_Interrupts,     -- Ioctl_Options => 2
      Disable_Interrupts);   -- Ioctl_Options => 3

   --  Parallel Port 'Ioctl' Data
   type Ioctl_Data is record
      Control : Control_Lines;
      Status  : Status_Lines;
      Level   : Logic_Status;
   end record;

end Parallel_Port_Driver;
```

Con la especificación ofrecida al usuario del '*driver*'del puerto paralelo ya vista, veamos ya cómo es el comportamiento interno del mismo, a través de su implementación. Al principio de ella se definen una serie de constantes que facilitan el manejo de datos internos, a través de las cuales se obtiene la información de los registros de memoria sobre los que se debe escribir para controlar el funcionamiento ('`PP_BASE_REG`', '`PP_DATA_REG`' '`PP_STATUS_REG`', '`PP_CONTROL_REG`') y también las máscaras con las que se identifican los posibles estados que pueden tomar las distintas líneas en estos registros de memoria ('`Control_Lines_Masks`', '`Status_Lines_Masks`'). Además también se define una variable denominada '`Last_Control_Byte`', que es la que se va usar para ir recordando cuál es el estado del registro de la línea de control cada vez que este se va actualizando con nuevos valores que son establecidos a lo largo de todas las posibles opciones.

Con respecto a las funciones de entrada/salida implementadas ya propiamente en el '*driver*', comentar en primer lugar que en la función de '*inicialización*' que es como se podría llamar a la función '`Create`' lo que se hace es deshabilitar las interrupciones, a través de la escritura en el registro de control de la máscara correspondiente y así evitar que al cargar el '*driver*' del dispositivo por primera vez se tengan las interrupciones del dispositivo habilitadas. Luego, con la función '`Open`', ya se habilitan las interrupciones para que al crearse un nuevo archivo de dispositivo, éste siempre sea creado con las interrupciones habilitadas, y ya posteriormente a través del '`Ioctl`' se puedan manejar las interrupciones como se desee desde la aplicación.

La lectura/escritura del '*driver*' es sumamente simple, puesto que sólo se necesita tomar la información de un único byte (definido en la especificación con el tipo '`Byte`'), lo cual supone no tener que llevar ninguna secuencia de control, ni cuenta alguna dentro de la implementación de estas funciones de entrada/salida del dispositivo. De este modo, si se hace una petición de escritura de una información que contenga varios bytes, el dispositivo únicamente comprenderá el significado de uno único, de manera que en este caso, el control para realizar la escritura correcta del conjunto de bytes debe llevarlo la aplicación, como se podrá ver en la siguiente sección, 4.2.2, en la que se estudiarán las aplicaciones desarrolladas para estudiar el comportamiento del '*driver*' del puerto paralelo.

De las funciones de entrada/salida que utiliza el '*driver*', la más destacada es, sin duda, '`Ioctl`' ya que es la que va a permitir realizar todas las operaciones especiales que necesita este dispositivo, como pueden ser '`Set_Control_Line`', '`Read_Status_Line`' o también '`Enable_Interrupts`' y '`Disable_Interrupts`' [19] . Estas operaciones se utilizan para que el dispositivo pueda tener un control sobre las distintas líneas sobre las que puede trabajar y los modos de operación que puede admitir. Para hacer que la '*línea de control*' tome los distintos estados que puede tener (`Strobe, Auto_Feed, Init_Printer, Select_Input`) se tiene la opción '`Set_Control_Line`'; para saber en que situación se encuentra la '*línea de estado*' (`Busy, Acknowledge, Paper_End, Device_Select, Error`) se utiliza la opción '`Read_Status_Line`'; y por último, para el manejo de las interrupciones del dispositivo se tiene las dos opciones: la que se usa para habilitar (`Enable_Interrupts`) y la que se utiliza para deshabilitar (`Disable_Interrupts`).

4.2.2. Aplicaciones para el uso del puerto paralelo.

Antes de realizar el diseño de las aplicaciones para comprobar el correcto funcionamiento del '*driver*' del puerto paralelo, se tuvo que preparar un circuito especial habilitado para tomar los datos que ofrece el puerto paralelo, con una serie de pilotos que se iluminan ofreciéndonos visualmente la información que se tiene en cada momento en los registros del puerto paralelo de la plataforma de ejecución ('*target*'). El diseño de este sistema de visualización de los datos del puerto paralelo es el que se tiene en la figura:

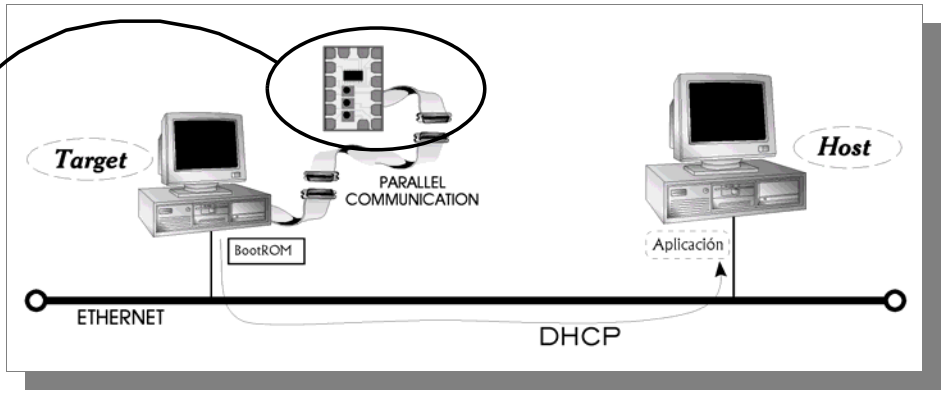

Figura 11 . Conexión entre la plataforma de ejecución ('*target*') y las tarjetas para comunicaciones a través del puerto paralelo.

A través de su esquema eléctrico y sus conexiones se puede observar mejor cómo funciona esta tarjeta de comunicaciones diseñada expresamente para funcionar en el puerto paralelo.

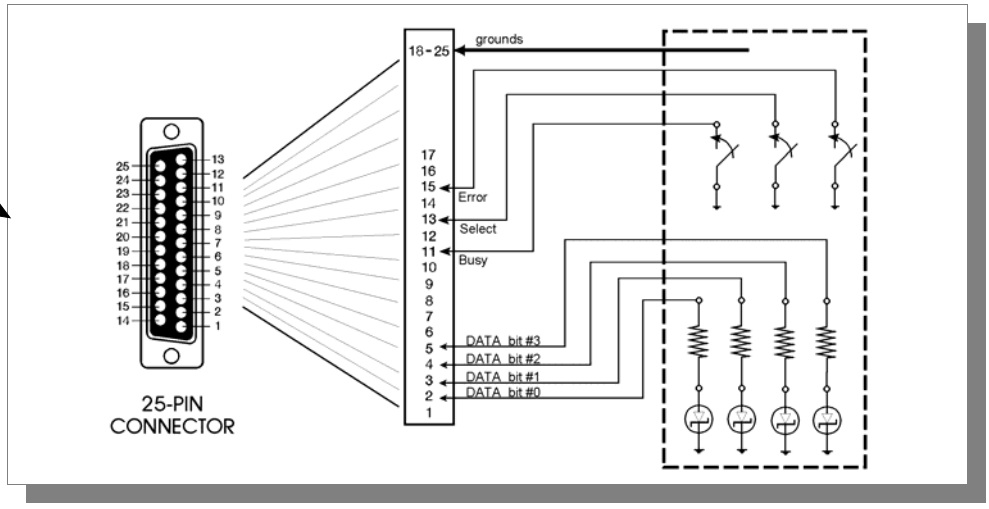

Figura 12 . Tarjeta de comunicaciones usada para comprobar los datos del puerto paralelo.

Además de poder visualizarse los datos de información a través de una serie de LED's, la tarjeta dispone de unos interruptores con los que se pueden generar cambios de estado en las distintas líneas especiales que tiene habilitadas el puerto paralelo, realizando estos cambios de manera manual. Mediante la obtención de los estados de estas líneas a través de las órdenes que maneja el '*driver*', se pueden comprobar los cambios de forma visual en la consola de la plataforma de ejecución ('*target*').

Hablando ya de las aplicaciones desarrolladas para la toma de datos del puerto paralelo, y en concreto comenzando por la que se desarrolló para la interfaz Ada, una de las características primordiales que se tiene en su diseño es que se utilizan todas las funciones genéricas que nos ofrece la interfaz POSIX-Ada. Así, se utiliza tanto el '`Ioctl`' genérico, además del uso de las funciones de entrada/salida '`Read`' y '`Write`' genéricas, instanciadas para el tipo de dato '`Byte`' ofrecido en la especificación vista anteriormente. Con todo ello, se evita el uso engorroso que suponen los '`Ada.Streams`' y de manera bien sencilla se realiza la toma de datos del puerto.

Para intentar comprender mejor el uso de esta aplicación, veamos un esquema en el cual se van situando cada una de las acciones que se van llevando en el desarrollo de la aplicación:

- Se definen las funciones genéricas necesarias, es decir '`Parallel_Port_Read`' y '`Parallel_Port_Write`', instanciadas utilizando el tipo 'Byte' de la especificación y después '`Parallel_Port_Ioctl`', que se instancia usando los tipos ofrecidos para el '`Ioctl`' en la especificación del '*driver*'. Además también se definen unas variables internas ('`PP_Ioctl_Data_Used`', '`Read_Byte`', '`The_Fd`') necesarias en la aplicación.
- Se crea un descriptor de fichero nuevo utilizándose el archivo de dispositivo de nombre "`parallel_port_driver`", asociado al puerto paralelo.
- Se genera un bucle en el cual, sucesivamente, se va escribiendo un número desde 0 hasta 7 en el dispositivo, después se va leyendo y se saca el valor leído por pantalla. Además, en la tarjeta de comunicaciones comentada anteriormente se puede ir observando cómo van iluminándose los LED's, asociados con la información que se va variando en el registro de datos del puerto paralelo.
- Se hace otro bucle en el cual se va a estudiar el comportamiento de las líneas de control del puerto. Se hace un bucle interno completo con el que se ponen todas las líneas a nivel bajo mediante la función '`Ioctl`' con la opción '`Set_Control_Line`', habiendo establecido previamente los valores apropiados de línea y de estado en el registro de datos que se le debe pasar al '`Ioctl`'. Después se va poniendo a nivel alto (también mediante la función '`Ioctl`') la línea que corresponda según el orden que se establezca en el bucle externo, y se presenta en pantalla la información de que esa determinada línea está a valor alto, de manera que sucesivamente va apareciendo que cada una de las líneas ha sido puesta a valor alto.

- A través de un bucle, en este caso un bucle que sólo ofrece continuidad, se comprueban los cambios de las líneas de estado que se generan a través de los interruptores de la tarjeta de comunicaciones. Se leen los estados que presentan las distintas líneas de estado, a través de la función '`Ioctl`' con la opción '`Read_Status_Line`' y se muestra en pantalla la información, la cual va variando según cuál sea el interruptor que se pulse, poniendose la línea correspondiente al interruptor pulsado a nivel alto.
- Se cierra el descriptor de fichero que se creó al inicio de la aplicación.

A continuación se ofrece el código completo de esta aplicación Ada comentada anteriormente, en la cual se comprueban los datos y las líneas de estado que se manejan en el puerto paralelo.

