ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Proyecto Fin de Carrera

GESTIÓN DE DISPOSITIVOS DE ENTRADA/SALIDA ANALÓGICA, DIGITAL Y POR EL BUS SERIE I2C

(Management of devices for digital and analog input/output, and through the I2C serial bus)

Para acceder al Titulo de

INGENIERO DE TELECOMUNICACIÓN

Autor: Daniel Sangorrín López
Febrero - 2006

Agradecimientos

Quisiera agradecer a Michael González Harbour su confianza, paciencia y buen hacer en la tarea de dirección de este proyecto.

Título en inglés[1]

Management of devices for digital and analog input/output, and through the I2C serial bus

Palabras clave

Driver, MaRTE OS, kernel, sistemas empotrados, tiempo real, Ada 95, C, POSIX.13, bus serie I2C, arranque por red, Linux, software libre, GNU

Organización de este documento

Tras una primera introducción, en el *capítulo* 1, a los conceptos básicos y objetivos del proyecto, se mostrará, en los *capítulos* 2 y 3, una visión práctica que nos permita ponernos en situación. Veremos en qué consiste MaRTE OS en la práctica, cuáles son los ficheros más importantes para nosotros y cómo configurar el entorno de trabajo. Los *capítulos* 4 y 5 son completamente independientes uno del otro. En el primero mostraremos el desarrollo del software encargado de controlar una tarjeta multifunción con capacidad de adquisición de señales analógicas y con dos puertos digitales de entrada/salida. En el segundo, veremos el desarrollo de un subsistema software encargado de controlar el **Bus Serie I2C**, muy extendido en sistemas empotrados, y crearemos un driver para un dispositivo I2C real que use este subsistema. Finalmente, en el *capítulo* 6 se enumerarán las conclusiones e ideas para un trabajo futuro. Se ha decidido añadir también un pequeño apéndice con pequeños *trucos* que pueden ser útiles para otros desarrolladores en un futuro.

Lectores noveles en la materia podrían comenzar por el *capítulo* 2 para una primera toma de contacto, a continuación volver al *capítulo* 1 para matizar y después continuar por el *capítulo* 3. El lector experto en sistemas empotrados quizá sólo necesite leer el *capítulo* 1 y saltar directamente a los *capítulos* 4 y 5. Los lectores que ya dominen MaRTE OS pueden comenzar directamente por los *capítulos* 4 y 5 sin necesidad de leer los capítulos iniciales.

Licencia

[1] Para que figure en el Suplemento Europeo al título

Índice general

Capítulo 1

Introducción

El objetivo de este capítulo de introducción consiste en mostrar una visión global del proyecto y explicar algunos conceptos y términos básicos para la comprensión del resto de capítulos

1.1. Motivaciones del proyecto

A la hora de afrontar la realización de este proyecto fin de carrera decidí que había varios aspectos que deseaba cumplir:

- Participar en el desarrollo de un sistema empotrado.
- Trabajar en software de bajo nivel interactuando con el hardware.
- Utilizar software libre para su realización.

El campo de los sistemas empotrados tiene cada día más importancia en el mundo moderno. Vivimos prácticamente rodeados de todo tipo de aparatos electrónicos que, sin saberlo, llevan uno o varios sistemas empotrados en su interior: móviles, MP3, coches, lavadoras... La lista es casi interminable. Así, no cabe duda de que, para un amante de la tecnología como yo, el trabajar en el desarrollo de un sistema empotrado resultaba algo tentador.

Uno de los aspectos más interesantes que tienen estos sistemas es la necesidad de combinar diversas disciplinas para lograr un mismo objetivo. Una lectura a [1] nos da idea de la cantidad de ciencias que engloba. Entre ellas, [1] destaca como núcleo central el campo del tiempo real (si bien, aunque la mayoría de sistemas de tiempo real son empotrados, muchos sistemas empotrados no son de tiempo real).

Resulta, pues, coherente que este proyecto haya sido desarrollado en el *Grupo de Computadores y Tiempo Real* de la Universidad de Cantabria.

1.1.1. RoboCup, un trabajo en equipo

Este proyecto se encuadra con carácter particular dentro de un grupo de otros proyectos destinados al diseño e implementación de un equipo de robots futbolistas capaces de participar en la liga *small-sized* del torneo RoboCup [2]. Sin embargo, sus resultados son aplicables a cualquier sistema empotrado en general, por ejemplo para aplicaciones industriales o de telecomunicación.

RoboCup es una competición internacional de fútbol robótico con diversas categorías y está destinado a promover la inteligencia artificial, la robótica y otros campos relacionados. Es un intento de fomentar la investigación en estas disciplinas mediante el planteamiento de un problema estándar donde se pueda integrar y examinar un amplio rango de tecnologías. Así, RoboCup decidió elegir el fútbol como tema central, dejando entrever también su interés en que las innovaciones resultantes pudieran ser aplicables en problemas sociales importantes. Se decidió crear el proyecto RoboCupRescue, encargado específicamente de promocionar la investigación en aspectos sociales importantes como, por ejemplo, la búsqueda y rescate de personas en desastres de gran escala mediante robots. El lema con que a menudo se presenta RoboCup es el de **crear, para 2050, un equipo de robots humanoides completamente autónomo capaz de vencer al mejor equipo de seres humanos**. Para crear un equipo de robots capaz de jugar al fútbol se deben utilizar varias tecnologías como: principios de diseño de agentes autónomos, colaboración multi-agente, creación de estrategia, razonamiento en tiempo real, robótica o sensores.

Dado que se trata de un trabajo en equipo se necesita una cierta organización. En este caso, la figura de Director ha sido la encargada de realizar los análisis de requerimientos y será la que establezca las especificaciones a cada proyecto. De este modo, no precisaremos conocer el funcionamiento completo del robot sino solamente la parte necesaria para poder cumplir estas especificaciones. Dado que otros proyectos se basarán en nuestro trabajo debemos realizar una interfaz lo más clara posible y bien documentada. Esto permitirá al usuario comenzar a trabajar rápidamente en un siguiente nivel de abstracción.

Hemos de recalcar, sin embargo, que los desarrollos de este proyecto conjunto son más ambiciosos que la simple participación en RoboCup, pudiéndose utilizar en todo tipo de desarrollos, por ejemplo en sistemas de automatización industrial.

1.2. Antecedentes

La primera fase en el desarrollo del robot consistía en montarlo físicamente y hacer que se moviera (ver 1.2.1). El robot básicamente es un PC sobre el que se ejecutarán de forma concurrente distintas tareas. Unas se encargarán de controlar los motores, otras de la visión, etc... Por ello necesitamos un sistema operativo que nos abstraiga del uso de este PC y nos ofrezca servicios para crear esas tareas, sincronizarlas, gestionar la memoria, etc... Sin embargo, dado que un robot es un entorno muy agresivo y cambiante no nos sirve un sistema operativo normal. Necesitamos un sistema operativo de poco tamaño, que sea código libre a ser posible y sobre todo que sea de tiempo real. Esto quiere decir que no sólo nos importa que se ejecuten las tareas sino que tienen que ejecutarse cumpliendo unos plazos temporales que nosotros marquemos.

El sistema MaRTE OS, desarrollado en el propio grupo, cumple con todos estos requerimientos. MaRTE OS además cumple con un estándar llamado POSIX.13 [3]. El estándar POSIX [4] (conocido en el ámbito internacional con la referencia ISO/IEC-9945), tiene como objetivo permitir la portabilidad de aplicaciones a nivel de código fuente entre diferentes sistemas operativos. Para pequeños sistemas empotrados se aprobó otro estándar llamado POSIX.13, usado por MaRTE OS, que es mucho más pequeño que el estándar POSIX general y adecuado a sistemas empotrados de tiempo real.

Por otro lado, para hacer que nuestro robot interactúe con su entorno necesitaremos usar dispositivos periféricos. MaRTE OS ofrece un entorno para crear manejadores de estos dispositivos, drivers, que permiten al usuario abstraerse de su funcionamiento interno.

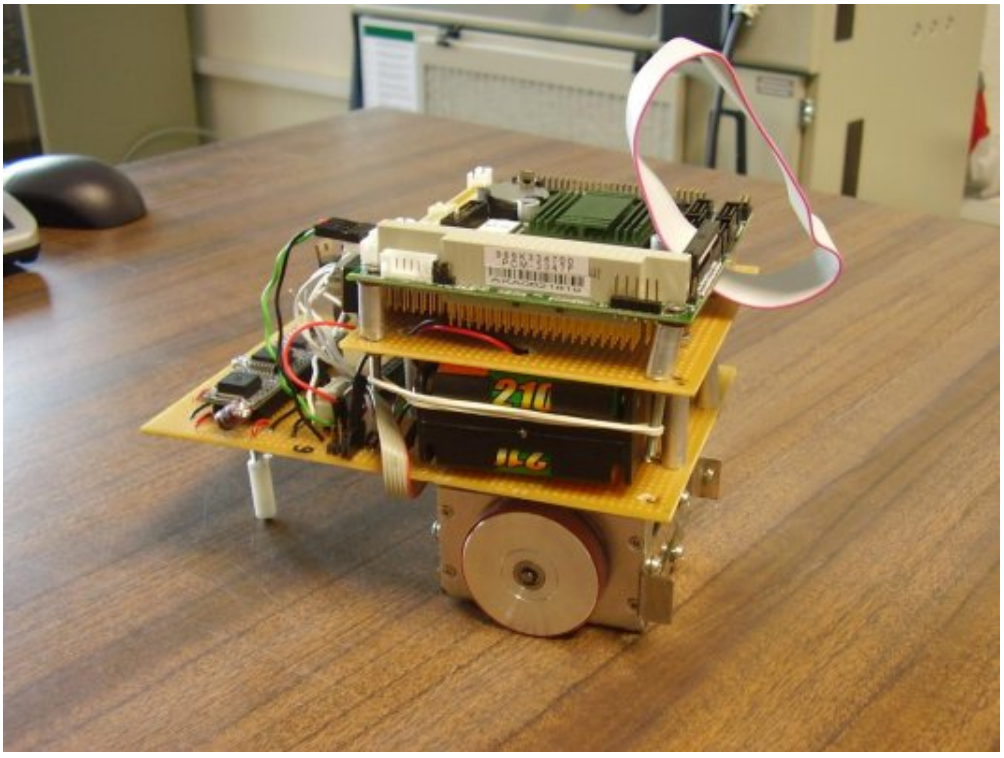

Figura 1.1: Prototipo por Bernardo Ruíz Abascal

1.2.1. Primer prototipo del Robot

En la figura 1.1 tenemos un primer prototipo del robot futbolista resultado del proyecto realizado por Bernardo Ruíz Abascal [5]. Además de la construcción física del prototipo, se realizó un software que permite ordenar al robot realizar diversas trayectorias. El esquema del hardware se muestra en la figura 1.2. En el proyecto de Bernardo, se utilizaba el puerto paralelo como forma de interactuar con el hardware. Dos de sus líneas se utilizaban para controlar la velocidad y giro de los motores a través de un chip. Para conseguir una gran precisión en el control de esta velocidad se realimentaba el sistema gracias a las salidas de dos encoders magnéticos, uno en cada motor. Estos encoders proporcionan dos señales de pulsos cada uno desfasadas un semiperiodo. Según sea la frecuencia de estos pulsos se

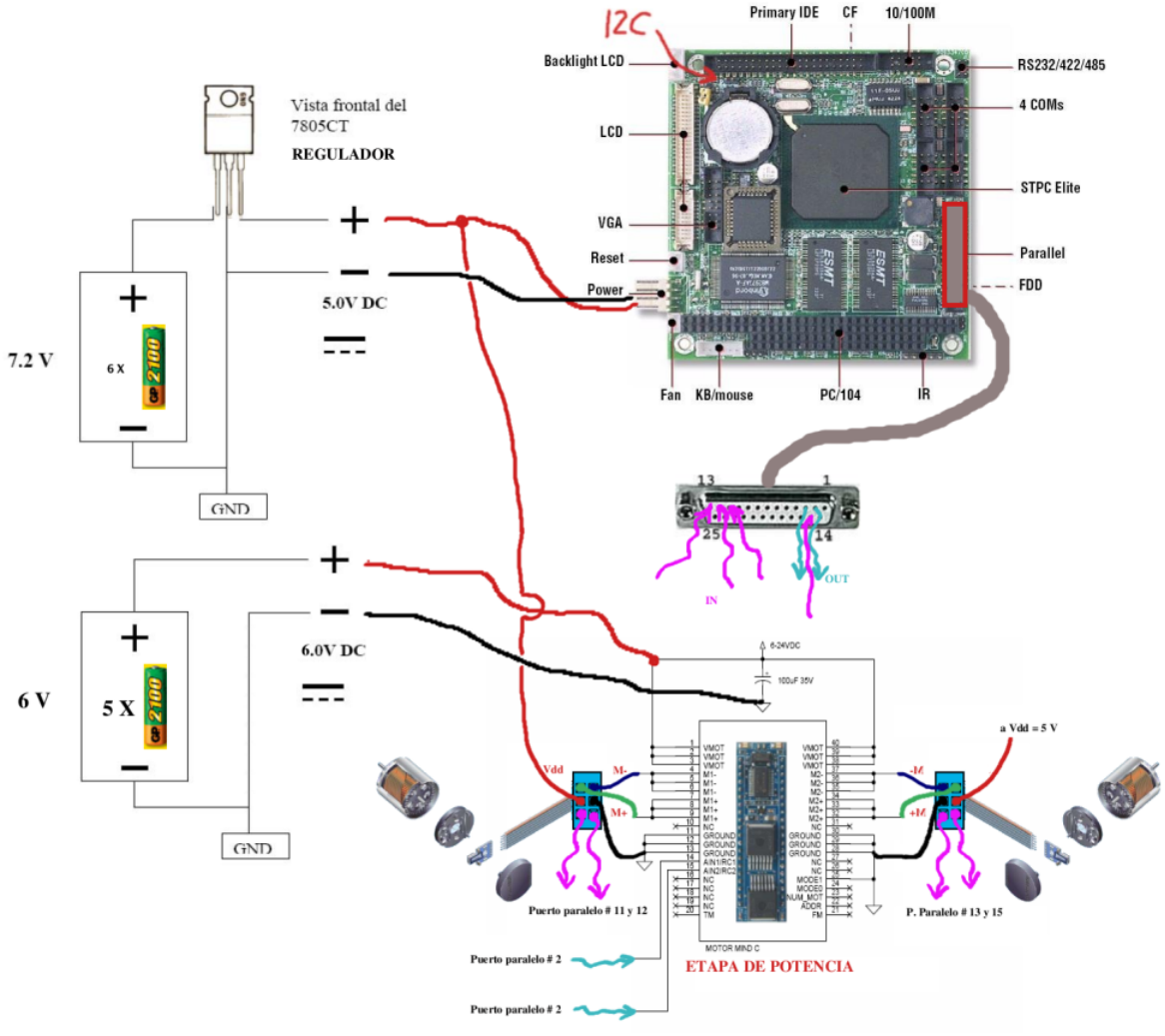

Figura 1.2: Esquema del prototipo

obtiene la velocidad de giro y según sea el desfase, positivo o negativo, se obtiene el sentido del giro. Como el puerto paralelo sólo tiene una línea para generar interrupciones se tuvo que realizar la lectura de los pulsos mediante sondeo. Esto causa una sobrecarga excesiva por lo que Bernardo dejó propuesto el uso de un contador Agilent HCTL-2032 que liberase a la CPU y disminuyese errores de medida en la cuenta de flancos.

1.2.2. MaRTE OS

MaRTE OS [6] es un sistema operativo empotrado de tiempo real distribuido bajo licencia GNU/GPL. Ha sido desarrollado principalmente por Mario Aldea Rivas, como uno de los resultados de su tesis doctoral, dentro del Grupo de Computadores y Tiempo Real, bajo la dirección de Michael González Harbour. Sus principales características son:

- Permite ejecutar aplicaciones en máquinas desnudas.
- Da y controla el acceso a los recursos hardware.
- Es bastante diferente de un sistema de propósito general como Linux ya que no soporta sistemas de ficheros ni tiene una consola de comandos.
- Sigue el subconjunto mínimo de tiempo real POSIX.13. Dentro de POSIX.13 MaRTE OS figura dentro de la categoría ”Mínimo”, destinada a sistemas empotrados pequeños, en la cual no es necesario ofrecer sistema de ficheros ni multiproceso. MaRTE OS no hace uso pues de la MMU de algunas arquitecturas, no usa disco duro sino que se aloja en memoria volátil y tampoco necesita de un terminal. Es lo que se denomina en jerga POSIX como controlador de un ”Tostador”.

  Funcionalidad del perfil mínimo del subconjunto POSIX.13 incluida en MaRTE OS:

  Threads: también llamados *Hilos* o *Hebras*. Un *Thread* es una secuencia de instrucciones ejecutada en paralelo con otras secuencias. Los *Threads* son una forma de dividir un programa en varias tareas que se ejecutan de forma concurrente.

  Mutexes, **Variables Condicionales**, **Semáforos**: estructuras utilizadas para la sincronización de varios *Threads*.

  Señales: servicio del núcleo que permite indicar la existencia de un evento.

  Relojes y contadores: ofrecen funcionalidad para controlar el tiempo de ejecución de cada *Thread*.

  Suspensión de *Threads*, retrasos absolutos y relativos.

  ”Ficheros”de dispositivo y entrada/salida Permiten tratar los dispositivos desde un punto de vista abstracto como si fueran un fichero al que realizar llamadas del tipo **open**, **read**, **write**, ...

  Funcionalidad extra: Manejadores de interrupciones hardware, planificación definida por la aplicación.
- Ofrece concurrencia a nivel de thread (tareas en Ada) pero no a nivel de procesos. En MaRTE OS sólo hay un proceso (que será el encargado de crear los threads-tareas necesarios), y por tanto un único espacio de direcciones de memoria.
- Todos los servicios tienen una respuesta y latencia temporal acotada, por lo que se puede utilizar para aplicaciones de tiempo real (tanto estricto como no).
- El espacio de direcciones es compartido por el *núcleo* (*kernel* en Inglés) y la aplicación con lo cual no se provee la protección de otros sistemas operativos y se debe probar a fondo el sistema final. La ventaja es que permite una mayor velocidad de ejecución.

- Es un *núcleo* monolítico. Los principales tipos de núcleos son los monolíticos, micronúcleos, híbridos y exonúcleos. Un núcleo monolítico se caracteriza porque cuando se está ejecutando una parte de su código no hay otras partes ejecutándose concurrentemente. Permite simplificar su desarrollo y ser muy eficientes en sistemas monoprocesador, aunque es complicado portarlo a sistemas multiprocesador. Linux en sus inicios también era un núcleo monolítico aunque hoy en día es un híbrido. Los sistemas basados en micronúcleos son muy interesantes desde un punto de vista teórico pero resultan muy complicados de llevar a la práctica. En ellos, el núcleo está formado por diversos procesos concurrentes que se comunican entre sí. Existe un interesantísimo debate entre Linus Torvalds y Andrew Tanenbaum al respecto [7].

- Está escrito en Ada 95 (salvo algo de código en C y ensamblador) pero ofrece interfaces tanto para aplicaciones Ada como C, o incluso aplicaciones que mezclen threads con tareas Ada.

- Es portable a diferentes plataformas gracias a una capa de abstracción hardware. Incluyendo microcontroladores, de gran importancia de en el mundo de los sistemas embebidos. Actualmente se encuentran implementadas para:

  Arquitectura x86 PC (bien en máquinas desnudas o bien en emuladores). El *núcleo* es compatible con cargadores que usen el protocolo Multiboot.

  Arquitectura Linux y LinuxLib, donde MaRTE OS corre dentro de Linux. Principalmente destinada a investigación y enseñanza (no se puede alcanzar requerimientos de tiempo real estricto dentro de un sistema que no lo es, como Linux).

  Arquitectura M683XX. Resultado de proyecto fin de carrera [8]. No se distribuye actualmente con MaRTE OS debido a que corresponde a una versión antigua que no ha sido actualizada.

- Es código libre con licencia GNU/GPL. Descargable en [6].

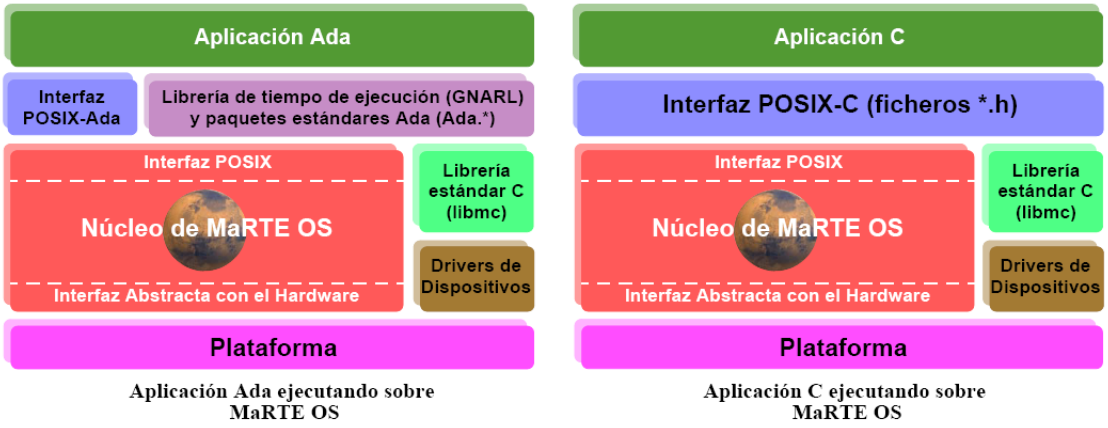

Figura 1.3: Arquitectura de MaRTE OS

En la figura 1.3 podemos ver la arquitectura en forma de capas de MaRTE OS para una aplicación escrita en Ada y en C. La principal diferencia entre ambas radica en la capa utilizada para comunicar la aplicación con el resto del sistema.

- En el caso de las aplicaciones Ada, esta interacción se realiza en su mayor parte de forma indirecta a través de la librería de tiempo de ejecución del compilador GNAT, llamada GNARL, y en parte a través de la interfaz POSIX-Ada. Es decir, para crear un hilo de ejecución no usamos una función como en C, sino que creamos una tarea Ada, mediante la palabra **task**, y será la librería GNARL la que se encargue de transformar adecuadamente esa instrucción según el sistema operativo donde se ejecute (o incluso puede ser una máquina desnuda). Sin embargo, si queremos sincronizar una tarea con un thread C, no podemos usar las utilidades de Ada (como los objetos protegidos) sino que necesitamos recurrir a servicios del sistema operativo, los **Mutexes** en este caso, mediante la interfaz POSIX-Ada. La librería de tiempo de ejecución es en su mayor parte independiente de la plataforma sobre la que se ejecuta. Sólo una pequeña parte, denominada GNULL (GNU Low-level Library), debe ser modificada para adaptarse a las diferentes plataformas de ejecución. Esta adaptación es necesaria para cada versión del compilador GNAT por lo que MaRTE OS es definitivamente dependiente del compilador GNAT. No podemos compilarlo con un compilador Ada cualquiera.[1]

- En el caso de las aplicaciones C, la interacción se realiza a través de la interfaz POSIX.1, una capa muy delgada consistente únicamente en un conjunto de ficheros de cabecera que definen directamente las funciones exportadas por el núcleo de MaRTE OS y la librería estándar C.

Como puede apreciarse en ambas figuras, el núcleo incluye una interfaz abstracta de bajo nivel para acceder al hardware. En ella se define la visión que del hardware tienen las partes del núcleo que son independientes de la plataforma de ejecución (Ej: x86, PowerPC, ARM, MIPS,...). Esta interfaz constituye la **única parte del núcleo que es dependiente del hardware**, lo que facilita el portado de MaRTE OS a distintas plataformas. Por otro lado, los gestores de dispositivos (drivers) también presentarán dependencias respecto al hardware sobre el que se ejecutan. Además, la librería estándar C depende del hardware sobre el que se ejecuta en lo referente a las operaciones de entrada/salida por consola. La E/S por consola no está incluida dentro de la interfaz abstracta con el hardware porque no es común a todas las posibles arquitecturas (los sistemas empotrados normalmente no disponen de dispositivo de interfaz con el usuario). Ver que GNARL se asienta también sobre esta librería para realizar su entrada/salida de texto (**`Ada.Text_IO`**) y que por tanto necesitaremos adaptarlo para cada compilador.

1.2.3. Subsistema de drivers driver de MaRTE OS

Los *drivers* son código software encargado de controlar un dispositivo periférico abstrayendo al usuario de las partes más internas de ese manejo. En MaRTE OS se ha elaborado un subsistema, resultado del proyecto fin de carrera de Francisco Guerreira Payo [9], que nos ofrece un entorno simple con una instalación y uso de los *drivers* (tanto en C como en Ada) estandarizada, y que permite una adaptación de *drivers* de sistemas abiertos como Linux de manera sencilla.

[1] Como curiosidad para el lector avanzado, esta adaptación se encuentra en el directorio libmgnat/ donde hay directorios para cada compilador compatible con MaRTE OS

Similar a sistemas operativos derivados de Unix, las tareas acceden a los dispositivos mediante una interfaz estándar de acceso a ficheros (**open**, **close**...). Dado que en MaRTE OS todo se encuentra en RAM esta especie de sistema de ficheros de dispositivos se encuentra enlazada de forma estática. En la sección 2.4 veremos este subsistema detalladamente.

1.3. Objetivos del Proyecto

El contador propuesto en el proyecto de Bernardo, ver 1.2.1, requiere **un puerto digital con muchas líneas** para su control. Esto motivó la compra de la **tarjeta PCM-3718-H**, que posee dos puertos digitales de entrada/salida de 8 bits. Además, dispone de 16 canales de entrada analógica que nos pueden ser útiles para capturar señales de todo tipo como, por ejemplo, de sensores.

Sin embargo, quedó constatado que la plataforma escogida no es tan fácilmente expandible como otras basadas en microcontroladores. Afortunadamente la plataforma posee un **bus I2C**, muy utilizado por periféricos típicamente destinados a sistemas empotrados por lo que sería muy interesante poder manejarlo.

Por otro lado, Bernardo comprobó una vez montado el prototipo que las ruedas del motor resbalaban. Esto impedía saber con exactitud la dirección del robot símplemente midiendo el giro de los motores a través de los encoders. Por todo ello, se decidió comprar **una brújula magnética** con interfaz I2C.

Finalmente, necesitamos configurar una nueva forma de arranque para cuando el robot esté terminado (no puede llevar un disco duro a cuestas) y otra para la fase de desarrollo, mediante Ethernet, sin necesidad de usar las ya obsoletas disqueteras. Dicho esto, los objetivos que se plantearon son:

- Configurar el arranque de MaRTE OS desde una tarjeta de memoria CompactFlash™y mediante Ethernet sin usar disquete.
- Diseñar e implementar un software capaz de gestionar la entrada analógica y la entrada/salida digital de la tarjeta multifunción PCM-3718-H de Advantech.
- Diseñar e implementar un subsistema software capaz de gestionar el uso de los buses I2C que se encuentren en un sistema empotrado, ofreciendo interfaz tanto a *drivers* escritos en C como en Ada.
- Programar un *driver* para una una brújula magnética con interfaz I2C, como demostración del funcionamiento real del subsistema I2C y para un control más preciso de la trayectoria del robot.

