Programa Oficial de Postgrado en Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

Manejadores de buses serie en MaRTE OS

Mónica Puig-Pey González

puigpeym@unican.es

Director:
Mario Aldea Rivas y
Michael González Harbour.
Grupo de Computadores y Tiempo Real
Departamento de Electrónica y Computadores.

Santander, Octubre de 2009

Curso 2008/2009

Agradecimientos

En primer lugar, agradecer a mis directores de tesis Mario Aldea Rivas y Michael González Harbour por haberme ofrecido la oportunidad de realizar este trabajo. Por su amabilidad y ayuda, orientándome sobre cómo afrontar cada objetivo a cumplir.

A todos los profesores que he tenido en el Máster de Computación, muchos de los conocimientos adquiridos me han sido útiles de una manera u otra para llevar a cabo este trabajo.

A todos los compañeros que trabajan en el grupo de Computadores y Tiempo Real, porque consiguen crear un ambiente de trabajo realmente agradable. En particular a Juan, porque hemos compartido tantas horas este curso realizando el máster que ya nos reímos sin necesidad de hablarnos.

Y para terminar a mi familia, a mis amigas y a Tato, por animarme cada vez que las cosas no salían bien. En definitiva por estar siempre ahí.

A todos ellos muchas gracias.

Resumen

Los buses serie son ampliamente utilizados en los sistemas empotrados como mecanismo de comunicación con numerosas tarjetas de captura de datos. En la presente Tesis se pretende desarrollar manejadores en el sistema operativo MaRTE OS para los buses serie RS-232 y SPI, sobre plataforma PC.

En lo referente al bus RS-232, se pretende desarrollar un controlador de hasta 8 puertos serie, que amplíe la capacidad del *hardware* estándar del PC que admite hasta 4 puertos. Para ello se utilizará la tarjeta de 8 puertos serie Parvus COM-1274-A1, usando como punto de partida el controlador ya existente en MaRTE OS para el dispositivo de interfaz serie (UART) del PC.

También se pretende desarrollar un controlador para el protocolo SPI (*Serial Peripheral Interface*), que es un protocolo serie de bajo coste ampliamente utilizado en sistemas empotrados pequeños. Se pretende implementar este protocolo por *software*, usando líneas digitales proporcionadas por una tarjeta de entradas y salidas digitales. Al objeto de realizar las pruebas se va a utilizar la tarjeta de I/O digital Acces 104-AIO12-8-DC.

Con el objeto de probar la implementación del protocolo SPI desarrollada se realizará un controlador del sensor inercial de 3 ejes Analog Devices ADIS16355.

Tabla de contenidos

1. Introducción 1

1.1. Sistemas operativos de tiempo real 1

1.2. Los buses serie. 2

1.3. Objetivos del proyecto 2

1.4. Organización de la memoria 3

2. MaRTE OS 5

2.1. El sistema operativo MaRTE OS 5

2.1.1. Arquitectura de una aplicación ejecutando sobre MaRTE OS 6

2.1.2. Subsistema de drivers de MaRTE OS 7

2.2. Entorno de trabajo. Cargando MaRTE OS 9

2.2.1. Computador de ejecución GEME-3000 10

3. Driver para la tarjeta de 8 puertos serie Parvus COM-1274-A1 11

3.1. Hardware 11

3.1.1. Interfaz RS-232 12

3.1.2. Tarjeta Parvus COM-1274-A1 configuraciones y registros 13

3.1.3. Montaje y pruebas iniciales 14

3.2. Software 15

3.2.1. Diseño de la arquitectura e implementación 16

3.2.2. Pruebas e integración 19

4. Subsistema de gestión del bus serie SPI 21

4.1. Hardware 21

4.1.1. Estándar SPI 22

4.1.2. Tarjeta entradas y salidas digitales Acces 104-AIO12-8-DC 25

4.1.3. Pruebas iniciales. Implementación del protocolo mediante bit banging 27

4.2. Software 29

4.2.1. Análisis de especificaciones 30

4.2.2. Diseño de la arquitectura 30

4.2.3. Diseño detallado y codificación. 36

4.3. Manejador para un dispositivo SPI: sensor inercial de 3 ejes Analog Devices ADIS16355 43

4.3.1. Hardware 43

4.3.2. Software 45

5. Conclusiones 49

5.1. Trabajo futuro 49

Bibliografía 51

1. Introducción

1.1. Sistemas operativos de tiempo real

Un sistema de tiempo de tiempo real se define como aquél que debe completar el trabajo a realizar y liberar los servicios que para ello utiliza, en un plazo determinado de tiempo[1]. Lo importante no sólo es que el trabajo se realice, sino que éste se realice a tiempo.

En nuestro día a día nos encontramos multitud de sistemas de tiempo real ofreciéndonos sus servicios. Por ejemplo, cuando conducimos un coche, son estos sistemas los que controlan la inyección, los frenos e incluso hasta luces. Es evidente que un retardo en la ejecución de la orden de frenado puede llegar a tener graves consecuencias. Más ejemplos de sistemas de tiempo real los podemos encontrar en aviones, controlando el despegue y aterrizaje de los mismos, o en hospitales, utilizando máquinas de monitorización. A diferencia de los PCs y estaciones de trabajo que ejecutan aplicaciones que no son de tiempo real como pueden ser el editor o el navegador, los computadores y redes que soportan aplicaciones con requisitos de tiempo real no están habitualmente al alcance de nuestra vista. Cuando un sistema de tiempo real funciona correctamente tiende a hacernos olvidar su existencia.

A lo largo de este memoria se habla del sistema operativo MaRTE OS[2], un sistema operativo de tiempo real que cumple con el estándar POSIX. Un sistema que implemente este estándar de manera completa es un sistema demasiado grande para su uso en pequeños sistemas empotrados y por ello se aprobó en el año 1997 el estándar POSIX.13[3], que define cuatro subconjuntos de servicios del sistema operativo. MaRTE OS entra dentro del perfil “Sistema de Tiempo Real Mínimo” y puede ser utilizado para desarrollar aplicaciones empotradas de tiempo real o como herramienta de docencia. Se distribuye como *software* libre bajo licencia GPL (“*GNU General Public License*”).

En la implementación de un sistema operativo, existe un apartado importante dedicado a la elaboración de *drivers*. Los *drivers* o manejadores son módulos *software* encargados de controlar un dispositivo periférico facilitando al usuario su manejo, sin que sea necesario que conozca el funcionamiento interno del dispositivo. En MaRTE OS se ha elaborado un subsistema, resultado del proyecto fin de carrera de Francisco Guerreira Payo[12], que nos ofrece un entorno simple, donde los dispositivos son vistos como ficheros, a los que se accede con las funciones POSIX de acceso a ficheros mediante *open, read, write, ioctl,*ect.; similar a lo que se hace en otros sistemas tipo UNIX.

La elaboración de manejadores es importante para permitir a los sistemas utilizar todo tipo de dispositivos periféricos, aumentando las posibilidades del equipo, en múltiples ocasiones equipos empotrados, de interactuar con el exterior.

1.2. Los buses serie.

Un bus serie es una interfaz de comunicaciones de datos digitales, frecuentemente utilizado por computadores y periféricos, en donde la información es transmitida bit a bit enviando un solo bit a la vez, en contraste con los buses paralelo que envían varios bits simultáneamente. En la comunicación serie sólo hay un canal por el que enviar o recibir datos, mientras que en la comunicación paralelo, existen varios canales permitiendo enviar múltiples bits a la vez. Por un lado, al utilizar comunicación paralelo conseguimos mayores tasas de flujos de datos, sin embargo esto contrasta con que el cableado que requiere es mayor que usando un bus serie, lo que encarece la comunicación, y lo que es peor, aumenta la posibilidad de interferencia entre los diferentes cables, reduciendo así la capacidad de transmisión. El objetivo es encontrar un equilibrio entre ancho de banda de la comunicación y el coste de la misma.

Existen tres tipos de comunicaciones serie:

- *Simplex*: En este caso el transmisor y el receptor están perfectamente definidos y la comunicación es unidireccional. Este tipo de comunicaciones se emplean usualmente en redes de radiodifusión, donde los receptores no necesitan enviar ningún tipo de dato al transmisor.

- *Duplex*, *half duplex* o *semi-duplex*: En este caso ambos extremos del sistema de comunicación cumplen funciones de transmisor y receptor y los datos se desplazan en ambos sentidos pero no simultáneamente. Este tipo de comunicación se utiliza habitualmente en la interacción entre terminales y un computador central.

- *Full Duplex*: El sistema es similar al *duplex*, pero los datos se desplazan en ambos sentidos simultáneamente. Para ello ambos transmisores poseen diferentes frecuencias de transmisión o dos caminos de comunicación separados, mientras que la comunicación *semi-duplex* necesita normalmente sólo uno. Para el intercambio de datos entre computadores, este tipo de comunicaciones son más eficientes que las transmisiones *semi-duplex*.

En el desarrollo de este trabajo vamos a conocer a fondo el bus serie SPI, creando una interfaz SPI en MaRTE OS. Existen, además de éste, otros buses serie como el I2C; un bus muy usado en la industria, principalmente para comunicar microcontroladores y sus periféricos en sistemas integrados. El bus SMBus es otro bus serie que tienen la mayoría de las placas base modernas, al que se conectan la mayor parte de los chips de monitorización del sistema. Estos chips sirven para medir temperaturas de componentes, velocidad de ventiladores, voltajes, etc.; en general toda clase de información sobre el *hardware*.

1.3. Objetivos del proyecto

El objetivo de este trabajo es participar en el proyecto “Sistema de Cálculo de Orientación basado en *GPSs* no dedicados y apoyado por Sensores Inerciales MEMs de bajo coste” que consiste en el diseño y fabricación de un prototipo de Sistema de Navegación de alta precisión y bajo coste. Este sistema estará formado por 4 receptores GPS de bajo coste, y 3 sensores inerciales tipo MEMS para la medición de las velocidades de rotación en 3 ejes.

Se trata de un proyecto parcialmente financiado por el Ministerio de Industria Turismo y Comercio y ha sido desarrollado por la Fundación Centro Tecnológico de Componentes en colaboración con la Universidad de Cantabria.

Nuestros objetivos dentro del mismo serán:

- Desarrollo del manejador en MaRTE OS para un tarjeta PC/104[6] que contiene ocho puertos series. Se trata de la tarjeta COM-1274-A1 de Parvus, que va a permitir conectar a nuestro equipo los cuatro receptores GPS.
- Estudio del bus serie SPI (*Serial Peripherical Interface*).
- Diseñar e implementar un subsistema *software* capaz de gestionar el uso de los buses SPI, ofreciendo una interfaz tanto a *drivers* escritos en C como en Ada.
- Programar un *driver* para un sensor inercial de tres ejes y alta precisión que utilice la interfaz SPI, como demostración del funcionamiento real del subsistema SPI. Servirá para proporcionar con precisión los movimientos del sistema de navegación.

1.4. Organización de la memoria

Organizamos la memoria de la siguiente manera:

- En el capítulo 2 se explica cuál es nuestro entorno de trabajo, conociendo los instrumentos que nos servirán en el desarrollo del mismo. Se hace una breve descripción sobre el sistema operativo a utilizar, MaRTE OS, y explicamos cuál es el método a seguir a la hora de instalar *drivers* en el mismo.
- En el capítulo 3, explicamos paso a paso cómo conseguir desarrollar el *driver* en MaRTE OS para la tarjeta COM-1274 de Parvus, que aumenta en ocho el número de puertos serie posibles.
- En el capítulo 4, vemos cómo funciona el bus serie SPI. Aplicamos los conocimientos adquiridos con el fin de implementar una interfaz SPI en MaRTE OS. Como prueba del correcto funcionamiento de la interfaz desarrollamos un *driver* para un dispositivo periférico que se comunique con el equipo a través del subsistema SPI. Se trata del sensor inercial ADIS-16355 de Analog Devices, capaz de medir velocidades angulares y aceleraciones en los tres ejes con alta precisión.
- Para finalizar, en el capítulo 5, repasamos lo objetivos propuestos analizando los resultados obtenidos.

2. MaRTE OS

2.1. El sistema operativo MaRTE OS

El sistema operativo MaRTE OS[2] 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.

Veamos sus principales características:

- Está pensado para aplicaciones principalmente estáticas.
- Presenta tiempos de respuesta acotados en todos sus servicios. En principio, va a ser utilizado en aplicaciones de tiempo real, en las que el cumplimiento de los plazos temporales constituye un requisito fundamental para el correcto funcionamiento del sistema.
- Existe un sólo espacio de direcciones de memoria compartido por el núcleo y la aplicación.
- Permite ejecutar aplicaciones desarrolladas tanto en lenguaje Ada como C.
- Portable a distintas arquitecturas:
  - Arquitectura x86 PC (bien en máquina desnuda o bien en emuladores).
  - Arquitectura Linux y LinuxLib, donde MaRTE OS corre sobre Linux. Principalmente destinada a investigación y enseñanza (no se pueden alcanzar requerimientos de tiempo real estricto dentro de un sistema que no lo es, como Linux).
- Se ha diseñado un núcleo de estructura monolítica, en el que las llamadas al sistema, o al menos la parte de ellas en la que se accede a estructuras globales, constituyen secciones críticas en las que las interrupciones se encuentran deshabilitadas.
- Toma la forma de una librería para ser enlazada con la aplicación.
- Como hemos dicho cumple el perfil mínimo del subconjunto POSIX.13 otorgando al sistema la siguiente funcionalidad:
  - *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 temporizadores: ofrecen funcionalidad para controlar el tiempo de ejecución de cada *thread*. Permiten medir el tiempo y programar actividades para ser ejecutadas en un instante futuro. Los relojes y temporizadores de tiempo de ejecución son los que permiten controlar el tiempo de ejecución de los *threads*.

- 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, ioctl...*

• Es código libre con licencia GNU/GPL. Descargable en [4]

2.1.1. Arquitectura de una aplicación ejecutando sobre MaRTE OS

En la figura 2.1 se muestra el esquema de capas de la arquitectura de una aplicación escrita en Ada y en C respectivamente cuando ejecutan sobre MaRTE OS. 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. También puede hacerse de una forma más directa utilizando la interfaz POSIX-Ada. 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.

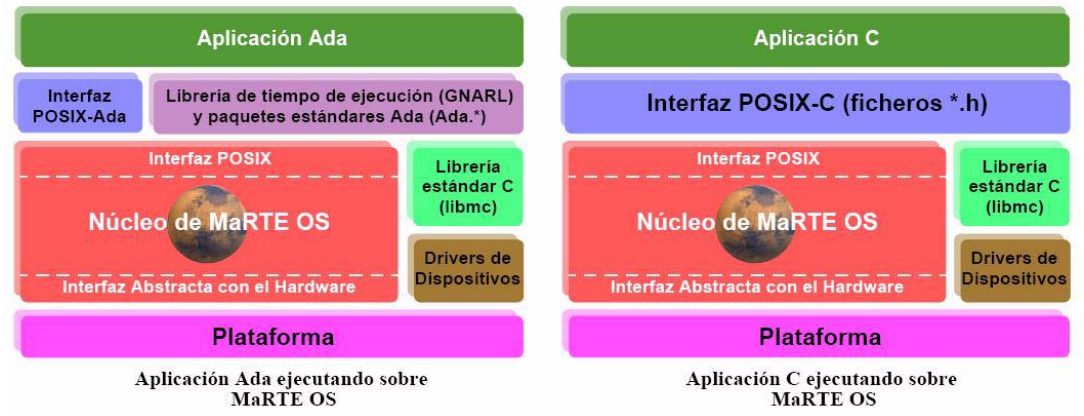


Figura 2.1: Arquitectura de MaRTE OS

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. 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). Lo normal es que esta funcionalidad sea aportada por el código encargado de manejar el dispositivo que se desea haga las veces de consola, en su caso.