```
procedure Parallel_Port_Read_Write_Ioctl_Ada is

   procedure Parallel_Port_Read is new
     POSIX_IO.Generic_Read (Parallel_Port_Driver.Byte);

   procedure Parallel_Port_Write is new
     POSIX_IO.Generic_Write (Parallel_Port_Driver.Byte);

   procedure Parallel_Port_Ioctl is new
     POSIX_IO.Generic_Ioctl (Parallel_Port_Driver.Ioctl_Options,
                             Parallel_Port_Driver.Ioctl_Data);

   PP_Ioctl_Data_Used : Parallel_Port_Driver.Ioctl_Data;
   Read_Byte : Parallel_Port_Driver.Byte;
   The_Fd : File_Descriptor;


begin

   The_Fd := Open ("parallel_port_driver", READ_WRITE);

   for Tmp in 0 .. 7 loop
      Parallel_Port_Write (The_Fd, Byte (Tmp));
      Parallel_Port_Read (The_Fd, Read_Byte);

      New_Line;
      Put ("BYTE READ := ");
      Put (Integer (Read_Byte));
      New_Line;
      delay 0.4;
   end loop;

   Full_Console_Management.Clear_Screen;


   for Ct_Line in Control_Lines loop
      for Ct_Line_Tmp in Control_Lines loop
         for Status in Logic_Status loop
            PP_Ioctl_Data_Used.Control := Ct_Line_Tmp;
            PP_Ioctl_Data_Used.Level := Low;
            Parallel_Port_Ioctl (The_Fd, Set_Control_Line,
                                 PP_Ioctl_Data_Used);
         end loop;
      end loop;
      PP_Ioctl_Data_Used.Control := Ct_Line;
      PP_Ioctl_Data_Used.Level   := High;
      Parallel_Port_Ioctl (The_Fd, Set_Control_Line, PP_Ioctl_Data_Used);
```

```
      case Ct_Line is
         when Strobe       => Put ("Strobe       -> High");
         when Auto_Feed    => Put ("Auto_Feed    -> High");
         when Init_Printer => Put ("Init_Printer -> High");
         when Select_Input => Put ("Select_Input -> High");
      end case;

      New_Line; New_Line;
      delay 1.0;

   end loop;


   for Tmp in 0 .. 50 loop

      for St_Line in Status_Lines loop
         PP_Ioctl_Data_Used.Status := St_Line;
         Parallel_Port_Ioctl (The_Fd, Read_Status_Line, PP_Ioctl_Data_Used);

         case PP_Ioctl_Data_Used.Status is
            when Busy          => Put ("Busy");
            when Acknowledge   => Put ("Acknowledge");
            when Paper_End     => Put ("Paper_End");
            when Device_Select => Put ("Device_Select");
            when Error         => Put ("Error");
            when others        => null;
         end case;

         case PP_Ioctl_Data_Used.Level is
            when High => Put ("  ->  High");
            when Low  => Put ("  ->  Low");
         end case;

         New_Line; New_Line;

      end loop;
      delay 0.5;
   end loop;


   New_Line;
   Close (The_Fd);

end Parallel_Port_Read_Write_Ioctl_Ada;
```

Como en todos los otros casos anteriores, y reiterándonos en el tema, todas las aplicaciones se pueden realizar en ambos lenguajes disponibles, tanto para Ada como para C, y también para este caso se realizaron ambos tipos, con la aplicación C diseñada de una manera muy similar a la vista en Ada, teniendo un perfecto comportamiento todas las funciones del '*driver*', pero no se va a recoger el código para no extenderse en exceso.

Además del estudio de los registros de datos y las líneas de estado, también se puede estudiar la generación de interrupciones por parte del puerto paralelo con otra tarjeta de prueba similar a la anteriormente utilizada, que en este caso está preparada únicamente con un interruptor conectado a la línea de interrupción, reflejándose también en la consola el momento en el que es generada la interrupción. Veamos cuál es el esquema de esta nueva tarjeta que se utiliza.

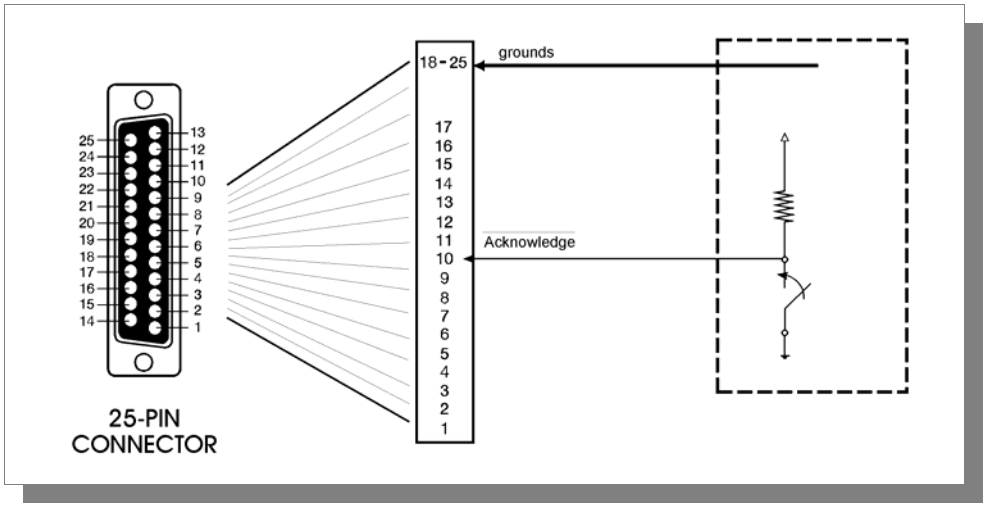

Figura 13 . Tarjeta de prueba utilizada para comprobar la generación de interrupciones del puerto paralelo.

Veamos a continuación una síntesis de las dos aplicaciones desarrolladas para el estudio del manejo de interrupciones, que presenta una pequeña dosis mayor de dificultad. Ambas aplicaciones siguen un esquema muy similar, estableciéndose en primera instancia el manejador de interrupciones definido en una subrutina definida previamente, para después habilitarse las interrupciones de las dos maneras necesarias, en el dispositivo a través del '`ioctl`', y en el sistema a través del '`Hardware_Interrupts`', dándose así la posibilidad de que el sistema reconozca la interrupción asociada. Tras ello se pone el sistema en espera hasta que se recibe la interrupción que se pretende, no reconociéndose ninguna otra interrupción.

Para comenzar a estudiar el código de estas aplicaciones diseñadas para la gestión de las interrupciones producidas por el puerto paralelo, veamos el código utilizado para la aplicación Ada del manejo de interrupciones:

```
procedure Parallel_Port_Hw_Interrupts_Ada is

   procedure Parallel_Port_Ioctl is new
     POSIX_IO.Generic_Ioctl (Parallel_Port_Driver.PP_Ioctl_Options,
                             Parallel_Port_Driver.PP_Ioctl_Data);

   PP_Ioctl_Data_Used : Parallel_Port_Driver.PP_Ioctl_Data;
   The_Fd : File_Descriptor;


   procedure Parallel_Port_Install_Handler is
   begin
      Put ("....Interrupt sent.");
   end Parallel_Port_Install_Handler;

   function Address_To_Interrupt_Handler_Procedure is new
     Ada.Unchecked_Conversion
     (System.Address, Hardware_Interrupts.Interrupt_Handler_Procedure);

   PP_Interrupt_Handler_Procedure :
     Hardware_Interrupts.Interrupt_Handler_Procedure :=
     Address_To_Interrupt_Handler_Procedure
     (Parallel_Port_Install_Handler'Address);


begin

   The_Fd := Open ("parallel_port_driver", READ_WRITE);

   Hardware_Interrupts.Install_Handler
     (PP_Interrupt, PP_Interrupt_Handler_Procedure);

   Parallel_Port_Ioctl (The_Fd, Enable_Interrupts, PP_Ioctl_Data_Used);
   Hardware_Interrupts.Enable_Hardware_Interrupt (PP_Interrupt);

   New_Line; New_Line;  Put ("Waiting for an interrupt......");
   Hardware_Interrupts.Wait (Parallel_Port_Driver.PP_Interrupt);

   New_Line; New_Line;  Put ("ENDING TASK...");

   Close (The_Fd);

exception

   when Hardware_Interrupts.RESERVED_INTERRUPT =>
      Put (" Error: Reserved Interrupt !!!");

   when Hardware_Interrupts.INTERRUPT_WAITING_ABORTED  =>
      Put (" Error: Interrupt Waiting Aborted !!!");

end Parallel_Port_Hw_Interrupts_Ada;
```

De esta aplicación Ada vista, destacar el uso que se hace de las excepciones, característica propia del lenguaje, que permite que cuando salte una excepción (cuando se genera un error), ésta pueda ser gestionada desde la aplicación, y en este caso se hace para los errores propios de MaRTE OS '`RESERVED_INTERRUPT`'('*Interrupción_Reservada*') y también '`INTERRUPT_WAITING_ABORTED`' ('*Espera_Interrupción_Abortada*').

Por otro lado, la aplicación desarrollada en C se asemeja de manera prácticamente idéntica al esquema inicial que se comentó para ambas aplicaciones, con el establecimiento del manejador, la habilitación de las interrupciones de las dor formas necesarias y luego, la puesta en espera del sistema para recibir la interrupción deseada. El código con el que se desarrolla esta aplicación del manejo de interrupciones se tiene a continuación:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <drivers/parallel_port.h>
#include <misc/console_management.h>
#include <misc/hwinterrupts.h>

int pp_install_handler ()
{
  printf("\n .....Interrupt sent.\n\n\n\n");
  return 0;
}

int main ()
{
  int file_desc;
  pp_ioctl_data_t pp_ioctl_data_used;

  if ((file_desc = open ("parallel_port_driver", O_RDWR)) == -1)
   {perror ("error in 'open'");
    exit (1);
   }



  hwinterrupts_install_handler (PP_INTERRUPT, pp_install_handler);

  hwinterrupts_enable (PP_INTERRUPT);
  ioctl (file_desc, ENABLE_INTERRUPTS, &pp_ioctl_data_used);

  printf ("\n\nWaiting for an interrupt......\n\n");
  hwinterrupts_wait (PP_INTERRUPT);

  if (close (file_desc) == -1)
   {perror ("error in 'close'");
    exit (1);
   }

  return 0;
}
```

Con estas últimas aplicaciones desarrolladas se concluye el estudio realizado del '*driver*' diseñado para el puerto paralelo, donde se controlaron el manejo de datos y la gestión de interrupciones.

4. 3. - Puerto serie.

4.3.1. 'Driver' para el puerto serie.

Aprovechando código que el desarrollo de la tesis inicial [1] había utilizado, tomado del proyecto 'OSKIT' [20], se tiene parcialmente cubierta la configuración del puerto serie, puesto que es necesario el uso de este puerto para la depuración del código ('*debug*') a través de la utilidad GDB. Pero esta información se encuentra bastante difusa, puesto que el código está muy disperso dentro de numerosos archivos de la ruta `$~\include\oskit`.

Por ello, se pretende realizar un único '*driver*' que recoja toda la información necesaria del puerto serie. Como esta información tomada de OSKIT está escrita en C completamente, este nuevo '*driver*' del puerto serie se ha realizado en este lenguaje C, primero para acortar tiempo de trabajo y además también para probar la conversión de un '*driver*' realizado en C, compilando sus funciones de entrada/salida y pasándose el código objeto al sistema de archivos de MaRTE OS, que utiliza Ada, todo ello siguiendo el mismo patrón que se vio en el apartado 3.3. del capítulo anterior.

El '*driver*' presenta una especificación que se ofrece al usuario a través de un fichero de cabeceras en el que se definen toda una serie de tipos de variables y constantes que van a permitir manejar todas las opciones que presenta el puerto serie:

```
#ifndef _MARTE_SERIAL_PORT_H_
#define _MARTE_SERIAL_PORT_H_

typedef unsigned int    tcflag_t;
typedef unsigned char   cc_t;
typedef unsigned int    speed_t;


/* Magic control character value to disable the associated feature */
#define VDISABLE    0xFF

/* Input flags (iflag) */
#define  IGNBRK   0x00000001          /* ignore BREAK condition */
#define  BRKINT   0x00000002          /* map BREAK to SIGINTR */
#define  IGNPAR   0x00000004          /* ignore (discard) parity errors */
#define  PARMRK   0x00000008          /* mark parity and framing errors */
#define  INPCK    0x00000010          /* enable checking of parity errors */
#define  ISTRIP   0x00000020          /* strip 8th bit off chars */
#define  INLCR    0x00000040          /* map NL into CR */
#define  IGNCR    0x00000080          /* ignore CR */
#define  ICRNL    0x00000100          /* map CR to NL (ala CRMOD) */
#define  IXON     0x00000200          /* enable output flow control */
#define  IXOFF    0x00000400          /* enable input flow control */
#define  IXANY    0x00000800          /* any char will restart after stop */


/* Output flags (oflag) */
#define  OPOST    0x00000001          /* enable following output processing */
#define  ONLCR    0x00000002          /* map NL to CR-NL (ala CRMOD) */
```

```
/* Control flags (cflag) */
#define CSIZE     0x00000300          /* character size mask */
#define CS5       0x00000000              /* 5 bits (pseudo) */
#define CS6       0x00000100              /* 6 bits */
#define CS7       0x00000200              /* 7 bits */
#define CS8       0x00000300              /* 8 bits */
#define CSTOPB    0x00000400          /* send 2 stop bits */
#define CREAD     0x00000800          /* enable receiver */
#define PARENB    0x00001000          /* parity enable */
#define PARODD    0x00002000          /* odd parity, else even */
#define HUPCL     0x00004000          /* hang up on last close */
#define CLOCAL    0x00008000          /* ignore modem status lines */


/* Local flags (lflag) */
#define  ECHOE    0x00000002          /* visually erase chars */
#define  ECHOK    0x00000004          /* echo NL after line kill */
#define  ECHO     0x00000008          /* enable echoing */
#define  ECHONL   0x00000010          /* echo NL even if ECHO is off */
#define  ISIG     0x00000080          /* enable signals INTR, QUIT, [D]SUSP */
#define  ICANON   0x00000100          /* canonicalize input lines */
#define  IEXTEN   0x00000400          /* enable DISCARD and LNEXT */
#define  TOSTOP   0x00400000          /* stop background jobs from output */
#define  NOFLSH   0x80000000          /* don't flush after interrupt */


/* Standard speeds */
#define B50       50
#define B75       75
#define B110      110
#define B150      150
#define B200      200
#define B300      300
#define B600      600
#define B1200     1200
#define B1800     1800
#define B2400     2400
#define B3600     3600
#define B4800     4800
#define B7200     7200
#define B9600     9600
#define B14400    14400
#define B19200    19200
#define B28800    28800
#define B38400    38400
#define B57600    57600
#define B115200   115200


/* Indexes into control characters array (cc) */
#define VEOF      0
#define VEOL      1
#define VERASE    3
#define VKILL     5
#define VINTR     8
#define VQUIT     9
#define VSUSP    10
#define VSTART   12
#define VSTOP    13
#define VMIN     16
#define VTIME    17
#define NCCS     20



/* Standard POSIX terminal I/O parameters structure. */
typedef struct {
  tcflag_t  iflag;
  tcflag_t  oflag;
  tcflag_t  cflag;
  tcflag_t  lflag;
  cc_t      cc[NCCS];
  speed_t   ispeed;
  speed_t   ospeed;
} termios_t;
```

```
/* Serial Ioctl Options */
#define  SETATTR     0 /* Set attributes to be written into the UART registers */
#define  GETATTR     1 /* Get attributes from the UART registers */
#define  SETSPEED    2 /* Set input/output speed */
#define  GETSPEED    3 /* Get input/output speed */
#define  EINTERRUPT  4 /* Enable interrupts */
#define  FLUSH       5 /* Flush the buffer */

#endif /* _MARTE_SERIAL_PORT_H_ */
```

Todo el código interno utilizado en la implementación del '*driver*' es bastante denso, porque supone tener que programar los registros de la UART del puerto serie [21], dándole todos los valores apropiados a cada uno de los estados de los bits de los registros de control y datos.