La aplicación de estos todos estos resultados, además del pretexto de la participación en RoboCup, podrá realizarse en todo tipo de desarrollos, por ejemplo en sistemas de automatización industrial o de telecomunicación.

Capítulo 2

Hola MaRTE

Este es un capítulo escrito con una visión muy práctica y con un estilo informal de tal manera que el lector pueda imaginarse rápidamente el entorno de trabajo, es decir, de dónde partimos y qué cosas podemos hacer.

2.1. *hello world*

Vale, algo fácil y rápido para entrar en calor. Vamos a correr un simple ejemplito de aplicación con MaRTE OS sobre un emulador[1] dentro de nuestro sistema GNU/Linux. Usaremos el emulador QEMU por ser de código libre y muy fácil de instalar. Otro emulador libre muy interesante es *BOCHS*. Dentro de los emuladores privativos uno de los mejores es VMWare. Instala el emulador *QEMU* como **root** :

```
-- hazte root =>su
cd /
wget http://fabrice.bellard.free.fr/qemu/qemu-0.8.0-i386.tar.gz
tar zxvf qemu-0.8.0-i386.tar.gz
-- salte de root =>exit)
```

Explicación: Nos hacemos **root** para tener privilegios de escritura en los directorios estándar de instalación de **qemu**. Mediante **wget** nos descargamos **qemu** (su distribución binaria) al directorio raíz. Finalmente utilizando el comando **tar** y las opciones **z** (descomprimir usando gzip), **x** (extraer archivos del .tar resultante), **v** (mostrar dónde se extraen los archivos) y **f** (usar el archivo que decimos), descomprimimos los archivos en los directorios adecuados (si no existe un directorio se crea automáticamente).

El software resultante de este proyecto se instalará de una manera similar en los directorios de MaRTE OS.

[1] Un emulador es un software que permite ejecutar programas en una plataforma (arquitectura hardware o sistema operativo. A diferencia de un simulador, que sólo trata de reproducir el comportamiento del programa, un emulador trata de modelar de forma precisa el dispositivo que se está emulando.) diferente de la cual fue escrito originalmente

Instala el compilador GNAT en tu cuenta de usuario:

```
cd
wget http://marte.unican.es/gnat-3.14p-i686-pc-linux-gnu-bin.tar.gz
mkdir gnat
cd gnat
tar zxvf ../gnat-3.14p-i686-pc-linux-gnu-bin.tar.gz
cd gnat-3.14p-i686-pc-linux-gnu-bin
./doconfig
-- Escoger opción 3 y como directorio base /home/tucuenta/gnat)
export PATH=/home/tucuenta/gnat/bin:$PATH
./doinstall
```

Explicación: en **./doconfig** escogemos nuestra cuenta de usuario para no interferir con otras versiones del compilador que pueda tener el sistema. Este script modifica una serie de archivos internos según el directorio base que le indiquemos. El segundo script, **./doinstall** se encarga justamente de copiar los archivos (que es lo que significa 'instalarlos') en los directorios adecuados, teniendo en cuenta los cambios hechos en el directorio base. Para que MaRTE OS pueda llamar al compilador hemos de indicarle su situación mediante una variable de entorno. El comando **`export`** funciona en la *shell* BASH (**`echo $SHELL`**) y crea una variable de entorno temporal. Para hacerla fija añade el comando a **`~/.bash_profile`**. Esto además permite que si hay varios compiladores instalados el nuestro sea el elegido.

Instala y compila el núcleo de MaRTE OS en tu cuenta de usuario:

```
cd
wget http://marte.unican.es/marte1.4.tgz
tar zxvf marte1.4.tgz
cd marte
./minstall
-- No pentium
-- Gnat = /home/tucuenta/gnat/lib/gcc-lib/i686-pc-linux-gnu/2.8.1
-- HOST IP cualquiera, no lo usaremos
-- /home/tucuenta/export
-- El resto no nos importa =>Ctrl + C
mkdir /home/tucuenta/export
export PATH=/home/tucuenta/marte/utils:$PATH
```

Explicación: el script **./minstall** se encarga de configurar y compilar MaRTE OS quedando listo para ser enlazado con nuestra aplicación.

Crea tu primer **mprogram** con un *hello world*:

```
mgnatmake examples/hello_world.adb
```

Explicación: con esta orden compilamos nuestra aplicación y la enlazamos al núcleo dando como resultado un archivo ejecutable llamado **mprogram**.

Crea una imagen de disquete con GRUB (ver 2.3.1) y **mprogram**:

```
cd ../export
cp ../marte/netboot/grubfloppy.img .
mkdir mfloppy
-- hazte root: su
mount -o loop grubfloppy.img mfloppy/
cp mprogram mfloppy/netboot (overwrite yes)
umount mfloppy
-- salte de root: exit)
```

Explicación: la orden **mount** permite montar la imagen del disquete de manera que lo podemos ver como un directorio más sobre el que leer o escribir. Copiamos nuestro **mprogram**, sobreescribiendo el archivo **netboot** que es el que está configurado para ser cargado por defecto.

Ejecuta el emulador **qemu** haciendo que arranque desde tu imagen de disquete:

```
qemu -fda grubfloppy.img
```

Explicación: la opción **fda** dice a qemu que utilice el archivo **`grubfloppy.img`** como la primera disquetera del PC que está emulando.

En la figura 2.1 podemos ver una captura de pantalla de QEMU emulando nuestro ejemplo.

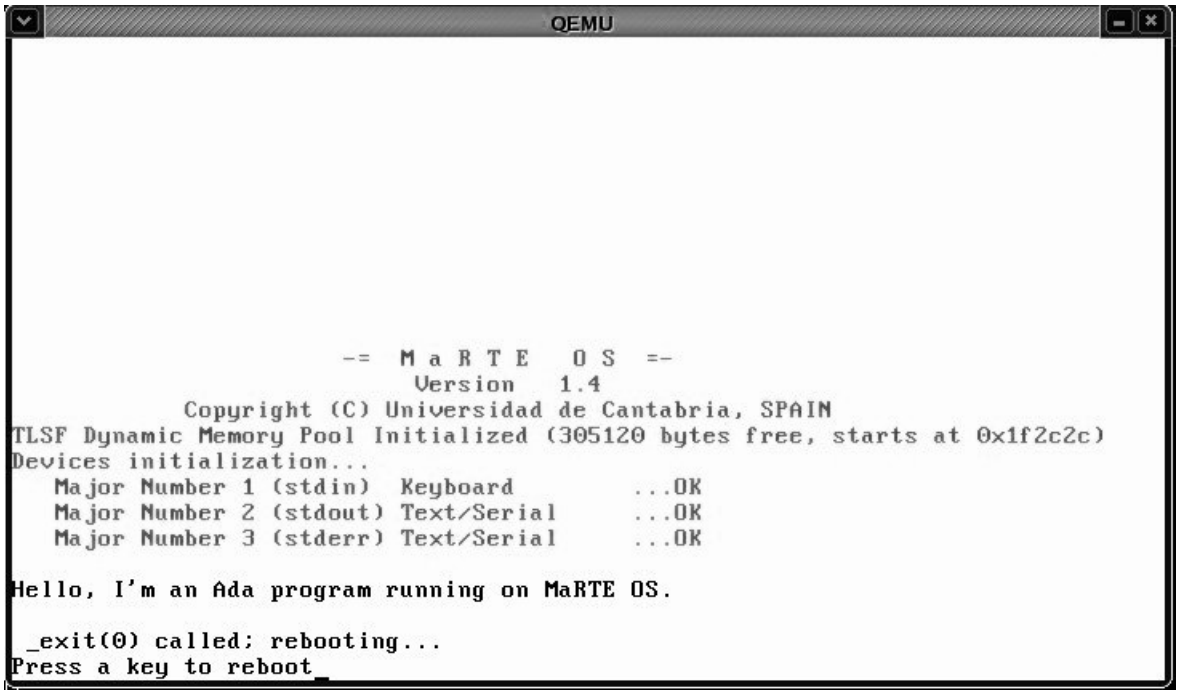

Figura 2.1: Qemu emulando un *hello world*

2.2. tar zxvf marte1.4.tgz

Ahora que ya tenemos una ligera primera idea de como funciona MaRTE OS lo *destriparemos* un poquito. Si echamos un vistazo (**`ls -R ~/marte | more`**) a los contenidos de MaRTE OS nos daremos cuenta de que MaRTE OS no es más que un conjunto de ficheros de código, ordenados por directorios. Los más importantes para nosotros son:

1. **`drivers/`**: donde escribiremos nuestros *drivers*.
2. **`utils/`**: los scripts que nos facilitan la compilación ordenada de MaRTE OS. Los más importantes son:
   - **`mgcc miprograma_c.c`**: Para compilar aplicaciones C.
   - **`mgnatmake miprograma.adb`**: Para compilar aplicaciones Ada.
   - **`mkkernel`**. Para recompilar el núcleo y los *drivers*.
3. **`x86/`** y **`kernel/`**: el núcleo del sistema operativo. **`x86/`** contiene la parte dependiente del hardware x86 y **`kernel/`** la parte independiente.
4. **`include/`** y **`posix5/`**: Interfaz POSIX al núcleo para C y para Ada.

Al compilar nuestra aplicación con los scripts mencionados se genera como vimos un archivo llamado **mprogram**. En el caso de la arquitectura x86, **mprogram** es un binario ejecutable ELF [10] (**`readelf -a ~/export/mprogram`**) y compatible con Multiboot (ver sección 2.3). Como cualquier otro programa, **mprogram**, tiene un principio y un final, y se ejecuta secuencialmente, instrucción tras instrucción:

```
cd
mgnatmake -g marte/examples/hello_world.adb
objdump -d export/mprogram | more
```

El comando **objdump** nos muestra nuestra aplicación en código ensamblador x86:

```
mprogram:     file format elf32-i386
Disassembly of section .text:
00100000 <_start>:
  100000:       eb 0e                   jmp    100010 <boot_entry>
  100002:       89 f6                   mov    %esi,%esi
...
```

Como hemos compilado con la opción **`-g`** podemos ver además ciertas etiquetas útiles intercaladas gracias a las cuales podemos seguir el flujo de ejecución de MaRTE OS desde el principio (una vez ha sido cargado en memoria por el cargador, ver sección 2.3). Sobre el ensamblador del x86 nos basta con saber el significado de **`jmp`**, **`call`** y **`ret`** para poder seguir el flujo de ejecución. Puedes consultar el manual de Intel, Volumen 2, para ver su significado [11].

Mediante la orden **`grep -R -n -i boot_entry *`** buscamos la primera etiqueta, **`boot_entry`**. Así descubrimos que el primer fichero en ejecutarse está programado en ensamblador y es **`x86/boot/multiboot.S`** (se puede ver que su código es el mismo al mostrado por **objdump**). Poco después se llama al fichero **`kernel/base_multiboot_main.c`**. En este fichero se realizan una serie de inicializaciones, se usan los datos proporcionados por el cargador Multiboot y se muestra el **mensaje de bienvenida**.

En la última línea se llama a **`exit(wrapper_main(argc, argv, environ));`** que corresponderá a la función del fichero **`wrapper_main_ada.c`**, si la aplicación está escrita en Ada o a **`wrapper_main_c.c`**, si está escrita en C. Esta diferenciación se utiliza, en la fase de enlazado, porque en el caso de las aplicaciones escritas en C, se necesita llamar a unas **funciones especiales**:

```
adainit();
ret = main(argc, argv, envp);
adafinal();
```

Estas dos funciones (**`adainit()`** y **`adafinal()`**) son necesarias para elaborar todos los paquetes Ada que exporten funciones que se usarán desde aplicaciones C (los paquetes Ada antes de poder usarse necesitan ser **elaborados**, lo que implica la creación de las entidades declaradas en el paquete, seguida por la realización de las operaciones iniciales asociadas a las mismas como la inicialización de variables, ejecución del código de inicialización del paquete o asignación de memoria para crear un objeto dinámico). En el caso de MaRTE OS, estas funciones se crean programando un paquete vacío que hace **`with`**'s de todos los paquetes que van a utilizar las aplicaciones C (al hacerse un **With** de un paquete su código se elaborará antes que se use desde el paquete que hace el **With**), en **`kernel/marte_os_c.ads`**. Los comandos necesarios para producir estas funciones se encuentran en **`utils/mkkernel`** y simplificando equivalen a:

```
$ gnatmake -c marte_os_c.ads
$ gnatbind -n marte_os_c.ali
$ gnatmake -c b~marte_os_c.adb
```

Para terminar con la explicación, hemos visto que en **`wrapper_main_[c,ada].c`** se llamaba a **`ret = main(argc, argv, envp);`**. Este **`main`** no es sino nuestra aplicación (por ejemplo, el *hello world* que hemos visto). Como vemos, si nuestra aplicación termina, se acabó MaRTE OS (excepto si quedan tareas o threads que hayamos creado, claro). Por tanto, podemos intuir que en la aplicación normalmente se programará un bucle infinito que nos mantenga *vivos* o bien se creen nuevos threads. Cuando hay varios threads, para que el sistema siga siendo *el que manda* se usa un pequeño truco. La única forma de obligar al thread a que deje de ejecutarse y se ejecute el sistema es porque se produzca un evento hardware, una interrupción (si no el thread seguiría ejecutándose a sus anchas). Por ello es necesario que el hardware tenga un *reloj de tiempo real*, que MaRTE OS puede programar para que al cabo de unos `ms` provoque una interrupción (temporizador o *timer*). Esta funcionalidad es la más básica de un núcleo y en MaRTE OS se encuentra principalmente en los paquetes **`Kernel.Scheduler`** y **`Kernel.Timed_Events_And_Timer`**.

Como curiosidad, el archivo correspondiente al mensaje que nos invita a rearrancar el sistema se encuentra en **`x86/boot/_exit.c`**.

Como hemos visto, MaRTE OS, al igual que cualquier programa, tiene un principio y un final y no son más que instrucciones una detrás de otra. No hay ninguna magia. Sin embargo, hay una gran diferencia respecto a los programas que se ejecutan en Linux. Hagamos una comparación con Linux para entender como funciona MaRTE OS.

En Linux los programas (ej: hola.exe) residen en un disco duro, guardados mediante un sistema de ficheros, y hay un software (Linux Loader) que se encarga (interactuando con el *núcleo*, claro) de acceder a ese fichero en disco (mediante operaciones de entrada salida **`inb`**, **`outb`**, como a cualquier otro periférico), crear las estructuras necesarias para ese futuro proceso, cargar el binario a memoria RAM, etc. . . , y otro software (Dynamic Loader, perteneciente al paquete Glibc **`http://www.gnu.org/software/libc/`**) que suele estar en **`/lib/ld-linux.so.2`** y que se encarga de buscar las librerías dinámicas que un programa pueda necesitar durante su ejecución (busca en unos sitios determinados claro, no en todo el disco. Generalmente en **`/lib`**, **`/usr/lib`** y donde digamos en **`/etc/ld.so.conf`**). En el *núcleo* Linux 2.6.0, por ejemplo, tras ser cargado en memoria ejecuta la función **`start_kernel`** en **`init/main.c`**, que hace numerosas llamadas a funciones con nombres como **`mirecurso_init()`** que se encargan de hacer inicializaciones. Al finalizar, se hace una llamada a un archivo ejecutable en el disco duro **`run_init_process("/sbin/init")`**.

En MaRTE OS, afortunadamente, las cosas son mucho más sencillas. Nuestro querido **mprogram**, contiene tanto el programa de usuario como el *núcleo* (el *núcleo* en realidad se comporta como si fuera una librería de funciones que podemos usar o no en nuestro programa) enlazados estáticamente. El programa no es un fichero aparte como **`/sbin/init`**, es un código enlazado al *núcleo*. Este **mprogram**, una vez cargado en memoria RAM, siempre estará allí, no accedemos a disco para nada. Por tanto, cuando MaRTE OS se está ejecutando, por ejemplo, un bucle poniendo ”hola”en la pantalla cada segundo, lo que tenemos es un trozo de código en RAM ejecutándose. Y los que mandamos, somos nosotros la aplicación. Que queremos que se acabe MaRTE OS, pues nos salimos de ese bucle, y nos aparecerá un mensaje para reiniciar el ordenador. Así de sencillo y de potente. El usuario de MaRTE OS se siente poderoso, puede iniciar 5 threads que se encarguen cada uno de una cosa (controlar un motor, tomar medidas, ...), saber cuándo se produce una interrupción del exterior y actuar en consecuencia. No tenemos otros procesos que nos echen, ni permisos que nos prohiban acceder a determinadas cosas. Somos nosotros, en definitiva, quienes decimos qué, cuándo y cómo se ejecuta nuestro PC.

2.3. Booooting. Cargando MaRTE OS.

Hemos explicado el funcionamiento de MaRTE OS una vez está en memoria RAM. Sin embargo, aún no sabemos cómo llegó nuestro **mprogram** desde nuestro disco duro hasta la memoria RAM y quién le pasó el control de la CPU. ¿Qué pasa a partir de que pulsamos el botón de encendido de nuestro PC?. Que una serie de instrucciones almacenadas en memoria no volátil son las encargadas de ponerlo todo en marcha. Al arrancar, la así llamada BIOS realiza una serie de chequeos y lee la información de una pequeña memoria CMOS que mantiene sus datos gracias a una batería de litio. Obtiene la fecha y la hora actualizada, configuración de los puertos, parámetros del disco duro y la secuencia de inicialización o

arranque. Esta última le indica el orden en que debe comenzar a buscar un programita que será el encargado de cargar el sistema operativo. Por tanto, el primer punto en el que podemos intervenir es seleccionando el orden de arranque. El segundo punto en el que podemos intervenir es diciendo el contenido de ese programita. Si es un disco duro, deberemos grabar el programita en el primer sector, llamado MBR:

```
dd if=/dev/hda bs=512 count=1 | ndisasm - | more
```

El comando anterior muestra el contenido del MBR (ver [12]). Para ello leemos los primeros 512 bytes del primer disco duro y los desensamblamos. En el MBR además está guardada la llamada Tabla de particiones, que permitirá a ese programita, al que llamaremos cargador, saber en cuántas particiones está dividido el disco duro.

2.3.1. GRUB

GNU GRUB [13] es uno de los cargadores multiboot más importantes. Ha sido derivado del llamado GRUB, GRand Unified Bootloader, que fue originalmente diseñado por Erich Stefan Boleyn. La especificación Multiboot [14] es un protocolo entre el cargador y el *núcleo* de un sistema operativo. Uno de las características más importantes es que permite la coexistencia de distintos sistemas operativos en un mismo ordenador, siempre que estos respeten el protocolo. Además podemos utilizar información que nos da GRUB dentro del *núcleo* (por ejemplo, cuáles son los límites superior e inferior de la memoria). MaRTE OS se ha basado en código de OSKit para soportar este protocolo así que podemos utilizar GRUB para cargarlo.

2.3.2. EtherBoot

EtherBoot [15] es otro cargador, especialmente programado para descargar nuestra aplicación a través de una red Ethernet. El cargador **Etherboot** tiene que ser configurado para nuestra tarjeta de red, protocolos que utilizaremos, etc... Para la descarga a través de la red necesitaremos usar el protocolo DHCP y o bien el protocolo tftp o bien el protocolo NFS. Por tanto, deberemos configurar los correspondientes servidores en nuestro host.

2.3.3. Otros cargadores para MaRTE OS

Otros cargadores que podemos utilizar son Netboot [16], que de hecho viene dentro de la distribución estándar de MaRTE OS, y Nilo [17]. La distribución de MaRTE OS actual (1.58) ocupa descomprimida unos 18 Mb. De estos, 7 MB pertenecen al directorio de netboot. Esto se debe a que se necesita un cargador diferente para cada tarjeta de red. Dada la variedad de tarjetas de red, es bastante probable que la que se use en la plataforma de ejecución (como de hecho nos ocurre) no esté soportada por netboot. Por ello, quizá sería mejor dejar al usuario descargarse el cargador **Etherboot** que necesite a través de internet.

2.3.4. Ejemplos de configuraciones posibles

A continuación se muestran algunas posibilidades para arrancar MaRTE OS. En el capítulo 3 veremos otras dos formas:

1. Imagen de disquete con GRUB y **mprogram** corriendo sobre un emulador: es la forma que ya hemos visto en la sección 2.1.

2. Escribir la imagen en un disquete real y utilizarlo en nuestro PC: simplemente consiste en grabar la imagen en un disquete mediante una de estas dos instrucciones:

```
cat grubfloppy.img >/dev/fd0
dd if=grubfloppy.img of=/dev/fd0 bs=1440k
```

3. GRUB de nuestro GNU/Linux + **mprogram**: esta opción consiste en añadir nuestro *núcleo* a la lista de sistemas operativos que hay en nuestro PC. Para ello editamos el fichero de configuración de nuestro GRUB en GNU/Linux /boot/grub/grub.conf (o sus enlaces en /etc/grub.conf y /boot/grub/menu.lst) y añadimos nuestro *núcleo*:

```
title MaRTE OS
root (hd0,0) <-- (Disco duro y partición donde esté el
mprogram contando desde cero)
kernel /home/tucuenta/export
```

4. Disquete con GRUB + netboot que cargan **mprogram** por ethernet: consiste en grabar grubfloppy.img pero metiendo el archivo netboot correspondiente a nuestra tarjeta, en lugar de un **mprogram**. En el directorio netboot/ de MaRTE OS hay varias tarjetas a escoger. Este disquete se colocará en la máquina prototipo o target y en nuestro entorno de desarrollo, host, deberemos configurar un servidor DHCP (o BOOTP) y un servidor NFS (se puede hacer con el script utils/mexport):

```
/etc/dhcpd.conf:
host mov1 {
      option host-name "mov1.ctr.unican.es";
      hardware ethernet 00:0B:AB:05:BB:B0;
      fixed-address 193.144.198.55;
      server-name "soc1.ctr.unican.es";
      next-server 193.144.198.98;
      filename "/home/tucuenta/export/mprogram";
      option root-path "/home/tucuenta/export"; }}
/etc/exports:
/home/tucuenta/export 193.144.198.55(ro,sync)
```

5. Disquete con **Etherboot** que carga **mprogram** por ethernet: sería lo mismo que la anterior pero en lugar de crear un floppy con GRUB+Netboot, se crearía un floppy a partir de la imagen **Etherboot**. Esta se obtiene en la Web de Etherboot [15] tras seleccionar una serie de opciones (nuestra tarjeta, DHCP y NFS).

2.4. Los Drivers en MaRTE OS

2.4.1. Introducción

Todo dispositivo periférico se controla mediante operaciones de escritura y lectura sobre registros. Por tanto, al final, en lo más profundo nuestro código habrá siempre unas pocas instrucciones encargadas de salir al exterior a interactuar con nuestro dispositivo mediante estos registros. Básicamente podríamos decir que un *driver* consiste precisamente en esconder ese conjunto de registros, instrucciones y demás aspectos de un dispositivo para ofrecer al usuario una manera más sencilla y abstracta de utilizarlo.

En la arquitectura x86 estos accesos al exterior se realizan mediante unas sencillas instrucciones máquina, **inb** y **outb**, que acceden a lo que se llama espacio de direcciones de E/S (a diferencia de otras arquitecturas donde los accesos de E/S están mapeados en direcciones de memoria). A nivel hardware no hay diferencia conceptual entre regiones de memoria o regiones de E/S, a ambas se accede mediante señales eléctricas en los buses de direcciones, control y datos. La diferencia está en que algunos procesadores (como los x86) tienen líneas eléctricas separadas para los puertos de E/S y unas instrucciones especiales para acceder a esos puertos.

Quizá te surja la pregunta: ¿entonces los *drivers* no son portables?. Efectivamente, las instrucciones de E/S son fuente de dependencias. Debido a que trabajan con los detalles de cómo el procesador mueve datos dentro y fuera, resulta muy difícil esconder las diferencias entre sistemas. Como consecuencia, gran parte del código fuente relacionado a los puertos de E/S será dependiente de la plataforma. Por ejemplo, en x86 un puerto es un unsigned short (8 bits sin signo) mientras que en otras plataformas es un unsigned long (32 bits sin signo) y son sólo localizaciones especiales del mismo espacio de direcciones que la memoria.

Los dispositivos periféricos se construyen para ajustarse a un determinado bus, y los más populares son los destinados a los computadores personales. Debido a ello, el resto de procesadores, que no poseen un espacio de direcciones separado para los puertos E/S necesitan emular este comportamiento, muchas veces mediante chipsets o circuitería extra. Por la misma razón, por ejemplo, en Linux se implementa el concepto de los puertos de E/S en cada plataforma sobre la que quiere correr, incluso para aquellas en las que la CPU usa un único espacio de direcciones.

El uso de puertos E/S es común en placas con el bus ISA, mientras que para dispositivos PCI se suelen mapear los registros en la memoria. Por otro lado, a pesar de las similitudes hay que tener cuidado pues existen ciertas diferencias entre un acceso a memoria y un acceso a un registro de E/S. Especialmente en la arquitectura x86, se pueden producir problemas cuando el procesador trata de transferir datos muy rápidamente hacia o desde el bus. Para ello se inserta un pequeño retardo después de cada instrucción. Las funciones de pausa añaden un **`_p`** a los nombres de las instrucciones: **`inb_p`** y **`outb_p`**.

2.4.2. Arquitectura del subsistema de drivers de MaRTE OS

En la figura 2.2 podemos contemplar la arquitectura del subsistema de drivers.