2.1.2. Subsistema de drivers de MaRTE OS

La interfaz POSIX de entrada/salida, permite a las aplicaciones acceder a los dispositivos como si fueran ficheros con un nombre (mediante *open, read, write, ioctl...*), ello permite independizar el *software* de aplicación del funcionamiento interno del *driver*.

El nombre con el que la aplicación accede al dispositivo se encuentra en la tabla "`The_Devices_File_Table`" contenida en el fichero `marte-kernel-device_table.ads` que encontramos en el directorio *x86_arch/drivers/arch_dependent_files.* Esta tabla asocia cada dispositivo periférico con su fichero de dispositivo. En este mismo fichero encontramos la tabla "`The_Driver_Table`" que contiene los *drivers* que se encuentran disponibles en ese momento en MaRTE OS. Aparentemente el número de elementos de esta tabla de *drivers*, debería coincidir con el número de elementos en la tabla con los descriptores de ficheros. Pero esto no es así, ya que un mismo *driver* puede dar uso a más de un dispositivo periférico. Es por esto que existen los denominados números mayores y menores de un descriptor de fichero. El mayor sirve para indicar qué *driver* utiliza y el menor para distinguirlo de otros dispositivos que también usen el mismo *driver*. Un ejemplo claro y que además se desarrolla en este trabajo, es el puerto serie. Un computador puede disponer de más de un puerto serie, normalmente dos como mínimo. Ambos puertos utilizan el mismo *software* para controlarlo, y por tanto el mismo *driver*. Los números menores nos sirven para distinguir de qué puerto se trata en cada caso.

En la figura 2.2 vemos las características de estas tablas y sus interrelaciones:

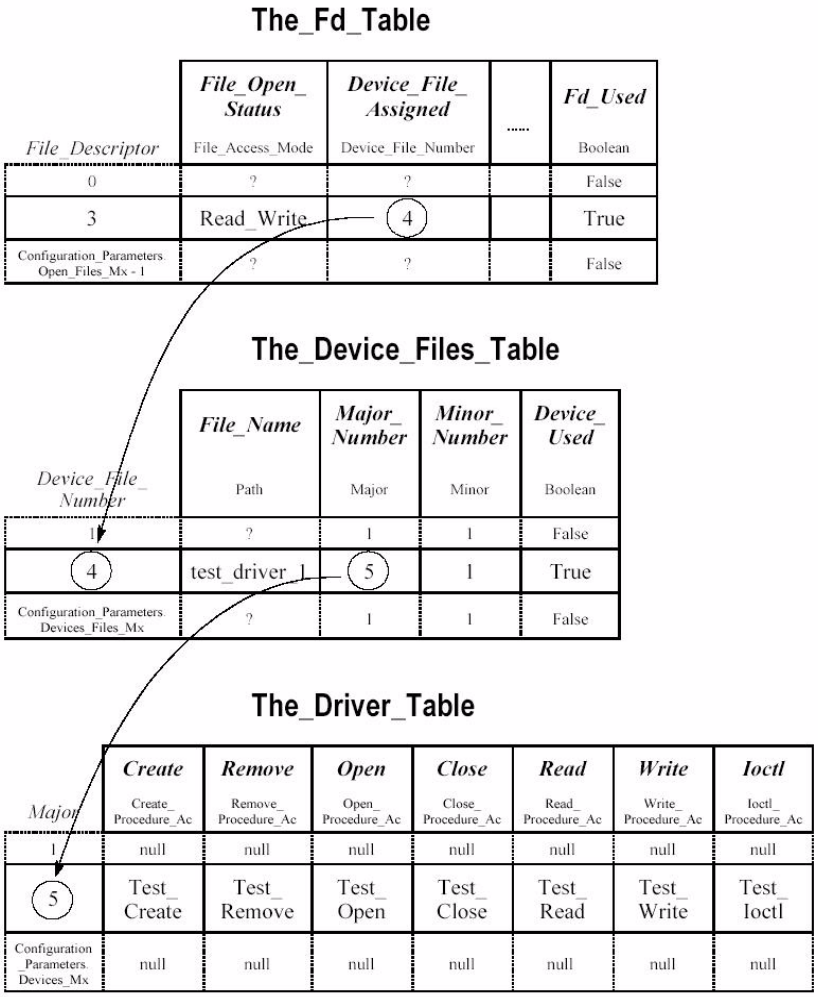


Figura 2.2: Tablas del subsistema de drivers

En el fichero `marte-kernel-device_table.adb`, se desarrolla el procedimiento encargado de inicializar los *drivers* del sistema. Consiste básicamente en llamar a las funciones "Create" de los *drivers* que se encuentren en la tabla "`The_Driver_Table`". Esta función debe ejecutarse antes de que un dispositivo sea usado por una aplicación, y se lleva a cabo en la elaboración de MaRTE, es decir, en la inicialización antes de que se empiece a ejecutar el código del usuario.

Los *drivers* escritos en C necesitan un fichero que importe sus funciones a Ada (`pragma import`), para que puedan ser incluidos en las tablas. En el caso de que estemos en aplicaciones que usen paquetes Ada que exporten funciones que se usarán desde aplicaciones C, los paquetes Ada necesitan ser elaborados antes de usarse, 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. Para realizar esta elaboración de los paquetes podemos hacer dos cosas: incluir el paquete en el fichero `marte_os_c.ads`, realizado con este fin, o hacer uso nada más comenzar el código de la función `ada_init()`.

La figura 2.3 muestra el funcionamiento de los *drivers* en MaRTE OS que hemos ido explicando a lo largo del apartado:

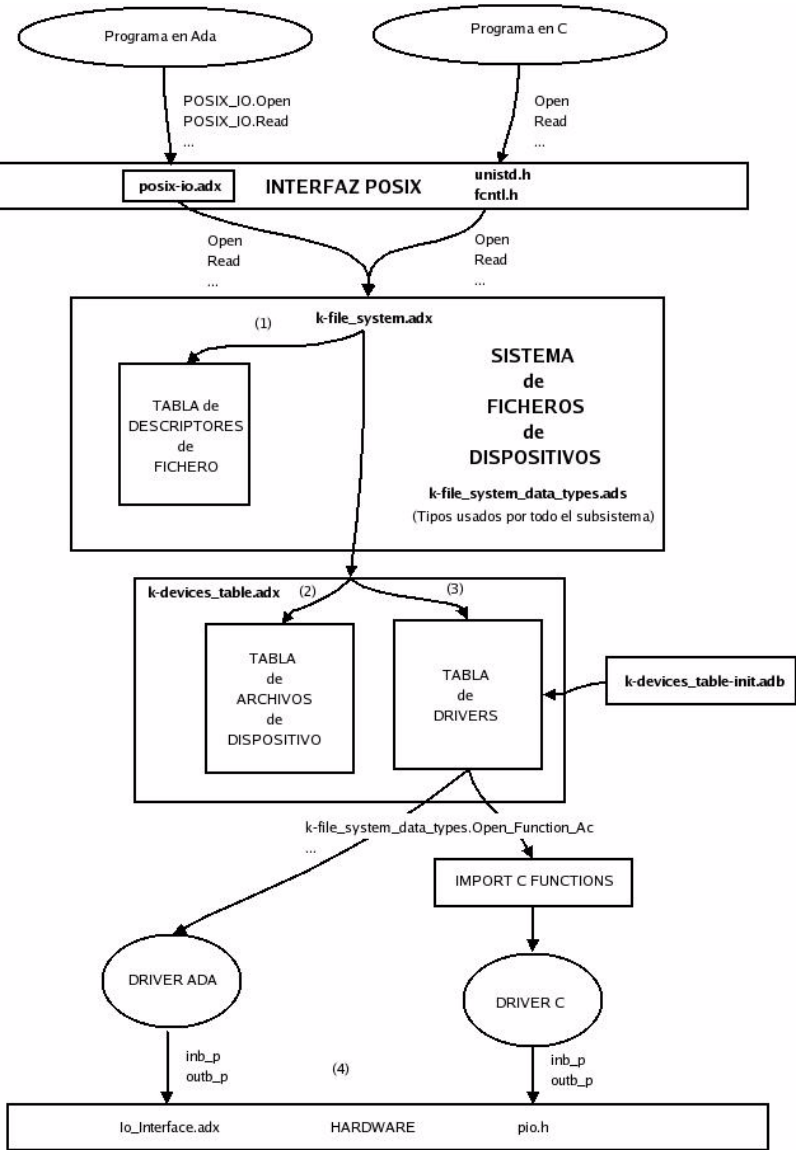


Figura 2.3: Arquitectura de los drivers en MaRTE OS

Los dispositivos periféricos funcionan a través del control de un conjunto de registros propios. En el manual de cada dispositivo se encuentra la funcionalidad de cada registro, y la forma de manejarlo. Al final, los dispositivos siempre se controlan a través de escrituras y lecturas sobre estos registros. Es por esto que cuando se realiza un manejador para un dispositivo en particular, en la parte de más bajo nivel del código existen un conjuntos de instrucciones trabajando directamente sobre estos registros. La función del *driver* es esconder este mecanismo de control al usuario y ofrecerle una interfaz fácilmente manejable para interactuar con el dispositivo.

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.

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 sufijo _p a los nombres de las instrucciones: `inb_p` y `outb_p`.

2.2. Entorno de trabajo. Cargando MaRTE OS

Lo habitual en los sistemas empotrados es que el desarrollo de las aplicaciones se realice en un entorno cruzado formado por dos computadores: el computador empotrado, sobre el que se ejecutarán las aplicaciones y el equipo de desarrollo, generalmente más potente y con un sistema operativo de propósito general, en el que se dispone de un conjunto de herramientas que posibilitan la creación, carga en el equipo de ejecución y depuración de las aplicaciones.

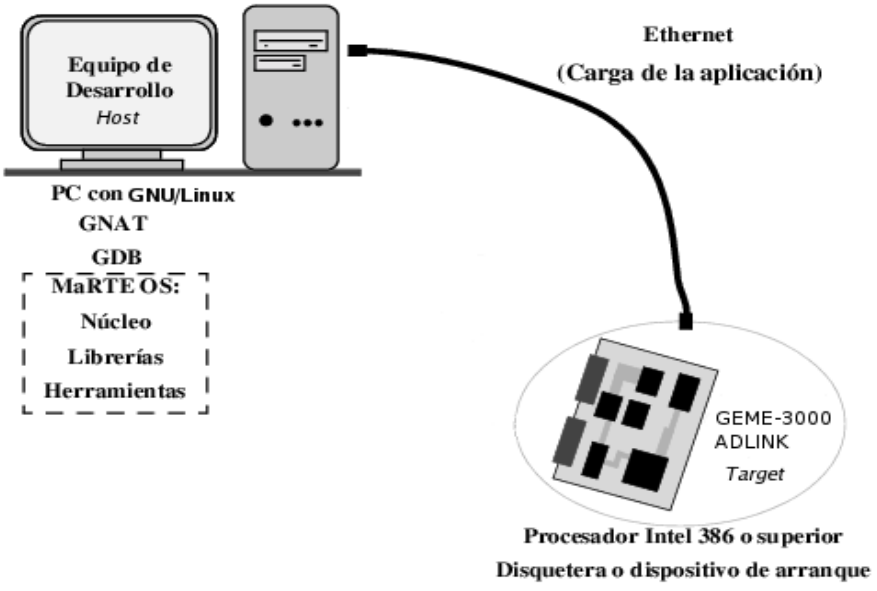


Figura 2.4: Entorno de desarrollo

El equipo de desarrollo es un PC con sistema operativo Linux en el que se encuentran instalados los compiladores GNAT y GCC. También se encuentra el núcleo y el resto de librerías que componen el sistema operativo MaRTE OS.

El proceso de creación de una aplicación consiste en la compilación de su código y su posterior enlazado con las librerías que componen MaRTE OS. Los archivos de órdenes `mgcc` y `mgnatmake` se encargan de automatizar este proceso mediante la invocación de los compiladores GNAT y GCC con las opciones apropiadas. Su modo de utilización es similar al de los comandos `gcc` y `gnatmake`, aceptando prácticamente las mismas opciones que éstos.

La aplicación se compila y enlaza en el equipo de desarrollo y se copia en el directorio desde el que será cargada en el computador de ejecución. En el momento que se arranque este equipo remoto, se localiza el ejecutable a través de una red *ethernet*, se descarga y así empieza a correr la aplicación sobre MaRTE OS. En sistemas autónomos, una vez desarrollada la aplicación, es posible copiarla en una memoria *FLASH* desde la que el dispositivo empotrado puede arrancar sin necesidad de estar conectado a la red.

2.2.1. Computador de ejecución GEME-3000

GEME-3000 de ADLINK[7] es el equipo que utilizamos como computador remoto. Sus características principales son:

- CPU: *Low Voltage Intel Pentium III* 800MHz.
- 512KB de memoria caché.
- 256 MB de memoria SDRAM.
- 2 puertos USB 1.1.
- Un puerto paralelo de alta velocidad.
- Conector PS/2 para teclado y/o ratón.
- Conector para disquetera.
- Dos puertos serie, COM1 y COM2, que utilizan la UART 16550 compatible con la interfaz RS-232.
- Interfaz PC/104[6], permitiendo aumentar el número de tarjetas de este tipo, acoplándose una sobre otra.
- Ranura para tarjeta Compact Flash.

Para establecer la red entre el equipo remoto y el *host*, deberemos configurar un servidor DHCP y un servidor NFS en el equipo de desarrollo. El arranque se realiza utilizando una tarjeta Compact Flash en la que se encuentra grabado el programa de arranque por red (*Etherboot*). Este programa realiza una petición por la red (utilizando el protocolo DHCP) que es contestada por el servidor. En respuesta a dicha petición, el servidor envía al computador de ejecución su dirección IP. Posteriormente se solicitará el programa ejecutable utilizando el protocolo NFS. Etherboot pone el programa ejecutable en ejecución, momento en el que se carga MaRTE OS y, a continuación, se ejecuta el código del usuario.

En la página oficial de MaRTE OS [4], podemos encontrar un documento[5] que explica detenidamente cómo cargar MaRTE OS en un equipo empotrado.

3. Driver para la tarjeta de 8 puertos serie Parvus COM-1274-A1

3.1. Hardware

La tarjeta COM-1274-A1 de Parvus es un módulo PC/104[6] desarrollado para soportar múltiples tareas de comunicación en un sistema embebido. Está pensada para un mínimo tamaño y consumo de potencia. Al ser un módulo PC/104, el montaje en un PC resulta muy sencillo, concretamente en nuestro equipo de ejecución GEME-3000 de ADLINK[7], basta con desatornillar el módulo, abrirlo para ver el interior, y encajar la tarjeta en el bus ISA del equipo.

Esta tarjeta viene con su correspondiente manual de usuario (también accesible por internet [8]), así como *software* para manejarla. Por tanto la tarea del programador de *drivers* se facilita si este manual está bien hecho. Tan sólo necesitaremos diseñar una buena interfaz de usuario, y en la implementación leer con detenimiento el manual para trabajar con la tarjeta adecuadamente. Las principales características de esta tarjeta son:

- 8 puertos serie configurables por *software* para los modos RS-232, RS-422 o RS-485.
- Máxima velocidad programable de 115 Kbaudios.
- Controlador I/O: 16C554C, formado por 4 UARTs (*universal asynchronous receiver and transmitter*).
- Hasta 2 puertos CAN.
- Bus ISA para PC/104.

Nuestro objetivo es proporcionar al usuario una interfaz sencilla y transparente para poder hacer uso de los ocho puertos serie trabajando en modo RS-232.