Con respecto a la implementación y al sistema de archivos de dispositivos de MaRTE OS, hay que destacar el hecho de que con este '*driver*' se introduce por primera vez el uso de los números menores, ya que se configura el '*driver*' para que al escribir sobre un registro de la UART, los números menores sean los que permitan diferenciar las direcciones de memoria base asociada a cada uno de los diferentes puertos COM [22]

(`COM1 -> 0x3F8, COM2 -> 0x2F8, COM3 -> 0x3E8, COM4 -> 0x2E8`)

Para realizar el estudio de las funciones de entrada/salida, éste se centrará sobre las tres principales funciones de entrada/salida, siendo la primera de todas la función '`read`'. En esta función se hace un bucle para ir obteniendo cada uno de los bytes que en total son pedidos en la llamada de la función y que son tomados de la información que va mandando el puerto serie. De esta manera se entra en un bucle del que se sale cada vez que llega un byte, evento que se detecta estando en modo DLAB=0, que se establece poniendo a '0' el bit #7 del registro #3, partiendo de la dirección base de memoria correspondiente al puerto, y a su vez, el bit #0 del registro #5 salta a valor '1'. En este momento se guarda internamente el byte recibido del puerto, que se encuentra en el registro #0, y ya posteriormente se van almacenando en una cola los sucesivos bytes que el puerto serie va mandando. Veamos la implementación completa de esta función '`read`' de entrada/salida:

```
unsigned int serial_port_read
(int file_descriptor, void *buffer, unsigned int bytes)
{
  int i, port, byte, bytes_read = 0;
  unsigned char mode;

  port = get_minor (file_descriptor) - 1;

  for(i=0; i<bytes; i++)
    { /* Wait for a character to arrive.  */
      for (;;)
       { mode = inb_p(ser_io_base[port] + 3);
         outb_p(ser_io_base[port] + 3, 0x7F & mode);     /* DLAB = 0 */
          if (inb_p(ser_io_base[port] + 5) & 0x01)
          {
           byte = inb_p(ser_io_base[port] + 0);
           break;
           }
       }
      ((char*)buffer)[i] = byte;
      bytes_read++;
    }
  return bytes_read;
}
```

Por otro lado, se tiene la función complementaria a la anterior función '`read`', es decir, la función '`write`' que se utiliza para mandar información al puerto serie para ser recibida en el otro equipo lejano. También aquí se entra en un bucle para poder enviar todos los bytes solicitados en la llamada a la función, y una vez dentro se comprueba el '*flag*' (identificador de opciones) `OPOST` que presenta la estructura global '`ser_termios`', con el cual los bytes correspondientes a '*nueva línea*' ('`\n`') deben ser enviados como bytes correspondiente a '*retorno de carro*' ('`\r`'). Fuera de esta opción, se envían todos los bytes de la misma manera, lo cual supone que se tiene que tener el modo DLAB=0, que se establece como se vio anteriormente con el bit #7 del registro #3 a '0', y después se tiene que esperar a que el '*buffer*' de transmisión se encuentre libre, cosa que sucede cuando el bit #2 del registro #5 se pone a valor '0', momento en el que se escribe en el registro #0 el byte correspondiente de la cola interna de almacenamiento. El código que implementa toda esta idea que se ha comentado sobre la función '`write`' es el siguiente:

```
unsigned int serial_port_write
(int file_descriptor, void *buffer, unsigned int bytes)
{
  unsigned int i, port, bytes_written = 0;
  unsigned char mode;

  port = get_minor (file_descriptor) - 1;

  for(i=0; i<bytes; i++)
    {
      if (ser_termios[port].oflag & OPOST)
       {
         if (((char *)buffer)[i] == '\n')
             /* Wait for the transmit buffer to become available.  */
           while (!(inb_p(ser_io_base[port] + 5) & 0x20));
           outb_p(ser_io_base[port] + 0, '\r');
       }
      else
       {
         mode = inb_p(ser_io_base[port] + 3);
         outb_p(ser_io_base[port] + 3, 0x7F & mode);   /* DLAB = 0 */

           /* Wait for the transmit buffer to become available.  */
         while (!(inb_p(ser_io_base[port] + 5) & 0x20));
         outb_p(ser_io_base[port] + 0, ((char *)buffer)[i]);
       }
      bytes_written++;
    }
  return bytes_written;
}
```

Por último nos queda por ver la función '`ioctl`' para completar el estudio de la implementación del '*driver*' del puerto serie. Esta función es la que ofrece toda la capacidad de control sobre el dispositivo, ya que a través de toda una serie de opciones se permite abordar todas las variantes que ofrece la programación del puerto serie. Su implementación es bastante densa puesto que supone tener que programar cantidad de registros de la UART, así que comentaremos las opciones que ofrece la función. Posteriormente se recoge el código completo de la función, para que quede constancia de cómo es la implementación real del mismo.

Las opciones principales de la función '`ioctl`' son establecer ('`SETATTR`') y obtener ('`GETATTR`') los atributos de configuración del puerto; establecer ('`SETSPEED`') y leer ('`GETSPEED`') de forma directa la velocidad de comunicación del puerto; habilitar las interrrupciones ('`EINTERRUPT`'); y una opción para vaciar el '*buffer*' de datos ('`FLUSH`').

```
int serial_port_ioctl (int file_descriptor, int request, void* argp)
{
  int port;
  unsigned char mode;
  unsigned freq_divisor;
  termios_t *com_params = (termios_t *) argp;

  port = get_minor (file_descriptor) - 1;

  switch (request)
    {
    case SETATTR:
      /* Determine what to plug in the data format register.  */
      if (com_params->cflag & PARENB)
       {
         if (com_params->cflag & PARODD)
           mode = 0x08;
         else
           mode = 0x18;
       }
      else
       mode = 0x00;

      if (com_params->cflag & CSTOPB)
       mode |= 0x04;

      switch (com_params->cflag & 0x00000300)
       {
       case CS5: mode |= 0x00; break;
       case CS6: mode |= 0x01; break;
       case CS7: mode |= 0x02; break;
       case CS8: mode |= 0x03; break;
       }

      /* Convert the baud rate into a divisor latch value.  */
      freq_divisor = 115200 / com_params->ospeed;

      /* Initialize the serial port.  */
      outb_p(ser_io_base[port] + 3, 0x80 | mode);            /* DLAB = 1 */
      outb_p(ser_io_base[port] + 0, freq_divisor & 0x00FF); /*LSB latch divisor*/
      outb_p(ser_io_base[port] + 1, freq_divisor >> 8);     /*MSB latch divisor*/

      outb_p(ser_io_base[port] + 3, 0x7F & mode);           /* DLAB = 0 */
      outb_p(ser_io_base[port] + 1, 0x00);       /* no interrupts enabled */
      outb_p(ser_io_base[port] + 4, 0x0B);       /* OUT2, RTS, and DTR enabled */

      /* make sure the FIFO is on */
      outb_p(ser_io_base[port] + 2, 0x41); /* 4 byte trigger (0x40); on (0x01) */

      /* Clear all serial interrupts.  */
      inb_p(ser_io_base[port] + 6); /* ID 0: read RS-232 status register */
      inb_p(ser_io_base[port] + 2); /* ID 1: read interrupt identification reg */
      inb_p(ser_io_base[port] + 0); /* ID 2: read receive buffer register */
      inb_p(ser_io_base[port] + 5); /* ID 3: read serialization status reg */
      break;


    case GETATTR:
      /* Clear struct data to get the parameters */
      com_params->iflag = 0x00000000;
      com_params->oflag = 0x00000000;
      com_params->cflag = 0x00000000;
      com_params->lflag = 0x00000000;
      com_params->ispeed = 0;
      com_params->ospeed = 0;

      mode = inb_p(ser_io_base[port] + 3); /* reading the Line Control Register*/
```

```
      switch (mode & 0x03)      /* Length of the word  (WLS1 - WLS0) */
       {
       case 0x00: com_params->cflag |= CS5; break;
       case 0x01: com_params->cflag |= CS6; break;
       case 0x02: com_params->cflag |= CS7; break;
       case 0x03: com_params->cflag |= CS8; break;
       }

      if (mode & 0x08)    /* Parity bit enable  (PEN) */
       {
         com_params->cflag |= PARENB;

         if (!(mode & 0x10))    /* Odd parity (ODD) */
           com_params->cflag |= PARODD;
       }


      if (mode & 0x04)    /* Stop bits number  (STB) */
       com_params->cflag |= CSTOPB;

      /* DLAB = 1 */
      outb_p(ser_io_base[port] + 3, 0x80 | mode);
      freq_divisor = inb_p(ser_io_base[port] + 1) << 8 |  /* MSB latch divisor */
                     inb_p(ser_io_base[port] + 0);        /* LSB latch divisor */


      /* Convert the divisor latch value into a baud rate. */
      com_params->ispeed = 115200 / freq_divisor;
      com_params->ospeed = 115200 / freq_divisor;
      break;


    case SETSPEED:
      /* Convert the baud rate into a divisor latch value.  */
      freq_divisor = 115200 / com_params->ospeed;

      /* Initialize the serial port.  */
      mode = inb_p(ser_io_base[port] + 3);  /*reading the Line Control Register*/
      outb_p(ser_io_base[port] + 3, 0x80 | mode);           /* DLAB = 1 */
      outb_p(ser_io_base[port] + 0, freq_divisor & 0x00FF); /*LSB latch divisor*/
      outb_p(ser_io_base[port] + 1, freq_divisor >> 8);     /*MSB latch divisor*/
      break;


    case GETSPEED:
      mode = inb_p(ser_io_base[port] + 3);  /*reading the Line Control Register*/
      outb_p(ser_io_base[port] + 3, 0x80 | mode);    /* DLAB = 1 */
      freq_divisor = inb_p(ser_io_base[port] + 1) << 8 |  /* MSB latch divisor */
                     inb_p(ser_io_base[port] + 0);        /* LSB latch divisor */

      /* Convert the divisor latch value into a baud rate. */
      com_params->ispeed = 115200 / freq_divisor;
      com_params->ospeed = 115200 / freq_divisor;
      break;


    case EINTERRUPT:
      outb_p(ser_io_base[port] + 1, 0x01);     /* interrupt on received data */
      break;


    case FLUSH:
      /*
       * Line Status Register (LSR)  :
       * bit #6 is 'HIGH' if there is no byte in hold register or shift register
       */
      while (!(inb_p(ser_io_base[port] + 5) & 0x40));
      break;


    default:
      break;
    }
}
```