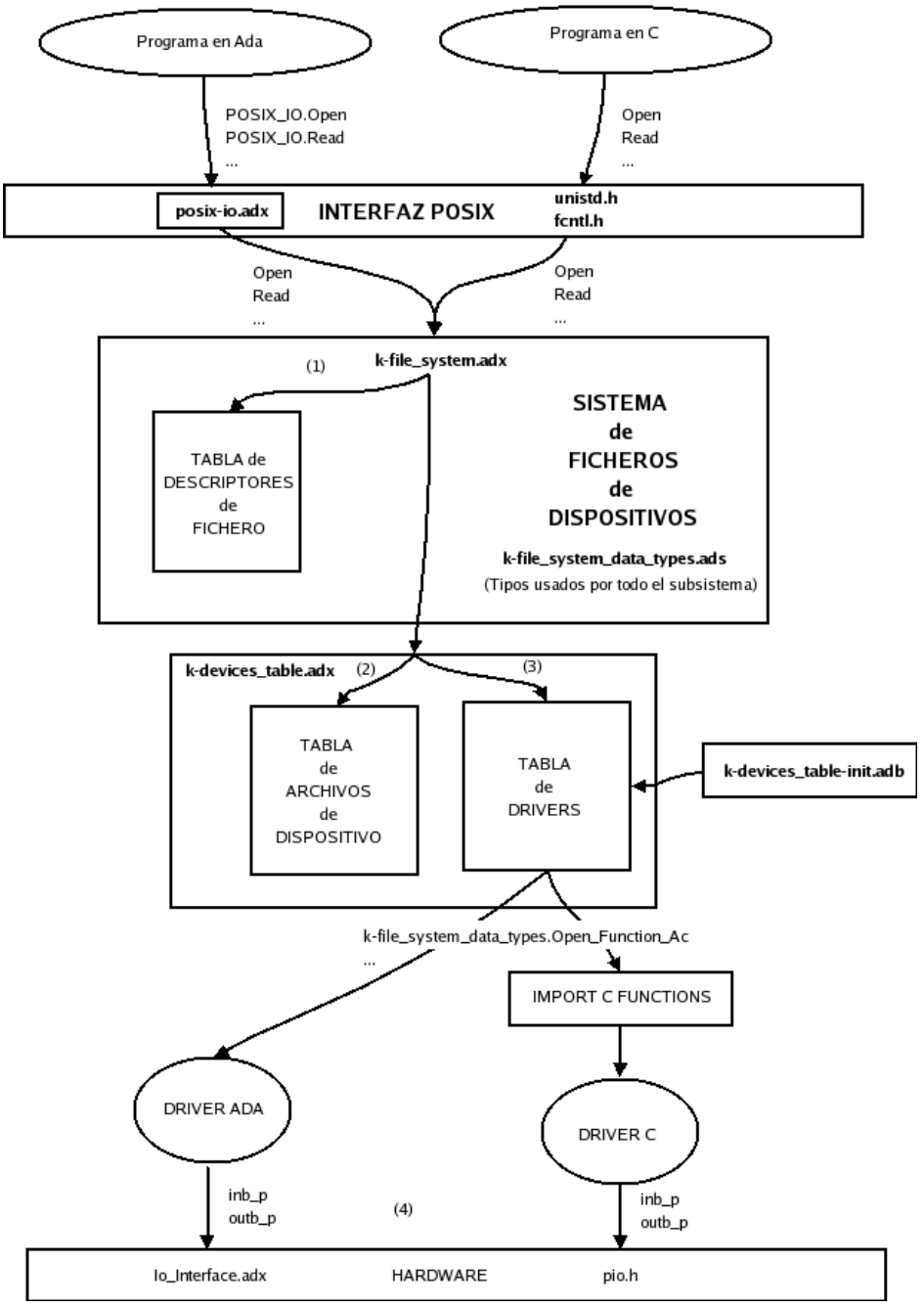

Figura 2.2: Arquitectura del subsistema de drivers

En lo más profundo tenemos nuestras queridas funciones **`inb_p`** y **`outb_p`**, contenidas en **`<sys/pio.h>`** para *drivers* escritos en C y en **`Io_Interface.ads`** para *drivers* escritos en Ada 95. Estas funciones se mapean directamente en las instrucciónes de lenguaje ensamblador **inb** y **outb**:

```
#define inb(port) .... asm volatile("inb %%dx, %0" ....
```

De todo esto las aplicaciones no se enteran en absoluto ya que acceden a través de la interfaz POSIX de entrada/salida, en la capa más superior. Esta interfaz permite acceder a los dispositivos como si fueran ficheros con un nombre (mediante **open**, **read** ...). La interfaz C se mapea directamente en las funciones ofrecidas por el sistema de ficheros mientras que la interfaz Ada realiza primero unas conversiones de tipos.

Cuando el sistema de ficheros recibe la llamada (**open**, **read** ...) utiliza las tablas mostradas en la figura 2.3.

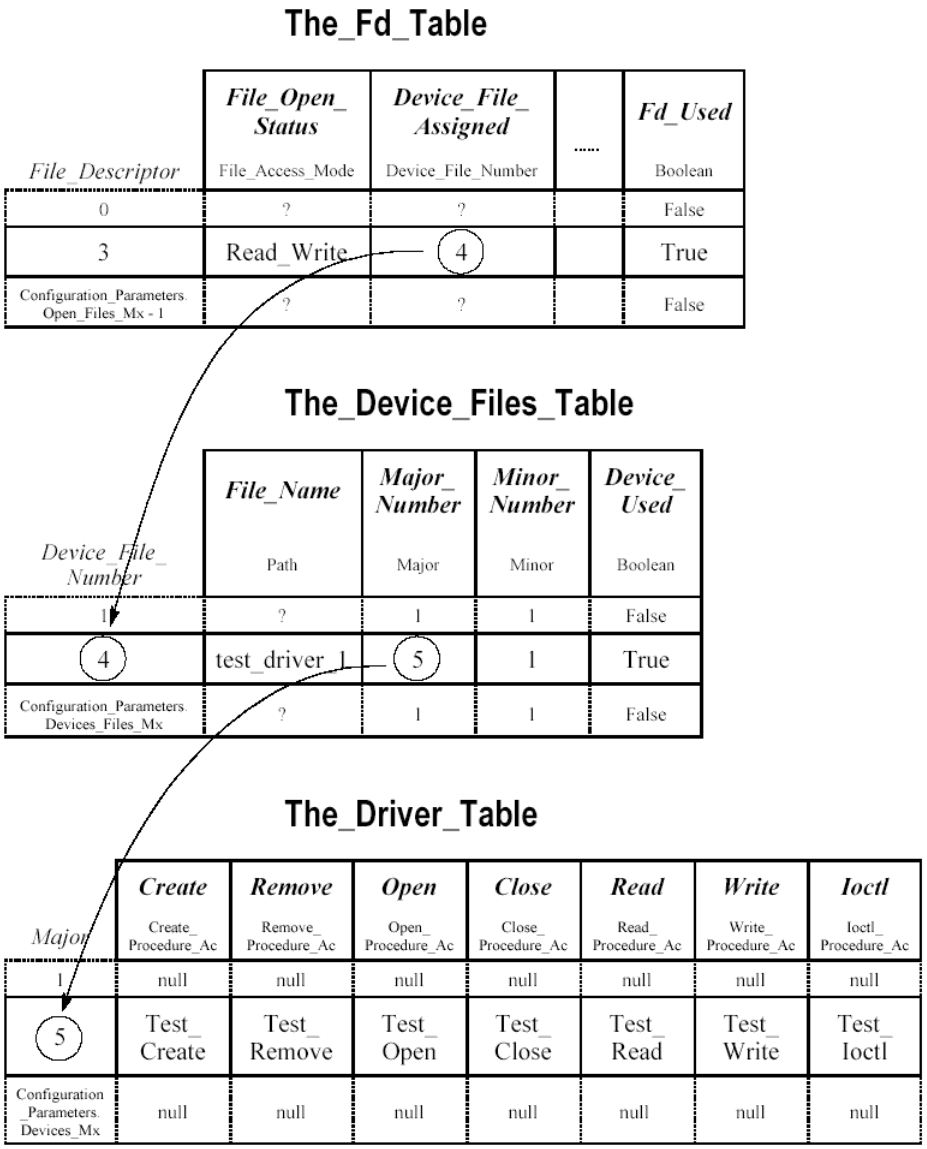

Figura 2.3: Tablas del subsistema de drivers

El orden en que las utiliza es el mostrado en la figura 2.2. En (1) se accede a la tabla de descriptores de ficheros. Esta tabla nos permite saber si un fichero está abierto, en qué modo está abierto (sólo lectura, ...) y almacena un número (Device File Number) que relaciona el descriptor con un driver. Una vez, se obtiene este número podríamos pensar en que ya podemos acceder a las funciones de nuestro driver.

Sin embargo, a veces ocurre que un mismo driver puede ser utilizado para manejar varios dispositivos iguales presentes en el sistema. Por ejemplo, podemos tener varios puertos Serie RS-232 que se utilizan igual. En lugar de crear varios drivers con el mismo contenido para cada puerto, podemos usar otro numerito para, dentro de nuestro driver, distinguir el manejo de varios puertos. Al número que identifica el driver se le llama **número MAYOR** y al número que identifica a cada instancia se le llama **número MENOR**.

La tabla de archivos de dispositivo es la encargada de asociar el nombre de cada dispositivo hardware con su número MAYOR y su número MENOR. Así en el paso (2) de la figura 2.2, el sistema de ficheros (dentro de la llamada **open**, **read** ..) accede a la tabla de archivos de dispositivo para obtener, a partir del nombre, el número MAYOR.

Una vez lo tiene, en el paso (3) accede a la tabla de drivers. Utilizando el número MAYOR obtiene los punteros a otras funciones **open**, **read**, ... (con distintos argumentos) del driver adecuado, que se encargarán de controlar al dispositivo. El driver recibe como argumento un descriptor de fichero que puede usar si quiere para obtener su número MENOR y así saber qué *subdispositivo* debe gestionar (4).

Lo único que nos queda por explicar de la figura 2.2 es **`k-devices_table-init.adb`**. Este archivo contiene código de inicialización (que se ejecuta al elaborarse el paquete), lo que básicamente significa llamar a las funciones **Create** de los drivers instalados estáticamente, que se encargarán de inicializar los dispositivos a su cargo y sus estructuras internas:

```
for I in Major loop
   if The_Driver_Table (I).Create /= null then
      Put ("   Major Number "); Put (Integer (I));
      case I is
        when 1 =>      Put (" (stdin)  ");
        when 2 =>      Put (" (stdout) ");
        when 3 =>      Put (" (stderr) ");
      ...
end loop;
```

Los *drivers* escritos en C necesitan un fichero que importe sus funciones a Ada (*pragma* **import**). Además, como deben poder usarse desde aplicaciones Ada y C hemos de aplicar lo explicado en la sección 2.2. Por ello, en **`marte_os_c.ads`** nos encontramos con los siguientes **`with`**'s:

```
with Drivers_MaRTE;
with Kernel.Devices_Table;
with Kernel.Devices_Table.Init;
with Kernel.File_System;
with Kernel.File_System_Data_Types;
```

Estos ficheros se elaboran antes de que se ejecute nuestra aplicación. Además de permitir la exportación de funciones a C, provocará la ejecución de la parte de inicialización de los paquetes (el orden de ejecución depende del compilador y no tiene porqué ser el orden en que están escritos en **`marte_os_c.ads`**). Por tanto, las llamadas a los Creates de nuestros drivers se efectúan antes que la aplicación.

2.4.3. Creación e instalación de drivers en MaRTE OS

A la hora de realizar un *driver* en MaRTE OS se han de seguir una serie de pasos. Si escribimos nuestro *driver* en Ada, los pasos son los siguientes:

1. Hacer un **`with Drivers_MaRTE`** e implementar las funciones del *driver* siguiendo las funciones prototipo. No es necesario implementar todas, solamente las necesarias.

```
function Open (Fd   : in File_Descriptor;
               Mode : in File_Access_Mode) return Int;
...
```

2. Instalar el *driver* en **`kernel.devices_table.ads`**:
   - **`The_Driver_Table`**: donde debemos agregar las funciones de nuestro *driver* (Ej: **`Create => midriver_functions.Create'Access`**).
   - **`The_Device_Files_Table`**: donde deberemos añadir una entrada para nuestro *driver* con su nombre (PATH) (Ej: **`/dev/midriver`**), el número MAYOR escogido en la tabla **`The_Driver_Table`**, y el número MENOR.

Para *drivers* en C, primero programamos las funciones prototipo que se encuentran en **`#include <drivers/drivers_marte.h>`** y añadimos un **GNUMakefile** para compilarlo. A continuación, las importamos a Ada en un fichero, **`midriver_import.ads`**, siguiendo el esquema de **`demo_driver_c`**, para poder instalarlas en **`The_Driver_Table`** (Ej: **`Create => midriver_import.Create_Ac`**).

Con esto ya tendremos instalado nuestro *driver*. Para usarlo deberemos hacerlo a través de la interfaz POSIX que en el caso de Ada es **`POSIX_IO.ads`** y en el caso de C es **`<fcntl.h>`** para **open** y **`<unistd.h>`** para **close**, **read**, **write** y **ioctl**.

Además se recomienda no utilizar dentro de los *drivers* la interfaz POSIX.Ada pues se compilaría la interfaz completa debido a las **With**'s, creando un ejecutable de mayor tamaño lleno de funciones que no utilizamos. Para el uso de algunas funciones interesantes de esta interfaz se han creado dos paquetes en **`/misc`**, **`Marte_Hardware_Interrupts`** y **`Marte_Semaphores`**, para ser usados en lugar de sus equivalentes POSIX.

Otro aspecto importante es que en MaRTE OS, al ser un sistema destinado eminentemente a aplicaciones estáticas existe un fichero de configuración que afecta a la cantidad de *drivers* que podemos instalar. Se trata de **`kernel/configuration_parameters.ads`** y las principales constantes que nos pueden afectar son:

```
Use_Devices_Filesystem :  constant Boolean := True;
Open_Files_Mx : constant := 8;
Devices_Files_Mx : constant := 16;
Devices_Mx : constant := 10;
Minor_Number_Mx : constant := 255;
Path_Mx : constant := 32;
```

La información sobre el significado de esas constantes, además de ser fácil de intuir por el nombre, viene comentada en el propio fichero de configuración.

Capítulo 3

El entorno de desarrollo

En este capítulo veremos nuestro entorno de desarrollo, es decir, las características del sistema empotrado que vamos a utilizar y cómo lo tendremos configurado para poder realizar el desarrollo de nuestro software.

3.1. Vista global del entorno de desarrollo

En la figura 3.1 se puede ver cuáles serán los elementos más importantes en nuestro entorno de desarrollo. Por un lado tenemos una placa madre que integra el procesador, y algunos de los puertos y buses más típicos de un PC. Es lo que compondría nuestra plataforma de ejecución o *target* (además de los periféricos que luego podamos añadir). Sin embargo, programaremos y compilaremos nuestro MaRTE OS en un ordenador diferente al que llamaremos sistema de desarrollo (*host*)[1]. En este sistema de desarrollo usaremos el sistema operativo GNU/Linux. En principio, cualquier *sabor* de GNU/Linux puede valer para la plataforma x86. Se han hecho pruebas satisfactorias con Ubuntu y con el LiveCD de LinuxFromScratch (hay que hacer unos pequeños cambios al script de instalación, cambiar los **`set`** por **`export`**, de gnat 3.14, pues está hecho para la Shell C, /bin/csh). En este caso usaremos Fedora Core 3, por ser el utilizado en el Grupo. Durante la fase de desarrollo lo más cómodo es que al compilar nuestra aplicación, MaRTE OS (el fichero **mprogram**) se copie a un directorio que estará *exportado* mediante un servidor NFS, es decir, que puede descargarse a través de una red. De esta manera nos bastará encender la plataforma de ejecución para que se descargue la aplicación y la ejecute.

Una vez tengamos nuestro **mprogram** completamente terminado y testeado, deberemos cambiar la forma de cargarlo a una basada en algún soporte portátil (si no el robot necesitaría ser conectado al sistema de desarrollo cada vez que se reinicie para cargarse la aplicación).

MaRTE OS además proporciona un método para depurar errores de forma cruzada usando un cable RS-232 y el depurador de alto nivel GDB. Otra forma algo chapucera de depurar errores si no se dispone de este cable consiste en pausar la ejecución del programa con funciones de entrada/salida de texto (**`Get_Immediate(H); New_Line;`**).

[1] A esto se le llama entorno de compilación cruzado.

Por otro lado, a la hora de programar *drivers*, a veces puede incluso resultar más efectivo el uso de un osciloscopio como método de depuración.

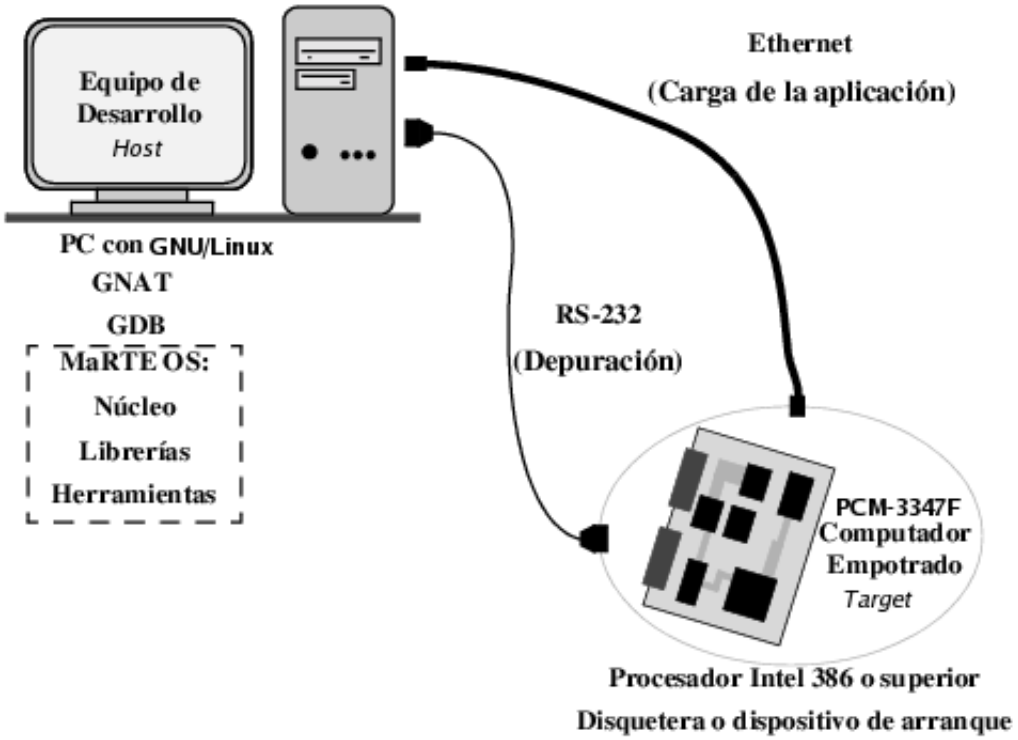

Figura 3.1: Visión global del entorno del desarrollo

En la figura 3.1 se muestra un esquema del entorno de desarrollo. Vemos nuestro sistema de desarrollo donde utilizamos un sistema GNU/Linux con un compilador GNAT y un depurador GDB. La plataforma de ejecución se descargará la aplicación a través de la red Ethernet y podrá ser depurada a través de la línea RS-232.

3.2. La placa madre PCM-3347F

La placa PCM-3347F es un módulo SBC (single board chip) PC/104 de Advantech que posee todas las funciones y potencia de un PC de tamaño normal, pero que sigue el factor de forma del estándar industrial PC/104 (90 x 96 mm). A pesar de ser tan pequeño, contiene todo tipo de funcionalidades. Su precio es de aproximadamente 336 Euros.

Dentro de sus especificaciones podemos destacar las siguientes:

- CPU ST Elite 133MHz, que es compatible con la familia x86. Con lo cual puede ser utilizado con MaRTE OS. Está basado en los 486 por lo que no podemos utilizar MaRTE OS con la funcionalidad de los Pentium. Básicamente, esto quiere decir que tendremos que usar su chip 8254 como contador (ver marte/x86/pit.ad[s,b]) en lugar del más preciso apic (ver marte/x86/local_apic.ad[s,b]).

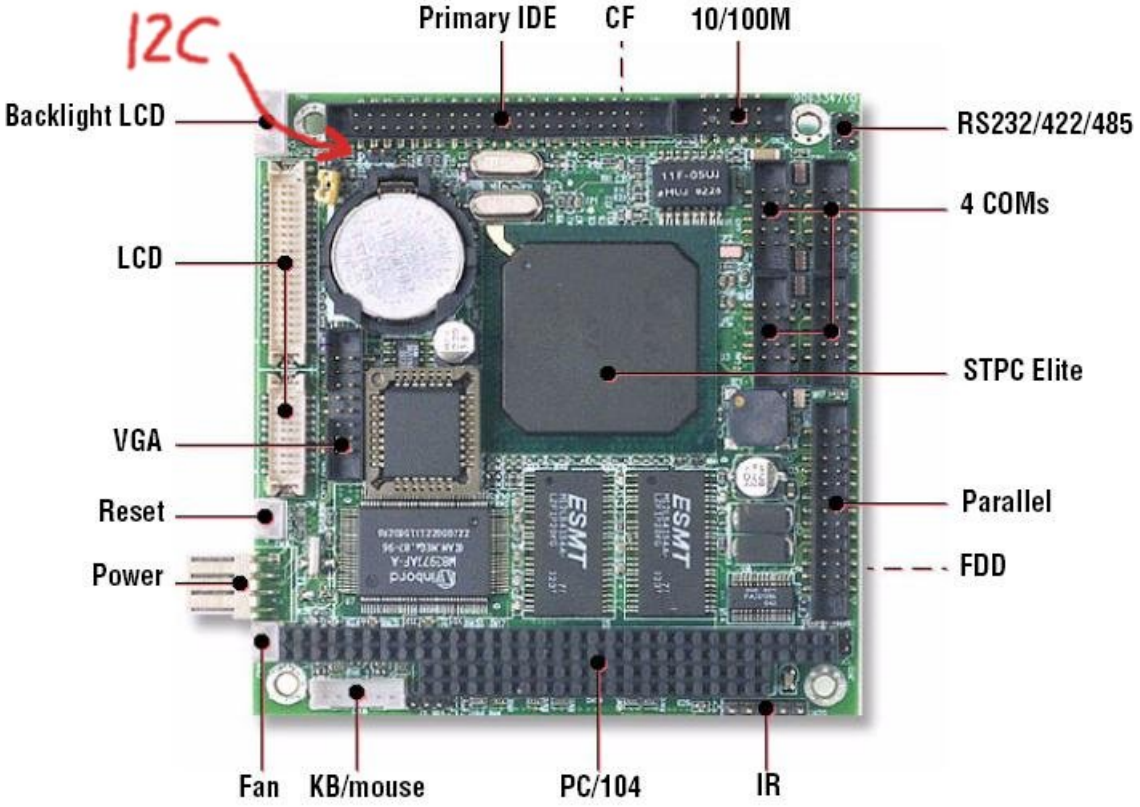

Figura 3.2: Placa PCM-3347

- No necesita ventilador, muy importante pues nos gastaría mucha batería cuando el robot funcione con ellas.
- Permite el uso de tarjetas CompactFlash™, lo cual nos será muy util para cargar desde ella MaRTE OS cuando los robots estén terminados.
- Gráficos: posee controlador para pantallas VGA, con lo cual podremos conectar un monitor, que siempre es útil a la hora de desarrollar. Además posee un controlador para pantallas LCD.
- Tarjeta de red RTL 8100: no tenemos *drivers* para esta tarjeta en netboot pero sí que los hay para etherboot así que podremos utilizarla para cargar nuestra aplicación en la fase de desarrollo.
- Funciona a +5 V. Necesitaremos una fuente de alimentación de 5 V durante el desarrollo y una configuración de baterías similar para cuando esté terminado.
- Su consumo de potencia es muy bajo: +5 @ 1.6 A
- Es fácilmente expandible conectando otros módulos PC/104 de forma apilada comunicándose a través del bus ISA.
- Conexión IDE, Floppy, Teclado, Ratón, Puerto paralelo, 4 puertos serie y puerto infrarrojos.
- Y aunque no lo diga en las especificaciones, también dispone de un bus serie I2C.

3.3. Arranque en la fase de desarrollo

En la fase de desarrollo lo más cómodo es conectar la plataforma de ejecución al sistema de desarrollo mediante un cable ethernet para que se descargue la última versión compilada. Como la placa dispone de conector a disquetera podríamos utilizar la configuración Disquete+Etherboot vista en la sección 2.3.4. Otra alternativa es que en la BIOS, existe la posibilidad de arrancar por LAN. Al seleccionarla y resetear, en la pantalla aparecía un mensaje donde lo más significativo eran las siglas PXE. Acrónimo de Pre-Boot Execution Environment, PXE es un pequeño software grabado en una memoria ROM que permite arrancar por red. PXE tiene la limitación de poder cargar archivos de hasta 32Kb. Por tanto, no podemos cargar nuestro **mprogram** directamente. Necesitaremos hacer un truco: cargar una imagen de EtherBoot especial para PXE. Una vez cargado, el proceso será similar a si se hubiera cargado por disquete. Pero existe otro problema. Tanto PXE como etherboot utilizan DHCP para obtener una IP y un directorio desde donde descargar. Necesitaremos distinguir en nuestro servidor DHCP cuándo somos llamados por PXE y cuando por Etherboot. Los pasos son:

1. Al arrancar, la ROM PXE llama a nuestro servidor DHCP. Éste le dice su dirección IP y cuál es el archivo que se tiene que descargar.
2. Con estos datos, el programita PXE se descarga por TFTP ese archivo, que es una imagen de etherboot. Necesitamos pues configurar un servidor TFTP. Cuando se la descarga le pasa el control.
3. Ahora es Etherboot quien manda y como no sabe ni dónde está, lo primero que hace es una llamada DHCP. Nuestro servidor ve que ahora quién llama es Etherboot por lo que además de darle la IP, le dice cual es el archivo que se tiene que descargar, en este caso **mprogram**.
4. Con estos datos, Etherboot se descarga por NFS ese archivo, el **mprogram**. Necesitamos pues configurar un servidor NFS. Cuando se lo descarga le pasa el control y comienza a funcionar MaRTE OS.

Los ficheros de configuración de los servidores necesarios son:

```
/etc/dhcpd.conf
/etc/xinetd.d/tftp
/etc/exports
```

Se han incluido en el software adjunto a esta memoria para que sirvan de guía aunque su configuración está bien documentada en el manual **man**. La parte más importante pertenece a la configuración DHCPD donde existe un IF que utiliza un campo del protocolo DHCP para distinguir las llamadas de PXE y Etherboot:

```
if substring (option vendor-class-identifier,0,9) = "PXEClient"
  {filename "/eb-5.0.8-rtl8139.lzpxe";}
else if substring(option vendor-class-identifier,0,9)="Etherboot"
  {filename "/home/dsl/export/mprogram";
  option root-path "/home/dsl/export";}
```

3.4. Arranque en la fase final

Configuraremos el arranque desde una tarjeta CompactFlash™[18] para cuando el robot esté terminado y necesite, por tanto, libertad de movimientos. Lo que haremos es instalar el cargador GRUB en ella y copiar nuestra aplicación **mprogram**:

1. Particionamos la tarjeta usando **`fdisk`**. Si no sabes qué fichero la representa utiliza el comando **`fdisk -l`**. Creamos una partición FAT16 y le ponemos la marca de bootable. Con esto lo que hacemos es reescribir la tabla de particiones.

   ```
   /sbin/fdisk /dev/sda
   ```

2. A continuación formateamos la tarjeta.

   ```
   /sbin/mkdosfs /dev/sda
   ```

3. Montamos la partición creada en un directorio cualquiera.

   ```
   mount -t msdos /dev/sda1 /pepito
   ```

4. A continuación creamos la siguiente estructura de directorios:

   `/pepito/boot/grub/stage1, stage2, menu.lst, grub.conf`
   `/pepito/mprogram`

   Los ficheros del directorio **`grub/`** constituyen el cargador (podemos copiarlos del GRUB [13] de nuestro GNU/Linux).

5. En los ficheros de configuración **`menu.lst`** y **`grub.conf`**, deberemos añadir nuestro *núcleo*:

   ```
   root (hd0,0)
   title MaRTE OS sobre CompactFlash
   kernel /mprogram
   ```

   Donde **root** indica el disco duro y partición base, **title** es el mensaje mostrado por GRUB en la pantalla de selección y **kernel** indica el núcleo que se deberá cargar.