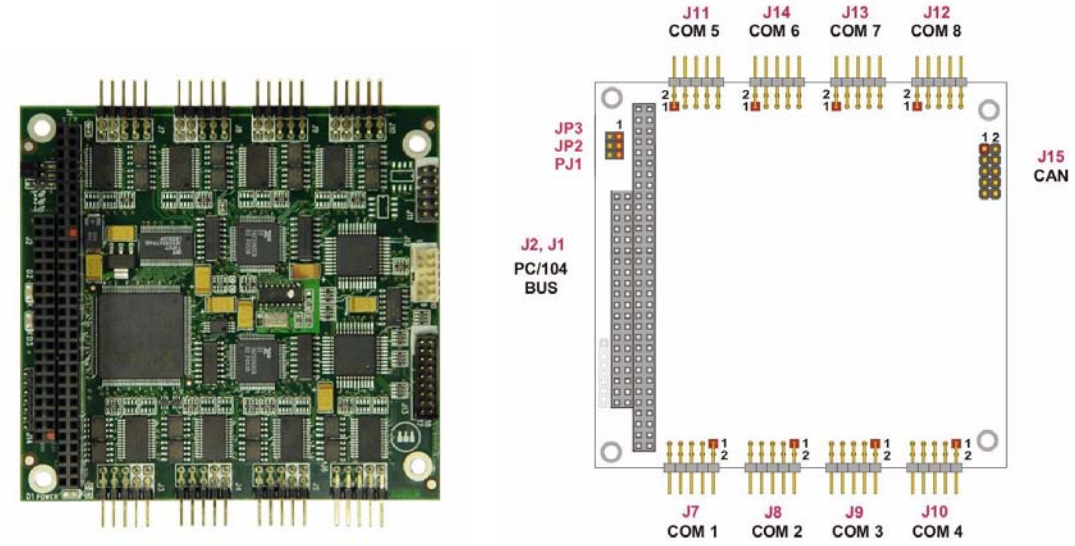
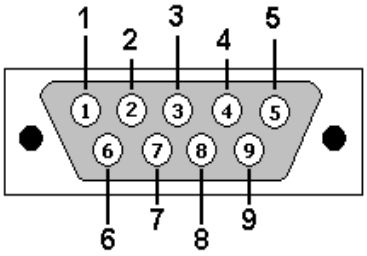


Figura 3.1: Tarjeta Parvus COM-1274, 8 puertos serie

3.1.1. Interfaz RS-232

La transmisión serie es un mecanismo de comunicación de datos codificados en bytes entre dos equipos y en el que los bits de cada byte se transmiten sucesivamente en el tiempo a través de un solo par de hilos físicos. El mecanismo de transferencia de un byte vía línea serie se inicia por la transferencia en paralelo desde un *buffer* de transmisión a un registro de desplazamiento que transfiere el dato como una sucesión de bits en serie de acuerdo con la frecuencia del reloj de transmisión. El proceso inverso de recepción es similar, los bits según van llegando son recogidos en un registro de desplazamiento de acuerdo con la frecuencia del reloj de recepción, y cuando ya ha llegado el byte completo se transfiere en paralelo al *buffer* de recepción, de donde el programa debe leerlo.

En ocasiones se utiliza un conjunto de líneas *hardware*, a través de las que se mantiene un protocolo de control del flujo de datos. En la figura 3.2 se muestran las líneas definidas en el estándar RS-232 para un conector de 9 pines:



pin DB 9	Signal	Function	In/Out
1	DCD	*Data Carrier Detect*	In
2	RX	*Receive Data*	In
3	TX	*Transmit Data*	Out
4	DTR	*Data Terminal Ready*	Out
5	GND	*Signal Ground*	--
6	DSR	*Data Set Ready*	In
7	RTS	*Request To Send*	Out
8	CTS	*Clear To Send*	In
9	RI	*Ring Indicator*	In

Figura 3.2: Frontal de un conector DB 9 macho

El puerto serie RS-232 (también conocido como COM), utiliza cableado simple desde 3 hilos hasta 25 y conecta computadores o microcontroladores a todo tipo de periféricos, desde terminales a impresoras y *modems*. En el manejador que queremos realizar no se utiliza ninguna de las líneas de control de flujo de datos *hardware* y el control de flujo se lleva a cabo a nivel de enlace por el *software* de comunicaciones.

La interfaz entre el RS-232 y el microprocesador generalmente se realiza mediante una UART 8250 (computadoras de 8 y 16 bits, PC XT) o 16550 (*IBM Personal Computer/AT* y posteriores). Se controla desde la CPU a través de un conjunto de registros[9] mapeados sobre el bus I/O. En la tabla 3.1se enumeran estos registros.

Tabla 3.1: Registros del puerto serie. UART 16550

Offset	Modo	DLAB	Registro	Nemónico
+0	Escritura	0	*Buffer* de transmisión	TX
+0	Lectura	0	*Buffer* de recepción	RX
+0	Escritura	1	*LSB Latch divisor*	DLL
+1	Escritura	1	*MSB Latch divisor*	DLM
+1	Escritura	0	Habilitación de interrupciones	IER
+2	Lectura	-	Identificación de interrupciones	IIR
+3	Escritura	-	Control de línea	LCR
+4	Escritura	-	Control de módem	MCR
+5	Lectura	-	*Status* de línea	LSR
+6	Lectura	-	*Status* de módem	MSR

3.1.2. Tarjeta Parvus COM-1274-A1 configuraciones y registros

La tarjeta COM-1274 permite configurar mediante *jumpers hardware* la selección de su dirección base. Para ello se utilizan los *jumpers* JP1 y JP2 que observamos en la figura 3.1. Con ellos podemos escoger entre cuatro direcciones de entrada/salida: 0x110, 0x150, 0x1A0, 0x1E0 (escritas en hexadecimal). Esta configuración *hardware* se registrará en nuestro *driver* de forma estática mediante constantes. Es decir, es el usuario de la tarjeta el que según qué configuración *hardware* desee deberá modificar los valores de dichas constantes.

Pero no por escoger la dirección base de la tarjeta a través de los *jumpers*, estamos escogiendo también la dirección base de los puertos serie. En realidad esta segunda dirección es configurable por *software*. El fabricante proporciona un *software* específico para cambiar la configuración de la tarjeta. Entre otras cosas se puede modificar el valor de esta dirección base de los puertos series y el modo de funcionamiento, ya sea RS-232, RS-422, RS-485 ó deshabilitado; pero en ningún caso este programa sirve para manejar la tarjeta, para eso están los *drivers* específicos de la misma.

El fabricante no proporciona el código fuente de este *software* de configuración. Es compresible ya que funciona para los sistemas operativos Windows y Linux, pero claro está, no para MaRTE OS que es lo que nos interesa. Al no tener los ficheros fuente no somos capaces de descifrar cómo se escribe en los registros de control para realizar cualquier cambio en la configuración que viene por defecto. Nos conformamos con la siguiente configuración por defecto ya que es válida para nuestro equipo:

- La dirección base de la tarjeta tiene valor 0x150.

- La dirección base de los puertos serie es 0x280.
- Todos los puertos serie se encuentran habilitados, utilizan la IRQ 10 y están trabajando en modo RS-232.
- Las interfaces CAN se encuentran deshabilitadas.

Además del programa de configuración, el fabricante proporciona el manejador de la tarjeta para la mayoría de sistemas operativos comunes. Para poder realizar nuestro *driver*, es necesario conocer los registros con los que manejarla. Desafortunadamente no aparece nada al respecto en el manual de la tarjeta.

Lo único que se encuentra en el manual es el espacio de direcciones requerido para las puertos serie a partir de la dirección I/O de los mismos. Este espacio es de 64 bytes, es decir 8 bytes por cada puerto.

Por suerte, sí que tenemos el código fuente de los *drivers* del fabricante y gracias a este código se puede conocer el *offset* entre los registros. Cada puerto serie a pesar de tener 8 registros reservados, tiene el mismo funcionamiento que los puertos habituales COM1, COM2, COM3 y COM4 (que utilizan los vistos en la tabla 3.1). Éste es un dato que se podía suponer porque la UART que utiliza es la 16C554C [apartado 3.1], y sabemos que los computadores modernos de tipo PC tienen el circuito 16550.

Si el funcionamiento es el mismo que el de un puerto serie común, lo primero es intentar adaptar el código del *driver* de MaRTE OS para el puerto serie, probar algunos de los puertos de nuestra tarjeta y ver qué sucede.

3.1.3. Montaje y pruebas iniciales

En los directorios *x86_arch/drivers/serial_port* y *x86_arch/include/drivers* de MaRTE OS se encuentran todos los ficheros donde se implementa el *driver* para el puerto serie, escrito en lenguaje C.

Lo primero es realizar un sencillo archivo de prueba, `test_serial_port_c.c`, para probar la comunicación de los COM1 y COM2 habituales del equipo, no los de nuestra tarjeta. Ambos puertos envían y reciben un array de caracteres. Con este test comprobamos que el *driver* hecho en MaRTE OS para puertos serie funciona correctamente en nuestro equipo GEME-3000.

Queremos reutilizar este código para alguno de nuestros puertos serie. La variable "`ser_io_base`" del fichero `serial_port_driver_c.c` contiene las direcciones I/O para los puertos COM. Este array debe contener ahora, en alguno de sus elementos, la dirección base de los puertos series en nuestra tarjeta, 0x280. Además la variable "`IRQ_number`" toma el valor del número de interrupción que se establece en la configuración por defecto, IRQ10, en el mismo elemento del array que en la variable "`ser_io_base`", ya que ambas posiciones se refieren al mismo puerto.

Para realizar esta prueba se conecta el COM1 de nuestra tarjeta, con el COM2 habitual del equipo a través de un cable RS-232 cruzado fabricado por nosotros, cruzando los pines transmisor y receptor y uniendo las tierras [figura 3.2 y 3.3].

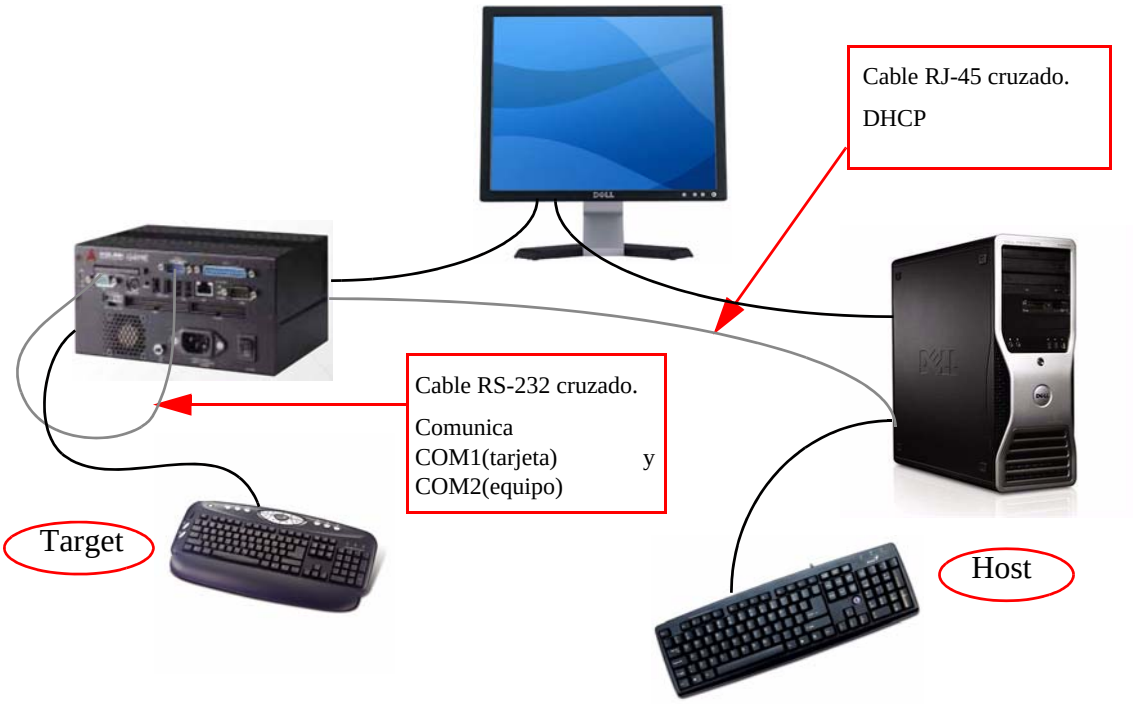


Figura 3.3: Entorno de trabajo

El objetivo es hacer funcionar el test anterior `test_serial_port_c.c` comunicando a través de los puertos de la tarjeta.

En este proceso se crea otro fichero de prueba, `test_status.c`, capaz de leer y mostrar por pantalla la configuración guardada en la EEPROM de la tarjeta. La información necesaria para aprender a leer de la EEPROM la sacamos profundizando en el *driver* que proporciona el fabricante para la tarjeta funcionando bajo el sistema operativo Linux. Gracias a este test, comprobamos que efectivamente la configuración que el manual define por defecto y que hemos visto en el apartado 3.1.2 es correcta.

Para conseguir establecer la comunicación a través del `test_serial_port_c.c` es necesario habilitar la interrupción en la BIOS del equipo donde estamos corriendo el test. Se entra en la opción "*PnP/PCI Configurations*" de la BIOS, ahí en"*IRQ Resources*", y se escoge para la interrupción IRQ10 la opción "*Legacy ISA*"[11], en lugar de "*PCI/ISA PnP*"[10].

Tras todos estos cambios se consigue comunicar el COM1 de la tarjeta y otro puerto serie del equipo, probando la comunicación en ambos sentidos.

3.2. Software

Una vez que nos hemos documentado bien sobre la tarjeta, sus características y requerimientos, es hora de crear el *software*. Gracias al correcto funcionamiento de las pruebas iniciales, vamos a intentar reutilizar toda la parte posible del código del *driver* para los puertos serie hechos en MaRTE OS. Este *driver* fue realizado inicialmente en 2003 por Francisco Guerreira en su proyecto fin de carrera[12] para obtener el título como Ingeniero de Telecomunicación. La versión actual que tiene el sistema operativo es una versión mejorada realizada por Mario Aldea en 2005, en la que se controla el puerto serie por medio de las interrupciones. Para nuestro caso,

nos ayudamos de la versión más actualizada, teniendo en cuenta cualquier tipo de documentación de ésta o anteriores versiones que pueda servirnos de ayuda.

Como esta información tomada de MaRTE OS se encuentra escrita en C, por ahorrar tiempo la nueva versión de *driver* para puertos serie también estará escrita en el mismo lenguaje. En cualquier caso, si hubiese estado escrita en Ada, hubiese sido necesario exportarlo a C, porque es el lenguaje que necesita la empresa interesada en el producto, Fundación Centro Tecnológico de Componentes.

3.2.1. Diseño de la arquitectura e implementación

Con respecto a la implementación y al sistema de archivos de dispositivos de MaRTE OS, hay que destacar el hecho de que con este *driver* es necesario el uso de los números menores para poder distinguir qué puerto serie vamos a utilizar. Son estos números menores los que permiten diferenciar las direcciones de memoria base asociada a cada unos de los puertos.

Tabla 3.2: Direcciones de E/S de los puertos serie

Puerto series habituales en un computador								
Puerto	COM1	COM2	COM3	COM4				
Dirección	0x3F8	0x2F8,	0x3E8	0x2E8				
Puerto series pertenecientes a la tarjeta Parvus COM-1274								
Puerto	COM1	COM2	COM3	COM4	COM5	COM6	COM7	COM8
Dirección	0x280	0x288	0x290	0x298	0x2A0	0x2A8	0x2B0	0x2B8

A pesar de que ya existía un *driver* para los puertos series en MaRTE OS, en esta nueva versión los hemos vuelto a incluir. Por tanto el *driver* comprende el funcionamiento de los ocho puertos serie de la tarjeta COM-1274 y los cuatro (COM1-COM4) que ya funcionaban.