4.3.2. Aplicaciones para el uso del puerto serie.

Al igual que sucedía con el caso del puerto paralelo, también en este caso se debe preparar un circuito especial para comprobar el correcto diseño del '*driver*' del puerto serie. Este circuito sigue el siguiente diseño:

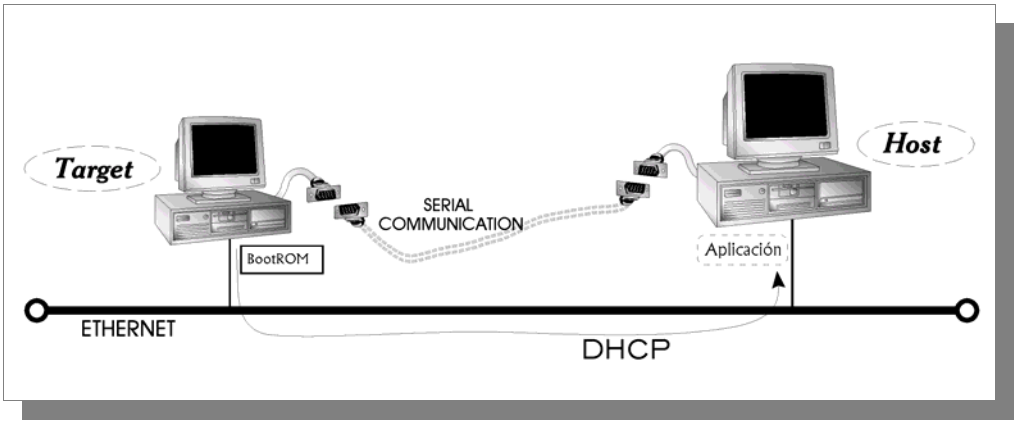

Figura 14 . Conexión entre dos PC's para establecer comunicación a través del puerto serie

Este diseño es el mismo que ya se utilizaba previamente al momento del desarrollo del proyecto para realizar la depuración del código ('*debug*') a través de la utilidad GDB, puesto que está preparado para que los dos PC's se comuniquen a través del puerto serie, pero de otra manera diferente a como lo realiza el '*driver*', con lo cual el '*driver*' debe establecer una comunicación nueva, configurada de manera distinta al GDB.

En el caso de esta aplicación, hay que establecer una sincronización entre ambos PC's para que cuando uno esté a la escucha el otro envíe la información, y viceversa, y todo ello trabajando a la misma velocidad de comunicación para que se tenga una información congruente. Esta sincronización en el host Linux se podría conseguir a través del diseño de un programa que lo haga de forma automática [23], que esté diseñado de acuerdo a la aplicación diseñada en MaRTE OS con escucha en el momento de enviar datos la aplicación MaRTE y al revés, como se ha dicho. Pero para simplificar este apartado, al final se realizó la sincronización manual, siguiendo el siguiente orden, debido al diseño de la aplicación MaRTE que veremos posteriormente:

- cambio de velocidad del puerto serie (a N bps):
  `$ stty speed N < /dev/ttyS0`
- escucha para recepción de datos del puerto serie :
  `$ cat < /dev/ttyS0`
- escritura de datos de un fichero '`file`' sobre el puerto serie :
  `$ cat file > /dev/ttyS0`

La aplicación MaRTE desarrollada, primero establece la velocidad deseada, para después escribir con '`write`' datos sobre el puerto serie (que son escuchados por el '*host*' con la orden: `$cat < /dev/ttyS0`), y después se pone a escuchar el puerto serie con '`read`' hasta que el host escribe con la orden: `$cat file > /dev/ttyS0`.

Todos los posibles cambios de velocidad de comunicación son aceptados por la aplicación perfectamente, manteniendo en todo momento la integridad de datos, hasta llegar al máximo régimen binario, que se establece a 115,400 bps. Se intentaron probar otros cambios en la configuración de manera bilateral, pero en el '*host*' que trabaja en modo Linux no se permitían excesivos cambios, ya que la orden que permite la manipulación de la configuración del puerto serie, la ya vista '`ssty`', aunque se supone que permite realizar multitud de cambios según su manual, a la hora de realizarlos sólo era posible cambiar únicamente los parámetros '`cstopb`', '`parodd`', '`crtscts`'. Estos parámetros tienen cada uno su significado propio: '`cstopb`' sirve para cambiar el número de bits de '*stop*'; '`parodd`' sirve para establecer la paridad impar; y '`crtscts`' sirve para establecer el control de flujo RTS/CTS (***R**eady **T**o **S**end / **C**lear **T**o **S**end*).

Veamos cuál es el código completo implementado para la aplicación MaRTE, desarrollada en lenguaje C con la que se estudia la sincronización del puerto serie:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <drivers/serial_port_driver.h>


int main ()
{
  int file_descriptor;
  ssize_t bytes_read, bytes_written;
  termios_t termios_used;
  char buffer[] = "0123456789\n";
  char *buffer_r = "\0";

  if ((file_descriptor = open ("serial_port_driver_com1", O_RDWR)) == -1)
   {perror ("error in 'open'");
    exit (1);
   }

  termios_used.ospeed = B1200;
  printf ("CHANGING OUTPUT SPEED... : %d\n\n\n\n\n\n\n\n", termios_used.ospeed);
  ioctl (file_descriptor, SETSPEED, ((void *)&termios_used));

  ioctl (file_descriptor, GETATTR, ((void *)&termios_used));
  printf ("INITIAL SPEED : %d bps\n\n", termios_used.ospeed);
  printf ("%d STOP bits\n\n\n\n\n\n\n\n", (termios_used.cflag & CSTOPB) + 1);

  printf("WRITE function sending info.... \n\nRemote host READY for receiving.");
  getchar ();
  if ((bytes_written = write (file_descriptor, &buffer, 12)) == -1)
   {perror ("error in 'write'");
    exit (1);
   }

  printf("Waiting for remote info to be sent for READ.....");
  if ((bytes_read = read (file_descriptor, buffer_r, 18)) == -1)
   {perror ("error in 'read' #1");
    exit (1);
   }