Finalmente señalar que ambos modos de arranque, mediante PXE y CompactFlash™han sido probados satisfactoriamente y utilizados en otros proyectos paralelos a este.

Capítulo 4

Driver para la tarjeta PCM-3718-H

En este capítulo se mostrará el desarrollo del software necesario para abstraer al usuario de los detalles de bajo nivel del manejo de la tarjeta PCM-3718-H.

4.1. HARDWARE

La tarjeta PCM-3718-H de Advantech es un módulo PC/104 [19] con capacidades de adquisición de señal analógica y entrada/salida digital. Está pensada para una alta precisión, fiabilidad y un mínimo tamaño y consumo de potencia. Además al ser un módulo PC/104, el montaje en el robot resulta muy sencillo. Esta tarjeta viene con su correspondiente manual de usuario (también accesible por internet), así como software para manejarla. Por tanto la tarea del programador de drivers es bastante sencilla si este manual está bien hecho. Tan solo necesitaremos diseñar una buena interfaz de usuario, y en la implementación leer con detenimiento el manual para manejar la tarjeta adecuadamente. Su precio es de alrededor de 430 Euros. Según el manual las principales características de esta tarjeta son:

- Dos puertos digitales de 8 bits de entrada/salida y compatibles TTL.
- 16 entradas analógicas con tierra común (8 en modo diferencial).
- Conversor A/D de 12 bits, frecuencia de muestreo de hasta 100kHz utilizando DMA.
- Rango de entrada para cada canal analógico de entrada programable por software.
- Varios tipos de disparo (*trigger*) (evento que provoca cada conversión A/D): *disparo* software, *disparo* mediante marcapasos, y *disparo* mediante un pulso externo.
- Transferencia de los datos capturados controlado por programa (consulta), mediante rutinas de atención a la interrupción o mediante DMA.

Todas estas características nos permitirán hacernos una idea del tipo de interfaz que podremos ofrecer a los usuarios de nuestro driver.

4.1.1. Montaje de desarrollo y pruebas

Durante el desarrollo del driver necesitaremos hacer pruebas contínuamente. Para ello hemos de ser capaces de introducir/extraer datos de la tarjeta. Es importante comprobar que no sobrepasamos las intensidades permitidas. Para el caso de la entrada/salida digital las conexiones de la figura 4.1 serían las adecuadas. También se podría utilizar una fuente regulada de voltaje con capacidad para limitar la intensidad. Para la entrada analógica se utilizarán las conexiones de la figura 4.2. Para medir los valores reales en las líneas se puede utilizar un multímetro o un osciloscopio. De esta manera podremos comprobar que lo que capturamos mediante nuestra tarjeta es lo mismo que nos dice el osciloscopio/multímetro.

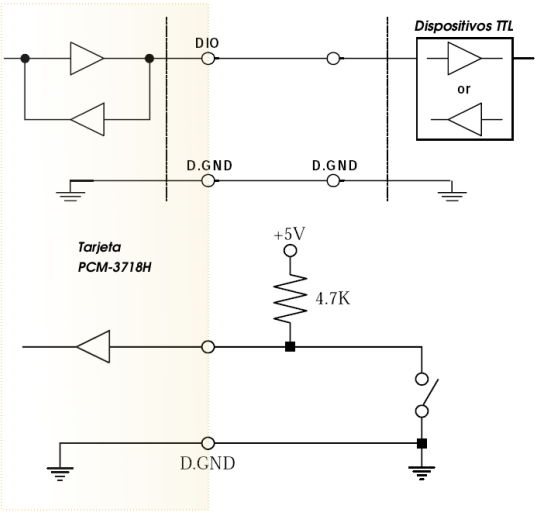

Figura 4.1: Montaje entrada/salida digital

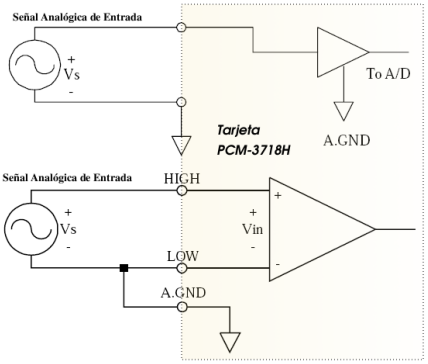

Figura 4.2: Montaje entrada analógica

En la figura 4.3 observamos el montaje completo final sobre el que se realizará el desarrollo de nuestro manejador.

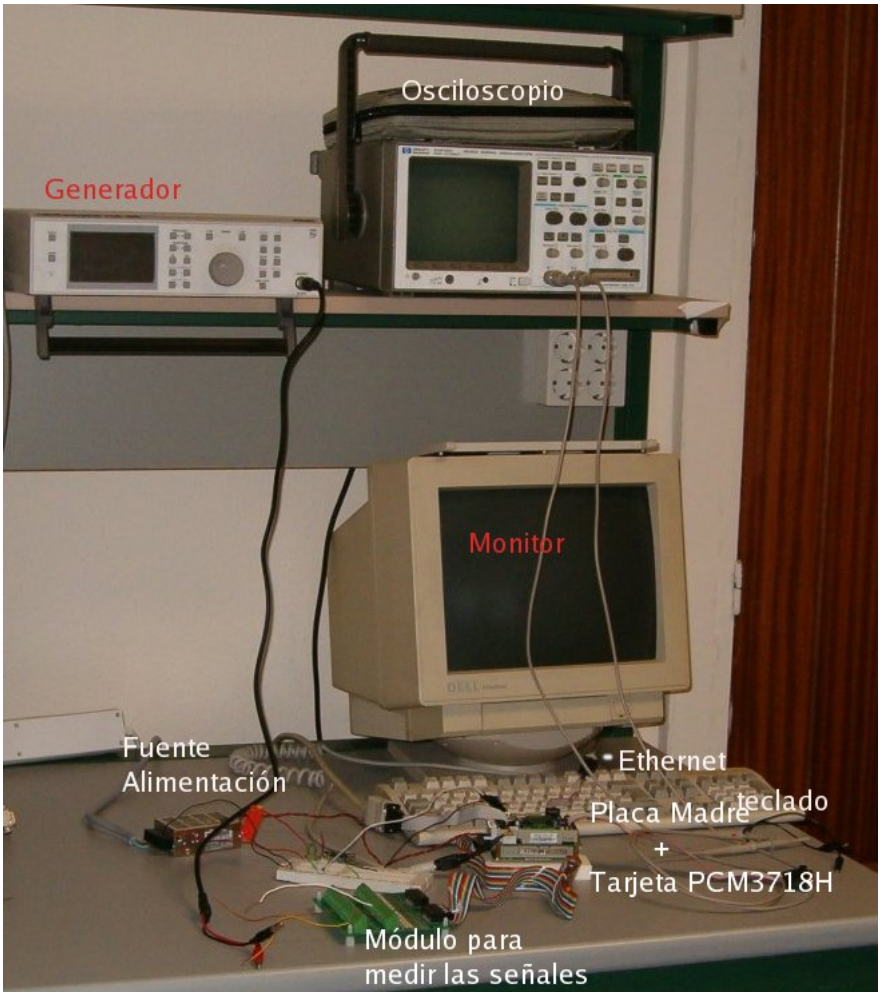

Figura 4.3: Montaje completo

4.1.2. Configuraciones Hardware

La tarjeta PCM-3718-H permite ser configurada mediante *jumpers* harware para ofrecer cierta flexibilidad. Los principales *jumpers* que podemos configurar son:

- Selección de la dirección base para acceder a los registros. La tarjeta requiere 16 direcciones consecutivas dentro del rango 0x000 - 0x3F0. Se debe escojer un rango que tengamos libre. (Para ello podemos consultar la BIOS del PC, que es la encargada de *engañar* a las líneas del bus de direcciones del procesador mediante unos registros internos).
- Selección del canal DMA.
- Selección de la frecuencia base del marcapasos (10Mhz ó 1Mhz).
- Selección de entrada analógica diferencial o simple.
- Selección de fuente de *disparo* externo.

Esta configuración hardware se registrará en nuestro driver de forma estática mediante constantes. Es decir, será el usuario de la tarjeta el que según qué configuración hardware desee deberá modificar los valores de dichas constantes. Quizá estas constantes podrían obtenerse mediante pruebas, pero en sistemas empotrados en los que las configuraciones se saben de antemano, esta otra forma resulta mucho más robusta y simple.

4.1.3. Registros de la tarjeta PCM-3718-H

Como cualquier otro dispositivo periférico la tarjeta PCM-3718-H se maneja mediante una serie de registros. Esta tarjeta, concretamente, posee 16 registros de direcciones consecutivas en el espacio de direcciones de entrada/salida del PC partiendo de la dirección Base (ver 4.1.2). La forma de acceder a estos registros ya ha sido descrita en 2.4

Registros de DATOS

- *Registros de datos A/D* (RO[1]): para cada conversión deberemos leer estos dos registros y extraer el valor, AD0 .. AD11 (12 bits), y el canal convertido, CH0 .. CH3 (hay 16 canales de entrada en modo tierra común y 8 en modo diferencial).

Base + 0	AD3	AD2	AD1	AD0	CH3	CH2	CH1	CH0
Base + 1	AD11	AD10	AD9	AD8	AD7	AD6	AD5	AD4

- *Registros de datos digitales* (RW): como la tarjeta tiene dos puertos digitales de 8 bits, existen los correspondientes dos registros de 8 bits, para recibir o enviar bytes.

Base + 3	DIO7	DIO6	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
Base + 11	DIO15	DIO14	DIO13	DIO12	DIO11	DIO10	DIO9	DIO8

Registros de ESTADO

- *Registro de estado A/D* (RO): da información del estado de la operación de conversión.

Base + 8	EOC	N/A	MUX	INT	CN3	CN2	CN1	CN0

  - EOC (End of Conversion): Será el bit que debamos mirar cuando efectuemos conversiones mediante *disparo* software para saber si ya se ha completado la conversión que ordenamos. Cuando el *disparo* sea el **marcapasos** (**pacer**, en el manual) o señal externa no se usará éste sino INT.

    0: La conversión se ha completado o no se ha iniciado ninguna conversión.

    1: La tarjeta está ocupada realizando una conversión.

[1] RO: Read Only (Sólo Lectura); RW: Read and Write (Lectura y Escritura); WO: Write Only(Sólo Escritura)

- MUX: Este nos sirve para consultar si la tarjeta está usando canales con tierra común o diferenciales (seleccionable mediante un *jumper*). Nosotros no lo utilizaremos pues es una característica que representaremos con constantes de configuración como el resto de *jumpers*.

- INT: Este bit se nos pondrá a '1' cada vez que se produzca una conversión y los datos estén listos. Para una nueva conversión deberemos borrarlo (para indicar que ya se han recogido los datos) escribiendo un valor cualquiera en el registro de estado A/D. Para habilitar las interrupciones utilizaremos el registro de control indicando, además, el nivel de interrupción a utilizar. Si usamos *disparo* software es mejor usar el bit EOC ya que así nos ahorramos la operación de borrar. Para los demás *disparos* sí usaremos este bit, bien consultándolo cada cierto tiempo o bien mediante interrupciones.

- CN3 .. CN0: Cuando EOC=0 estos bits nos indican el número del siguiente canal a ser convertido. La tarjeta va convirtiendo rotativamente los canales entre el canal de inicio y final. Éstos se configuran mediante el registro de control Mux Scan.

Registros de CONTROL

- *Disparo Software* (WO): Una vez escojemos el *disparo* software en el registro de control A/D (Base + 9), éste registro nos servirá para, mediante su escritura (cualquier valor), iniciar una nueva conversión. La dirección de este registro es la misma que la del registro de datos A/D (Base + 0), pero no se solapan porque uno es RO y otro WO. Por tanto, la forma de realizar una conversión software consistirá en seleccionar el *disparo* software (Base + 9), escribir cualquier valor en este registro (Base + 0), esperar a que se termine la conversión (bit EOC de Base + 8) y finalmente leer el dato capturado (Base + 0 y Base + 1).

- *Control del Rango* (WO): Este registro nos sirve para seleccionar el rango de entrada de cada canal, controlado por un código de rangos que se guarda en una memoria RAM en la tarjeta.

  Base + 1 N/A N/A N/A N/A G3 G2 G1 G0

  El método de cambiar el rango de un canal consiste en seleccionar el canal que queremos modificar como canal de inicio (registro Mux Scan, Base+2) y a continuación escribir el código del rango deseado en el registro de control de rango (Base+1). Los rangos posibles son:

  $$\pm 5\ \pm 2{,}5\ \pm 1{,}25\ \pm 0{,}625\ 0a10\ 0a5\ 0a2{,}5\ 0a1{,}25\ Voltios$$

- *Registro Mux Scan* (RW): Este canal sirve para establecer el canal de inicio y fin. La tarjeta A/D irá realizando conversiones de forma rotativa entre estos dos canales. Es importante recordar que cada vez que se escribe en este registro, la cuenta se reinicia y se comienza a convertir desde el inicio otra vez (incluyendo cuando queramos cambiar de rango de entrada a un canal).

  Base + 2 STOP3 2 1 0 START3 2 1 0

- *Registro de control A/D* (RW): Este registro se encarga de configurar la forma en que se realizan las conversiones.

Base + 9	INTE	I2	I1	I0	N/A	DMAE	ST1	STO

- INTE: sirve para habilitar (1) o deshabilitar las interrupciones.

- I2 .. I0: sirve para indicar el nivel de interrupción que deseamos utilizar (2 .. 7). Hay que tener cuidado de escojer uno que no esté siendo usado por otro dispositivo.

- DMAE: sirve para habilitar (1) o deshabilitar la funcionalidad DMA (nosotros no la utilizaremos).

- ST1 .. ST0: sirve para establecer la fuente de *disparo* (software, externa o mediante **marcapasos**).

- Registros para controlar el **marcapasos**: La tarjeta PCM-3718-H utiliza el chip de Intel 8254 para la generación de los pulsos que hagan de *disparo* de las conversiones A/D. Este chip tiene una funcionalidad mucho mayor por lo que aquí nos restringiremos a lo que necesitamos. Los contadores 1 y 2 del chip 8254 están dispuestos en cascada y operando en una configuración divisora fija. El contador 1 está conectado al reloj de 1 Mhz o 10Mhz y su salida está conectada a la entrada del contador 2. La salida del contador 2 está configurada de forma interna para que sus pulsos sirvan como *disparo* a la conversión A/D (ver figura 4.4). La frecuencia de estos pulsos vendrá determinada por la ecuación 4.1 donde los valores de cada contador pueden variar entre 2 y 65535.

$$Rate = \frac{F_{clk}}{contador1 * contador2} \tag{4.1}$$

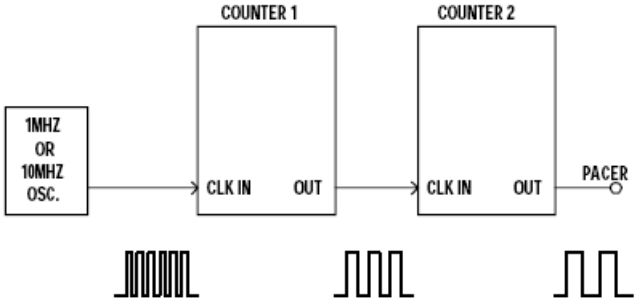

Figura 4.4: Configuración del marcapasos

- *Registro de los contadores* (RW): Cada contador tiene 16 bits, puede valer entre 2 y 65535 y está compuesto por un registro de 8 bits al que se accede dos veces seguidas (ver bits RW1-RW0 del registro de control del **marcapasos**). En el primer acceso se escribe la parte menos significativa y en el segundo acceso la parte más significativa.

Base + 13: Contador 1

Base + 14: Contador 2

- *Registro de control del marcapasos* (WO): Con este registro configuraremos el chip 8254 en el modo 3, dentro de la diversa funcionalidad que nos ofrece. Este modo nos sirve para generar pulsos cuadrados que hagan de *disparo*.

Base + 15	SC1	SC0	RW1	RW0	M2	M1	M0	BCD

SC1-SC0: Selecciona un contador (01=contador1 y 10=contador2)

RW1-RW0: Selecciona modo operación de lectura/escritura (escoger 11 para escribir los contadores en dos fases, primero la parte menos significativa y luego la parte más significativa).

M2-M1-M0: Sirve para seleccionar el modo en que funcionará el chip 8254. Como hemos dicho, debemos escoger el modo 3 (011).

BCD: Permite hacer una cuenta binaria (0..2**16-1) o BCD (0..9999). A nosotros nos interesará una cuenta binaria por lo que deberemos ponerlo a 0.

4.2. SOFTWARE

Una vez nos hemos empapado de todos los aspectos hardware de nuestro dispositivo es la hora de crear el software. El software existente para la placa está realizado para usarse dentro de un sistema operativo privativo basado en ventanas y no se basa en ningún estándar sino en una API basada en funciones. Por supuesto, no ofrece el código fuente así que no nos servirá ni siquiera para los detalles de bajo nivel. Afortunadamente, existe código libre en GNU/Linux para manejar la tarjeta. Se encuentra dentro del proyecto Comedi, destinado a la creación de drivers, herramientas y librerías para la adquisición de datos. El diseño de esta gran colección de software no nos permite adaptar fácilmente el código a nuestro entorno de programación de drivers, ya que existe bastante código intermedio entre las llamadas del usuario y las funciones específicas de cada tarjeta, pero al menos nos resultará útil como fuente de consulta en aspectos de bajo nivel.

4.2.1. Análisis de especificaciones

Esta tarea consiste en determinar la forma en la que se resolverá el problema, pero sin entrar aún en su implementación informática, y determinar asímismo la interfaz con el usuario. Se debe realizar de forma consensuada con el director del proyecto pues será la parte más importante de cara a su uso dentro de otros sistemas. La primera conclusión que se obtuvo es la de desarrollar dos drivers, uno para la parte digital y otra para la parte analógica. Además de la decisión mencionada respecto a registrar la configuración hardware de la tarjeta de forma estática, se decidió también no implementar la funcionalidad DMA pues resultaba algo complicado, y el ritmo de operación no lo requería.

Entrada/Salida Digital

La tarjeta dispone de dos puertos de 8 bits y se ha decidido permitir el uso de los 16 bits como un puerto digital único o de ambos puertos por separado. Se podrá establecer mediante

una llamada IOCTL el puerto con el que deseamos trabajar. Una vez establecido este puerto se podrá escribir y leer en él mediante las funciones estándar POSIX de entrada/salida. En la interfaz Ada, se facilitarán tipos de datos para instanciar las funciones POSIX genéricas, si así se desea.

Entrada Analógica

La entrada analógica es bastante más complicada. Dadas las características hardware de la tarjeta podemos establecer hasta cinco modos de funcionamiento:

1. **Software+Fixed**: En este modo cada vez que se efectúe una llamada de lectura (mediante la función POSIX **read**) se capturarán un número fijo de muestras de forma inmediata y se devolverán los resultados.

2. **Pacer+Fixed**: En este modo cada vez que se efectúe una llamada de lectura (mediante la función POSIX **read**) se capturarán un número fijo de muestras al ritmo de un generador de pulsos (**marcapasos**) y se devolverán los resultados.

3. **Pacer+Scan**: En este modo la tarjeta se encontrará realizando conversiones al ritmo de un generador de pulsos (**marcapasos**) continuamente y almacenándolas en un buffer interno. Cada vez que se efectúe una llamada de lectura (mediante la función POSIX **read**) se leeran datos de dicho buffer. Si el buffer se llena, los datos antiguos se machacarán.

4. **External+Fixed**: Similar a **Pacer+Fixed** pero ahora el ritmo de conversión lo marca una señal digital externa[2].

5. **External+Scan**: Similar a **Pacer+Scan** pero ahora el ritmo de conversión lo marca una señal digital externa.

Utilizaremos las siguientes llamadas IOCTL para controlar nuestro driver según nuestras necesidades:

1. **Set_Range_Of_Channel**: Nos permitirá ajustar el rango de voltaje de entrada de cada canal.

2. **Set_Parameters**: Con esta llamada ajustaremos los parámetros de funcionamiento: la fuente de *disparo*, el canal inicial, el canal de fin, el modo (Fixed o Scan), el número de muestras a convertir y la frecuencia de muestreo.

3. **Get_Status**: Se utiliza dentro del modo Scan. Este comando nos dice el número de muestras dentro del buffer. Nos permite consultar el estado del buffer para extraer o no muestras.

4. **Flush**: Sirve para borrar las muestras del buffer interno.

[2] Se puede seleccionar mediante un *jumper* entre una señal externa o la línea cero del puerto digital de la propia tarjeta.

4.2.2. Diseño de la arquitectura

Dividiremos nuestro código en varios módulos que nos ayuden tanto a descomponer el problema en partes más manejables, como a ofrecer una interfaz clara al usuario ocultando la información que no necesita. En la figura 4.5 podemos ver la arquitectura utilizada de forma gráfica.

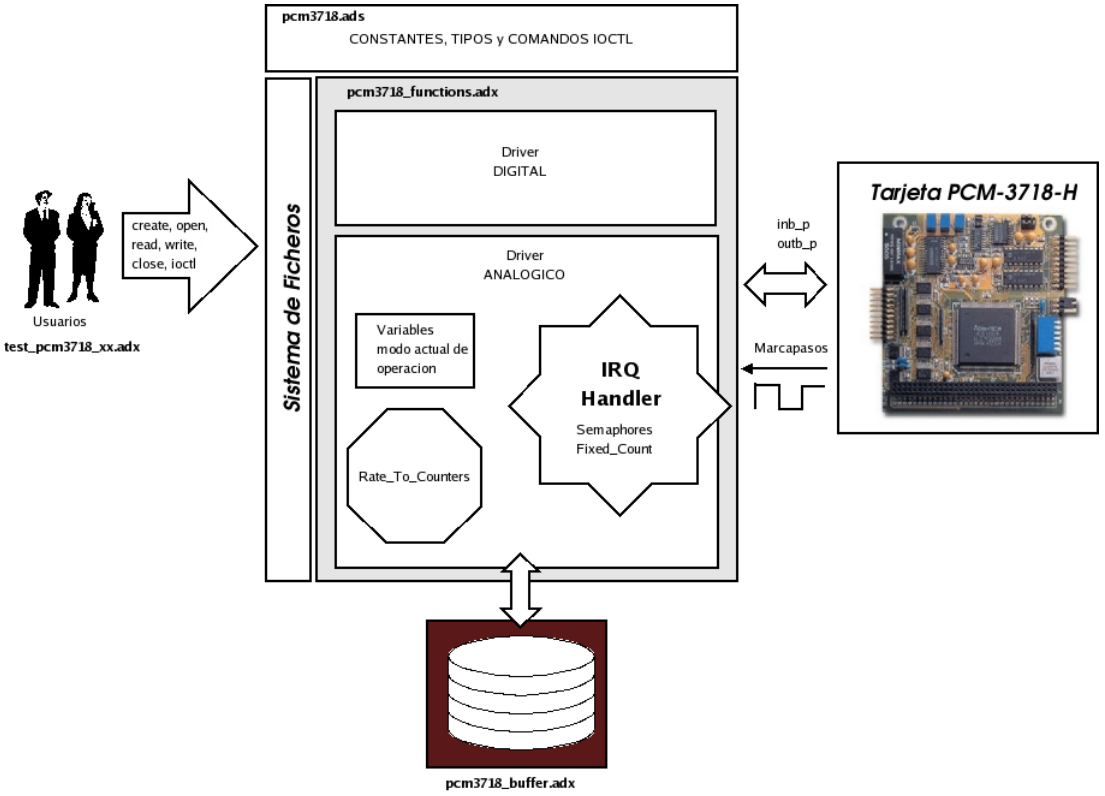

Figura 4.5: Arquitectura del driver de la pcm3718H

Los archivos relacionados con este driver son:

1. **`pcm3718.ads`**
2. **`pcm3718_functions.ad[s,b]`**
3. **`pcm3718_buffer.ad[s,b]`**
4. **`test_pcm3718_ai.adb`**
5. **`test_pcm3718_dio.adb`**
6. **`test_pcm3718_ai.c`**
7. **`test_pcm3718_dio.c`**

La veremos a continuación más detalladamente.

pcm3718.ads

En este fichero se definen las constantes, tipos y comandos que necesitarán usar las aplicaciones que usen nuestro driver. Las dependencias de este módulo son:

- **`Io_Interface`**: para el tipo IO_Port de Pcm3718_Base
- **`Basic_Integer_Types`**: para los tipos Unsigned_8/16
- **`MARTE_Hardware_Interrupts`**: para el nivel IRQ de interrupción

Las constantes que se definen en el fichero son las siguientes:

- CONSTANTES HARDWARE:
  - **`Pcm3718_Base`**: la dirección del puerto de E/S base (Ej: hex 300)
  - **`F_Clock_Pacer`**: la frecuencia del generador de pulsos (1Mhz o 10Mhz)
  - **`Irq`**: el nivel de interrupción de nuestra tarjeta (Ej:Parallel2_Interrupt)
  - **`Num_Channels`**: 16 canales normales u 8 canales diferenciales
- CONSTANTES SOFTWARE
  - **`Buffer_Mx`**: Capacidad en muestras del buffer interno (Ej: 256)
  - **`AD_Code_Min, AD_Code_Max y AD_Code_Range`**: Constantes necesarias para convertir los valores de las muestras en variables reales como temperatura, voltaje, etc...(0,4095,4096). No modificar.

Los tipos de datos que se definen son los siguientes:

- DATOS DIGITALES
  - **`Digital_Data`**: aunque los puertos digitales son de 8 bits, queremos poder utilizar los dos juntos. Por ello definiremos un dato digital como un número de 16 bits.
- DATOS ANALÓGICOS
  - **`Channel_Type`**: tipo para referirnos a cada uno de los 16 canales de la tarjeta.
  - **`Sample_Type`**: tipo de las muestras que capturemos.
  - **`Analog_Data_Type`**: cada dato analógico está compuesto de la muestra en sí más el canal del que se ha obtenido.
  - **`Num_Conv_Type`**: tipo para contar el número de conversiones hechas.
  - **`Analog_Data_Index`**: índice de un array de datos analógicos.
  - **`Analog_Data`**: tipo que utilizaremos como buffer de datos analógicos.

Finalmente, los tipos definidos para los comandos y argumentos IOCTL. En el caso de las aplicaciones Ada deberán instanciar la función IOCTL genérica con estos tipos.:

- COMANDOS IOCTL

  1. **`Ai_Ioctl_Cmd`**: comandos para la entrada analógica. Entre paréntesis se muestran los argumentos que deberán ser suministrados. Si éstos van entre corchetes significa que son opcionales, es decir, que se dan según el modo en que se esté trabajando.

     `Set_Range_Of_Channel`: establece el rango de un canal (Input_Range, Start_Ch).

     `Set_Parameters`: establece parámetros de funcionamiento (Trigger, Start_Ch, Stop_Ch, [Mode, Count, C1, C2]).

     `Get_Status`: obtiene el número e muestras en el buffer (Num_Conv).

     `Flush`: borra todas las muestras del buffer.

  2. **`Dio_Ioctl_Cmd`**: comandos para la entrada/salida digital.

     `Mode_Byte_1`: el primer registro de 8 bits.