Los archivos relacionados con este *driver* son los siguientes:

- `serial_port_driver.h`
- `serial_port_driver_c.c`
- `serial_port_driver_import.ads/adb:`
- `serial_port_driver.ads`

El *driver* presenta una especificación que se ofrece al usuario a través de un fichero de cabeceras, `serial_port_driver.h`, en el que se definen toda una serie de tipos de variables y constantes que van a permitir manejar todas las opciones que presenta el puerto serie. Haciendo uso de los atributos de control, podemos escoger el número de bits que constituyen el dato, el número de bits de stop, habilitar/deshabilitar las interrupciones y seleccionar la paridad. También se encuentra definidas las constantes a utilizar en la llamada a la función '`ioctl`'. Esta función es la que ofrece la capacidad de control sobre el dispositivo, permite abordar todas las variantes que ofrece la programación del puerto serie. Su implementación es bastante densa puesto que supone tener que programar los registros de la UART. Las opciones principales que ofrece son establecer ('SETATTR') y obtener ('GETATTR') los atributos de configuración del puerto; establecer ('SETSPEED') y leer ('GETSPEED') de forma directa la velocidad de comunicación del puerto; habilitar las interrupciones ('EINTERRUPT'); y una opción para

vaciar el *buffer* de datos (‘FLUSH’). En este archivo se encuentra la información necesaria para que el usuario pueda hacer un uso correcto del *driver*, sin tener que conocer a fondo cómo está éste hecho.

En el fichero ‘`serial_port_driver_c.c`’ se encuentra el desarrollo del *driver*. Vamos a analizar las funciones principales:

- `serial_port_create()`: esta función se ejecutará cada vez que se inicialice MaRTE OS, siempre que el *driver* se encuentre en la tabla del fichero `marte-kernel-devices_table.ads`. En ella se inicializa un *buffer* de entrada y uno de salida para cada puerto serie, existiendo un total de doce puertos serie. Cada vez que el usuario quiera transmitir o recibir información por el puerto serie lo hará a través de estos *buffers*, que están implementados en el paquete `ring-buffers.h` de MaRTE OS, común en *drivers* de entrada salida. Concretamente en esta función se utilizan los métodos:
  - `void init_input_buffer(ring_input_buffer_id_t buff_id)`
  - `void init_output_buffer (ring_output_buffer_id_t buff_id)`
- `serial_interrupt_handler(void * area, intr_t intr)`: el manejador de interrupciones controla el funcionamiento de la UART a través de las interrupciones.

  En la tabla 3.3 tenemos el registro de interrupciones de las UART 16550. Leyendo este registro se puede identificar la causa que ha generado una interrupción asociada con la transmisión y recepción de datos por el puerto serie. El bit menos significativo b_0 indica si existe una interrupción pendiente. Los dos bits restantes indican cuál es la interrupción de mayor prioridad pendiente. En la tabla 3.4 se muestra el significado de estos bits.

Tabla 3.3: Registro de identificación de interrupciones del puerto serie

Registro IIR, offset #2. Registro de lectura							
0_{b7}	0_{b6}	0_{b5}	0_{b4}	0_{b3}	$IDI1_{b2}$	$IDI0_{b1}$	$\overline{IP}_{b0}$

Cuando ocurre una interrupción de ‘estado de línea’(#6) el procedimiento a seguir para tratarla es leer el registro de status de línea, LSR.

Si salta la interrupción ‘dato recibido disponible’(#4) se lee el *buffer* de recepción y seguidamente el bit menos significativo del registro LSR. Este bit recibe el nombre de ‘dato recibido dispuesto’ (DR), y toma valor uno cuando se recibe un nuevo dato en el *buffer* de recepción. Esto se hace porque mientras leíamos el primer dato que provocó la interrupción pudo haber llegado otro nuevo dato que no provocara interrupción. Por esto se lee este bit y en caso de encontrarse a valor alto, se lee nuevamente el *buffer* de recepción, y así continuamente hasta que el DR tome valor cero.

Cuando salta la interrupción ‘*buffer* de transmisión libre’(#2), lo primero que se hace al atender esta interrupción es leer el bit DR que acabamos de explicar y por el mismo motivo, por si llegó un dato y no saltó la interrupción, ya que es prioritario atender los datos recibidos. Posteriormente escribimos en el *buffer* de transmisión.

Puesto que en nuestro sistema no hacemos uso de las líneas de control, en el caso de que se produjese una interrupción ‘*status modem*’ (#0) el manejador informaría al sistema operativo de que esa interrupción no está manejada, y en caso de que no estuviese

manejada por ningún otro dispositivo se mostraría un mensaje por pantalla indicando la interrupción producida.

Tabla 3.4: Interrupciones del puerto serie

b_2	b_1	b_0	Prioridad	Tipo	Origen	Cancelación
0	0	1	-	Ninguna	-	-
1	1	0	Más alta	Status de línea	Error *overrun* Error paridad Error marco *Break*	Lectura LSR
1	0	0	Segunda	Dato RX disponible	Dato RX disponible	Lectura *buffer* RX
0	1	0	Tercera	*Buffer* TX libre	*Buffer* TX libre	Lectura IIR Escritura *buffer* TX
0	0	0	Más baja	*Status* de módem	*Clear to send* *Data set ready* *Ring Indicator* *Carrier detect*	Lectura MSR

En el manejador de interrupción cabe destacar que es el argumento `void *area`, el que da la información de qué puerto serie es, en el que se ha producido la interrupción. Este valor coincide con el número menor asignado a cada puerto.

- `serial_port_open (int file_descriptor, int file_access_mode)`: Se asocia el manejador de interrupciones a la interrupción correspondiente a cada puerto serie. Esto es, los puertos de la tarjeta Parvus COM-1274 se asocian a la interrupción por defecto de la tarjeta que hemos dicho que era la IRQ10, los COM1 y COM3 se asocian a la interrupción IRQ4 y COM2 y COM4 a la IRQ3.

  Además se establece una configuración por defecto para cualquier puerto serie que se abra: 8 bits por palabra, 1 bit de stop, sin paridad, velocidad de transmisión de 9600 baudios e interrupciones habilitadas.

- `ssize_t serial_port_read (int file_descriptor, void *buffer, size_t bytes)`: Cuando queremos leer información recibida por el puerto serie, se lee el *buffer* de entrada que se inicializó en el *create*. Como hemos explicado antes, la función manejadora de interrupción `serial_interrupt_handler`, es la encargada de que cada vez que se recibe un dato por un puerto, sea guardado en el *buffer* de entrada correspondiente al puerto. De esta forma, cuando el usuario desee leer la información que ha llegado por un puerto serie, es suficiente con que lea el *buffer* de entrada de ese puerto serie.

- `ssize_t serial_port_write (int file_descriptor, void *buffer, size_t bytes)`: A través de la función `write_to_output_buffer` se escribe en el *buffer* de salida del puerto. Esta función se encarga de depositar el primer carácter directamente en el *buffer* de transmisión del puerto serie, registro TX de la tabla 3.1, y el resto en el *buffer* de salida que inicializamos en el *create* y pertenece al paquete `ring-buffers`. De esta forma al depositar el primero de los caracteres en el registro TX del puerto serie, las interrupciones se van desencadenando una tras otra hasta enviar todo la información que

se desea. Es entonces el manejador de interrupción, como vimos antes, quien se encarga de sacar cada dato del *buffer* de salida y enviarlo por el puerto serie.

- `int serial_port_ioctl (int file_descriptor, int request, void* argp)`: ya se han explicado las opciones de control que establece esta función cuando hablamos del fichero `serial_port_driver.h`, pero destacaremos la de poder cambiar los atributos de configuración del puerto serie con SETATTR.

En el fichero `serial_port_driver_import.ads/adb` se importan las funciones creadas en el desarrollo del *driver* al lenguaje Ada, y en `serial_port_driver.ads` se definen un conjunto de constantes necesarias para hacer posible la importación de las funciones. Algunas de ellas son las mismas que encontrábamos en el fichero `serial_port_driver.h`.

3.2.2. Pruebas e integración

En el apartado 3.1.3, vimos cómo es posible hacer funcionar el puerto COM1 de la tarjeta, reutilizando al máximo el *driver* ya existente en MaRTE OS para el puerto serie. Lo que nos interesa ahora es probar el correcto funcionamiento de todos los puertos de la tarjeta.

Se crea un fichero de prueba muy similar al que ya teníamos, `test_serial_port_driver_c.c`, donde cada uno de los puertos de la tarjeta establece comunicación con el COM1 y con el COM2 del nuestro cubo GEME-3000.

No conseguimos comunicar a través del par J9-J10 (COM3- COM4, ver figura 3.1), por eso creamos el fichero `test_lectura_mem.c`, con el fin de mostrar por pantalla todos los registros del puerto serie vistos en la tabla 3.1. Obtenemos los siguientes valores:

Tabla 3.5: Lectura de los registros de puertos series de la tarjeta COM-1274

	TX-RX / DLL	DLM/ IER	IIR	LCR	MCR	LSR	MSR	-
COM1 0x280	0	0	1	0	0	96	0	255
COM2 0x288	0	0	1	0	0	96	0	255
COM3 0x290	255	255	255	255	255	32	72	255
COM4 0x298	255	255	255	255	255	255	255	255
COM5 0x2A0	0	0	1	0	0	96	0	255
COM6 0x2A8	0	0	1	0	0	96	0	255
COM7 0x2B0	0	0	1	0	0	96	0	255
COM8 0x2B8	0	0	1	0	0	96	0	255

Vemos en esta lectura de memoria que para los puertos 3 y 4 los registros toman valores muy diferentes. Pero en la interpretación de estos valores no obtenemos información de por qué hay un mal funcionamiento. Se resetean las interrupciones, pero el funcionamiento sigue siendo incorrecto.

Después de haber revisado la documentación, de haber hecho variaciones en los tests de pruebas y no obtener ninguna explicación llegamos a dos posibles causas del problema:

- Las direcciones 0x290-0x29F están siendo utilizadas por algún otro dispositivo en nuestro equipo.
- La tarjeta no funciona correctamente para estos puertos serie.

Hablamos con los encargados del proyecto en el Centro Tecnológico de Componentes, por si tuviesen otra tarjeta igual y poder probarla en nuestro equipo, pero ésto no es posible. Son ellos quienes prueban en su equipo la tarjeta y el *software* recién fabricado por nosotros. En su caso el puerto serie que no funciona es otro, el COM1 (J7), y el resto responden al test comunicándose correctamente. Este hecho hace que descartemos la segunda de las posibles causas que originaban el problema. Efectivamente la tarjeta funciona correctamente, y es nuestro equipo el que ocupa parte de esas direcciones de memoria de entrada/salida, ocasionando el mal funcionamiento de la tarjeta.

Se da por finalizado así el *driver* para la tarjeta Parvus COM-1274, teniendo en cuenta que al menos con nuestro equipo GEME-3000 hemos conseguido hacer funcionar 6 puertos serie de los ocho que tiene la tarjeta. Es un número suficiente de puertos serie, ya que en el sistema de navegación para el que se desarrolla este proyecto sólo se necesitan 6 puertos serie (4 para los receptores GPS, 1 para comunicaciones con dispositivos externos y uno para depuración).

Es importante recordar que, desafortunadamente, como no se dispone de información para cambiar las direcciones de memoria de la tarjeta, éste es un problema que no tiene solución actualmente.

4. Subsistema de gestión del bus serie SPI

4.1. Hardware

El objetivo de este trabajo es ofrecer nuevas posibilidades de interaccionar con el exterior a los equipos que utilicen MaRTE OS. Ya hemos visto en el capítulo 3, cómo utilizando la tarjeta Parvus COM-1274, podemos disponer de hasta 8 puertos serie además de los que ya tenga nuestro equipo, normalmente dos puertos, COM1 y COM2. Los *buses* serie son una buena forma de conseguir conectar todo tipo de periféricos. Existe ya una interfaz I2C en MaRTE OS, un bus serie que utilizan multitud de dispositivos. Dicha interfaz fue realizada por Daniel Sangorrín como parte de su proyecto fin de carrera para obtener el título como Ingeniero de Telecomunicación[13]. Este capítulo está dedicado al desarrollo de un nuevo bus de comunicaciones, el bus serie SPI. Se trata de crear un Interfaz SPI capaz de emular el protocolo independientemente del adaptador que usemos, consiguiendo así una estructura genérica a la que se puedan añadir con facilidad distintos adaptadores a los que conectar dispositivos periféricos SPI.

Existen varias diferencias entre el bus SPI y el I2C. El bus SPI consume más terminales, 4 hilos como mínimo, frente a los dos, sin contar la tierra, del I2C. El direccionamiento se hace mediante líneas específicas a diferencia de lo que ocurre en I2C, en el que se selecciona cada chip mediante una dirección de 7 bits que se envía por las mismas líneas del bus. No hay señal de asentimiento. El *maestro* podría estar enviando información sin que estuviese conectado ningún *esclavo* y no se daría cuenta de nada. Pero aparte de estas desventajas frente al bus I2C, tiene otras ventajas. Es posible realizar una comunicación *full duplex y* se pueden alcanzar mayores velocidades de transmisión. Estamos ante un protocolo flexible, ya que es posible elegir el tamaño de la trama de bits, su significado y propósito. Además consume menos energía que el I2C debido que posee menos circuitos y éstos son más simples. En definitiva es muy interesante que el sistema operativo MaRTE OS posea ambas buses, teniendo en cuenta lo útiles que son cuando se trabaja con equipos empotrados.

Como adaptador para soportar el protocolo vamos a utilizar la tarjeta Acces 104-AIO12-8. En ella encontramos un conversor analógico/digital, un conversor digital/analógico y tres puertos de entrada/salida digital llamados A, B y C. Cada puerto está formado por ocho bits, donde cada bit es una entrada/salida digital. Lo que hace un total de 24 entradas y salidas digitales, con las que representaremos cada una de las líneas del protocolo.

A lo largo del capítulo conoceremos el funcionamiento del protocolo, el diseño de la interfaz, y la forma de añadir diferentes adaptadores sobre los que soportar el bus SPI. Finalmente para probarlo vamos a crear un *driver* para un periférico SPI, que haga uso de nuestra interfaz. Se trata del sensor inercial ADIS16355 de *Analog Devices*, capaz de medir velocidades y aceleraciones angulares en los tres ejes con alta precisión.

4.1.1. Estándar SPI

SPI es un bus de al menos cuatro líneas [14], sobre el cual se transmiten paquetes de información cuyo tamaño puede ser múltiplo de ocho bits. Cada una de estas líneas transporta la información entre los dispositivos conectados al bus. Cada dispositivo conectado al bus puede actuar como transmisor y receptor al mismo tiempo, por lo que este tipo de comunicación serie es *full duplex*. Dos de estas líneas transfieren los datos (una en cada dirección), la tercera línea es la del reloj. Algunos dispositivos sólo pueden ser transmisores y otros sólo receptores; generalmente un dispositivo que transmite datos también puede recibir.

Un ejemplo podría ser una memoria EEPROM, el cual es un dispositivo que puede transmitir y recibir información.

Los dispositivos conectados al bus se definen como maestros y esclavos. Un maestro es aquel que inicia la transferencia de información sobre el bus y genera las señales de reloj y control. Un esclavo es un dispositivo controlado por el maestro. Cada esclavo es controlado sobre el bus a través de una cuarta línea selectora llamada *Chip Select* o *Select Slave*, por lo tanto el esclavo es activado sólo cuando esta línea está seleccionada. Generalmente se dedica una línea de selección para cada esclavo. Por eso el bus tiene al menos cuatro líneas, esto sucede cuando solamente hay un esclavo y por tanto una única línea *Chip Select*.