  printf("\n\nINFO RECEIVED : %s\n\n\n\n\n\n\n\n", buffer_r);
  getchar ();
}
```

4. 4. - Sincronización entre tareas, con uso de funciones de entrada/salida bloqueantes.

4.4.1. *'Driver' con cola circular de tamaño fijo, con funciones bloqueantes controladas con variables condicionales.*

En este '*driver*' se ha seguido en su mayor parte el diseño del '*driver*' que se comentó en la sección 4.1.1., en el que se tiene un '*driver*' con una cola circular en la que se va guardando la información con una serie de contadores, de manera que al llegar al final físico de la cola se sigue escribiendo por el principio de la cola, siempre que se haya realizado una lectura que haya liberado espacio.

Pero este nuevo '*driver*' se ha diseñado para que se gestionase el error en caso de intentar leer sobre la cola vacía o escribir sobre la cola llena. Estas dos condiciones son las que serán controladas con las variables condicionales y los '*mutexes*', es decir, los métodos de sincronización que tiene el POSIX para comunicación entre procesos.

De este modo se diseña el '*driver*' de manera que al realizar una lectura sobre una cola vacía, se quede en espera hasta que se escriba algo sobre ella. Obviamente este sistema está pensado para ser utilizado en "*aplicaciones multi-thread*" puesto que en un único '*thread*' nunca se podría salir de la condición de espera, ya que no se iba a poder escribir nada con la aplicación esperando en la variable condicional. Para que se despierte de la espera, al finalizar cada función se debe señalizar la variable condicional, de manera que sea recogida esta señalización por la función opuesta, puesto que esto supondría que se ha roto la condición para desbloquear la variable condicional que está en espera.

Veamos en un esquema de qué modo se llevan a cabo estas nuevas condiciones, pero únicamente veremos detalladamente cómo actúa la función '`Read`', puesto que la función '`Write`' va a ser prácticamente complementaria a esta.

- Se define un procedimiento interno '`Checking_Counters`' con el cual se hacen todos los cálculos de los valores de los contadores internos sobre los que se establecen los límites para la lectura de la cola. De esta manera se calcula la porción de la cola sobre la que se pide un número de bytes de lectura, como parámetro de entrada y partiendo desde el último punto que se ha leído se hace el cálculo, teniendo en cuenta que se trata de una cola circular y al llegar al final real de la cola los datos pueden seguir siendo tomados desde el principio, siempre que no se de la vuelta completa a la cola y se llegue otra vez hasta el punto inicial de lectura.
- Se comienza el desarrollo propiamente dicho del procedimiento, comprobandose si la cola no se encuentra vacía, lo que supone que habrá datos para leer y entonces tendrá sentido hacer la llamada al procedimiento interno '`Checking_Counters`' para la comprobación de los contadores

- En el caso de encontrarse la cola vacía se entra en un bucle de espera, controlado por una variable condicional y su '*mutex*' asociado, del cual no se sale hasta que no se señaliza esta variable condicional, hecho que se producirá a la salida de la función '`Write`', ejecutada por otro '*thread*', puesto que el hecho de salir de la función de escritura implica que ya se tenga algún dato para leer, de manera que se señaliza la variable condicional desde esta función. De esa manera se rompe la condición de espera, a la cual le sigue una comprobación de los contadores para poder hacer la lectura correcta con los nuevos valores actualizados.
- En este punto es donde se realiza la lectura de la información, como tal, del dispositivo, bien sea llegando de manera directa o después de una espera por tener la cola que haya sido rota por una señalización desde la función '`Write`'. De este modo, se toman los datos correctos de la cola del '*driver*', correspondientes a la parte de la cola marcada por los valores de los contadores. Previamente han sido comprobados y establecidos mediante el procedimiento interno, y se guardan como contenido del puntero que es pasado como parámetro de entrada para almacenar esta información.
- Se establece el valor '`False`' a la variable global '`Buffer_Full`', que servirá para el control de la cola dentro de la función de escritura, puesto que al haber leído algún dato ya no se cumple el hecho de que la cola esté llena, y por esta misma razón, se manda una señalización a la variable condicional para sacar de la condición de espera a posibles '*threads*' esperando en operaciones de escritura que se encuentren bloqueados por tener la cola llena.

```
----------
-- Read --
----------
function Cond_Variable_Buffer_Read
  (Fd         : in File_Descriptor;
   Buffer_Ptr : in Buffer_Ac;
   Bytes      : in Buffer_Length)
  return Int is

   procedure Checking_Counters;
   Bytes_Read : Buffer_Length;
   Begin_Read, End_Read : Integer;

   procedure Checking_Counters is
   begin
      Begin_Read := Integer (Read_Counter);
      End_Read   := Integer (Read_Counter) + Integer (Bytes) - 1;

      if Begin_Read < Integer (Write_Counter) and
          ((End_Read mod Integer (Buffer_Driver_Length)) >
            Integer (Write_Counter)) then

         End_Read := Integer (Write_Counter - 1) mod
                     Integer (Buffer_Driver_Length);
         Read_Counter := Write_Counter;
         Bytes_Read := Buffer_Length
                       ((End_Read - (Begin_Read - 1)) mod
                        Integer (Buffer_Driver_Length));
         Buffer_Empty := True;
      else
         Read_Counter :=
           Buffer_Length ((Integer (Read_Counter) + Integer (Bytes) - 1) mod
                          Integer (Buffer_Driver_Length) + 1);
         Bytes_Read := Bytes;
      end if;
   end Checking_Counters;
```

```
begin

   if not Buffer_Empty then
      Checking_Counters;
   end if;

   while Buffer_Empty loop
      New_Line;
      Put ("Waiting.......  BUFFER EMPTY");
      New_Line;
      Put ("Not ready for READING");
      New_Line;
      New_Line;

      Wait (Descriptor_Of (Conditional_Variable),
            Get_Mutex_Descriptor (Fd));
      Checking_Counters;
      Buffer_Empty := False;
   end loop;


   for I in Begin_Read .. End_Read loop
      Buffer_Ptr.all (Buffer_Length (I - (Begin_Read - 1))) :=
        Buffer_Driver
        (Buffer_Length ((I - 1) mod Integer (Buffer_Driver_Length) + 1));
   end loop;

   Buffer_Full := False;
   Signal (Descriptor_Of (Conditional_Variable));

   return Int (Bytes_Read);