     `Mode_Byte_2`: el segundo registro de 8 bits.

     `Mode_Word`: ambos registros formando uno de 16 bits.

- ARGUMENTOS IOCTL

  - **`Range_Type`**: tipo para definir los rangos de entrada de cada canal analógico.

```
type Range_Type is
(Bipolar_5,     --            -5 < Vin < 5
Bipolar_2_5,    --          -2.5 < Vin < 2.5
Bipolar_1_25,   --         -1.25 < Vin < 1.25
Bipolar_0_625,  --        -0.625 < Vin < 0.625
Unipolar_10,    --             0 < Vin < 10
Unipolar_5,     --             0 < Vin < 5
Unipolar_2_5,   --             0 < Vin < 2.5
Unipolar_1_25,  --             0 < Vin < 1.25
Bipolar_10);    --           -10 < Vin < 10
```

  - **`Trigger_Type`**: fuentes de *disparo* disponibles, software (nosotros decimos cuándo se hace la conversión), external (una señal externa marca el ritmo de conversiones) o **marcapasos** (las conversiones se hacen la ritmo del generador de pulsos de la placa).

  - **`Mode_Type`**: modos de funcionamiento ya explicados. Fixed convierte N muestras y nos las devuelve. Scan convierte muestras continuamente y las almacena en un buffer.

  - **`Scan_Rate_Type`**: la frecuencia a la que se capturan los datos cuando se usa el *disparo* tipo **marcapasos**. Si F_Clock_Pacer = 1E6Hz entonces el valor vendrá dado en microsegundos (seg*10e-6) y si F_Clock_Pacer = 10E6Hz, vendrá dado en décimas de microsegundos (seg*10e-7). El driver deberá calcular internamente los valores necesarios de cada contador del chip 8254 para ajustar con el mínimo error el valor que queramos. El valor aproximado será devuelto por la misma variable del argumento IOCTL.

  - **`Ai_Ioctl_Arg`**: este es el argumento IOCTL con el que se instanciará la función IOCTL genérica del paquete POSIX_IO.

```
type Ai_Ioctl_Arg is record
   Input_Range : Range_Type;
   Start_Ch    : Channel_Type;
   Stop_Ch     : Channel_Type;
   Trigger     : Trigger_Type;
   Mode        : Mode_Type;
   Count       : Num_Conv_Type;
   Scan_Rate   : Scan_Rate_Type;
end record;
```

pcm3718_functions.ad[s,b]

Este fichero es el núcleo del driver. En él se implementarán las funciones de interfaz POSIX, abstrayendo al usuario de las características hardware de la tarjeta. Las dependencias de este paquete son:

- **`Drivers_Marte`**: para diversos tipos extraidos del paquete Kernel.File_System_Data_Types como File_Descriptor, Buffer_Ac, ...
- **`Ada.Unchecked_Conversion`**: para realizar la conversión sin comprobación entre el puntero a los argumentos IOCTL y nuestro tipo Ai_Ioctl_Arg.
- **`MaRTE_Hardware_Interrupts`**: para el uso de funciones asociadas al manejo de interrupciones.
- **`MaRTE_Semaphores`**: para la sincronización entre la rutina de atención a la interrupción y el resto del driver.
- **`Io_Interface`**: para las funciones y tipos de E/S Inb_P, Outb_P y IO_Port.
- **`Basic_Integer_Types`**: para los tipos Unsigned_8/16 y operaciones de desplazamiento de bits.
- **`System`**: para el tipo System.Address utilizado como argumento en el manejador de interrupciones.
- **`Basic_Console_Io`**: para mostrar mensajes de error mediante Put, New_Line. En los drivers no se debe usar Ada.Text_Io.
- **`Pcm3718_Buffer`**: para el *buffer* interno donde se almacenan las muestras en el modo scan.
- **`Pcm3718`**: para los tipos y constantes definidos para la tarjeta.

Las funciones interfaz[3] que se definirán son:

- **`Dio_Open`**: reserva la entrada/salida digital y la inicializa. Por defecto se comienza usando el modo Word.

[3]Es importante recalcar que estas funciones no se usan directamente por los usuarios, sino a través del sistema de ficheros

- **`Dio_Close`**: libera la entrada/salida digital.
- **`Dio_Read`**: lee un dato del puerto digital.
- **`Dio_Write`**: escribe un dato en el puerto digital.
- **`Dio_Ioctl`**: cambia de puerto digital (hay dos de 8 bits o uno de 16 bits)
- **`Ai_Open`**: reserva la entrada analógica y la inicializa. Por defecto se comienza en el modo Software+Fixed.
- **`Ai_Close`**: libera la entrada analógica. En caso de que estuviera en modo Scan, desactiva la captura de datos.
- **`Ai_Read`**: lee muestras de la tarjeta.
- **`Ai_Ioctl`**: controla la tarjeta mediante los comandos vistos.

Además existirán dos funciones internas

- **`Pcm3718_Handler`**: se encargará de atender a las interrupciones cuando se use *disparo* por **marcapasos** o externo.
- **`Rate_To_Counters`**: se encarga de obtener los valores de los contadores del chip 8254 que ofrecen una mejor aproximación al valor pedido como frecuencia de muestreo del **marcapasos**.

pcm3718_buffer.ad[s,b]

Con este paquete implementaremos el *buffer* donde se guardarán las muestras en el modo Scan. Para sincronizar la lectura con la escritura de manera que el acceso sea mutuamente exclusivo no nos vale un objeto protegido pues la rutina de atención a las interrupciones no le haría caso. Tampoco nos sirven los **mútex**. Lo que se hará es deshabilitar las interrupciones cuando se quiera leer del *buffer* y después volver a habilitarlas. Además, se usará un semáforo para indicar la existencia de muestras válidas en el *buffer*. Las dependencias del *buffer* son:

- **`Pcm3718`**: para los tipos de la tarjeta.
- **`Basic_Integer_Types`**: para el tipo Unsigned_8.

La interfaz funcional de este paquete será la siguiente:

- **`Flush`**: vacía el *buffer*
- **`Write`**: escribe una muestra (dos bytes) en el *buffer*

  Datos de entrada: Byte_Low y Byte_High (Unsigned_8)

- **`Get_Status`**: obtiene el número de muestras en el *buffer*

  Datos de salida: Pcm3718.Num_Conv_Type;

- **`Read`**: lee muestras del *buffer*

  Datos de salida: The_Data (Pcm3718.Analog_Data) y Count (Pcm3718.Num_Conv_Type)

Un aspecto interesante es que cuando el manejador de interrupciones escriba la muestra capturada en el *buffer*, la escribirá sin procesar. Cuando la aplicación lea un conjunto de N muestras, el *buffer* las convertirá al tipo Analog_Data_Type, con el canal y el valor separados. De esta manera se evita estar mucho tiempo en el manejador de interrupciones, lo que nos puede suponer perder otras interrupciones, y dejar el procesado para momentos más tranquilos. Si no leemos las muestras y el *buffer* se llena, se machacarán las muestras más antiguas.

pcm3718.h

Este fichero es el equivalente a pcm3718.ads pero para la interfaz C del driver. Contiene los mismos tipos y constantes. Para que haya equivalencia a la hora de representar los tipos de Ada y C en la memoria se necesita utilizar clausulas especiales en pcm3718.ads.

```
for Channel_Type'Size use 8;
for Sample_Type'Size use 16;
for Analog_Data_Type'Size use 32;
for Num_Conv_Type'Size use 32;
for Ai_Ioctl_Cmd'Size use Integer'Size;
for Dio_Ioctl_Cmd'Size use Integer'Size;
for Range_Type'Size use 8;
for Range_Type use (
   Bipolar_5     => 0,  Bipolar_2_5   => 1,  Bipolar_1_25  => 2,
   Bipolar_0_625 => 3,  Unipolar_10   => 4,  Unipolar_5    => 5,
   Unipolar_2_5  => 6,  Unipolar_1_25 => 7,  Bipolar_10    => 8);
for Trigger_Type'Size use 8;
for Mode_Type'Size use 8;
```

4.2.3. Diseño detallado y codificación

En esta sección sólo nos detendremos en desgranar las funciones más importantes del driver. Las funciones de la parte digital son muy sencillas ya que simplemente se basan en hacer las operaciones Inb_P y Outb_P para leer/escribir datos en los puertos. Las funciones de la parte analógica están diseñadas de la siguiente forma (se excluyen comprobaciones de errores y otro código más o menos supérfluo):

- **`Ai_Ioctl`**: para empezar haremos un *Case* entre cada uno de los comandos.

  `when Set_Range_Of_Channel =>` escribiremos el rango del canal. Para ello seleccionamos el canal como canal de inicio y a continuación escribimos el rango.

Finalmente debemos restablecer los canales de inicio y fin (debemos guardarlos en una variable estática interna para poder recordarlos).

`when Set_Parameters =>` aquí prepararemos la tarjeta para muestrear según el modo escogido. Por tanto deberemos diferenciar cada caso.

`when Software =>` en este caso debemos programar el registro de control para usar *disparo* software, y bloquear las interrupciones en caso de que las estuvieramos usando.

`when Pacer | External =>` lo primero que haremos es programar los contadores del generador de pulsos si el *disparo* es por **marcapasos**. Para ello usaremos la función Rate_To_Counters que nos dirá los números a escribir y luego los escribiremos. A continuación, debemos diferenciar entre si estamos en el modo Fixed o Scan. En el modo Fixed, dejaremos la tarjeta preparada pero con las interrupciones deshabilitadas. En el modo Scan, inicializamos el semáforo que indica si hay muestras en el *buffer* y a continuación habilitamos las interrupciones. A partir de ese momento ya se estarán realizando conversiones.

`when Get_Status =>` deshabilitamos las interrupciones, leemos el estado del *buffer* y volvemos a habilitar las interrupciones.

`when Flush =>` deshabilitamos las interrupciones, borramos el *buffer* y volvemos a habilitar las interrupciones.

- **`Ai_Read`**: al igual que en **`Ai_Ioctl`** deberemos distinguir cada modo de funcionamiento de la tarjeta.

  `when Software =>` primero borramos el *buffer* de muestras antiguas. A continuación hacemos un bucle donde vamos convirtiendo las N muestras y guardándolas en el *buffer*. Para comprobar la conversión usamos Polling ya que sólo se tarda 7.5us en realizar cada conversión. El motivo de guardarlas en el *buffer* es para arañar tiempo que se gasta en convertir los dos bytes obtenidos en una variable con una muestra y un canal. Cuando se termina de convertir las N muestras se lee el *buffer*.

  `when Pacer | External =>` En el caso Fixed, se coloca en una variable estática el número de muestras a convertir. A continuación se habilitan las interrupciones y se espera en el semáforo Fixed_Sem a que se terminen de realizar. Finalmente se deshabilitan las interrupciones de nuevo y se lee el *buffer*. En el caso Scan, se espera en el semáforo Scan_Sem a que haya al menos una muestra en el *buffer*. A continuación se deshabilitan las interrupciones, se lee el *buffer* y se vuelven a habilitar. Es importante restablecer el valor del semáforo si no hemos vaciado el *buffer* completamente.

- **`Pcm3718_Handler`**: este es el manejador de interrupciones. Con interrupciones habilitadas, cada vez que nos llegue un pulso se ejecutará a la máxima prioridad. Durante su ejecución se deshabilita por hardware esa interrupción en particular en el controlador de interrupciones del PC. Si se produce una interrupción quedará pendiente y se atenderá nada más habilitarse. Si se producen 2 o más, sólo se atenderá una (no se encolan). Distinguiremos dos casos:

  `Fixed =>` Guardará muestras en el *buffer* e irá decrementando la variable estática global hasta convertir todas las muestras. En ese momento, señalizará el semáforo Fixed_Sem, para indicar que ya ha terminado.

`Scan =>` Simplemente se dedicará a coger la muestra y guardarla en el *buffer*. Señalizará el semáforo Scan_Sem para indicar que hay muestras disponibles.

- **`Rate_To_Counters`**: esta función se encarga de obtener los valores de los contadores del chip 8254 que mejor aproximen un valor de frecuencia de muestreo dado. El problema a resolver es el siguiente:

$$\text{minimizar } f(c) = |C1 * C2 - \frac{F_{clock}}{Rate}|$$
$$(2,2) < (C1, C2) < (65535, 65535)$$

Este problema se trata en teoría de la *Optimización* dentro de la subclase *Programación No-Lineal Entera*. Desgraciadamente es uno de los campos con mayor dificultad y aún en investigación. Por ello, el algoritmo que se utilizó se basa en aplicar un bucle para todos los valores posibles de un contador y calcular el error, quedándonos con el menor. Además, resulta fácil obtener la cota máxima de su tiempo de ejecución. En la plataforma utilizada 500 ms.

4.2.4. Pruebas e integración

Antes de llegar a esta solución, se pasó por varios rediseños en los que tuve que resolver algunos problemas inesperados. Para probar el driver se realizarón programas de test en Ada y C tanto para la parte digital como para la parte analógica (test_pcm3718_ai.adb, test_pcm3718_dio.adb, test_pcm3718_ai.c y test_pcm3718_dio.c). Para la comprobación de los valores capturados se tomaron medidas con un osciloscopio. En el caso de la adquisición de datos analógicos se distinguieron las cinco formas de funcionamiento ya descritas.

En los diseños preliminares se había optado por utilizar un *buffer* protegido que era escrito por una tarea interna (Bottom_Half) y leido por la función Ai_Read. Esta tarea permitía que el manejador de interrupciones sólamente se dedicase a coger la muestra y guardarla en una variable atómica, con lo que el tiempo en máxima prioridad era muy pequeño. Correspondía a la tarea Bottom_Half (de una prioridad menor) coger esa muestra y guardarla en el *buffer*. Esta tarea podía ser activada/desactivada de forma asíncrona por la aplicación según el modo de captura que se seleccionase en las llamadas IOCTL (También podía matarse mediante un nuevo comando IOCTL, *kill*). Para ello se creó otro objeto protegido (Persistent_Signals) y se utilizó junto a una característica especial de Ada llamada ATC[4] (Asynchronous Task Control). A continuación podemos ver un extracto del código:

```
IRQ_Sample   : Pcm3718.Analog_Data_Type;
pragma Atomic(IRQ_Sample);
Stop_Command : Persistent_Signals.Persistent_Signal;

function Irq_Handler:
  Fetch_Sample (IRQ_Sample);
  Outb_P(Pcm3718_Base + Pcm3718_Status, 1);
```

[4]Ver en el anexo A una nota sobre los problemas de configuración de GAP que pueden provocar que el ATC no funcione correctamente y que me provocaron bastantes quebraderos de cabeza durante algunos dias.

```
task body Bottom_Half:
...
loop
  accept Start;
  PHI.Associate (Pcm3718.Irq, Irq_Handler'Access ...
  select
    Stop_Command.Wait;
  then abort
    PHI.Unlock (Pcm3718.Irq ...
    loop
      PHI.Timedwait(0,null,Intr'access, Handler'access ...
      Sample := IRQ_Sample;
      Pcm3718_Buffer.Fifo.Write(Sample);
    end loop;
  end select;
  PHI.Lock...
  PHI.Disassociate (Pcm3718.Irq,Irq_Handler'Access)...
end loop;
```

En la figura 4.6 se muestra esta arquitectura.

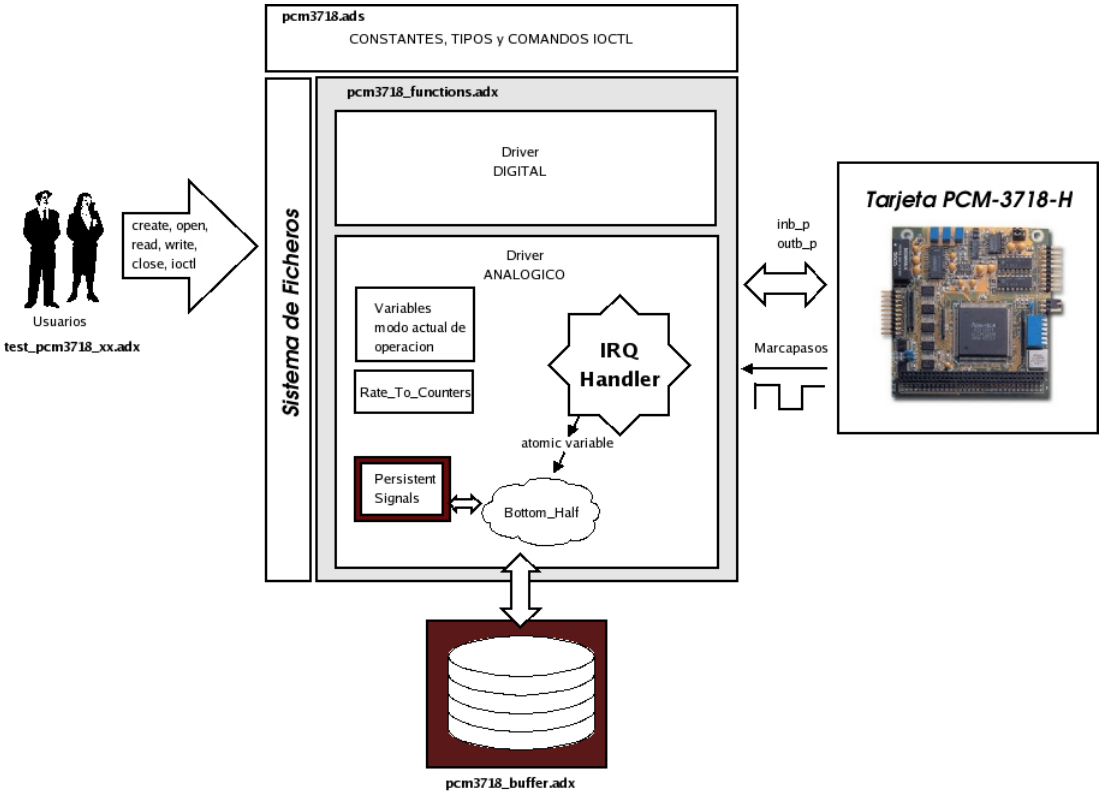

Figura 4.6: Arquitectura usando Bottom Halves

La idea es la siguiente. Al inicio la tarea Bottom_Half, se queda esperando en su entry Start. Cuando la aplicación escoge el modo **Pacer** o **External** mediante la correspondiente

llamada IOCTL, se activa la bottom half llamando a Bottom_Half.Start, que lo que hace es asociarse al nivel de interrupción y esperar a que se produzca una interrupción mediante la llamada Timedwait. Cuando desde la aplicación se activen las interrupciones y se produzca una, se ejecutará Irq_Handler a la máxima prioridad, que cogerá la muestra, la guardará en una variable atómica[5] IRQ_Sample y terminará. En ese momento la tarea Bottom_Half estará lista para ejecutarse si es la de mayor prioridad en ese momento. Lo que hará será guardar la muestra en el *buffer* y volverá a llamar a Timedwait para esperar la siguiente interrupción. Cuando la aplicación quiera cambiar a modo software, donde no se usan las interrupciones, le mandará una señal de Stop, Stop_Command.Send;, con lo que la tarea volverá a la posición inicial.

Desgraciadamente, surgieron dos problemas con este diseño:

1. Debido a que la CPU no es muy rápida, 133Mhz, no se puede utilizar la tarjeta A/D a ritmos excesivamente altos pues se pierden muchas muestras durante la rutina de atención a la interrupción. Con este diseño las muestras se perdían cuando bajabamos de 200 us, lo cual se consideró demasiado. Gracias al nuevo diseño se consiguió que funcionase sin pérdida de muestras a partir de unos 38-40 us, lo cual estaba bastante bien.

2. El otro problema era todavía más grave. Al realizar los tests iniciales desde la interfaz C, el driver se comportaba perfectamente. Sin embargo, cuando llegó el momento de poner prioridades, se quedaba completamente colgado. Esto se debía a que cuando la aplicación principal estaba en C, aunque le pusieramos una prioridad y aunque el código del driver estuviera en Ada, al final se trataba de un Thread accediendo a un objeto protegido (recordemos que en la función Read se accede al *buffer* y en las IOCTL se accede a Persistent_Signals). El entorno sobre el que corre Ada solo puede tener en cuenta las tareas que se crean desde ella y no los threads que se han creado por el sistema operativo. Es como si estuvieran en niveles distintos. Para solventarlo había que cambiar los objetos protegidos por paquetes con **Mútexes** y **Variables Condicionales** o bien hacer un truco que consistía en que la aplicación inicial esté en Ada y que desde ella se cree un thread que contenga la verdadera aplicación, esta ya en C. Sin embargo, dado que esto era una chapucilla y además no se conseguían ritmos adecuados, se decidió finalmente rediseñar el driver por completo.

[5] Una variable atómica es aquella que se modifica de un golpe, sin ser interrumpido en mitad.

Capítulo 5

Subsistema de gestión del Bus Serie I2C

En este capítulo estudiaremos el diseño e implementación de un subsistema software que se encargará de ofrecer una interfaz sencilla para el acceso a dispositivos que usen el Bus Serie I2C.

5.1. HARDWARE

Nuestro sistema empotrado basado en PC ha adquirido nuevas posibilidades de interacción con el exterior gracias a la tarjeta PCM-3718-H. Sin embargo, si nos fijamos en las posibilidades de otras arquitecturas más típicas en sistemas empotrados, podemos pensar que un PC (a pesar de la ventaja de ser veteranos y por tanto fiables) no es en absoluto la mejor vía ni mucho menos la más barata para realizarlo. Por ejemplo, algunos microcontroladores de Microchip de gama alta incluyen dentro del propio procesador todo tipo de puertos, módulos A/D integrados, módulos PWM, etc... Necesitamos pues una forma de poder expandir los recursos de nuestro sistema empotrado y poderle conectar todo tipo de periféricos de una forma sencilla y barata. Una posible forma es mediante un *bus* tan sencillo como potente, el Bus Serie I2C.

5.1.1. El bus serie I2C

El *bus* I2C (Inter-IC) fue desarrollado por Philips (ver manual en [20]) a principios de los 80 para resolver la conexión entre una CPU y varios periféricos en equipos de electrónica de consumo. Se trataba de simplificar las conexiones entre periféricos (reduciendo el número de pistas, decodificadores, etc) y al mismo tiempo aumentar la inmunidad al ruido en sistemas de audio y vídeo. La iniciativa de Philips ha sido seguida por muchos fabricantes, por lo que la variedad de periféricos existentes en la actualidad es muy amplia: memorias RAM y E2PROM, microcontroladores, pantallas LCD y LED, puertos de E/S, codificadores DTMF, tranceptores IR, conversores A/D y D/A, relojes de tiempo real, monitorizadores hardware de temperatura, etc.

Nivel físico

El *bus* I2C está formado por 3 cables [21]:

- SDA (System Data): por la cual viajan los datos entre los dispositivos.
- SCL (System Clock): por la cual transitan los pulsos de reloj que sincronizan el sistema.
- GND (Masa): Interconectada entre todos los dispositivos *enganchados* al *bus*.

Las líneas SDA y SCL son bidireccionales y en el caso del sentido de salida son salidas en colector (o drenador) abierto. Esto permite hacer la AND cableada de las señales de los distintos dispositivos. Se deben poner en estado alto (conectar a la alimentación por medio de resistencias "*Pull-Up*") para construir una estructura de *bus* tal que se permita conectar en paralelo múltiples entradas y salidas. Las dos líneas disponen de niveles lógicos altos

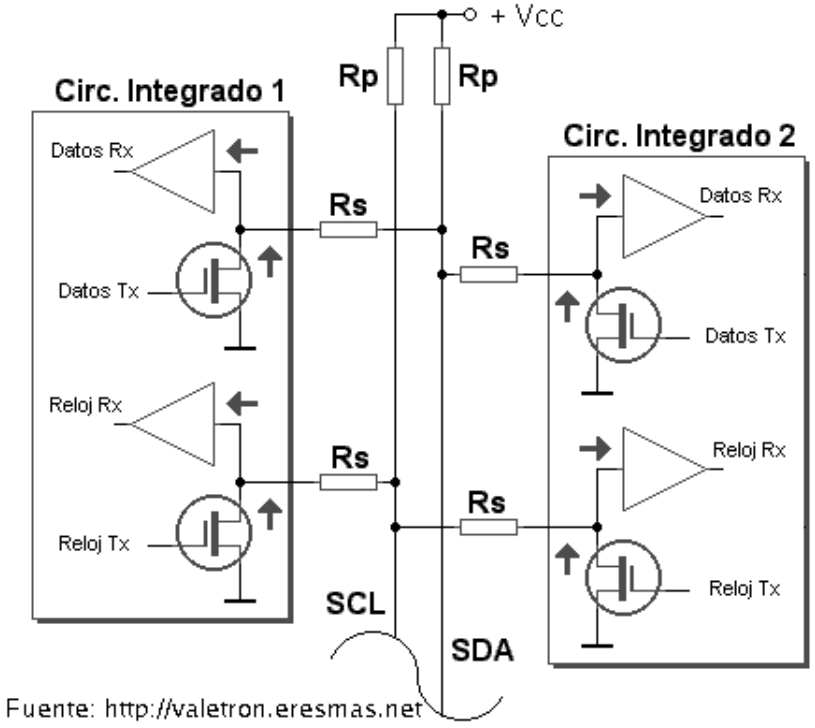

Figura 5.1: Nivel físico del *Bus* I2C

cuando están inactivas. Además, es fácil determinar si otro elemento está ocupando el *bus* ya que si se intenta fijar un nivel alto y sigue bajo se debe a que otro elemento que ocupa el *bus* está fijando ese nivel.

El número de dispositivos que se puede conectar al *bus* es teóricamente ilimitado, pero obsérvese que las líneas tienen una especificación máxima de 400pF en lo que respecta a capacidad de carga. La velocidad de transmisión estándar es de 100Kbps aunque hoy día se alcanzan velocidades superiores.

Se pueden utilizar puertos digitales TTL, como por ejemplo el puerto paralelo, para emular el bus I2C. Para ello se necesita una pequeña circuitería adicional.