En un tiempo determinado T1, solo podrá existir un maestro sobre el bus. Cualquier dispositivo esclavo que no esté seleccionado, debe deshabilitarse (ponerlo en alta impedancia) a través de la línea selectora (*chip select*) que deberá entonces estar a valor alto.

4.1.1.1. Especificaciones del Bus

Todas las líneas del bus transmiten la información en una dirección.

La señal sobre la línea de reloj (SCLK) es generada por el maestro y sincroniza la transferencia de datos.

La línea MOSI (*Master Out Slave In*) transporta los datos del maestro hacia el esclavo.

La línea MISO (*Master In Slave Out)* transporta los datos del esclavo hacia el maestro.

Cada esclavo es seleccionado por un nivel lógico bajo (‘0’) a través de la línea CS (*Chip Select*) o SS (*Slave Select*).

Los datos sobre este bus pueden ser transmitidos desde velocidades muy lentas hasta velocidades de 1 Mbits/segundo. Los datos son transferidos en bloques múltiplos de 8 bits, en donde el bits mas significativo (MSB) se transmite primero.

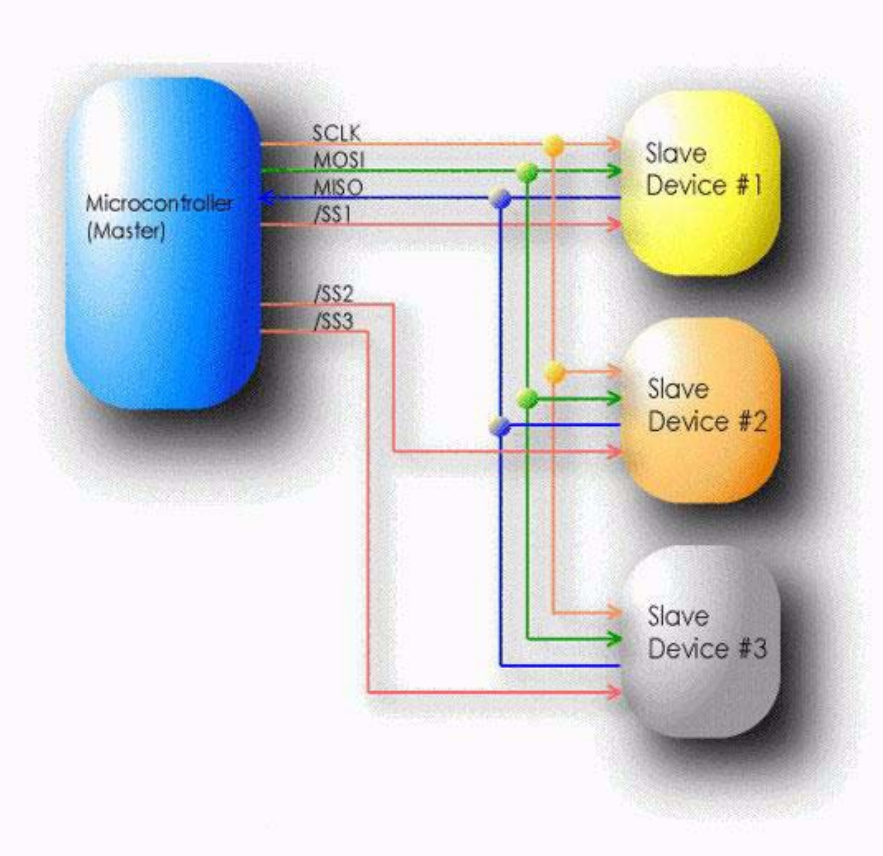


Figura 4.1: Bus Serie SPI

Toda la transferencia de los datos, es sincronizada por la línea de reloj de este bus. Se transfiere un bit por cada ciclo de reloj.

La mayoría de las interfaces SPI tienen 2 bits de configuración, llamados CPOL (*Clock Polarity* = Polaridad de Reloj) y CPHA (*Clock Phase* = Reloj de Fase). CPOL determina si el estado de reposo (*Idle)* de la línea de reloj está en bajo (CPOL=0) o si se encuentra en un estado alto (CPOL=1). CPHA determina en qué flanco de reloj los datos son desplazados hacia dentro o hacia fuera.

Si CPHA es igual a 0 los datos sobre la línea MOSI se detectan en cada flanco de bajada y los datos sobre la línea MISO se detectan en cada flanco de subida, si CPHA toma valor 1 se hace al revés.

Cada bit tiene 2 estados, lo que permite 4 diferentes combinaciones, las cuales son incompatibles una de la otra. Por ello si dos dispositivos SPI desean comunicarse entre sí, deben tener la misma Polaridad de Reloj (CPOL) y la misma fase de Reloj (CPHA).

Los cuatro modos de reloj definidos por el protocolo SPI son: modo A ó 0, modo B ó 1, modo C ó 2 y modo D ó 3.

La mayoría de los dispositivos SPI pueden soportar al menos 2 modos de los 4 antes mencionados. Los diferentes modos se ilustran a continuación. El modo requerido para una

determinada aplicación, está dado por el dispositivo esclavo. La capacidad de multi-modo combinada con un simple registro de desplazamiento hace que el bus SPI sea muy versátil.

Si CPOL vale cero, la línea de reloj debe estar a valor bajo mientras no se deseen enviar datos. Por el contrario si vale 1, la línea SCLK debe estar a valor alto.

CPHA, conjuntamente con CPOL, controlan cuándo se colocan los nuevos datos en el bus. Si CPHA es igual a un '1' lógico, los datos son desplazados sobre la línea MOSI según lo determinado por el valor de CPOL.

- Para CPHA = 1, si CPOL = 1, los nuevos datos se colocan sobre la línea cuando el flanco del reloj es descendente y se leen cuando el flanco del reloj es ascendente. Si CPOL = 0, los nuevos datos se ponen en la línea cuando el flanco del reloj es ascendente y se leen cuando el reloj tiene un flanco descendente.

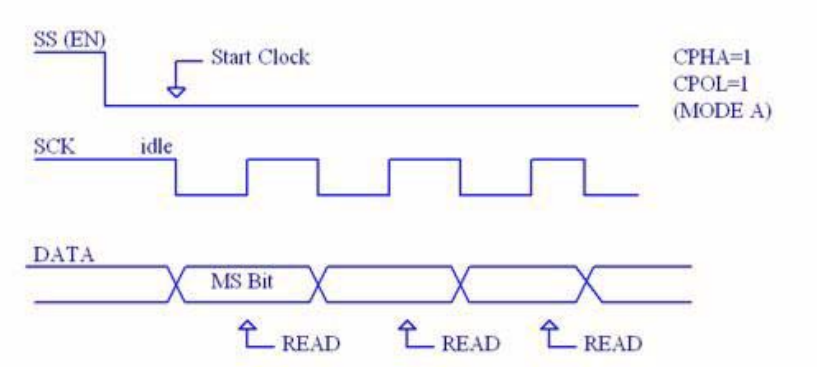


Figura 4.2: Funcionamiento SPI modo A

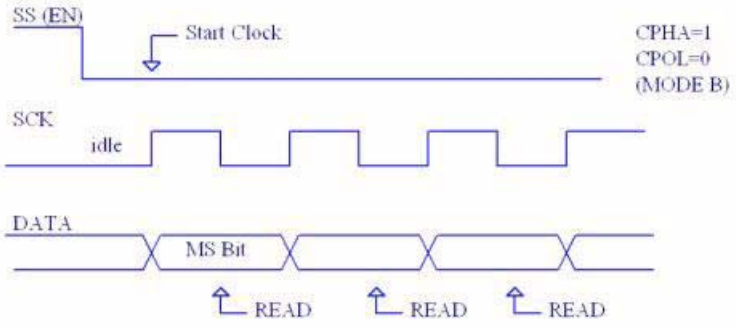


Figura 4.3: Funcionamiento SPI modo B

- Si CPHA = 0, el reloj de cambio es la operación lógica OR de SCLK con el terminal *Chip Select*. Tan pronto como el terminal *Chip Select* se coloca en un nivel lógico 0, los nuevos datos se ponen en la línea y en el primer flanco de reloj se leen los datos. Si CPOL se activa a un nivel lógico '1', el primer borde de reloj baja y los bits de datos subsecuentes se leen en cada flanco de bajada sobre la línea de reloj. Cada nuevo bit se pone en la línea cuando el reloj tiene un flanco ascendente de Reloj. Si CPOL es cero, el primer flanco de reloj ascendente y los bits de datos subsecuentes se leen en cada flanco ascendente de reloj. Cada nuevo bit se coloca en la línea cuando el flanco del reloj baja.

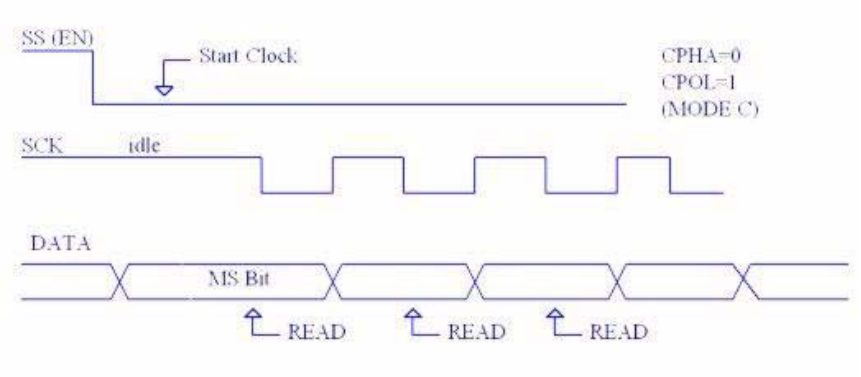


Figura 4.4: Funcionamiento SPI modo C

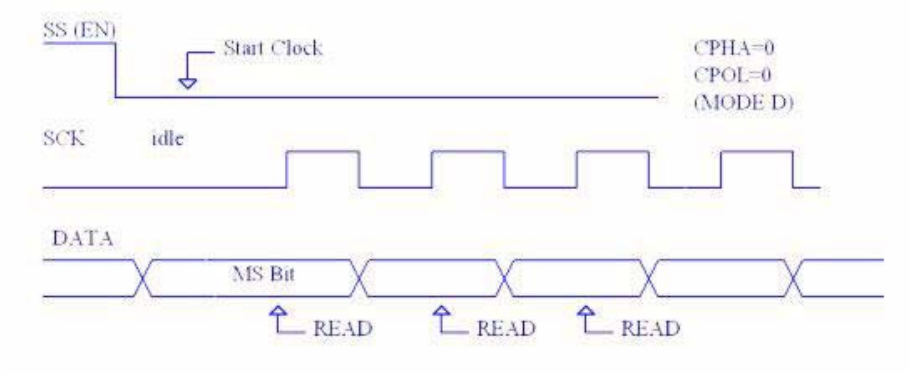


Figura 4.5: Funcionamiento SPI modo D

En resumen, Si CPHA=1, la transferencia (datos válidos leídos por el receptor) comienza en el segundo flanco de reloj. Si CPHA=0, la transferencia comienza en el primer flanco de reloj. Todas las transferencias subsecuentes dentro del byte ocurren en cada flanco de reloj.

4.1.2. Tarjeta entradas y salidas digitales Acces 104-AIO12-8-DC

En el manual de la tarjeta[15] encontramos la información necesaria para poder emular en ella el protocolo SPI.

Las características principales de la tarjeta son:

- Ocho entradas analógicas o cuatro entradas diferenciales.
- Rango de entrada programable a: 0-5V, 0-10V, +-5V, +-10V.
- Cuatro salidas analógicas.
- *Jumpers* para seleccionar el rango de salida: 0-5V, 0-10V, +-5V, +-10V.
- 24 líneas digitales de entrada salida.

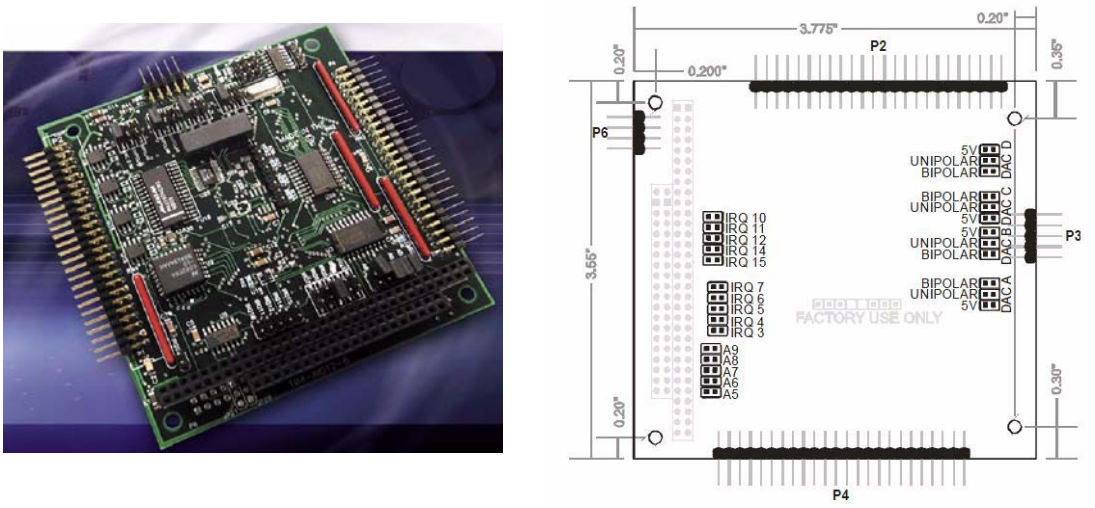


Figura 4.6: Tarjeta Acces 104-AIO12-8

Seleccionamos la dirección base de la tarjeta a través de los *jumpers* A5-A9. Nos aseguramos de escoger una dirección que esté libre en nuestro equipo para no tener luego problemas de funcionamiento. Dejamos libres los *jumpers* A9 y A8 y ponemos todos los demás, que corresponden a la dirección de entrada/salida 0x300.

Offset	Read Function	Write Function
00h	Board Status / Clear Board Status	IRQ Clear
01h	Interrupt Status	Interrupt Enables
02h	A/DC Low Byte	A/D Control / Start Conversion
03h	A/DC High Byte	
04h		D/AC 1 LSB
05h		D/AC 1 MSB
06h		D/AC 2 LSB
07h		D/AC 2 MSB
08h		D/AC 3 LSB
09h		D/AC 3 MSB
0Ah		D/AC 4 LSB
0Bh		D/AC 4 MSB
0Ch	Counter/Timer 0 Value	Counter/Timer 0 Load Value
0Dh	Counter/Timer 1 Value	Counter/Timer 1 Load Value
0Eh	Counter/Timer 2 Value	Counter/Timer 2 Load Value
0Fh		Counter Control
10h	Digital I/O Port A	Digital I/O Port A Output Value
11h	Digital I/O Port B	Digital I/O Port B Output Value
12h	Digital I/O Port C	Digital I/O Port C Output Value
13h	Digital I/O Status (Modes 1 & 2)	Digital I/O Command Byte
14h		Digital I/O Buffer Control
15h		A/DC Command

Figura 4.7: Mapa con la direcciones de registros de la tarjeta

Escogemos la configuración de las entradas y salidas digitales en el registro de offset 13 sobre la dirección base. Configuramos las puertas A y B como puertas de salida digital. Por tanto en ellas irán las líneas MOSI, SCLK y CS del protocolo. Y la puerta C se configura como puerta de entrada para emular la línea MISO. Las líneas de salida son en total 16, por lo que

Para probar la comunicación utilizamos nuestra tarjeta Acces, con la que emulamos el funcionamiento del protocolo a través de *bit banging*[16] (emulación del protocolo serie conseguida escribiendo uno por uno desde programa los bits que deseamos transmitir, así como las señales de reloj y selección necesarias), y para comprobar que el funcionamiento es el correcto le conectamos nuestro sensor inercial de tres ejes; en cuyo manual[17] encontramos unos ejemplos de comunicación. El objetivo de este apartado es probarlos hasta conseguir el correcto funcionamiento.