end Cond_Variable_Buffer_Read;
```

El comportamiento complementario que se ha comentado que realiza la función '`Write`', se refleja en el hecho de que la variable global sobre la que se debe comprobar la condición es distinta, ya que en este caso se debe mirar el hecho de que la cola se encuentre llena ('`Buffer_Full`') en lugar de comprobarse si la cola está vacía ('`Buffer_Empty`'), como sucedía este caso visto previamente con la función '`Read`'.

Además los valores de los contadores cambian de manera complementaria, de manera que todos los que corresponden a valores asociados con la lectura ('`Begin_Read`', '`End_Read`', ..) se convierten en escritura, y valores de escritura que se han utilizado dentro de la función de lectura ('`Write_Counter`'), para esta función de escritura deben ser tomado los valores complementarios correspondientes a la lectura.

Todo el resto del algoritmo se sigue de manera similar, chequeándose y estableciéndose los valores de los contadores internos para el manejo de la cola circular; bloqueándose la tarea en caso de ser necesario; realizándose después la acción de escritura, para permitir que en el caso de haber sido bloqueada la función, la información escrita sea transmitida de forma correcta; y por último lanzándose la llamada para despertar a posibles tareas, mediante la señalización de la variable condicional, para que si otras funciones están a la espera de que se cumpla la condición, éstas puedan ser alertadas de que ya se ha realizado la escritura.

4.4.2. Aplicación para la sincronización entre 'threads'.

Como se comentó en la sección anterior, este '*driver*' controlado con variables condicionales tiene sentido usarlo para "*aplicaciones multi-thread*", puesto que mientras una aplicación esté a la espera, p.ej. por el hecho de tener la cola llena para la escritura, es necesario que otro '*thread*' ejecute una lectura y consiga vaciar parte de esa cola para que se pueda realizar la escritura deseada y no se quede bloqueado.

De esta manera, se tiene que diseñar una aplicación que cree y ejecute al menos dos '*threads*' en los que se realice en cada uno de ellos, una lectura en uno, y una escritura en otro, utilizandose la funciones de entrada/salida del dispositivo visto anteriormente con las funciones bloqueantes, de manera que se tendrá que abrir un nuevo descriptor de fichero para poder tener uso del archivo de dispositivo asociado. Este descriptor de fichero único tendrá que ser pasado como argumento a través de un puntero a cada uno de los '*threads*' para que puedan disponer de esta información y puedan usar las funciones de entrada/salida de este dispositivo.

La aplicación debe introducir una periodicidad relativa para así controlar el hecho de que se pueda llenar la cola, o bien vaciarse, o medio completar, etc.. controlándose todas estas circunstancias a través de los períodos de repetición que se establezcan. Además también se tienen que controlar los tamaños de las lecturas y escrituras, para que juntándose con los períodos temporales, se produzcan en algún momento todas las condiciones establecidas, y se pueda comprobar de manera correcta el buen funcionamiento de las condiciones bloqueantes.

El '*thread*' desarrollado para la lectura (denominado '`reader`') se diseña para que se hagan en total cinco lecturas de la cola, una cada segundo, comenzando desde el instante inicial, de modo que se extraigan cada vez que se ejecuta la función de entrada/salida un total de 14 bytes, presentándose en pantalla. Estos 14 bytes no siempre son leídos de forma completa puesto que la función de lectura diseñada en el '*driver*' antes de concluir, no espera a que se tenga el número completo pedido en los parámetros de entrada, sino que hace la lectura del número de bytes que se tengan disponibles en ese momento en la cola.

El '*thread*' diseñado para la lectura (denominado '`writer`') se desarrolla de manera que se ejecuten cuatro escrituras sobre el dispositivo de 24 bytes, cada 0.4 segundos. Este período inicial de 0.4 segundos que transcurre hasta la primera escritura hace que se pueda cumplir siempre que se llega a la condición de bloqueo de la lectura, ya que se empieza a leer desde el instante inicial, sin tener información en el dispositivo. Cuando se escribe esta información por primera vez ya se sale de la condición de bloqueo que se da por leer sobre la cola vacía, y ya se va siguiendo con continuidad todo el proceso de lecturas/escrituras, hasta llegar a la última escritura en la que se da un bloqueo por intentar escribir sobre la cola llena (esta condición se da porque en las constantes del sistema se estableció el tamaño de la cola a 32 bytes), de manera que el bloqueo llega en la cuarta escritura. De forma similar a lo visto para la función '`read`', en la escritura tampoco se garantiza que todo el número de bytes pedidos se entreguen, ya que sólo se escribe hasta el final de la cola.

A través de un diagrama de eventos se podrá comprender mejor todo el proceso que sigue la cola durante todas las lecturas/escrituras, para este tamaño de cola de 32 bytes:

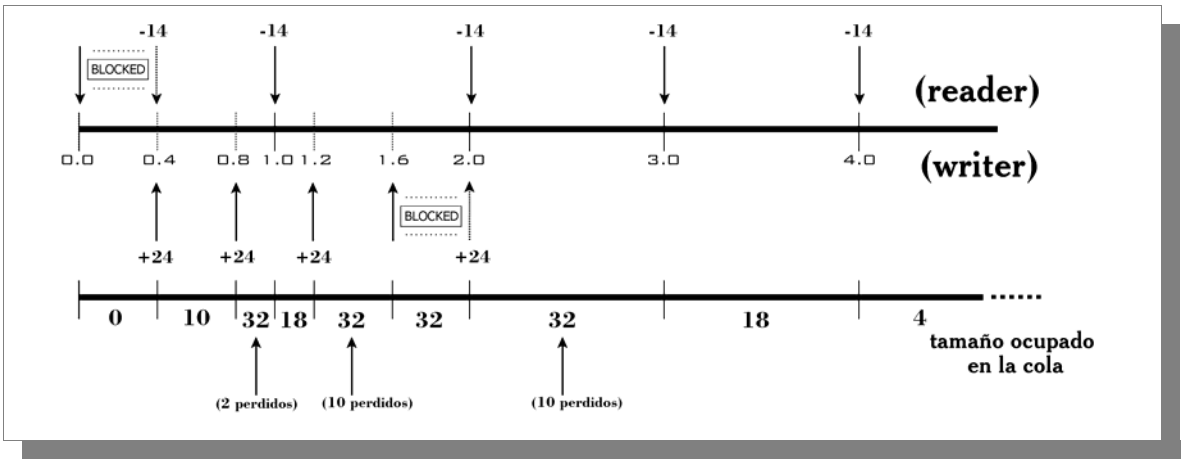

Figura 15 . Eventos para la sincronización entre los 'threads' con funciones bloqueantes

Este es el código completo de la aplicación '*multi-thread*', que se ha diseñado en lenguaje C, para comprobar este sistema bloqueante controlado por la variables condicionales:

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <pthread.h>

#define SEC 1E9

void *reader (void *arg)
{
  char *buffer = "\0";
  ssize_t bytes_read;
  int file_desc, i, j;
  struct timespec delay;

  file_desc = *((int *)arg);
  delay.tv_nsec = 1.0 * SEC;

  for (j=0; j<5; j++)
  {
    if ((bytes_read = read (file_desc, buffer, 14)) == -1)
     {perror ("error in 'read'");
      exit (1);
     }
    printf("\n\nBuffer READ (%d bytes) -> ", bytes_read);
    for (i=0; i<bytes_read; i++)
    {printf("%c", buffer[i]);
    }
    printf("\n\n");
    nanosleep (&delay, NULL);
  }
  pthread_exit ((void *)0);
}
```

```
void *writer (void *arg)
{
  char *written_buffer = "\0";
  ssize_t bytes_written;
  struct timespec delay;

  int file_desc, i;

  file_desc = *((int *)arg);
  delay.tv_nsec = 0.4 * SEC;

  for (i=0; i<4; i++)
  {
    nanosleep (&delay, NULL);
    written_buffer = "abcdefghijklmnopqrstuvwxyz";

    if ((bytes_written = write (file_desc, written_buffer, 24)) == -1)
     {perror ("error in 'write' #2");
      exit (1);
     }
  }
  pthread_exit ((void *)0);
}


int main()
{
  int file_desc;
  pthread_t read_th, write_th;


  /* Opening FILE */
  if ((file_desc = open ("cond_variable_buffer_driver", O_RDWR)) == -1)
   {perror ("error in 'open'");
    exit (1);
   }

  /* Threads Creation */
  if (pthread_create (&write_th, NULL, writer, &file_desc) != 0)
   {perror ("error in 'write' pthread create");
    exit (1);
   }
  if (pthread_create (&read_th, NULL, reader, &file_desc) != 0)
   {perror ("error in 'read' pthread create");
    exit (1);
   }


  /* Threads Join */
  if (pthread_join (write_th, NULL) != 0)
   perror ("error in 'write' JOIN");

  if (pthread_join (read_th, NULL) != 0)
   perror ("error in 'read' JOIN");


  /* Closing FILE */
  if (close (file_desc) == -1)
   {perror ("error in 'close'");
    exit (1);
   }

  return 0;
}
```

Con esta aplicación de sincronización, además de comprobar el correcto uso del '*driver*' diseñado para la sincronización entre procesos con '*mutexes*' y variables condicionales, también se ha podido observar el modo de usar la capacidad del sistema MaRTE OS de generar "*aplicaciones multi-thread*", creándose los '*threads*' a través de '`pthread_create`' y dando la capacidad de terminar a la aplicación cuando terminen los '*threads*' mediante la orden '`pthread_join`'

4. 5. - Resumen de la implementación de los '*drivers*' de dispositivos.

En este extenso capítulo se han estudiado todas las aplicaciones y '*drivers*' que se han desarrollado con posterioridad al diseño del sistema de archivos, para comprobar sus variados modos de funcionamiento, disponibles para cualquier tipo de dispositivo, tanto '*hardware*' como virtuales.

Se comienzan a desarrollar '*drivers*' de dispositivos, mediante el diseño de '*driver*' para un dispositivo virtual: una cola para almacenamiento temporal de información utilizable en tiempo de ejecución, que puede ser configurada con el tamaño que se desee desde la propia aplicación. Para este '*driver*' se implementa una aplicación en la que se hacen solamente se hace uso de las funciones de entrada/salida, en especial de lecturas/escrituras, aplicación que puede servir para comprobar en primera instancia el comportamiento inicial de las funciones de entrada/salida de cualquier dispositivo.

Después se diseñó un '*driver*' para un dispositivo físico, para el puerto paralelo, con aplicaciones que comprueban los cambios de estado de las líneas del puerto y los datos enviados por el mismo. Para probarlo se hace uso de una tarjetas especialmente diseñada para las comprobar las comunicaciones con el puerto paralelo de la plataforma de ejecución ('*target*'), y con otra tarjeta similar, preparada para generar interrupciones, se comprueba también la capacidad del uso de interrupciones por el puerto paralelo.

Continuando con los distintos dispositivos '*hardware*' que se pueden encontrar en cualquier sistema, se escribe posteriormente un '*driver*' para el puerto serie, aprovechando parte del código ya realizado siguiendo el proyecto OSKIT. Para la comprobación de este nuevo '*driver*' se utiliza en aplicaciones diseñadas para la escucha del puerto serie y la escritura sobre él, todo ello dentro de todo el rango de posibles velocidades de transmisión que ofrece el puerto para las comunicaciones.

Por último, se toma la idea original del dispositivo virtual diseñado anteriormente, y se cambia parte de su implementación, para conseguir hacer que disponga de funciones de entrada/salida bloqueantes y controladas por variables condicionales. Para poder utilizar este '*driver*' con funciones bloqueantes, se tiene que diseñar una aplicación '*multi-thread*' que consiga hacer que las funciones bloqueantes no dejen la aplicación en una espera activa, sino que la espera puede ser pasiva, dedicándose así todo el tiempo de la CPU a los restantes '*threads*'

Con todo esto se concluye toda la parte de desarrollo de proyecto realizado, quedando sólo por comentar las conclusiones finales y la línea futura de trabajo que queda abierta con este proyecto.