Nivel de enlace: el protocolo I2C

La comunicación entre dispositivos mediante las líneas SDA y SCL, se realiza mediante un protocolo maestro-esclavo (ver [21] y [20]). Es decir, hay un dispositivo maestro, que generalmente es el microcontrolador o microprocesador, que controla la comunicación diciendo qué dispositivo quiere leer o escribir. El dispositivo maestro es el encargado de generar la señal de reloj (SCL), es decir un conjunto de pulsos cuadrados que, cuando pase de ’0’ a ’1’ indicará la validez del valor del bit en la línea de datos (SDA). Por ejemplo, yo soy el maestro y pongo un ’0’ en la línea SDA, después subo el reloj SCL a ’1’, el esclavo entiende que le he enviado un ’0’, y así sucesívamente. Es importante darse cuenta de que no es imprescindible que los pulsos de reloj sean periódicos perfectos, el esclavo los seguirá de igual manera. Resumiendo, **el protocolo I2C se basa en enviar y recibir bits por la línea SDA al ritmo que el maestro marca con la línea SCL**. Y se especifican tres formas de enviar/recibir esos bits:

1. *Maestro envía datos a un esclavo*

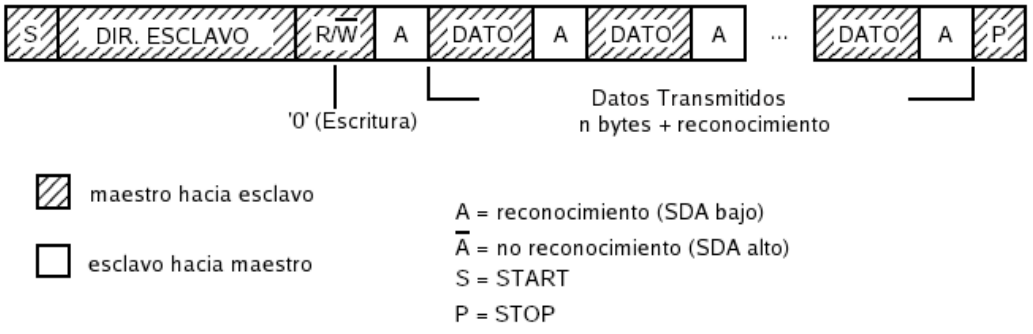

Figura 5.2: Maestro envía datos a un esclavo

2. *Maestro recibe datos de un esclavo*

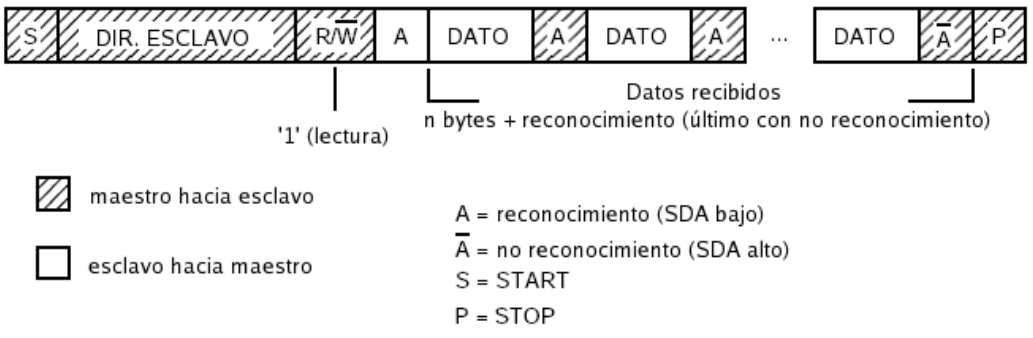

Figura 5.3: Maestro recibe datos de un esclavo

3. *Maestro escribe y recibe datos de un esclavo*

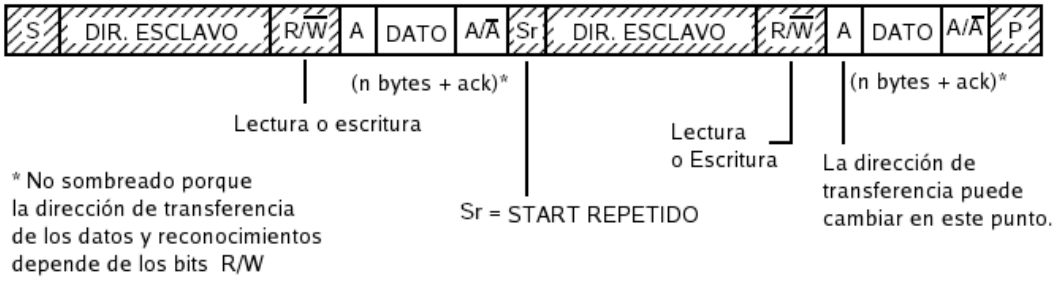

Figura 5.4: Maestro escribe y recibe datos de un esclavo

- **Bit de START (S) y RESTART (Sr)**: es una transición de ’1’ a ’0’ en la línea de datos (SDA) mientras la línea del reloj (SCL) esta a ’1’. Es decir, tenemos el *bus* libre, SDA=1 y SCL=1, bajamos SDA=0 y los esclavos deben saber que se va a iniciar una transferencia. El bit Sr permite realizar varias transferencias sin perder el *bus*.
- **Dirección del Esclavo**: los esclavos saben que se va a iniciar una transferencia por el bit de Start pero no con qué esclavo. Así que cada esclavo tiene una dirección de fábrica de 7 bits (con los que podemos direccionar hasta 112 dispositivos pues hay direcciones reservadas. Recientemente se añadió el uso de direcciones de 10 bits). Tras el Start, el maestro escribe esa dirección para seleccionar al esclavo.
- **Bit de lectura/escritura (R/W)**: el maestro añade un octavo bit para indicar si quiere leer (’1’) o escribir (’0’) en el dispositivo esclavo.
- **Bit de reconocimiento (ACK)**: consiste en poner la línea SDA a valor ’0’. El que ha de recibir el reconocimiento debe desbloquear la línea SDA poniéndola a valor ’1’ (para que con la AND cableada el resultado sea ’0’).
- **DATO**: los datos se transmiten por bytes seguidos por reconocimientos. No hay un límite de bytes a enviar. El byte de datos se transfiere empezando por el bit de más peso (7). Cuando el maestro lee los datos del esclavo el último byte lo contesta con un no-reconocimiento, en lugar de un reconocimiento, indicando que ya no desea recibir más bytes. Si un dispositivo esclavo no puede recibir o transmitir un byte de datos completo hasta que haya acabado alguno de los trabajos que realiza, puede mantener la línea SCL a ’0’ lo que fuerza al maestro a permanecer en un estado de espera. Los datos continúan transfiriéndose cuando el dispositivo esclavo está listo para otro byte de datos y desbloquea la línea de reloj SCL. Esto es especialmente útil, por ejemplo, en dispositivos basados en microcontrolador que pueden estar ocupados en otras tareas.
- **Bit de STOP (P)**: una transición de ’0’ a ’1’ en la línea de datos (SDA) mientras la línea de reloj (SCL) está a ’1’.

Las condiciones de Start y Stop son siempre generadas por el maestro. El *Bus* I2C se considera ocupado después de la condición de Start. El *bus* se considera libre de nuevo después de un cierto tiempo tras la condición de Stop. Puede que haya varios maestros en un mismo *bus*. En ese caso, deben esperarse a que el *bus* esté libre. Existe un sistema de arbitraje para el caso en que varios maestros intentan poner información en el *bus* a la vez.

5.1.2. Soporte de I2C en la Placa PCM-3347F

Conseguir utilizar el *Bus* I2C en nuestra placa base fue una pequeña odisea. Aún así la intentaré resumir pues creo que puede servir como muestra del trabajo sucio que a veces es necesario hacer para realizar un driver. Especialmente debido a que los manuales suelen esconder muy bien la información que realmente nos interesa.

El primer lugar en el que miro es, lógicamente, en la hoja de especificaciones de la placa. En esta hoja, se describen todos los puertos, *buses*, esquemas internos, etc, de la placa y sin embargo no aparece ningún *Bus* I2C. El siguiente paso consiste en mirar el manual de usuario de la placa (`http://www.advantech.com`) donde obtengo una primera **pista**: un conector con el nombre I2C BUS. Con lo cual ya sabemos que sí que existe; cómo se usa es otro asunto. Desgraciadamente, no pone nada más. Tampoco pone nada en la web del fabricante. Así que escribo al fabricante Advantech, que tiene una sección de soporte. Contestan con un correo estándar diciendo que para los productos estándar es mejor consultar a nuestro distribuidor (un poco absurdo). Escribo al distribuidor. Me contesta de una forma mucho más personalizada diciendo que tampoco ha podido encontrar la información ni en el manual, ni en la zona de *partners* de la web. Así que, dice, han enviado una consulta directamente al fabricante (de locos).

Como último remedio me pongo a investigar los chips de la placa a ojo, buscando extraños números en internet. Busco algún microchip encargado de manejar el *Bus* I2C, como los módulos MSSP (puerto serie síncrono maestro) de los PIC's. Tampoco encuentro solución de esta forma.

Voy a tirar la toalla cuando se me ocurre que quizá el controlador I2C esté integrado en la propia CPU. La CPU de la placa es una STPC ELITE 133 de la empresa STMicroelectronics (`http://www.st.com`). Descargo su datasheet y bingo: **I2C INTERFACE**. Me estoy acercando. Sigo buscando en el datasheet y me encuentro lo siguiente:

> SCL, SDA Interfaz I2C: estos pins bidireccionales están conectados al registro 22h/23h índice 97h. Son compatibles con las especificaciones eléctricas I2C, tienen salida en colector abierto y están conectados internamente a Vdd a través de sendos resistencias de "*pull-up*".
> SCL / DDC[1] Entrada
> SDA / DDC[0] Entrada

Como no sé muy bien qué significa eso de un registro doble con un índice, me descargo el manual del programador de su página Web. Busco I2C en el manual pero no aparece por ningún lado. No pasa nada, busco las letras DDC que me aparecían en el datasheet y obtengo algunos resultados.

Registro de control DDC

1. DDCR Access = 022h/023h Regoffset = 0x95h
2. Bits 7-6 DDCWD (DDC Write Data): Estos dos bits están conectados a salidas de colector abierto. Escribir en estos bits afecta a los pins DDC[1:0]. Los pins DDC[1:0] son salidas en colector abierto que están conectados a resistencias de pull-up. Así, programar cualquiera de estos bits a '1' deshabilita el driver de salida y permite que el pin actúe como entrada cuyo estado puede ser leído vía los bits 5-4 de este registro. Notar que las lecturas desde estos bits retornan el valor del último dato escrito en el registro, lo cual no tiene porqué ser igual al dato que hay realmente en el *bus*. Los bits 5-4 de este registro reflejan de forma precisa los datos en el *bus* sin importar quién lo está utilizando.

Dado que yo busco un conjunto de registros para interactuar con alguna controladora del *Bus* Serie I2C, no acabo de entender como podría yo usar este registro. Así que me vuelvo a quedar atascado.

La placa PCM-3347 podía adquirirse también con otro procesador de ST, el STPC-Consumer II. Al buscar el registro DDC en su manual descubro que sus siglas significaban Display Data Channel, y forma parte de la controladora VGA (alucino). Pero además, y esto es lo importante, acerca de los bits 7-6 pone lo siguiente:

> Pueden ser utilizados en su lugar para acceder a dispositivos I2C conectados. DDC1 y DDC0 corresponden a SCL y SDA respectivamente.

Me doy cuenta, por fin, de que la placa ofrece un *bus* físico I2C, pero **no una controladora** para manejarlo. En su lugar, lo que permite controlar son los valores de las dos líneas SDA y SCL escribiendo en 2 bits de un registro. Es decir, que tendré que ser yo quien emule el protocolo I2C.

La forma de escribir/leer en los registros viene explicada (algo escondida) en el manual. La primera dirección (022h) es el lugar donde se escribe un offset (95h), y la segunda dirección (023h) es donde leemos o escribimos el registro. Así que se podía decir que ya he descubierto, al fin, la manera de manejar las líneas I2C. Sin embargo, al hacer pruebas y medir con el osciloscopio, las líneas no cambian de nivel.

Casi renuncio. Estoy seguro de hacerlo bien y sin embargo, las líneas no cambian de nivel. Al cabo de un par de días me doy cuenta de que en el datasheet el offset era 97h, a diferencia de lo que aparecía en el manual del programador, donde indicaba que era 95h. Es decir, había una preciosa **errata** en el manual. Efectivamente, cambio el offset y las líneas se mueven (aleluya). Otra errata, ésta no viene bien en ninguna parte: los bits que se usan son distintos de los que se indica en el manual. Hago unas pruebas hasta deducir el funcionamiento, lo apunto y pienso: ¡lo conseguí!.

5.2. SOFTWARE

El subsistema se encargará de gestionar los *Buses* I2C que haya en el sistema. No se trata de un driver, sino de un software intermedio entre los drivers y los dispositivos I2C. Por tanto, tendrá una interfaz propia que nosotros crearemos, a través de la cual, los drivers escribirán y leerán los dispositivos I2C. En el *kernel* Linux este subsistema equivale al contenido en el directorio **`drivers/i2c`** y **`include/linux/i2c*.h`**. Tal como está implementado, lleno de punteros *void, resulta complicado adaptarlo para que sea utilizable desde Ada. Además, se usan *Mútexes* para el acceso al *bus*, mientras que nosotros utilizaremos un modelo no necesariamente bloqueante.

5.2.1. Análisis de especificaciones

El subsistema presentará una interfaz basada en comandos. Las tareas podrán mandar comandos de los tres tipos de transmisiones del protocolo I2C (ver sección 5.1.1). Éstos se irán guardando y ejecutándo en un orden de prioridad por una tarea interna (a partir de ahora *demonio*[1]). Una vez enviado el comando, la tarea podrá dedicarse a otros cálculos y chequear la operación cada cierto tiempo, o bien, bloquearse hasta que finalice.

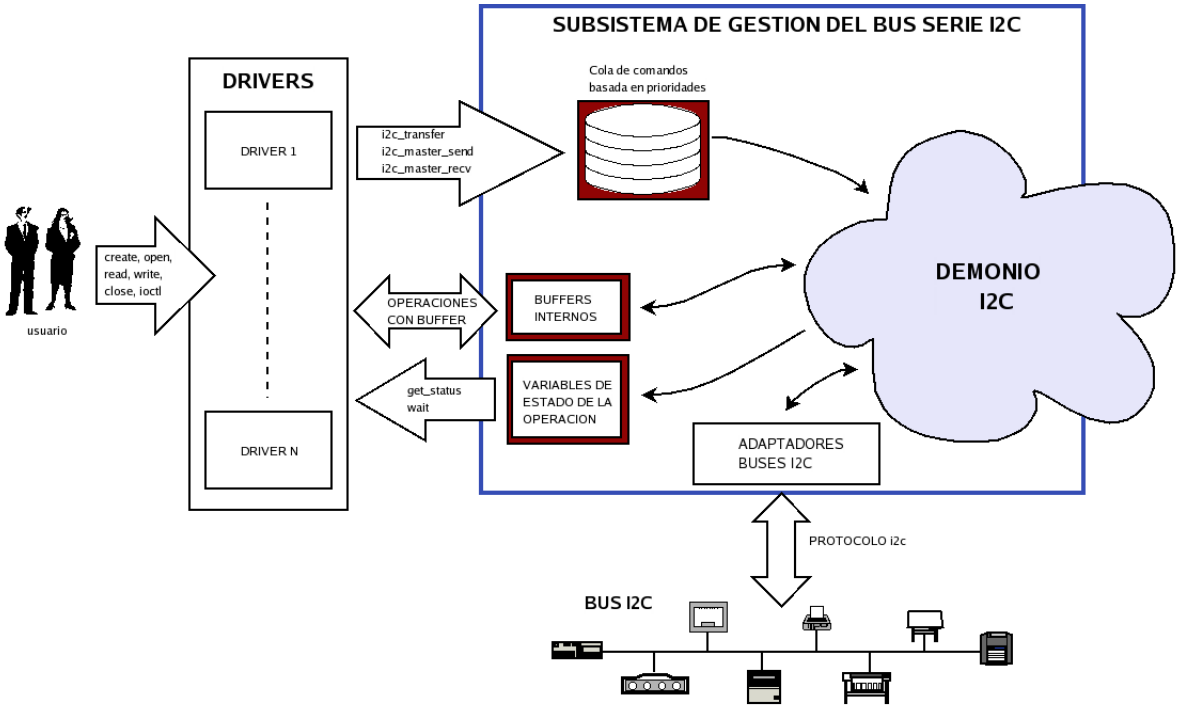

Figura 5.5: Esquema del subsistema I2C

Deberá haber una interfaz tanto para Ada como para C. Con una función general para el envío de varios comandos y dos funciones más simples para los casos de envío o recepción de bytes. El esquema de funcionamiento se muestra en la figura 5.5.

[1]Programa que permanece en segundo plano ejecutándose continuamente para dar algún tipo de servicio

5.2.2. Diseño de la arquitectura

Dividiremos nuestro código en varios módulos que nos ayuden tanto a descomponer el problema en partes más manejables, como a ofrecer una interfaz clara al usuario ocultando la información que no necesita. Se ha intentado aprovechar las características de alto nivel que ofrece Ada para realizar una arquitectura estructurada y fácilmente legible (al contrario, por ejemplo, de la arquitectura usada en Linux).

La facilidad de usar paquetes hijos, nos permite crear una estructura jerárquica donde en lo más alto se encuentren los tipos usados por todo el subsistema y en la parte más baja el código más específco como, por ejemplo, las interfaces a las aplicaciones en C y Ada o la interfaz hardware al *bus*. Además, se han utilizado objetos y polimorfismo para poder representar los distintos *Buses* I2C que puede haber en el sistema empotrado. Se ha utilizado también una tarea y la parte de inicialización de los paquetes Ada. No se han utilizado objetos protegidos pues no se podrían utilizar desde threads C, así que se han usado **Mútexes** y **Variables condicionales** en su lugar.

Los archivos relacionados con este driver son:

1. **`i2c.ads`**
2. **`i2c-adapters.ad[s,b]`**
3. **`i2c-adapters-bit.ad[s,b]`**
4. **`i2c-adapters_table.ads`**
5. **`priority_queues.ad[s,b]`**
6. **`i2c-daemon.ad[s,b]`**
7. **`i2c-daemon-ada_interface.ad[s,b]`**
8. **`i2c-daemon-c_interface.ad[s,b]`**
9. **`i2c.h`**
10. **`i2c_c_interface.c`**
11. **`i2c_elite.ads`**
12. **`i2c_elite.h`**
13. **`i2c_elite_c.c`**
14. **`i2c_parport.ad[s,b]`**
15. **`i2c_pcm3718.ad[s,b]`**
16. **`test_i2c.adb`**
17. **`test_i2c_c.c`**

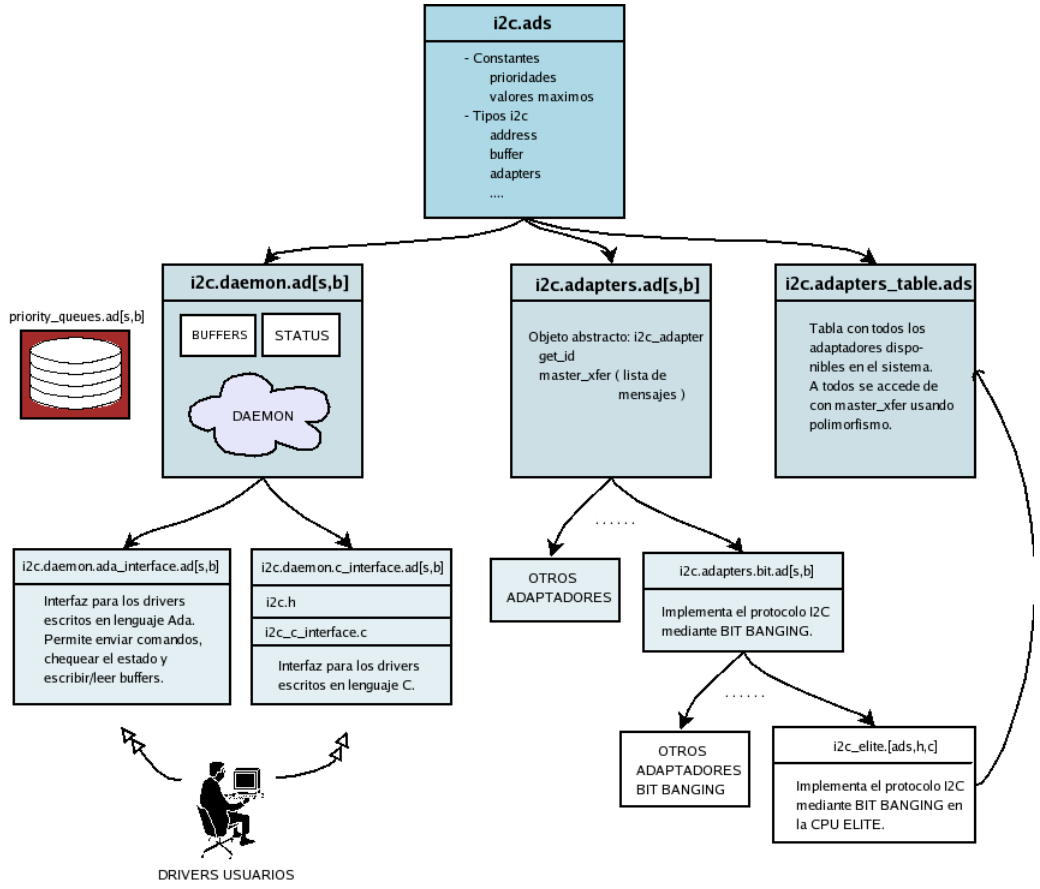

Figura 5.6: Esquema de la arquitectura del subsistema I2C

Las relaciones entre los módulos principales se muestran en la figura 5.6. Comencemos viendo las funciones interfaz de los drivers usuarios:

```
procedure I2C_Transfer ( Adap     : in I2C_Adapter_ID;
                         Op       : in I2C_Operation_ID;
                         Prio     : in I2C_Priority;
                         Msg_list : in I2C_Msg_Ada_List);

procedure I2C_Master_Send ( Adap   : in I2C_Adapter_ID;
                            Op     : in I2C_Operation_ID;
                            Prio   : in I2C_Priority;
                            Addr   : in I2C_Address;
                            Buffer : in I2C_Buffer;
                            Flags  : in I2C_Flags);

procedure I2C_Master_Recv ( Adap   : in I2C_Adapter_ID;
                            Op     : in I2C_Operation_ID;
                            Prio   : in I2C_Priority;
                            Addr   : in I2C_Address;
                            Buffer : in I2C_Buffer;
                            Flags  : in I2C_Flags);
```

`I2C_Transfer` permite hacer cualquier tipo de transferencia del protocolo I2C. Las otras dos funciones sirven para enviar o para recibir bytes y tienen la ventaja de ser más simples. El significado de cada parámetro es el siguiente:

- **`I2C_Adapter_ID`**: representa un *bus* I2C físico presente en el sistema. Los adaptadores, serán añadidos estáticamente a una tabla y el usuario deberá poder escoger por cual realizar la transferencia (obviamente por el *bus* en el que esté el dispositivo).
- **`I2C_Operation_ID`**: representa una operación de transferencia. Nos permitirá chequear el estado de la transferencia o quedarnos bloqueados hasta que termine mediante la siguiente interfaz:

  ```
  function Get_Status (Op : in I2C_Operation_ID)
                      return I2C_Operation_Status;

  function Wait (Op : in I2C_Operation_ID)
                      return I2C_Operation_Status;

  procedure Read_Buffer (Data   : out I2C_Data;
                         Buffer : in I2C_Buffer;
                         Op     : in I2C_Operation_ID;
                         Stat   : out I2C_Operation_Status);
  ```

  El tipo **`I2C_Operation_Status`** representa el estado de la operación:

  1. **`NOT_IN_USE`**: el **`I2C_Operation_ID`** no representa a ninguna operación enviada al subsistema.
  2. **`WORKDONE`**: la operación indicada por **`I2C_Operation_ID`** ha sido completada con éxito.
  3. **`WAITING`**: la operación indicada por **`I2C_Operation_ID`** está siendo ejecutada por el subsistema.
  4. **`I2C_ERROR`**: la operación indicada por **`I2C_Operation_ID`** ha sufrido un error interno (bien físico o bien porque los datos facilitados no son correctos).

- **`I2C_Priority`**: indica la prioridad de la operación. Ésta determinará, cuando varias tareas envíen comandos al subsistema, el orden en el que se deben ejecutar.
- **`I2C_Address`**: representa la dirección del dispositivo esclavo al que se accede.
- **`I2C_Buffer`**: los bytes se envían o reciben a través de buffers internos. Para leer el buffer ya hemos visto la función **`Read_Buffer`**. Para crearlo y liberarlo la interfaz es la siguiente:

  ```
  function Create_Recv_Buff (Num_Bytes : in I2C_Data_Count)
                             return I2C_Buffer;

  function Create_Send_Buff (Data : in I2C_Data) return I2C_Buffer;

  procedure Free (Buffer : in I2C_Buffer);
  ```

El tipo **`I2C_Data`** es un array de bytes y **`I2C_Data_Count`** es el índice de ese array.

- **`I2C_Flags`**: esta variable se incluye para poder soportar variaciones del protocolo, malfuncionamientos de algunos dispositivos, etc... Se basa en la misma variable del subsistema I2C de Linux. También dispone de una interfaz con algunas de estas características (aunque nosotros no las implementaremos pues no disponemos de dispositivos reales para probarlos):

```
procedure Flags_Init (Flags : in out I2C_Flags);

procedure Flags_Set_TenBitAddress (Flags : in out I2C_Flags);

procedure Flags_Set_NoStart (Flags : in out I2C_Flags);

procedure Flags_Set_RevDirAddr (Flags : in out I2C_Flags);

procedure Flags_Set_IgnoreNack (Flags : in out I2C_Flags);

procedure Flags_Set_NoRdAck (Flags : in out I2C_Flags);
```

- **`I2C_Msg_Ada_List`**: se utiliza para pedir al *demonio* que realice un conjunto de operaciones con bits de RESTART en medio. Es decir, corresponde al método de transferencia combinado visto en 5.1.1. La interfaz es la siguiente en el caso de Ada:

```
subtype I2C_Msg_Ada is I2C_Msg;

type I2C_Msg_Ada_List is private;

procedure Create_Msg ( Msg    : in out I2C_Msg_Ada;
                       Addr   : in I2C_Address;
                       Buffer : in I2C_Buffer;
                       Flags  : in I2C_Flags);

procedure Create_Msg_List ( Msg_Max  : in Positive;
                            Msg_List : in out I2C_Msg_Ada_List);

procedure Add ( Msg      : in I2C_Msg_Ada;
                Msg_List : in out I2C_Msg_Ada_List);

procedure Empty ( Msg_List : in out I2C_Msg_Ada_List);
```

En el caso de C la interfaz es similar pero la lista de mensajes se representa como un puntero a un mensaje:

```
typedef struct {
  i2c_address addr;
  i2c_flags flags;
  i2c_buffer buffer;
} i2c_c_msg;
```

Además existen una serie de constantes en **`i2c.ads`** que podrán ser modificadas:

- **`I2C_Mx_Prio  : constant := 256;`**: sirve para indicar cuántos niveles de prioridad pueden existir. Estas prioridades son usadas por el *demonio* I2C para servir los comandos en un orden. Ver que no son lo mismo que las prioridades de las tareas.
- **`I2C_Mx_Number_Of_Buffers : constant := 20;`**: establece el número máximo de buffers internos posibles. Si un usuario se olvida de liberar los buffers, se descubrirá rápido el olvido pues llegará un momento en que no le queden más buffers para usar.
- Prioridades internas: establece las prioridades del *demonio* I2C y de las estructuras de datos protegidas. Los usuarios deberán asegurarse de que su prioridad y la del *demonio* sean menores o iguales a las prioridades de las estructuras de datos protegidas.

```
I2C_Daemon_Prio : constant := 10;
I2C_Operation_List_Prio : constant := 11;
I2C_Commands_List_Prio : constant := 11;
I2C_Buffer_List_Prio : constant := 11;
```

- **`I2C_Operation_ID_Mx : constant := 10;`**: número máximo de identificadores de operación. Normalmente cada usuario del sistema solamente necesitará usar un identificador para sus operaciones.