Tanto el bit CPOL como el bit CPHA toman valor 1 en el periférico ADIS16355. Lo que significa que responde al modo A visto en la figura 4.2, el dispositivo no acepta otros modos. Gracias al manual sabemos cómo debe ser la secuencia para poder leer un comando del dispositivo. Se trata de tramas de 16 bits, con ciertos requisitos temporales y comunicación *full-duplex*. En la figura 4.9 vemos la secuencia SPI de comunicación.

El sensor inercial es capaz de medir aceleraciones y velocidades angulares en los tres ejes. Si se hace coincidir el eje Z del sensor con la vertical, estando el dispositivo en reposo, la aceleración medida en ese eje debe ser igual al valor de la gravedad. Es decir debemos leer -1g (-9.8 m/s^2).

Un test *bit banging* consiste en un *software* que mueva las líneas simulando el protocolo deseado. En nuestro test, `test_bitbanging_spi.c`, hemos creado una función que recibe 16 bits a transmitir. Como SPI es un protocolo *full-duplex*, según se van colocando los bits en las líneas SCLK y MOSI, se va leyendo la información recibida a través de MISO, que corresponde al anterior comando solicitado.

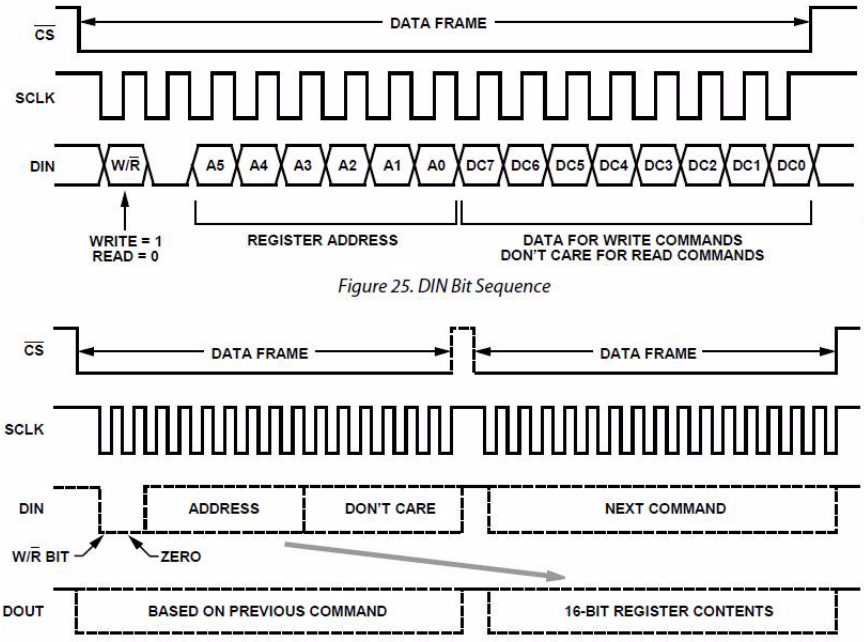


Figura 4.9: Secuencia SPI para leer comando con el ADIS16355

Para conseguir establecer comunicación con el dispositivo es necesario cumplir estrictamente los requisitos temporales. Entre ellos destaca el tiempo total de la trama, $t_{DATARATE}$, que debe ser de al menos 40 μs si estamos trabajando en modo rápido, y de 160 μs trabajando en modo normal, y el tiempo entre tramas $t_{DATASTALL}$ que ha de ser de al menos 9ns en *fast mode* y de 75ns en *normal mode*. A no ser que deseemos cambiarlo, el modo por defecto es el rápido, que permite frecuencias de reloj (`SCLK_line`) de hasta 2MHz.

A la hora de la implementación del protocolo se debe recordar que la línea de reset en el dispositivo está negada, por tanto debemos tenerla a valor alto siempre, salvo cuando tengamos interés en resetear el dispositivo.

En esta parte de pruebas nos ayudamos de un osciloscopio, muy útil para detectar posibles errores a la hora de mover las líneas mediante el test *bit banging*. Podemos observar la señales creadas y las recibidas y compararlas con los datos interpretados en el test.

A continuación se muestra la captura de estas señales con el osciloscopio, cuando estamos intentando medir la gravedad con el proceso que se explica a continuación:

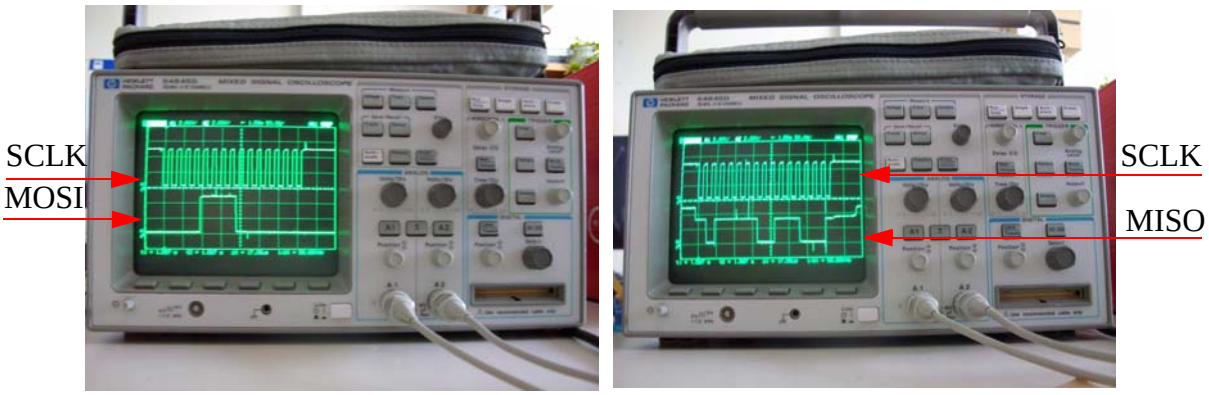


Figura 4.10: Captura con el osciloscopio de las tramas SPI en las pruebas iniciales

En la figura 4.10 por la línea MOSI se envía el comando solicitando la aceleración en el eje Z, ZACCL_OUT. La trama para solicitar este comando según el manual es 0x0F00, acompañada por la señal de reloj en cada flanco y (aunque no se vea en la figura) la línea CS a valor bajo. En la imagen de la derecha por la línea MISO se envía la respuesta del sensor inercial al comando anterior. En la trama de respuesta (‘1011111001110000’) la información la constituyen los 14 últimos bits, esto es 0x3E70, que según las instrucciones del manual corresponde a un valor de -0.9937g (g= valor de la gravedad 9.8 m/s^2) prácticamente -1g, el valor esperado.

Con este test de prueba hemos conseguido medir la gravedad a través de la aceleración que hay en el eje Z, cuando éste coincide con la vertical, y el dispositivo está en reposo. La prueba realizada nos ha ayudado enormemente para continuar con el diseño de la interfaz SPI.

4.2. Software

El subsistema se va a encargar de gestionar los *buses* SPI que haya en el sistema. Como ya hemos dicho antes no se trata de un *driver*, sino de un *software* intermedio entre los *drivers* y los dispositivos SPI. Por tanto, va a tener una interfaz propia que nosotros crearemos, a través de la cual, los *drivers* escribirán y leerán los dispositivos SPI. En MaRTE OS existe ya una interfaz de este tipo para el bus serie I2C, la idea es seguir la misma estructura que tiene esta interfaz; por un lado para mantener la coherencia en MaRTE OS, y por otro porque tiene una estructura muy estudiada originando una interfaz I2C realmente completa, adaptable a diferentes dispositivos y diferentes tarjetas de entrada/salida digital.

4.2.1. Análisis de especificaciones

El subsistema va a presentar una interfaz basada en comandos. Las tareas podrán mandar comandos al transmisor del protocolo SPI que elijan. De momento sólo tenemos el transmisor de entradas y salidas digitales Acces, pero como ya se ha dicho y se explicará con detenimiento en siguientes apartados pueden existir otros adaptadores SPI.

Los comandos se irán guardando y ejecutando en un orden de prioridad por una tarea interna que llamaremos también 'demonio'. Éste permanecerá en segundo plano ejecutándose continuamente. Una vez enviado el comando, la tarea de usuario podrá dedicarse a otros cálculos y chequear la operación SPI cada cierto tiempo, o bien, bloquearse hasta que esta finalice.

Podrán existir múltiples tareas de usuario enviando comandos SPI al subsistema. En cada comando que se envía se especifica la prioridad de la operación, el número que identifica la operación, el adaptador a través del cual se quiere establecer la comunicación, los *flags* indicando el modo de funcionamiento del periférico y el puntero a la trama o tramas a enviar y/o que se esperan recibir. Este conjunto de datos que constituyen el mensaje u operación, es encolado por el usuario y desencolado para llevar a cabo por el demonio.

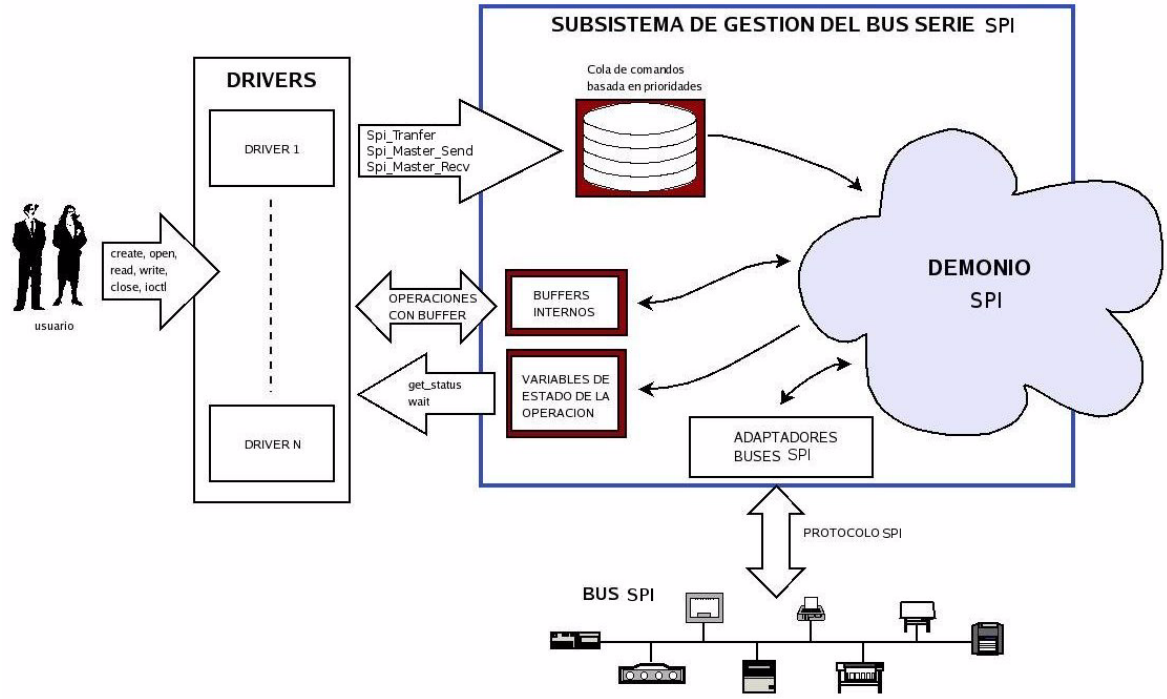


Figura 4.11: Esquema del subsistema SPI

Debe 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 4.11.

4.2.2. Diseño de la arquitectura

Dividimos 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.

La facilidad de usar paquetes hijos, nos permite crear una estructura jerárquica donde en el paquete raíz de la jerarquía se encuentren los tipos usados por todo el subsistema y en los paquetes hijos el código más específico como, por ejemplo, las interfaces a las aplicaciones en C y Ada o la interfaz *hardware* al *bus*. Además, hemos utilizado objetos y polimorfismo para poder representar los distintos *buses* SPI que puede haber en el sistema empotrado. Tenemos también una tarea y la parte de inicialización de los paquetes Ada. No hemos utilizado objetos protegidos pues no se podrían utilizar desde *threads* C, así que usamos mutexes y variables condicionales en su lugar.

- Los archivos relacionados con este *driver* son:

- `spi.ads`: contiene los tipos SPI, definidos para usar por todo el subsistema, y el valor de algunas constantes que pueden modificarse según los requerimientos del usuario.

- `spi-adapters.ad[s,b]`: contiene la declaración de la función genérica `Master_Xfer` de un adaptador para el *bus* SPI, definida como abstracta para que sea implementada en los paquetes hijos. Se define también el tipo `Spi_Adapter_Ref` que es un puntero a la clase `spi-adapters.`

- `spi-adapters-bit.ad[s,b]`: implementa el protocolo de comunicación SPI a través de *bit banging*.

- `spi-adapters_table.ads`: contiene una tabla con los adaptadores SPI que tenga el subsistema.

- `protected_priority_queues.ad[s,b]`: paquete donde se definen las colas de prioridad que se utilizan para encolar los mensajes SPI.

- `spi-daemon.ad[s,b]`: tarea encarga de llevar a cabo la comunicación SPI. Desencola el mensaje más prioritario, lo envía, recibe lo necesario y lo guarda en el *buffer* de recepción.

- `spi-daemon-ada_interface.ad[s,b]`: interfaz que han de usar los *drivers* escritos en Ada para hacer uso del subsistema SPI.

- `spi-daemon-c_interface.ad[s,b]`: interfaz que han de usar los *drivers* escritos en C para hacer uso del subsistema SPI.

- `spi_c_interface.c`: algunas de las funciones de la interfaz C para los usuarios y *drivers*, escritas en lenguaje C.

- `spi.h`: contiene el equivalente en C de muchos de los tipos definidos en `spi.ads` así como la declaración de las funciones que han sido exportadas de Ada a C en el paquete anterior.

- `spi-slaves_table.ads`: contiene una tabla con el número de adaptadores SPI que existen, especificando cuáles son las líneas de *chip select* para cada unos de ellos. De esta forma el periférico tiene un número de esclavo según qué línea `CS_line` utilice.

- `spi_acces.ads`: importación a lenguaje Ada del código específico para el adaptador constituido por la tarjeta 104-AIO12-8 de Acces

- `spi_acces_c.c`: código específico para el adaptador Acces escrito en lenguaje C.

- `spi_acces.h`: cabecera del fichero anterior.

- `test_spi.adb`: test de prueba de la interfaz SPI para un programa escrito en Ada.

- `test_spi_c.c`: test de prueba de la interfaz SPI para un programa escrito en C.