Capítulo 5

Conclusiones del proyecto.

5. 1. - Comentarios y conclusiones.

Se llega al final de esta memoria, y las conclusiones a realizar del trabajo realizado se centran principalmente en recalcar el cumplimiento de forma global de los objetivos fijados inicialmente en cuanto a la intención de desarrollar de manera exhaustiva un sistema de manejadores de dispositivos para el sistema operativo MaRTE OS.

La primera conclusión es que se ha conseguido integrar perfectamente todo este nuevo entorno dentro de todo el conjunto anterior que ofrecía el sistema MaRTE OS, principalmente por el hecho de hacer un diseño lo más independiente posible de las partes que ya estaban desarrolladas en el sistema, y que su unión con el sistema se ha pretendido hacer de una manera muy limpia y transparente a través de una serie mínima de cambios.

Una de las partes principales a destacar del diseño es que se realiza de tal manera que la aplicación pueda manejar las funciones de entrada/salida de los dispositivos a través de una única interfaz externa, que permite que en ningún momento se tenga que entrar dentro del núcleo del sistema para gestionar el uso de los dispositivos, ya que esta es la función que desempeña este nuevos sistema de archivos.

También comentar que una de las características básicas del sistema MaRTE, que se tiene en el hecho de ofrecer a las aplicaciones una doble interfaz POSIX para poder ser desarrolladas tanto en lenguaje Ada como en lenguaje C, es tenida muy en cuenta en todo momento y es por ello que el diseño de este sistema de archivos de dispositivos incide especialmente en el hecho de mantener en todo momento la integridad perfecta de la información que se le pasa a la aplicación, sea cual sea la interfaz que se utilice. De este modo, es posible que las aplicaciones escritas tanto en Ada como en C, hagan uso de '*drivers*'de entrada/salida que, a su vez, pueden estar escritos indistintamente en Ada o C.

Por otro lado, otra característica que se puede destacar es que el diseño del entorno aporta una única interfaz para la '*instalación de drivers*', acción que se lleva a cabo en un solo paquete, en el que se recoge toda la información que los distintos archivos de dispositivo deben manejar. De esta manera, cualquier variación que se desee realizar acerca de cualquier '*driver*' ya instalado, o bien, si se desea agregar un nuevo dispositivo al sistema mediante la creación de un nuevo archivo de dispositivo, todas estas acciones se centran sólo en un paquete único, con lo que se potencia la simplicidad a la hora de incluir nuevos '*drivers*'.

Por último, terminaremos comentado el desarrollo de nuevos '*drivers*' llevado a cabo, gracias a los cuales se ha podido constatar perfectamente la integración de este '*nivel de dispositivos*' dentro del sistema MaRTE. Cada uno de los '*drivers*' diseñado ha ido intentando aportar características especiales que ha sido recogidas en el diseño, y que gracias al uso de estos nuevos '*drivers*' se pudo comprobar y perfeccionar el funcionamiento global del entorno de archivos de dispositivos.

5. 2. - Línea futura de trabajo.

Como principal línea de trabajo que se puede establecer para el futuro se tiene el desarrollo de todos los posibles '*drivers*' que se quieran incluir dentro del sistema operativo MaRTE OS. Ya dentro de esta perspectiva se está de hecho comenzado a desarrollar '*drivers*' de red para el protocolo RT-EP, diseñado con anterioridad para el sistema Linux pero que ya se encuentra adaptado al sistema MaRTE.

Dentro de estos posibles nuevos '*drivers*' a desarrollar, se tiene la idea de realizar en breve un '*driver*' que controle la consola y el teclado, dispositivos que ahora mismo se encuentran en uso dentro del sistema, pero con sus funciones bastantes dispersas, lo cual supone que realizar cualquier intento de programar con sus funciones resulte complejo, como sucedía con el '*driver*' del puerto serie, que se comentó en el apartado 4.3.

Como retoques específicos al diseño del entorno, se podría intentar completar todas las posibilidades que ofrece la interfaz POSIX-Ada, que actualmente sólo cubre el uso de las funciones de entrada/salida principales que debe soportar el sistema. En la especificación de la interfaz POSIX completa se ofrecen otras muchas funciones secundarias, que quizá pudieran implementarse para el sistema MaRTE, pero que no son necesarias en la actualidad. Como ejemplo de estas funciones no implementadas, se tienen '*funciones de posicionamiento en archivos*', como '`Seek`', '`File_Size`' o también '`File_Position`'; y también se tienen '*funciones de control de archivos*', como son '`Get_File_Control`', '`Get_Close_On_Exec`', '`Change_Permissions`', '`Truncate_File`'..

De los '*drivers*' realizados en este proyecto, se podría ampliar alguna función adicional que pueda surgir como idea, ya que queda perfectamente abierta su ampliación a través de la generalidad que da la función '`ioctl`', de manera que principalmente para los '*drivers*' de los puertos serie y paralelo, alguna función adicional perfectamente se le puede agregar, pero su funcionamiento básico está más que cubierto con el diseño realizado en este proyecto.

Algunos de los nuevos '*drivers*' que se tiene intención de poner en marcha bajo la interfaz proporcionada con este proyecto son los siguientes:

- entrada/salida digital
- entrada/salida analógica
- codificadores de posición ('*resolvers*')
- pantalla gráfica
- captura de imágenes
- .....

- Referencias

[1] Mario Aldea, "*Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas*", Tesis doctoral, Dpto. Electrónica y Computadores, Grupo de Tiempo Real, Universidad de Cantabria, Mar.2003

[2] IEEE Standard 1003.1b:1993 "*Standard for Information Technology -Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]. Realtime Extension*" The Institute of Electrical and Electronics Engineers, 1993.

[3] IEEE Standard 1003.1c: 1995 "*Standard for Information Technology -Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]. Threads Extension*" The Institute of Electrical and Electronics Engineers, 1995

[4] IEEE Standard 1003.13:1997 "*Standard for Information Technology -Standardized Application Enviroment Profile- POSIX Realtime Application Support (AEP)*" The Institute of Electrical and Electronics Engineers, 1997.

[5] "*MaRTE OS home page, (Minimal Real-Time Operating System for Embedded Applications)*", Mario Aldea y Michael González, Dic.2001.
`http://marte.unican.es/`

[6] "*GNU General Public License - GNU Project - Free Software Foundation (FSF)*"
`http://www.gnu.org/copyleft/gpl.html`

[7] *"Real-Time Executive for Multiprocessor Systems: Reference Manual"*, U.S. Army Missile Command, Redstone Arsenal, Alabama, EE.UU., Enero 1996.

[8] "*FSMLabs, Real-Time Linux Operating System Web Page*"
`http://fsmlabs.com`

[9] J. Javier Gutiérrez, "*Programación orientada al sistema*", en 'Desarrollo de software para entorno industrial: programación concurrente y de tiempo real', Oct.1999

[10] Wind River Systems, Inc. "*VxWorks. Programmer's Guide v5.4*", Edition 1, Mar.1999

[11] Alessandro Rubini, Jonathan Corbet, "*Linux Device Drivers*", 2nd Edition, June 2001

[12] IEEE Standard 1003.5b:1996 "*IEEE Standard for Information Technology -- POSIX® Ada Language Interfaces -- Part 1: Binding for System Application Program Interface (API)*" The Institute of Electrical and Electronics Engineers, 1996.

[13] "*Download MaRTE OS 1.0*" [tar gziped, 4Mb], Dic.2001.
http://marte.unican.es/marte-1.0.tar.gz

[14] "*Download MaRTE OS 1.2*" [tar gziped, 4Mb], Feb.2003.
http://marte.unican.es/marte-1.2.tgz

[15] "*The Open Group Base Specifications Issue 6*", <fcntl.h>
http://www.opengroup.org/onlinepubs/007904975/basedefs/fcntl.h.html

[16] "*The Open Group Base Specifications Issue 6*", <unistd.h>
http://www.opengroup.org/onlinepubs/007904975/basedefs/unistd.h.html

[17] "Chapter 5, MultiNet v4.4, Messages and Logicals Guide", Error Codes
http://www.multinet.process.com/ftp/docs/html/messages/Ch05.htm

[18] "*Ada95 - Interfacing Pragmas*"
http://www.adaic.org/standards/95lrm/html/RM-B-1.html

[19] "*Use of a PC Printer Port for Control and Data Acquisition*"
http://et.nmsu.edu/~etti/fall96/computer/printer/printer.html

[20] "*The OSKit Project*"
http://www.cs.utah.edu/flux/oskit/

[21] "*Configuración del puerto serie*"
http://www.fortunecity.es/virtual/hardware/386/serie/manual/manual.htm#c2

[22] José María Drake y Elena Mediavilla. "*Técnicas de entrada/salida de tiempo real*", Apuntes de la asignatura "Instrumentación de Tiempo Real", Santander, Feb.1998.

[23] "*Cómo programar el puerto serie*"
http://www.insflug.org/COMOs/Programacion-Serie-Como/Programacion-Serie-Como-3.html