5.2.3. Diseño detallado y codificación

- **Los Adaptadores:** los adaptadores son la parte del subsistema encargada de realizar la transferencia a los dispositivos siguiendo el protocolo I2C. Existen distintos tipos de chips controladores del *Bus* I2C, para cada uno se necesitará implementar un adaptador. Afortunadamente la mayoría de los chips se pueden agrupar según algoritmos para manejarlos (esta información está extraída del propio *kernel* Linux). Por ejemplo, hay adaptadores, como es el nuestro, que se basan en emular el protocolo subiendo y bajando las líneas SCL y SDA (*Bit Banging*). Podemos ahorrarnos mucho código si separamos el algoritmo de la parte hardware.

  1. **`i2c.adapters.ad[s,b]`**: en este módulo se define un adaptador como una clase abstracta con una función principal que sirve para comunicar la transferencia I2C a realizar. Esta función se declara abstracta y será implementada por cada adaptador en particular. Además se provee un tipo que apunta a toda la clase. De esta manera, el *demonio* podrá utilizar los adaptadores de forma polimórfica.

```
type I2C_Adapter is abstract tagged record ...
type I2C_Adapter_Ref is access all I2C_Adapter'Class;
function Master_Xfer (Adap       : I2C_Adapter;
                      Msg_List   : I2C_Msg_List)
                      return Int is abstract;
```

A la función **`Master_Xfer`** se le pasa una lista de mensajes como parámetro. Cada mensaje indica un envío o recepción de bytes al dispositivo. Si hay varios mensajes, entre cada envío se incluirá un bit de ReStart tal como se indica en el protocolo I2C.

2. **`i2c.adapters.bit.ad[s,b]`**: en este módulo se implementa la parte algorítmica de los adaptadores conocidos como BIT BANGING. Estos adaptadores se basan en emular el protocolo I2C controlando directamente las líneas del *bus*, SDA y SCL. Para ello en primer lugar creamos una clase hija, que tome como datos los procedimientos para controlar las líneas del *Bus* I2C. Estos procedimientos serán la parte Hardware del adaptador, es decir, donde se encuentren las famosas instrucciones **`inb_p`** y **`outb_p`**:

```
type I2C_Adapter_BIT is new I2C_Adapter with record
   Wait_Semiperiod : Wait_Semiperiod_Ac := Null;
   Set_SDA_High : Set_I2C_Lines_Ac := Null;
   Set_SDA_Low  : Set_I2C_Lines_Ac := Null;
   Set_SCL_High : Set_I2C_Lines_Ac := Null;
   Set_SCL_Low  : Set_I2C_Lines_Ac := Null;
   Get_SDA      : Get_I2C_Lines_Ac := Null;
   Get_SCL      : Get_I2C_Lines_Ac := Null;
end record;
```

La función **`Wait_Semiperiod`** es la que delimita el tiempo de medio ciclo de reloj. Se ofrece una función por defecto, **`Wait_Default_Semiperiod`**. Esto nos permite tener varios *buses* cada uno a una frecuencia de reloj diferente con lo cual podremos tener dispositivos rápidos en un *bus* y lentos en otro. Recordemos que existen dispositivos que pueden alcanzar velocidades de varios Mbit/s.

A continuación creamos unas funciones auxiliares que se encargan de cada fase del protocolo. Estas funciones utilizan los procedimientos Hardware vistos:

```
procedure Start (Adap : in I2C_Adapter_BIT);

procedure ReStart (Adap : in I2C_Adapter_BIT);

procedure Stop (Adap : in I2C_Adapter_BIT);

procedure Write (Adap : in I2C_Adapter_BIT;
                 Byte : in Unsigned_8);

procedure Read (Adap : in I2C_Adapter_BIT;
                Byte : out Unsigned_8);

function Get_Ack (Adap : in I2C_Adapter_BIT) return Boolean;

procedure Set_Ack (Adap : in I2C_Adapter_BIT);

procedure Set_Nack (Adap : in I2C_Adapter_BIT);
```

Finalmente implementamos la función **`Master_Xfer`** que consistirá en ir leyendo los mensajes que se le han pasado y emular el protocolo I2C utilizando las funciones anteriores.

El algoritmo en pseudocódigo es el siguiente:

```
Start
for i in Msg_List'Range loop
    if "Mensaje de lectura" then
      Escribir direccion del esclavo con bit de lectura
      Ack := Get_Ack()
      for j in "N-1 bytes" loop
          Leer Byte
          Mandar reconocimiento
      end loop;
      Leer ultimo byte
      Mandar no reconocimiento
      Guardar datos en el buffer
    else "Mensaje de escritura"
      Escribir direccion del esclavo con bit de escritura
      Ack := Get_Ack()
      for j in "N bytes" loop
          Escribir Byte
          Ack := Get_Ack()
      end loop;
    end if;
    if "hemos terminado" then
      Stop
    else
      ReStart
    end if;
end loop;
```

Como vemos es bastante sencillo y se basa en el protocolo descrito en las figuras 5.2, 5.3 y 5.4.

3. **`i2c_elite.[ads,c,h]`**: este es un ejemplo de la parte Hardware de un adaptador del tipo BIT BANGING. Para realizarlo, sólo tenemos que saber subir y bajar las líneas, y como vimos en la sección 5.1.2, sabemos hacerlo. En este caso hicimos las funciones en C, aunque también se han hecho otros dos adaptadores (usando el puerto paralelo y usando la entrada/salida digital de la tarjeta PCM-3718-H) utilizando Ada. En C, al igual que cuando instalábamos drivers, es necesario realizar una importación de las funciones y convertirlas usando el paquete **`Ada.Unchecked_Conversion`**, para luego poder añadirlas a los campos del objeto instanciado de la clase **`I2C_Adapter_BIT`**.

4. **`i2c.adapters_table.ads`**: al igual que la tabla de drivers, esta tabla permite almacenar de forma estática, en RAM, un directorio con los adaptadores I2C disponibles. Para ello deberemos *instalar* nuestro adaptador, que significa darle un nombre identificativo en el tipo **`I2C_Adapter_ID`** dentro del fichero **`i2c.ads`**, por ejemplo '**Elite**', y agregarlo al array así:

```
The_I2C_Adapter_List : I2C_Adapter_List :=
  (Elite   => I2C_Elite.I2C_Adapter_BIT_Elite'Access,
   Parport => I2C_Parport.I2C_Adapter_BIT_Parport'Access,
   Pcm3718 => I2C_Pcm3718.I2C_Adapter_BIT_Pcm3718'Access);
```

- **El *Demonio* y sus estructuras:** El *demonio* es una tarea interna que se encarga de recoger los comandos enviados por los drivers y ejecutarlos en un orden de prioridad. Su pseudocódigo es muy fácil de seguir:

```
Dequeue(Cmd, Prio, The_Commands_Queue);
Adap := The_I2C_Adapter_List(Cmd.Adap_ID);
if (I2C.Adapters.Master_Xfer (Adap.all,Cmd.Msg_List) = -1) then
  Set_Status (Cmd.Operation_ID, I2C_ERROR);
else
  Set_Status (Cmd.Operation_ID, WORKDONE);
end if;
```

  Básicamente lo que hace es desencolar el siguiente comando (si la cola está vacía se dormirá hasta que alguien envíe un nuevo comando), ordenar la transferencia al adaptador adecuado y cuando ésta termine, establecer si se realizó con éxito u ocurrió algún error. Las principales estructuras de datos son:

  1. ***La cola de comandos***: se trata de una cola de prioridad cuyos elementos son de la siguiente forma (se utilizan punteros para que sea más eficiente):

```
type I2C_Command is record
   Adap_ID       : I2C_Adapter_ID;
   Operation_ID  : I2C_Operation_ID;
   Msg_List      : I2C_Msg_List;
end record;
type I2C_Command_Ref is access I2C_Command;
```

     La cola es una modificación de las colas de prioridad del software MAST [22], se encuentra implementada en **`priority_queues.ad[s,b]`** como un paquete genérico y tiene la siguiente interfaz:

```
procedure Init(Prio : in     Kernel.Mutexes.Ceiling_Priority;
               Q    : in out Queue);

function Empty (Q : Queue) return Boolean;

procedure Enqueue(E : in Element;
                  P : in Priority;
                  Q : in out Queue);

procedure Dequeue(E : out Element;
                  P : out Priority;
                  Q : in out Queue);
```

     Como esta cola es accedida concurrentemente por las tareas usuarias y por el *demonio*, es necesario protegerla utilizando un **Mútex**. Además, se utiliza una Variable condicional en la función **`Dequeue`**, para que el *demonio* se quede bloqueado si no hay comandos disponibles. La función **`Init`** es necesaria para inicializar el **Mútex** y la Variable condicional, así como la estructura interna de la cola.

2. ***La lista de estado de las operaciones***: esta lista permite a los drivers usuarios del subsistema chequear el estado de las transferencias que han ordenado mediante el envío de un comando. Como será accedida concurrentemente por las tareas usuarias y por el *demonio*, es necesario protegerla con un **Mútex**. Además, se utiliza una Variable condicional para que la función **`Wait`**, se quede bloqueada hasta que el estado de la operación sea distinto de **`WAITING`**.

```
type Status_List is array (I2C_Operation_ID)
                          of I2C_Operation_Status;

type Operation_List is record
   List      : Status_List;
   Mutex_Ref : KM.Mutex_Descriptor;
   Cond_Ref  : KCV.Condition_Descriptor;
end record;
```

3. ***La lista de Buffers internos***: el subsistema ofrece Buffers para que las aplicaciones C y Ada envíen o reciban los bytes del protocolo I2C. Al usuario le decimos que un Buffer es:

```
type I2C_Buffer is private;
Null_I2C_Buffer : constant I2C_Buffer;
```

   Sin embargo, lo que realmente se esconde detrás de **`I2C_Buffer`** es un identificador del Buffer interno que vamos a usar. De esta manera, tenemos una serie de buffers internos que identificamos con un valor. Esto nos permite que la compatibilidad con C y Ada sea más fácil y que si el usuario se olvida de liberar sus Buffers, se de cuenta rápidamente (pues existe un número máximo).

   Un Buffer en realidad no es más que un puntero a un array de bytes. Y una lista de Buffers guarda, además del Buffer, una variable que indica si está en uso o no, y otra que indica si es de lectura o de escritura. Aprovecharemos el campo **`I2C_Flags`**, al igual que en el *kernel* Linux, que nos servía para distinguir variaciones en el protocolo I2C.

```
type I2C_Buffer_Cell is record
    In_Use : Boolean;
    Data   : I2C_Data_Ref;
    Flags  : I2C_Flags;
end record;
```

   Los drivers usuarios del subsistema accederán a esta lista de forma concurrente para crear y liberar Buffers. Por ello es necesario protegerla con un **Mútex**. Podemos pensar que el *demonio* también accede de forma concurrente.

   Pero hay una pequeña diferencia. El *demonio* no accede nunca a la lista, si no que él recibe directamente el valor de la variable **`Data : I2C_Data_Ref;`**, o sea el Buffer, en cada mensaje y lo reenvía, a través de la función **`Master_Xfer`**, a los adaptadores. La sincronización en el acceso al Buffer es implícita pues la única forma que tienen las aplicaciones de leer el Buffer es a través de la función **`Read_Buffer`**, y ésta espera en una variable condicional (la de la lista de estado de operaciones) hasta que la transferencia se ha terminado y el Buffer está listo. Esto lo veremos más claro al estudiar las interfaces.

- **Las interfaces:** los módulos interfaz permiten el envío de comandos al *demonio*, la escritura/lectura de los buffers y el chequeo del estado de la operación. Se ha distinguido una interfaz para Ada y una interfaz para C debido a que algunas características de Ada, como los atributos de los arrays, la diferencia entre Access y los punteros C, no eran fácilmente exportables a C. La conversión de los mensajes a un formato único se realiza dentro de las funciones para que así el *demonio* no necesite hacer distinción alguna. El hecho de usar buffers internos con identificadores se debe también, en parte, a la búsqueda de compatibilidad con C.

  Interfaz ADA: hemos visto ya cuáles eran las funciones interfaz Ada. Pero no sabemos cómo funcionan por dentro. En la interfaz Ada se utiliza un parámetro para pasar la lista de mensajes del siguiente tipo:

```
type I2C_Msg_Ada_List is record
   Msg_List : I2C_Msg_List;
   Num      : Natural;
end record;
```

  La variable **Num** sirve para llevar la cuenta de los mensajes que se han añadido a la lista. Las funciones de transferencia lo que tienen que hacer es transformarlo todo en un comando que entienda el *demonio* I2C, poner el estado de la operación a **WAITING** y encolar el comando. Como se trata de un paquete hijo, tiene visibilidad a la cola de prioridad del *demonio* I2C:

```
procedure I2C_Transfer:
Cmd.Adap_ID := Adap;
Cmd.Operation_ID    := Op;
Cmd.Msg_List := new I2C_Msg_List_Array(1 .. Msg_list.Num);
Cmd.Msg_List(1 .. Msg_list.Num) :=
                     Msg_list.Msg_list(1 .. Msg_list.Num);
Set_Status (Op, WAITING);
Enqueue(Cmd, Prio, The_Commands_Queue);
```

  Interfaz C: en C no podríamos utilizar el tipo **I2C_Msg_Ada_List** porque contiene un tipo access que no entiende. En su lugar necesitamos usar punteros. Para poder utilizar los punteros desde Ada, existe un paquete llamado **Interfaces.C.Pointers** que es lo que vamos a utilizar:

```
type C_Msg is record
   Addr  : I2C_Address;
   Flags : I2C_Flags;
   Buffer: I2C_Buffer;
end record;

subtype C_Msg_Count is Integer;
type C_Msg_List is array (C_Msg_Count range <>) of aliased C_Msg;

package Msg is new Interfaces.C.Pointers(C_Msg_Count,
                                         C_Msg,
                                         C_Msg_List,
                                         C_Msg'(0,0,0));
```

En la función interfaz se deberá realizar la conversión de formatos. Para ello crearemos una nueva lista de mensajes y los iremos copiando del puntero C al tipo Ada:

```
function I2C_Transfer ( ... Msg_list : in Msg.Pointer;)
...
  Cmd.Adap_ID := Adap;
  Cmd.Operation_ID    := Op;
  Cmd.Msg_List := new I2C_Msg_List_Array(1 .. Count);
  for I in Cmd.Msg_List'Range loop
     Tmp_Msg.Addr := P.all.Addr;
     Cell := Get_Buffer_Cell (P.All.Buffer, The_Buffer_List);
     Tmp_Msg.Flags := P.all.Flags Or Cell.Flags;
     Tmp_Msg.Buffer := Cell.Data;
     Tmp_Msg.Count := Cell.Data'Length;
     Cmd.Msg_List(i) := Tmp_Msg;
     P := P + 1;
  end loop;
  Set_Status (Op, WAITING);
  Enqueue(Cmd, Prio, The_Commands_Queue);
```

Se necesitará hacer otra conversión al leer los datos del Buffer para convertirlos a un puntero que entienda el usuario C:

```
   function Read_Buffer ( Data : in Char.Pointer;
                          Count : in I2C_Data_Count;
                          Buffer : in I2C_Buffer;
                          Op   : in I2C_Operation_ID)
                        return I2C_Operation_Status is
      Stat : I2C_Operation_Status;
      Cell : I2C_Buffer_Cell;
      P : Char.Pointer := Data;
   begin
      Stat := I2C.Daemon.Wait (Op);
      Cell := Get_Buffer_Cell (Buffer, The_Buffer_List);
      for i in 1 .. Count loop
         P.all := Cell.Data(i);
         P := P + 1;
      end loop;
      return Stat;
   end Read_Buffer;
```

Las funciones que no necesitan realizar conversiones a tipos Ada las podemos programar en C. Por ejemplo, se creó el fichero **`i2c_c_interface.c`** para programar las funciones de control de los Flags. De esta manera podemos pasar valores por referencia (la variable **`flags`**) sin tener que crear más tipos de punteros mediante el paquete Ada de **`Interfaces.C.Pointers`**. Cuando utilizamos código C, hay que añadir la correspondiente línea de compilación al fichero GNUMakefile y exportar las funciones mediante la palabra clave **extern** en el fichero de cabecera.

5.2.4. Pruebas e integración

Al igual que en el caso del driver de la tarjeta multifunción PCM-3718-H, el diseño de este subsistema pasó por varias fases antes de llegar a la versión actual.

Se comenzó, como es lógico, elaborando la parte de más bajo nivel para asegurarnos de que podíamos emular el protocolo I2C con éxito. Para ello utilizamos una brújula con interfaz I2C sobre la que hacer nuestras pruebas. Fruto de este trabajo se crearon las funciones correspondientes al adaptador ELITE, tanto las de bajo nivel como las algorítmicas.

Sobre esta base, podríamos haber terminado añadiendo un simple **Mútex** que se encargara de controlar el acceso al *bus*. Sin embargo, se quiso implementar un sistema más complejo que fuera extensible y que cumpliera las especificaciones vistas. Se creó una primera versión basada en objetos protegidos, llamados Mailboxes, que servían de Buffers y de indicadores de su estado. Sin embargo, cuando estaba el subsistema completamente programado, al poner las prioridades se comprobó que no funcionaba debido a que los usuarios C, a pesar de convertirse sus datos a Ada, al final accedían, como threads, a los Mailboxes.

En un segundo intento, se creó una *clase* para los mensajes. De esta clase nacían dos hijos, uno para Ada y otro para C. En esta versión, los buffers los creaba el usuario a su antojo y sin limitación. Los usuarios Ada creaban un buffer a partir de un tipo Access y los usuarios C a partir de un tipo Char *. El truco estaba en que la escritura/lectura de los buffers se realizaba a través de los métodos de la clase. De esta manera, el *demonio* podía utilizar polimorfismo para escribir los datos sin tener que distinguir si el comando procedía de un usuario Ada o C.

Sin embargo, esto tenía un problema y era que los buffers residían en el espacio de usuario, lo cual no es tan seguro. Si por ejemplo el driver usuario manda el comando utilizando un buffer creado como una variable local en la llamada IOCTL, al intentar leerlo en la llamada READ el buffer habrá perdido visibilidad. Por tanto, se obliga al usuario a crear buffers globales. Por otro lado, si el usuario no libera la memoria de los buffers, no sería tan fácil darse cuenta como en la versión de buffers internos (ya que en esta versión el número de buffers es acotado).

De modo que en la versión final el usuario escogerá uno de los buffers internos para realizar sus operaciones y enviará su identificador en los comandos. Si se necesitan más buffers de los disponibles bastará con modificar la constante I2C_Mx_Number_Of_Buffers.

5.3. Driver para un dispositivo I2C real: Brújula CMPS03

Realizaremos un driver muy sencillo para manejar una brújula magnética mediante su interfaz I2C. Nos servirá como prueba del buen funcionamiento del subsistema y su función en el robot podría ser conseguir una mejor orientación para moverse por el terreno de juego.

5.3.1. HARDWARE

Conexionado físico

La brújula digital CMPS03 es un sensor de campos magnéticos que una vez calibrado ofrece una precisión de 3-4 grados y una resolución de décimas. Su precio es de unos 44.25 Euros (IVA incluido). Tiene dos interfaces, mediante pulsos temporizados (modulación en anchura), o bien por medio de un *Bus* I2C. Este sensor magnético está especifícamente diseñado como sistema de navegación para robots. La brújula está basada en los sensores KMZ51 de Philips que son lo suficientemente sensibles como para captar el campo magnético de la tierra. Usando dos de estos sensores colocados en ángulo de 90 grados, permite al microprocesador calcular la dirección de la componente horizontal del campo magnético natural.

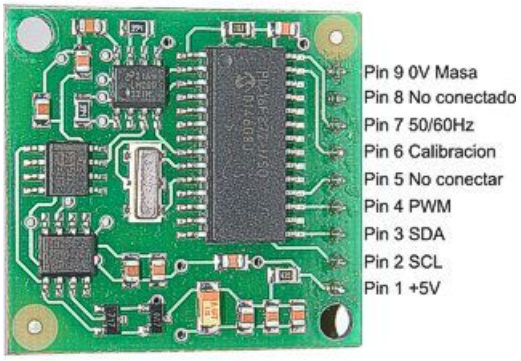

Figura 5.7: Conexiones de la brújula

En la figura 5.7 se puede observar las conexiones de los diferentes pines del CMPS03 cuyo significado es:

- Pin 1 y 9: son la alimentación y la tierra, que al ser de 5 V, podemos conectar directamente a nuestra fuente.
- Pin 4: es una de las dos posibles interfaces, el PWM (Pulse Witdh Modulation - Modulación por anchura de pulso). En este pin se obtiene una señal en la que el pulso positivo representa el ángulo de la brújula. El pulso varia en duración desde 1mS (0 grados) hasta 36,99 ms (359,9 grados), o dicho de otra forma, el pulso es igual a 100 uS por cada grado más 1ms de tara. La señal permanece a cero durante 65 ms entre pulsos, por lo que el periodo de trabajo es de 65mS + la anchura del pulso. El pulso es generado por un contador de 16 bits del propio procesador, con una resolución de

1 uS, aunque en la práctica no es recomendable hacer mediciones con una resolucion de más de 0,1 grados (10uS). Cuando use el interfaz PWM, es necesario conectar a +5V mediante 2 resistencias de 47 Kohm, los pines 2 y 3 (SCL - SDA) del interfaz I2C, ya que no se incluye resistencias de pull-up en el circuito.

- Pin 2 y 3: corresponden al *Bus* I2C formado por las señales SDA (señal de datos) y SCL (señal de reloj).
- Pin 7: nosotros lo pondremos a masa. Sirve para seleccionar entre 50 Hz (puesta a cero) o 60 Hz (puesta a uno). Esto es debido a una desviación errónea de unos 1,5 grados causada por el campo generado por la red eléctrica. Sincronizando la conversión con la frecuencia en herzios de la red, se consigue disminuir el error a tan solo 0,2 grados. El pin sí tiene un resistencia interna de pull up, por lo que si se deja sin conectar, funcionará a 60 Hz. El circuito realiza una conversión interna cada 40ms (50 Hz) o cada 33,3 ms (60Hz) de acuerdo con la conexión de esta entrada. No hay ningún tipo de sincronismo entre la realización de la conversión y la salida de los datos, ya que cuando estos son leídos se devuelven el valor mas reciente que este almacenado en su respectivo registro.
- Pines 5 y 8: están marcados como no conectados, aunque el pin 8 es en realidad el reset del microprocesador, con el fin de poder programarlo una vez soldado al circuito impreso. Esta entrada no tiene resistencia de pull up.

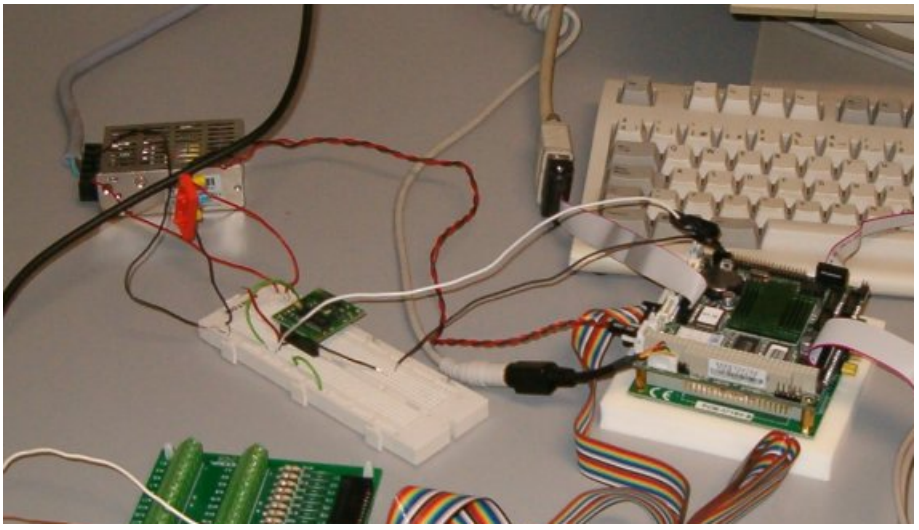

Figura 5.8: Entorno de desarrollo del driver para la CMPS03

- Pin 6: se usa para calibrar el sensor magnético. Esta entrada tiene su propia resistencia de polarización (pull up) y puede dejarse sin conectar una vez realizada la conversión. Antes de realizar la calibración, el modulo deberá mantenerse perfectamente horizontal con los componentes hacia arriba y los dos sensores en la cara inferior. Mantener el modulo alejado de objetos metálicos y muy especialmente de objetos magnéticos como imanes y altavoces. También es necesario conocer con precisión la dirección. en la que se encuentran los cuatro puntos cardinales, por lo que es absolutamente necesario comprobarlo con una brújula magnética. La calibración de la brújula digital puede hacerse por cualquiera de los siguientes dos métodos:

1. *El Método del pulsador*: es el que usaremos nosotros por ser más sencillo. Consiste en utilizar un pulsador entre masa y el pin 6 del circuito, con el fin de iniciar la calibración. Puede dejarse sin conectar una vez realizada la calibración. Para realizar la calibración, bastará con poner a masa el pin 6 momentáneamente por cada uno de los puntos cardinales en cualquier orden. Ejemplo:

    a) Apunta el circuito hacia el Norte. Pulsa momentáneamente en pulsador.
    b) Apunta el circuito hacia el Este. Pulsa momentáneamente en pulsador.
    c) Apunta el circuito hacia el Oeste. Pulsa momentáneamente en pulsador.
    d) Apunta el circuito hacia el Sur. Pulsa momentáneamente en pulsador.

2. *El Método I2C*: este método consiste en escribir 255 en el registro 15 del módulo por cada uno de los cuatro puntos cardinales.

El *Bus* I2C del circuito no incorpora las necesarias resistencias de pull-up, por lo que será necesaria su implementación en el mismo, para ello es recomendable utilizar dos resistencias de 1K8 en caso de utilizar el *bus* a 400 KHz y de 1K2 o 1K si se utiliza a una frecuencia de 1Mhz. Sólo son necesarias 2 resistencias en total para todo el *bus*, no por cada circuito que esté conectado al mismo (en nuestro caso las resistencias de pull-up están incluidas en el propio *Bus* I2C del procesador ELITE).

El sensor de brújula digital esta diseñado para ser compatible con la velocidad estándar de reloj de 100 Khz, aunque ésta puede aumentarse si se tiene en cuenta algunos factores que se especifican en el manual (nosotros no necesitaremos tanta velocidad).

Registros de la brújula

La brújula tiene un total de 16 bytes de registros, algunos de los cuales forman registros de 2 bytes tal y como puede verse en la tabla:

Registro	Función
0	Numero de revisión del Software
1	Dirección en 1 byte 0-255 para 0 - 360 grados
2,3	Dirección en 2 bytes 0-3599 para 0 - 359,9 grados
4,5	Test interno señal diferencial sensor 1
6,7	Test interno señal diferencial sensor 2
8,9	Test interno, valor de calibración 1
10,11	Test interno, valor de calibración 2
12	Sin usar, devuelve 0
13	Sin usar, devuelve 0
14	Sin usar, devuelve 0
15	Comando de calibración, escribir 255 para calibrar

El registro 0 es la revisión del software que actualmente es el 8. El registro 1 es la dirección, en grados, convertida en un valor entre 0 y 255 y que puede ser muy útil en ciertas aplicaciones donde resulta complicado utilizar la escala de 0 a 360 grados que requiere dos

bytes y que esta disponible en los registros 2 y 3 (el 2 es el más significativo) con un valor que va entre 0 y 3599 que equivale a 0 - 359,9 grados. Los registros 4 a 11 son de uso interno y del 12 al 14 no se usan, por lo que no deberán leerse con el fin de no consumir el ancho de banda del *Bus* I2C. El registro 15 se usa para calibrar la brújula.