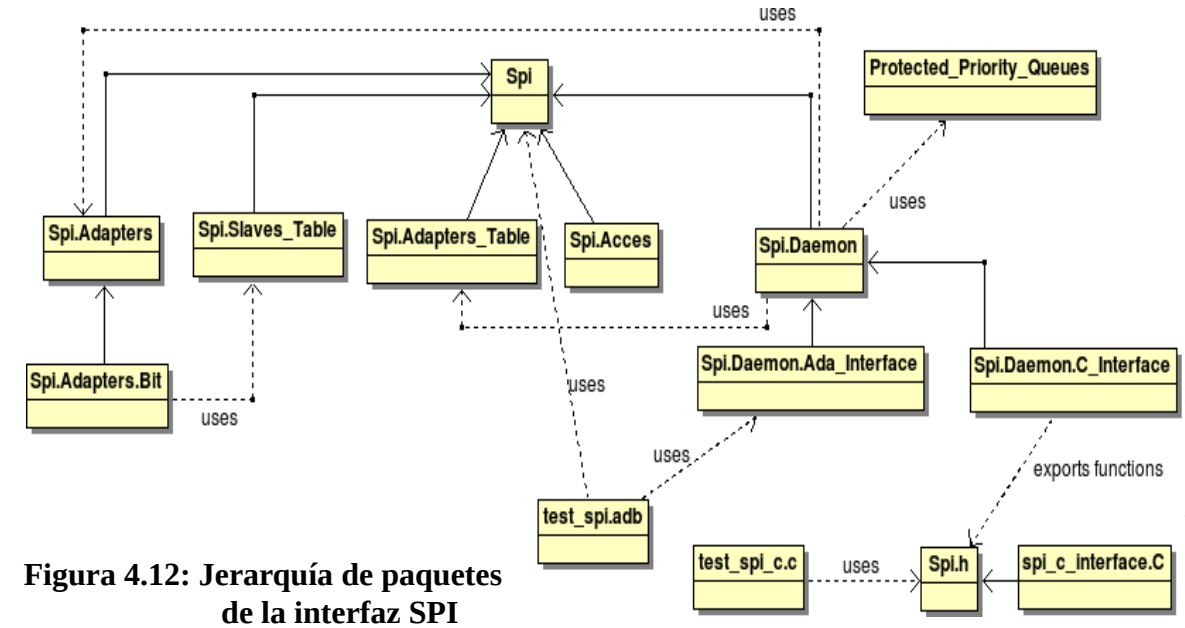


Figura 4.12: Jerarquía de paquetes de la interfaz SPI

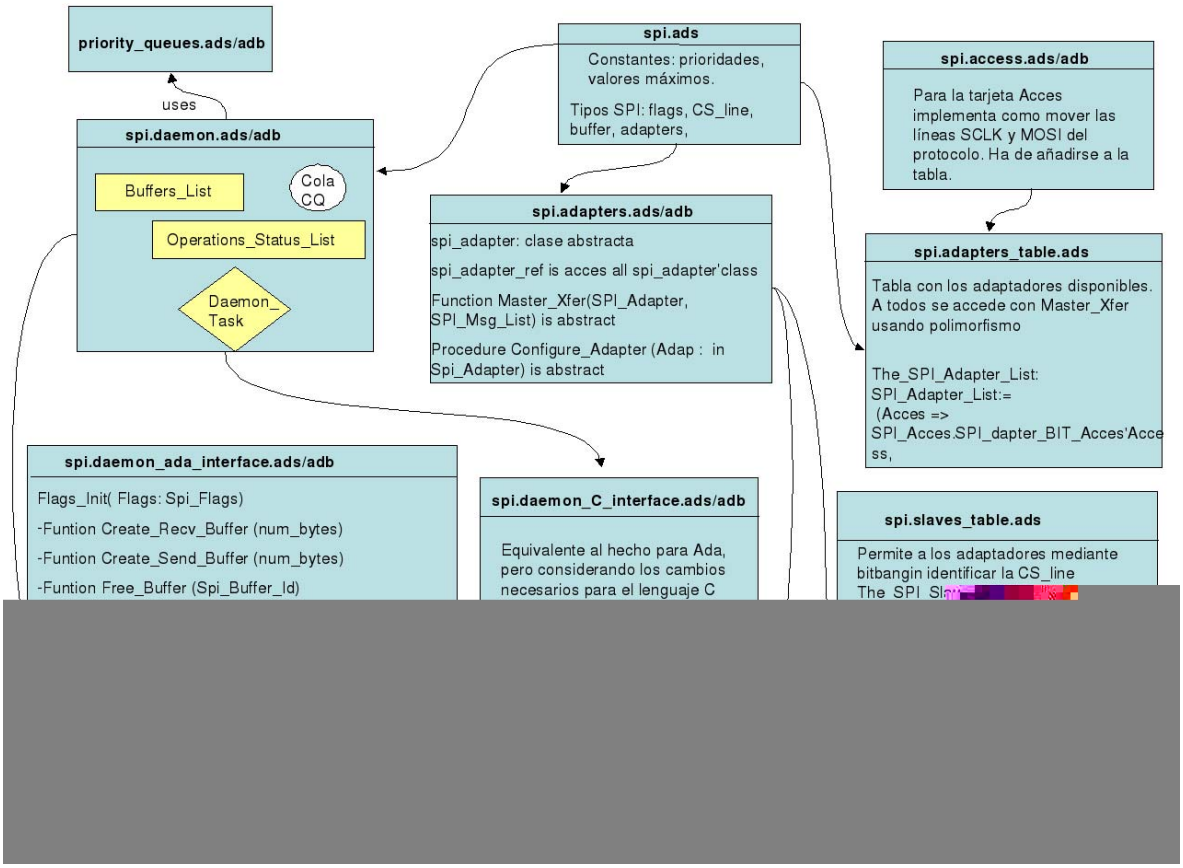


Figura 4.13: Esquema de la arquitectura del subsistema SPI

Las relaciones entre los módulos principales se muestran en la figura 4.13. Comencemos viendo las funciones interfaz de los *drivers* usuarios.

- **Interfaz SPI para los drivers usuarios**

Las siguientes funciones son las que han de utilizar los *drivers* para mandar comando al demonio:

```
procedure Spi_Transfer (
     Adap     : in     Spi_Adapter_Id;
     Op       : in     Spi_Operation_Id;
     Prio     : in     Spi_Priority;
     Msg_List : in     Spi_Msg_Ada_List);

procedure Spi_Master_Send (
     Adap   : in     Spi_Adapter_Id;
     Op     : in     Spi_Operation_Id;
     Prio   : in     Spi_Priority;
     Slave  : in     Spi_Slave;
     Buffer : in     Spi_Buffer;
     Flags  : in     Spi_Flags);

procedure Spi_Master_Recv (
     Adap   : in     Spi_Adapter_Id;
     Op     : in     Spi_Operation_Id;
     Prio   : in     Spi_Priority;
     Slave  : in     Spi_Slave;
     Buffer : in     Spi_Buffer;
     Flags  : in     Spi_Flags);
```

`Spi_Transfer` permite hacer cualquier tipo de transferencia del protocolo SPI. Las otras dos funciones, `Spi_Master_Recv` y `Spi_Master_Recv`, sirven para enviar o para recibir bytes y tienen la ventaja de ser más simples.

El significado de cada parámetro es el siguiente:

- `Spi_Adapter_ID`: representa un *bus* SPI físico presente en el sistema. Los adaptadores, son añadidos estáticamente a una tabla, `spi-adapters_table.ads`, y así el usuario puede escoger por cuál realizar la transferencia (obviamente por el *bus* en el que esté el dispositivo).
- `Spi_Operation_ID`: representa una operación de transferencia que pide el *driver* usuario. Dar una identificación a la operación nos permite, una vez hecha la petición, chequear el estado de la transferencia o quedarnos bloqueados. Para realizar estas acciones está la siguiente interfaz:

```
function Get_Status (
     Op : in     Spi_Operation_Id) return Spi_Operation_Status;

function Wait (
     Op : in     Spi_Operation_Id) return Spi_Operation_Status;

procedure Read_Buffer (
     Data   :    out Spi_Data;
     Buffer : in     Spi_Buffer;
```

```
Op     : in     Spi_Operation_Id;
Stat   :    out Spi_Operation_Status);
```

El tipo `Spi_Operation_Status` representa el estado de la operación:

1. `NOT_IN_USE`: el `Spi_Operation_ID` no representa a ninguna operación enviada al subsistema.

2. `WORKDONE`: la operación indicada por `Spi_Operation_ID` ha sido completada con éxito.

3. `WAITING`: la operación indicada por `Spi_Operation_ID` está siendo ejecutada por el subsistema.

4. `SPI_ERROR`: la operación indicada por `SPi_Operation_ID` ha sufrido un error interno (bien físico o bien porque los datos facilitados no son correctos).

- `Spi_Priority`: indica la prioridad de la operación. Ésta determinará, cuando varias tareas envíen comandos al subsistema, el orden en el que se deben ejecutar.
- `Spi_Slave`: representa el número de esclavo del dispositivo. Este valor junto con el `Spi_Adapter_Id` permite, gracias a la tabla `spi-slaves_table.ads`, conocer la línea *chip select* (CS) a la que se encuentra conectado el dispositivo.
- `Spi_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     Spi_Data_Count) return Spi_Buffer;

function Create_Send_Buff (
     Data : in     Spi_Data) return Spi_Buffer;

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

El tipo `Spi_Data` es un array de bytes y `Spi_Data_Count` es el índice de ese array.

- `Spi_Flags`: esta variable se incluye para poder soportar futuras variaciones del protocolo. De momento se utiliza para establecer el modo SPI de funcionamiento del esclavo, es decir, los valores de CPOL y CPHA. También dispone de una interfaz:

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

procedure Flags_Set_Spi_Mode (
     Flags : in out Spi_Flags; CPHA : in Boolean ; CPOL : in Boolean
);
```

Es necesario inicializar la variable `Spi_Flags` a cero (haciendo uso de la función `Flags_Init`) antes de usarla. Ésto es porque la función `Flags_Set_Spi_Mode` sólo trabaja sobre los bits 1 y 2, que son los que representan los valores de CPOL y CPHA, dejando el resto de bits de la variable igual que estaban. El bit cero se utiliza para indicar cuándo un mensaje es de lectura, se establece en la función vista anteriormente `function Create_Recv_Buff`.

- `Spi_Msg_Ada_List`: se utiliza para pedir al *demonio* que realice un conjunto de operaciones.

  La interfaz es la siguiente en el caso de Ada:

```
subtype Spi_Msg_Ada is Spi_Msg;
type Spi_Msg_Ada_List is private;

procedure Create_Msg (
     Msg    : in out Spi_Msg_Ada;
     Slave  : in     Spi_Slave;
     Buffer : in     Spi_Buffer;
     Flags  : in     Spi_Flags);

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

procedure Add (
     Msg      : in     Spi_Msg_Ada;
     Msg_List : in out Spi_Msg_Ada_List);

procedure Empty (
     Msg_List : in out Spi_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, y es necesario llamar a una función para crearlo:

```
typedef struct {
     spi_slaveslave;
     spi_flags flags;
     spi_buffer buffer;} spi_c_msg;

int spi_msg_c_init (spi_c_msg *msg,spi_slave slave,
     spi_buffer buffer,spi_flags flags)
     {
          msg->slave = slave;
          msg->flags = flags;
          msg->buffer = buffer;
          return 0;}
```

Además existen una serie de constantes en `spi.ads` que pueden ser modificadas según necesiten los *drivers* usuarios:

- `Spi_Mx_Prio : constant := 256`: sirve para indicar cuántos niveles de prioridad pueden existir en las operaciones que realizan los usuarios. Estas prioridades son usadas por el *demonio* SPI para atender los comandos en un orden. Ver que no son lo mismo que las prioridades de las tareas.
- `Spi_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 pues llegará un momento en que no queden más *buffers* disponibles.

- Prioridades internas: establece las prioridades del *demonio* SPI y de las estructuras de datos protegidas. Los usuarios deben tener una prioridad menor o igual a la del *demonio* y las estructuras de datos protegidas.

```
Spi_Daemon_Prio         : constant := System.Priority'Last-1;
Spi_Operation_List_Prio : constant := System.Priority'Last;
Spi_Commands_List_Prio  : constant := System.Priority'Last;
Spi_Buffer_List_Prio    : constant := System.Priority'Last;
```

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

4.2.3. Diseño detallado y codificación.

- **Los adaptadores**

Los adaptadores son la parte del subsistema encargada de realizar la transferencia a los dispositivos siguiendo el protocolo SPI. Existen distintos tipos de chips controladores del *bus* SPI, para cada uno se necesitará implementar un adaptador. Por ejemplo, hay adaptadores, como es el nuestro, que se basan en emular el protocolo subiendo y bajando las líneas CS, SCLK y MOSI (*Bit Banging*). Podemos ahorrarnos mucho código si separamos el algoritmo de la parte *hardware*.

1. `spi.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 SPI 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. También debe configurar todos los adaptadores que se encuentren en el sistema en ese momento. Ésto se lleva a cabo en la parte de elaboración de los paquetes.

```
type Spi_Adapter is abstract tagged record
     Id : Spi_Adapter_Id;
end record;
type Spi_Adapter_Ref is access all Spi_Adapter'Class;

function Master_Xfer (
     Adap     : Spi_Adapter;
     Msg_List : Spi_Msg_List) return Int is abstract;
procedure Configure_Adapter (Adap :  in Spi_Adapter) 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.

2. `spi.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 SPI controlando directamente las líneas del *bus* CS, MOSI, SCLK. Para ello en primer lugar creamos una clase hija, que tome como datos los procedimientos para controlar las líneas del *bus* SPI. Estos procedimientos serán la parte de comunicación con el *hardware* del adaptador, es decir, donde se encuentren las instrucciones `inb_p` y `outb_p`:

```
type Spi_Adapter_Bit is new Spi_Adapter with
record
     Configuration_Adap: Configure_Adapter_Ac   := null;
     Wait_Semiperiod   : Wait_Semiperiod_Ac     := null;
     Set_CS_High       : Set_Spi_CSline_Ac      := null;
     Set_CS_Low        : Set_Spi_CSline_Ac      := null;
     Set_Mosi_High     : Set_Spi_Lines_Ac       := null;
     Set_Mosi_Low      : Set_Spi_Lines_Ac       := null;
     Set_Sclk_High     : Set_Spi_Lines_Ac       := null;
     Set_Sclk_Low      : Set_Spi_Lines_Ac       := null;
     Get_Miso          : Get_Spi_Lines_Ac       := null;
     Get_Sclk          : Get_Spi_Lines_Ac       := null;
     Wait_Tdatastall   : Wait_Spi_Tdatastall_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.

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

```
procedure Chip_Select_ModesAC ( Adap      : in     Spi_Adapter_Bit;
     The_Slave : in     Spi_Slave);

procedure Chip_Select_ModesBD ( Adap      : in     Spi_Adapter_Bit;
     The_Slave : in     Spi_Slave);

procedure Write_ModeA ( Adap : in     Spi_Adapter_Bit;
     Byte : in     Unsigned_8);

procedure Read_ModeA ( Adap      : in     Spi_Adapter_Bit;
     Byte_Read : out     Unsigned_8);

procedure Write_ModeB ( Adap : in     Spi_Adapter_Bit;
     Byte : in     Unsigned_8);

procedure Read_ModeB ( Adap      : in     Spi_Adapter_Bit;
     Byte_Read : out     Unsigned_8);

procedure Write_ModeC ( Adap : in     Spi_Adapter_Bit;
     Byte : in     Unsigned_8);

procedure Read_ModeC ( Adap      : in     Spi_Adapter_Bit;
     Byte_Read : out     Unsigned_8);

procedure Write_ModeD ( Adap : in     Spi_Adapter_Bit;
     Byte : in     Unsigned_8);

procedure Read_ModeD ( Adap      : in     Spi_Adapter_Bit;
     Byte_Read : out     Unsigned_8);
```

```
procedure Write (
     Adap      : in     Spi_Adapter_Bit;
     Byte      : in     Unsigned_8;
     Mode      : in     Character);

procedure Read (
     Adap      : in     Spi_Adapter_Bit;
     Byte_Read :    out Unsigned_8;
     Mode      : in     Character);
```

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

El algoritmo en pseudocódigo es el siguiente:

```
Start
for i in Msg_List'Range loop
     if "CPOL=1" then
          ChipSelect para los modos A y C
          if "CPHA=1" then
               modo_funcionamiento =A
          else
               modo_funcionamiento =C
          end if;
     else
          ChipSelect para los modos B y D
          if "CPHA=1" then
               modo_funcionamiento =B
          else
               modo_funcionamiento =D
          end if;
     end if;
     if "Mensaje de lectura" then
          for J in Data'Length loop
               Read (Adaptador, modo_funcionamiento);
               Guardar datos en el buffer
          end loop;

     else "Mensaje de escritura"
          for J in Data'Length loop
               write (Adaptador,byte, modo_funcionamiento);
          end loop;
     end if;
if "hemos terminado" then
     Stop
else
     ReStart
end if;
end loop;
```

3. `spi_acces.[ads,c,h]`: este es un ejemplo de la parte *hardware* de un adaptador de tipo *bit banging*. Para realizarlo, sólo tenemos que saber subir y bajar las líneas, y como vimos en el apartado 4.1.3, sabemos hacerlo. En este caso hicimos las funciones en C y 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 `Spi_Adapter_Bit`.