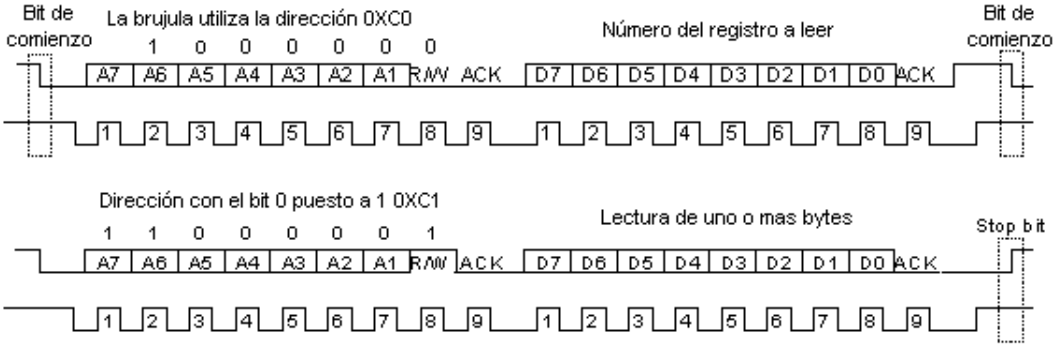

Figura 5.9: Diagrama temporal de la interfaz I2C de la brújula

En la figura 5.9 podemos ver el diagrama temporal del *Bus* I2C para leer los registros de la brújula. Consiste en un mensaje de escritura del registro que queremos leer, seguido de uno de lectura donde leemos el registro. Como vemos se ajusta a la tercera forma de transferencia del protocolo I2C (ver la sección 5.1.1).

5.3.2. SOFTWARE

Análisis de especificaciones

En principio los únicos registros que nos van a interesar son los registros que devuelven el valor de la dirección de la componente horizontal del campo magnético predominante. Esta dirección o rumbo una vez calibrada la brújula y en ausencia de campos magnéticos parásitos apuntaría al Norte magnético (si sabes la declinación magnética de tu región puedes calcular el geográfico también). Si existen campos magnéticos alrededor (basta con que esté en un edificio para que existan) no podemos usarla para calcular el Norte pero sí para calcular giros relativos. En este caso la calibración no es necesaria. El funcionamiento desde el punto de vista del usuario será el siguiente:

1. Creamos un *buffer* para almacenar el valor de la portadora.
2. Abrimos el fichero del driver mediante OPEN.
3. Enviamos un comando IOCTL indicando que se inicie la captura de la portadora, bien en el modo de un byte o bien en el modo de dos bytes.
4. Chequeamos mediante otro comando IOCTL el estado de la captura. El estado podrá ser que la captura está en progreso, que no se ha iniciado ninguna, que se ha producido un error o que ya está realizada.

5. Leemos el valor de la portadora mediante READ. Si intentamos leer cuando la captura no se ha completado la llamada READ será bloqueante[2].

Diseño de la arquitectura

Programaremos el driver en C, que siempre causará más dificultades y por tanto nos servirá para testear mejor nuestro subsistema. En todo caso también se ha testeado el subsistema desde pruebas en Ada. Los archivos relacionados con este driver son:

1. **`cmps03.h`**: en este fichero se declaran las principales constantes y tipos que necesitará el driver, incluyendo los comandos y argumentos IOCTL.
2. **`cmps03_c.c`**: este es el driver, propiamente dicho, donde programamos las funciones prototipo.
3. **`cmps03_functions.ads`**: archivo necesario para poder hacer las importaciones de las funciones a Ada y así poder instalarlas en el subsistema de drivers.
4. **`cmps03.ads`**: contiene los tipos equivalentes en Ada a los declarados en **`cmps03.h`**.
5. **`test_cmps03.adb`**: prueba del driver desde una aplicación Ada.
6. **`test_cmps03_c.c`**: prueba del driver desde una aplicación C.

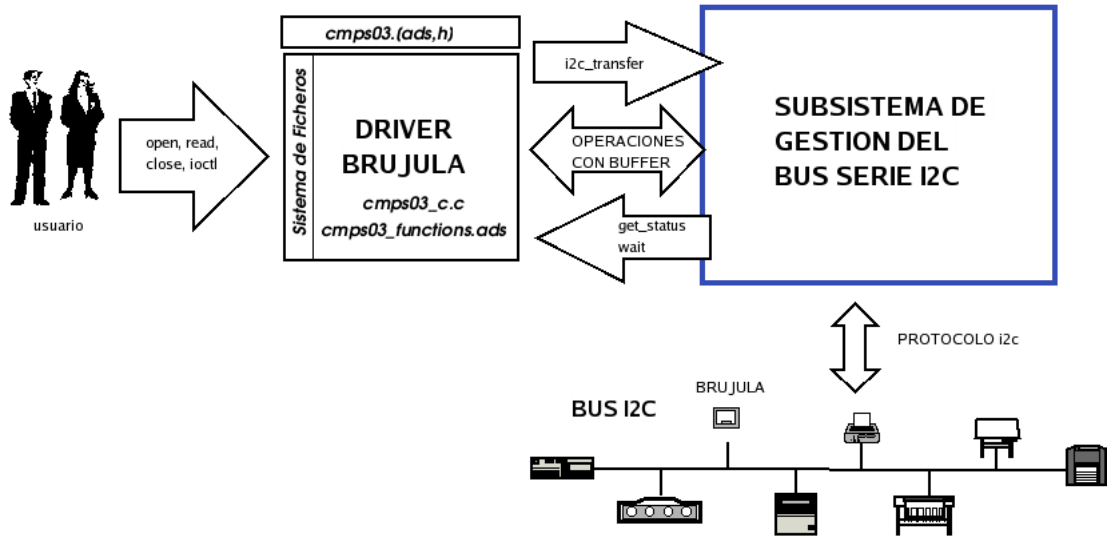

Figura 5.10: Arquitectura Brújula CMPS03

En la figura 5.10 se muestra la arquitectura del driver.

[2]La distinción entre llamadas bloqueantes debería ser posible mediante la variable del tipo Open_Option_Set de la llamada OPEN de POSIX. En la última versión de MaRTE OS ya está implementado.

Diseño detallado y codificación

Definiremos las siguientes constantes de tal manera que si cambia alguna no tenemos que cambiarla en todo el driver:

```
#define SLAVE_ADDRESS 0x60
#define REG_CMPS03_BEARING_BYTE 1
#define REG_CMPS03_BEARING_WORD 2
#define CMPS03_I2C_PRIORITY 4
#define I2C_ADAPTER ELITE
#define I2C_OPERATION OP_1
```

Las más importantes son las últimas tres ya que las otras normalmente no tendrán porqué cambiar. La prioridad será la utilizada a la hora de enviar comandos I2C al *demonio*, no tiene nada que ver con la prioridad del thread. El adaptador indica en qué *bus* se encuentra nuestra brújula. Como vemos lo hacemos de forma estática ya que estamos tratando con un sistema empotrado. Quizá en entornos de propósito general lo correcto hubiera sido usar una función de sondeo que buscase la brújula en los *Buses* I2C disponibles. El identificador de operación es uno de los disponibles. Hay que tener mucho cuidado de no interferir con otros drivers y escoger uno que no esté en uso. Para ello en el fichero i2c.h y i2c.ads, es conveniente añadir un comentario indicando que está siendo utilizado. Otra posibilidad, que de hecho se llegó a implementar, era realizar una función que daba identificadores a partir de un grupo de identificadores libres.

Los tipos simplemente son los correspondientes a las portadoras:

```
typedef uint8_t bearing_byte;
typedef uint16_t bearing_word;
```

En cuanto a las llamadas IOCTL tenemos los siguientes dos comandos:

```
typedef int cmps03_ioctl_cmd;
#define START_CONVERSION  0     /*Args: mode*/
#define GET_STATUS  1           /*Args: status*/
```

Es decir, iniciar la conversión y obtener el estado de ésta. Los argumentos sirven para indicar el modo en que se realiza la conversión y para ver el estado de la conversión:

```
typedef uint8_t bearing_mode;
#define BEARING_MODE_BYTE 0
#define BEARING_MODE_WORD 1

typedef uint8_t cmps03_status;
#define NO_CONVERSION_STARTED 1
#define CONVERSION_DONE 2
#define CONVERSION_IN_PROGRESS 3
#define CMPS03_ERROR 4
```

```
typedef struct {
  bearing_mode   mode;
  cmps03_status status;
} cmps03_ioctl_arg;
```

En cuanto a la programación del driver, necesitaremos dos variables globales estáticas (no visibles desde fuera) para guardar los identificadores de los buffers que vamos a utilizar. Necesitamos un buffer de escritura, para enviar el número de registro a leer, y un buffer de lectura, para leer la portadora:

```
static i2c_buffer buffer_read;
static i2c_buffer buffer_send;
```

La estructura de la función IOCTL será un CASE de los dos comandos, y se basará en utilizar el subsistema I2C para enviar los comandos adecuados:

```
  ret = flags_init (&flags);
  case START_CONVERSION:
    case BEARING_MODE_BYTE:
      create_send_buff ("registro uno");
      create_recv_buff ("un byte");
      mensaje_1 (SLAVE_ADDRESS, buffer_send, flags);
      mensaje_2 (SLAVE_ADDRESS, buffer_read, flags);
    case BEARING_MODE_WORD:
      create_send_buff ("registro dos y tres");
      create_recv_buff ("dos bytes");
      mensaje_1 (SLAVE_ADDRESS, buffer_send, flags);
      mensaje_2 (SLAVE_ADDRESS, buffer_read, flags);
    }
    ret = i2c_transfer ( I2C_ADAPTER,
                         I2C_OPERATION,
                         CMPS03_I2C_PRIORITY,
                         msg_list,
                         2);
  case GET_STATUS:
    ioctl_arg->status = i2c_get_status(I2C_OPERATION);
```

En el caso de la función READ deberemos leer el buffer, distinguiendo el modo en el que estemos (uno o dos bytes). El bloqueo nos lo proporciona el subsistema I2C:

```
  case 1:
    stat = read_buffer (&byte_buffer,1,buffer_read,I2C_OPERATION);
  case 2:
    stat = read_buffer (word_buffer,2,buffer_read,I2C_OPERATION);

  free_buffer (buffer_read);
  free_buffer (buffer_send);
```

Pruebas e integración

Para las pruebas se utilizaron programas de prueba tanto en C como en Ada. Surgió un error bastante importante al desarrollar el driver. Se debía a que algunos paquetes del subsistema I2C no se habían elaborado cuando se llamaba a la función Create y al estar nuestro driver programado en C, no podía usar las funciones exportadas correctamente. La solución es añadir las siguientes líneas:

1. **`marte_os_c.all`**:

   ```
   with I2C.Daemon.C_Interface;
   ```

2. **`k-devices_table.ads`**:

   ```
   with I2C;pragma Elaborate_All(I2C);
   with I2C.Daemon;pragma Elaborate_All(I2C.Daemon);
   with I2C.Daemon.C_Interface;
   pragma Elaborate_All(I2C.Daemon.C_Interface);
   with I2C.Daemon.Ada_Interface;
   pragma Elaborate_All(I2C.Daemon.Ada_Interface);
   ```

Capítulo 6

Conclusiones y líneas de futuro

6.1. Conclusiones y resultados

Podemos decir que se han cumplido satisfactoriamente tanto las motivaciones personales (ver 1.1) como los objetivos (ver 1.3) con que se inició este proyecto. Se ha trabajado en el desarrollo de un sistema empotrado (el robot), creando software de bajo nivel (haciendo drivers y gestionando un bus) y utilizando herramientas de código libre (MaRTE OS, GNU/Linux).

Aunque este proyecto es un granito de arena para la construcción final del robot, para el que todavía se necesitarán muchos nuevos driver, su aplicación a sistemas industriales o de telecomunicación ya es posible. Espero que esta detallada memoria, así como la distribución del código fuente, facilite a futuros desarrolladores su trabajo.

Los resultados concretos de este trabajo son:

1. Se ha configurado dos nuevas formas de arrancar MaRTE OS. A través de una tarjeta de memoria (apto para cuando el robot esté terminado) y por Ethernet sin disquetera (lo mejor en la fase de desarrollo).
2. Se ha creado un driver para controlar los puertos digitales de la tarjeta PCM-3718-H.
3. Se ha creado un driver para controlar la entrada analógica de la tarjeta PCM-3718-H. Optimizando el código se han conseguido llegar a los límites de la propia CPU pudiendo realizar conversiones a frecuencias de 38-40 us (25 khz).
4. Se ha diseñado e implementado un entorno para la gestión de uno o varios buses serie I2C. Está preparado para ofrecer una interfaz sencilla tanto a drivers escritos en Ada como en C. Además permite la creación e instalación de nuevos adaptadores del bus I2C de forma sencilla, centralizada y sin cambios en la interfaz.
5. Se ha programado un driver para un dispositivo I2C, la brújula CMPS03, que demuestra la funcionalidad del subsistema creado.

El código de los drivers y el subsistema I2C está estructurado en **43 archivos** que contienen **3085 líneas de código** además de abundante documentación en forma de comentarios.

El presupuesto de este proyecto se muestra en la siguiente tabla:

	Coste
Placa base	335.98 Euros
Tarjeta A/D	430 Euros
Brújula	44.25 Euros
Ingeniería	72808 Euros
otros: luz, internet...	400 Euros
TOTAL	74018.23 Euros

6.2. Trabajo futuro

Se proponen los siguientes puntos para futuros desarrollos:

- El mercado de periféricos con interfaz I2C es muy amplio. Mediante este subsistema se abre enormemente la capacidad de incorporar nuevos artilugios de utilidad al futuro robot. Por tanto, como trabajo futuro se propone la creación de nuevos drivers para dispositivos con esta interfaz.
- Se necesitan realizar nuevos adaptadores para otros tipos de controladores del bus I2C que hay en el mercado. Para ello, como guía el subsistema cuenta con varios ejemplos tanto en C como en Ada.
- Sobre el desarrollo del driver de la brújula deberá crearse otro proyecto que la utilice para mejorar el control de las trayectorias del robot.
- También se propone arreglar la posibilidad de establecer el comportamiento bloqueante o no de los drivers mediante la llamada POSIX OPEN, pues ya se encuentra implementada en la última versión de MaRTE OS.
- Como último detalle se propone realizar otro proyecto dedicado a aumentar, clarificar y actualizar la documentación de MaRTE OS, pues la considero tan importante como el propio código. Espero que con esta memoria aporte mi parte en ese sentido.

Apéndice A

Pequeños consejos

1. Cuando realices drivers ten siempre en cuenta la interfaz C. En C no se puede representar tanta variedad de tipos como en Ada (tipos arrays, access de cualquier tipo, objetos protegidos, etc).

2. Utiliza objetos protegidos cuando estés completamente seguro de que sólo serán accedidos por tareas Ada. Si un thread accede, aunque sea a través de una función escrita en Ada y exportada, a un objeto protegido, no respetará su protocolo. Para sincronizar threads y tareas sustituye el comportamiento del objeto protegido mediante Mutexes, para la exclusión mútua, y Variables condicionales para la sincronización y añade una función para inicializarlos.

3. Para realizar una macro en Ada (es decir, obligar al compilador a que despliegue el código en lugar de llamar a una subrutina) se utiliza el pragma **inline**. Úsalo solamente al final del desarrollo y comprueba que efectivamente estás ganando en tiempo de proceso (a costa del tamaño del archivo claro).

4. Algunos de los fallos más típicos que me han ocurrido son:

   - Cuando ocurren cosas extrañas con tus variables (por ejemplo, les asignas un valor y al poco rato ves que tiene otro valor sin que lo hayas cambiado), es probable que tengas un puntero en un código en C (ej: *buffer) al que no has hecho una llamada **malloc** con la suficiente capacidad y se están produciendo sobreescrituras de memoria.
   - Lo mismo te puede pasar a veces debido a que has declarado tu buffer como una variable local (ej: char buffer[5]) en una función (ej: IOCTL) y lo estás intentando usar en otra función (ej: READ). Convierte esa variable local a global añadiendo el atributo **static**, para que no sea visible fuera del archivo.
   - Cuando creas dos aplicaciones, una en C y otra en Ada, con el mismo nombre y distinta extensión, a veces se producen errores de linkado. Se debe a que se ha generado anteriormente un archivo objeto que el compilador no distingue. Esto pasa por ejemplo con **`hello_world.adb`** y **`hello_world.c`**. La solución puede ser borrar esos archivos creados al compilar, o mejor aún cambiar un poco el nombre del programa C, **`hello_world_c.c`**. Esto es aplicable por ejemplo a la hora de hacer los tests de tus drivers.

- Si el programa se queda colgado puede deberse a que un thread C está accediendo a un objeto protegido. Piensa que esto puede pasar incluso al principio del todo, en la inicialización de los drivers, si tu driver tiene un Create que llama a un objeto protegido.
- Un error muy típico lo puedes detectar cuando se produce al instalar un nuevo driver. Al iniciarse MaRTE OS, durante la inicialización de los drivers (que son llamadas a sus Creates), se produce una excepción y se termina con un mensaje de **`exit(1)`**. Se debe probablemente a que tienes que cambiar las constantes que determinan, estáticamente, el número de drivers que puedes instalar en **`configuration_parameters.ads`**.
- Otro error que surge a menudo al utilizar tus drivers desde C se debe a que los paquetes Ada que exportan funciones C no se encuentran elaborados. Normalmente se necesitan añadir al fichero **`marte_os_c.ads`** y a veces hacer un with con el pragma **`elaborate_all`** en **`k-devices_table.ads`**. Ten en cuenta que este error no sólo puede aparecer al utilizar el driver desde la tarea de usuario sino también en la inicialización, cuando se llama a su función Create (si la tiene).
- Cuando se intenta utilizar las instrucciones del ATC de Ada (ej: select-then-abort) te puede ocurrir que incluso los compilando correctamente, no funcione según lo esperado. Esto se debe probablemente a un error en la configuración del compilador GNAT. Vete al directorio:

  ```
  $ cd gap.../lib/gcc-lib/i686-pc-linux-gnu/3.2.3/
  ```

  y comprueba que los enlaces simbólicos '**adainclude**' y '**adalib**' apuntan a '**rts-sjlj/adainclude**' y '**rts-sjlj/adalib**'. Si no es así, modifícalos y recompila todo MaRTE con '**mkall**' o reinstala con '**minstall**'.

5. Utiliza cuando puedas Identificadores. Facilitan la compatibilidad con C y además evitan mover datos de un lado para otro.

6. El argumento de la función IOCTL es tanto de entrada como de salida.

7. Los nombres de los paquetes de MaRTE OS son a menudo gigantescos, lo cual puede provocar que tu programa sea poco legible. Es mejor utilizar **renames** para cambiar el nombre a uno de unas pocas letras, que usar la cláusula use, ya que ésta hace menos legible aún el código.

8. La documentación sobre la parte interna de MaRTE OS es escasa. Por ello es necesario navegar a través del código. Este comando puede facilitarte las cosas:

   ```
   grep -R -n -i loquebusco *
   ```

   Otras utilidades interesantes on el script **gnathtml.pl** y el programa **Ada2html**.

9. Si tu código te ha quedado descolocado puedes reindentarlo de forma automática mediante el editor emacs. Para ello ábre un fichero de código, selecciona el código a indentar, pulsa Alt+x y escribe la orden **ada-indent-region** (si tu programa está en Ada) o **indent-region** (si está en C).

10. Hay una función muy interesante en **`Posix_Hardware_Interrupts.ads`**. Gracias a ella podemos dormir una tarea a la espera de que se produzca una interrupción. Cuando se produzca, primero se ejecutará el manejador (**`Handler`**) y a continuación nuestra tarea se despertará (compitiendo por el uso de la CPU según la prioridad que tenga). Permite pues responder al evento de una interrupción. Se trata de la siguiente función:

```
procedure Wait (Intr    : out Hardware_Interrupt;
                Handler : out Interrupt_Handler);
```

Sin embargo, como hemos dicho, en los drivers es mejor no usar la interfaz POSIX y utilizar los dos ficheros ya mencionados del directorio **`misc/`**. Puede parecer que esta función no viene en esos ficheros pero hay una equivalente:

```
function Timedwait
  (Flags       : in     Int;
   Abs_Timeout : in     Type_Timespec.Timespec_Ac_Const;
   Intr        : access Hardware_Interrupt;
   Handler     : access Interrupt_Handler_Function)
  return Int
```

Uso: **`Timedwait(0,null,Intr'access, Handler'access)`**

11. Ya hemos comentado que desde una función manejadora de interrupciones no podemos utilizar Mutexes ni Variables condicionales. Si nos queremos sincronizar con una función manejadora de interrupciones sólo podemos hacer lo siguiente:

    - Para obtener exclusión mútua en el acceso a un recurso compartido podemos hacer que ese recurso sea una variable atómica, o bien, enmascarar la interrupción cuando queramos acceder al recurso.
    - Para señalización y espera podemos utilizar los semáforos contadores de MaRTE OS (los semáforos contadores también sirven para exclusión mútua entre threads pero no deben usarse ya que adolece de inversiones de prioridad y su uso es complejo y proclive a errores. En su lugar se deben usar Mutexes y Variables Condicionales.) Por ejemplo, un semáforo contador nos puede servir para llevar una cuenta de datos listos, o utilizarlos como semáforos binarios para indicar un evento. Para usarlos en señalización hay que pensar que un semáforo es una especie de contador. Si los usamos desde drivers es mejor utilizar el paquete **`misc/marte_semaphores.ads`**, que es equivalente al POSIX. Las funciones más importantes son:
        - Post: incrementa el contador del semáforo.
        - Getvalue: lee el valor del contador del semáforo.
        - Wait: decrementa el contador del semáforo. Si el contador vale 0, se queda bloqueado hasta que alguien lo incremente mediante un Post.

      Si, por ejemplo, queremos utilizar el semáforo como binario, para señalizar un evento (ej: muestra lista) haremos lo siguiente:

      Manejador de interrupción (señaliza que hay muestras en el buffer):

```
Escribo muestra en el buffer
Ret := MaRTE_Semaphores.Getvalue(Scan_Sem'Access,Value'Access);
if Value = 0 then
  Ret := MaRTE_Semaphores.Post (Scan_Sem'Access);
end if;
```

Tarea que espera el evento de que haya alguna muestra para leerla. Para la exclusión mútua ver que enmascaro la interrupción:

```
if MaRTE_Semaphores.Wait (Scan_Sem'Access) /= 0 then
  return -1;
end if;
Lock_IRQ;
Leo algunas muestras del buffer
if quedan muestras sin leer then
  Ret := MaRTE_Semaphores.Getvalue(Scan_Sem'Access,Value'Access);
  if Value = 0 then
     Ret := MaRTE_Semaphores.Post (Scan_Sem'Access);
  end if;
end if;
Unlock_IRQ;
```

12. Fedora también tiene un método de instalación de programas automatizado como el apt-get de las distros basadas en Debian. Se llama **yum** y es muy fácil de utilizar. Resulta muy útil para instalar nuevos programas, servidores, etc...

Bibliografía

[1] Artist. *Guidelines for a Graduate Curriculum on Embedded Software and Systems*. 2003. **http://www.artist-embedded.org/Education/index.html**.

[2] *RoboCup*. **http://www.robocup.org/**.

[3] Michael González Harbour y C. Douglass Locke. *Tostadores y POSIX*. 1997. **http://www.ctr.unican.es/publications/mgh-cdl-1997a.pdf**.

[4] *PASC, POSIX status report*. **http://www.pasc.org/standing/sd11.html**.

[5] Ruíz, Bernardo. *Control de trayectoria de un robot móvil en ada 95 sobre MaRTE OS*. Master's thesis, UC, 2005.

[6] *Código Fuente MaRTE OS*. **http://marte.unican.es**.

[7] *Debate entre Linus y Tanenbaum*. **http://www.oreilly.com/catalog/opensources/book/appa.html**.

[8] Alberto Gutiérrez Castro. *Migración de un sistema operativo de tiempo real, MaRTE OS, a un microcontrolador*. 2003. **http://marte.unican.es/port_marte_os.pdf**.

[9] Francisco Guerreira. *Entorno para la instalación y utilización de manejadores de dispositivos en MaRTE OS*. 2003. **http://marte.unican.es/memoria_drivers.pdf**.

[10] Hongjiu Lu. *ELF: From The Programmer's Perspective*. 1995.

[11] *Documentos de Intel*. **http://www.x86.org/intel.doc/inteldocs.htm**.

[12] Eduardo Suárez. *Dynamic partitions with grub: a guide for multiboot systems*. **https://www.ctm.ulpgc.es/~eduardo/grub/grub.html**.

[13] *Cargador GNU/GRUB*. **http://www.gnu.org/software/grub/**.

[14] *Especificación Multiboot*. **http://www.gnu.org/software/grub/manual/multiboot/multiboot.html**.

[15] *Cargador ETHERBOOT*. **http://etherboot.sourceforge.net/**.

[16] *Cargador NETBOOT*. **http://netboot.sourceforge.net**.

[17] *Cargador NILO*. **http://www.nilo.org/**.

[18] *Arranque por Compact Flash*.
http://leaf.sourceforge.net/doc/guide/bugrub.html.

[19] *Estándar PC/104*. **http://www.pc104.org/**.

[20] *Manual Bus I2C*. **http://www.semiconductors.philips.com**.

[21] Raúl Mateos Gil y otros. *Microcontroladores PIC. Aplicaciones en ingeniería eléctrica*. 2004.

[22] *Modeling and Analysis Suite for Real-Time Applications*.
http://mast.unican.es/.

[23] Mario Aldea. *Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas*. 2002. **aldeam_at_unican.es**.

[24] Jonathan Corbet y otros. *Linux Device Drivers 3 - O'Reilly*. 2005.
http://lwn.net/Kernel/LDD3/.

[25] Alan Burns. *Concurrency in Ada*. 1998. University Press. Cambridge.

[26] *Arranque por Red con PXE*.
http://www.kegel.com/linux/pxe.html.

[27] *Manual Referencia Ada95*. **http://www.adahome.com/rm95/**.

[28] *Ada95 Quality and Style guide*.
http://www.adaic.com/docs/95style/html/cover.html.

[29] *Manual DAQ PCM3718H*. **http://www.advantech.com/**.

[30] *Manual placa PCM3347F*. **http://www.advantech.com/**.

[31] *Manual CPU STPC Elite y Consumer II*. **http://www.st.com/**.

[32] *Manual Brújula CMPS03*. **http://www.superrobotica.com**.

Índice alfabético

Índice de figuras