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

```
The_Spi_Adapter_List : Spi_Adapter_List :=
     (Acces=> Spi_Acces.Spi_Adapter_BIT_Acces'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_Spi_Adapter_List(Cmd.Adap_ID);
if (Spi.Adapters.Master_Xfer (Adap.all,Cmd.Msg_List) = -1) then
     Set_Status (Cmd.Operation_ID, SPI_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 Spi_Command is record
     Adap_ID           : Spi_Adapter_ID;
     Operation_ID      : Spi_Operation_ID;
     Msg_List    : Spi_Msg_List;
end record;
type Spi_Command_Ref is access Spi_Command;
```

La cola es una modificación de las colas de prioridad del *software* MAST[18], 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 (Spi_Operation_ID)
     of Spi_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 SPI. Al usuario le decimos que un *buffer* es:

```
type Spi_Buffer is private;
Null_Spi_Buffer : constant Spi_Buffer;
```

Sin embargo, lo que realmente se esconde detrás de `Spi_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 dé cuenta rápidamente (pues como hemos visto 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 `Spi_Flags`, que nos servía para distinguir variaciones en el protocolo SPI.

```
type Spi_Buffer_Cell is record
     In_Use : Boolean;
     Data : Spi_Data_Ref;
     Flags : Spi_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, sino que él recibe directamente el valor de la variable `Data : Spi_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 Spi_Msg_Ada_List is record
     Msg_List : Spi_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* SPI, 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* SPI:

```
procedure Spi_Transfer:
     Cmd.Adap_ID := Adap;
     Cmd.Operation_ID := Op;
     Cmd.Msg_List := new Spi_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 podíamos utilizar el tipo `Spi_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
     Slave  : Spi_Slave;
     Flags  : Spi_Flags;
     Buffer : Spi_Buffer;
end record;
pragma Convention( C, C_Msg);
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'(1,0,0));
use Msg;
```

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 Spi_Transfer ( Adap     : in     Spi_Adapter_Id;
     Op       : in     Spi_Operation_Id;
     Prio     : in     Spi_Priority;
     Msg_List : in     Msg.Pointer;
     Count    : in     C_Msg_Count)return Int is
     Cmd : Spi_Command_Ref := new Spi_Command;
     Tmp_Msg : Spi_Msg;
     P       : Msg.Pointer     := Msg_List;
     Cell    : Spi_Buffer_Cell;
     begin
          Cmd.Adap_Id := Adap;
          Cmd.Operation_Id    := Op;
          Cmd.Msg_List := new Spi_Msg_List_Array(1 .. Count);
          for I in Cmd.Msg_List'range loop
               Tmp_Msg.Slave := P.All.Slave;
               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);
          return 0;
     end Spi_Transfer;
```

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     Spi_Data_Count;
     Buffer : in     Spi_Buffer;
     Op     : in     Spi_Operation_Id) return Spi_Operation_Status is
     Stat : Spi_Operation_Status;
     Cell : Spi_Buffer_Cell;
     P    : Char.Pointer         := Data;
     begin
          Stat := Spi.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. Se creó el fichero `spi_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.

4.3. Manejador para un dispositivo SPI: sensor inercial de 3 ejes Analog Devices ADIS16355

Realizamos un *driver* muy sencillo para manejar el sensor inercial mediante su interfaz SPI. Así podremos hacer uso y probar el correcto funcionamiento de nuestro reciente subsistema SPI.

4.3.1. Hardware

El sensor inercial de tres ejes[17] es un completo sistema capaz de funcionar como giróscopo, midiendo velocidades angulares con tres escalas de rangos +-75º/s, +-150º/s y +-300º/s. Funciona también como acelerómetro con una rango de medida de +-10g, todo esto con una resolución de 14 bits.

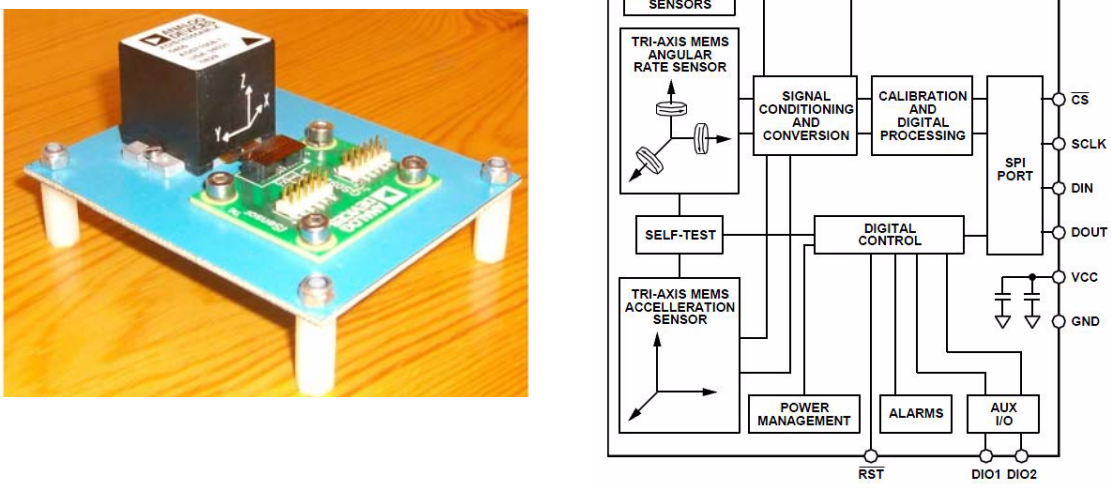


Figura 4.14: Sensor inercial ADIS16355

Hemos utilizado la tarjeta ADIS1635x/PCB[19] para facilitar las conexiones de los pines del sensor inercial, utilizamos el conector J1:

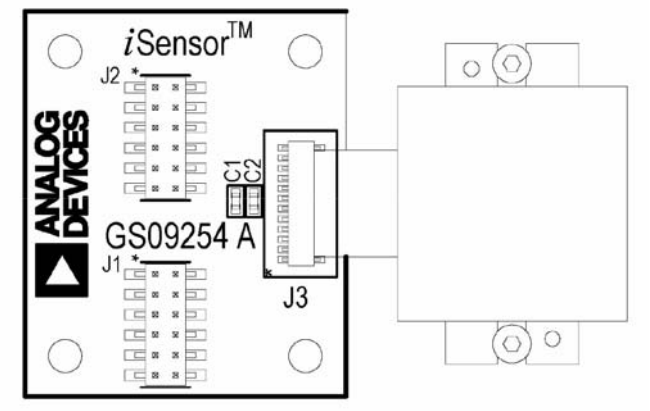


Figura 4.15: Placa ADIS1635x/PCB

El montaje en nuestro sistema se puede ver en la figura 4.16.

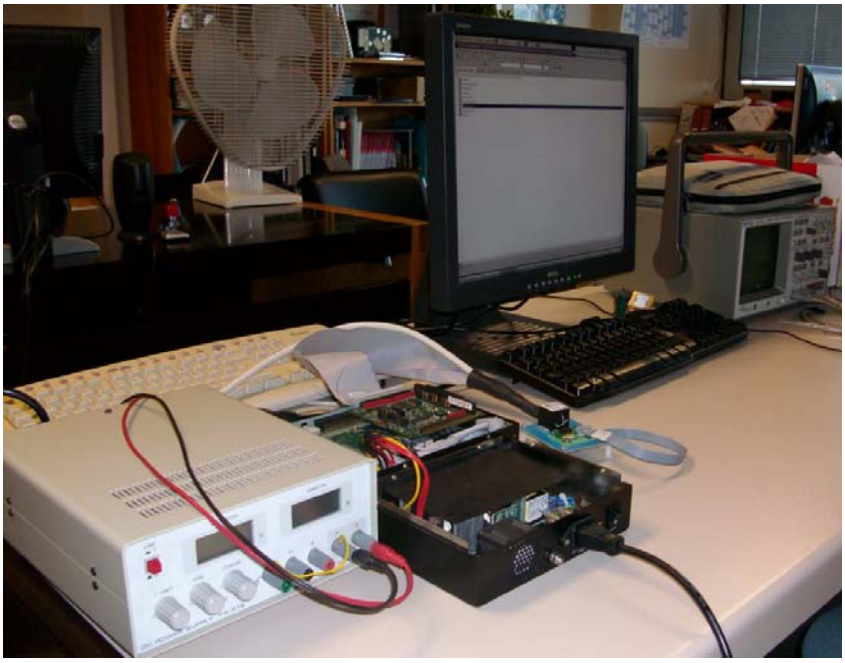


Figura 4.16: Montaje el sistema con el sensor inercial ADIS16355

El conexionado entre los pines del sensor ADIS16355 y el bus SPI se ve en la siguiente tabla. La placa ADIS1635x/PCB dispone de varios conectores, utilizamos el conector J1:

Tabla 4.1: Líneas SPI

Señal SPI	Pin bus SPI (tarjeta Acces)	Pin ADIS1635x/ PCB.
MOSI	23	6
CS	21	3
SCLK	19	2
Reset	17	1
MIS0	15	4

4.3.2. Software

En lugar de hacer un *driver* que debe instalarse añadiéndolo en el fichero `marte-kernel-devices_table.ads` vamos a hacer una interfaz que ofrezca las funciones para interactuar con el dispositivo. Este elección la hemos hecho para que resulte más sencillo el funcionamiento al usuario. Si no, tendríamos que hacer uso de la función *ioctl* y establecer una serie de constantes para identificar las funciones. Esta idea parece algo engorrosa, por eso escogemos el diseño de un interfaz, en lenguaje C, que ofrezca una función para cada uno de los comandos del dispositivo.

En la tabla 4.2 podemos ver la información sobre los registros de salida que necesitamos para pedir esos comandos.

En los directorios *x86_arch/drivers/adis16355* y *x86_arch/include/drivers* se encuentran los archivos relacionado con el *driver*. Para poder usarlo es suficiente con incluir el fichero `adis16355.h: #include "adis16355.h"`.

Tabla 4.2: Registros de salida del sensor inercial

Nombre	Función	Dirección	Bits de datos	Formato de datos	Factor de escala
SUPPLY_OUT	Potencia suministrada	0x03,0x02	12	Binario	1.8315mV
XGYRO_OUT	Giróscopo eje X	0x05,0x04	14	Complemento A2	0.07326º/s
YGYRO_OUT	Giróscopo eje Y	0x07,0x05	14	Complemento A2	0.07326º/s
ZGYRO_OUT	Giróscopo eje X	0x09,0x08	14	Complemento A2	0.07326º/s
XACCL_OUT	Aceleración eje X	0x0B,0x0A	14	Complemento A2	0.002522g
YACCL_OUT	Aceleración eje Y	0x0D,0x0C	14	Complemento A2	0.002522g
ZACCL_OUT	Aceleración eje Z	0x0F,0x0E	14	Complemento A2	0.002522g
XTEMP_OUT	Sensor de temperatura eje X	0x11,0x10	12	Complemento A2	0.1453ºC
YTEMP_OUT	Sensor de temperatura eje Y	0x13,0x12	12	Complemento A2	0.1453ºC
ZTEMP_OUT	Sensor de temperatura eje Z	0x15,0x14	12	Complemento A2	0.1453ºC
AUX_ADC	Entrada analógica auxiliar	0x17,0x16	12	Binario	0.6105mV

Hacemos uso de la interfaz SPI, concretamente del paquete `Spi.Daemon.C_interface`. Como el *driver* está escrito en lenguaje C, en el fichero `spi.h` se encuentra la declaración de todas las funciones exportadas desde Ada y las que estás directamente escritas en C que podemos utilizar, resultado de subsistema SPI descrito en el anterior apartado.

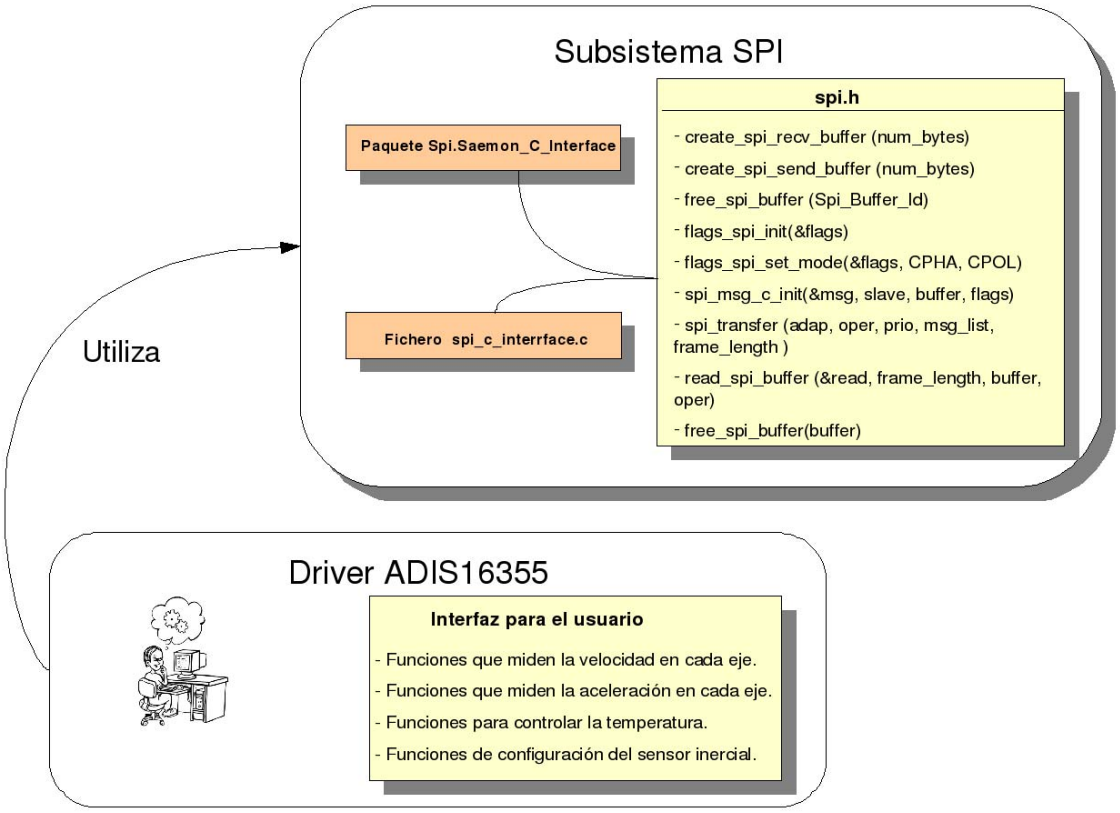


Figura 4.17: Relación entre el driver ADIS1635 y el subsitem aSPI

Haciendo uso de la interfaz SPI para programas escritos en lenguaje C se crean las siguientes funciones:

- `short send_request( uint8_t frame, int frame_length,int OP_id, uint16_t priority_op ):`
  Envía una trama de `frame_length` bytes donde los 8 primeros toman el valor `frame`. Se especifica el número de operación y la prioridad de la misma. La función devuelve en 16 bits la respuesta del dispositivo al comando.
- `void send_command( uint8_t *frame, int frame_length, int OP_id, uint16_t priority_op ):`
  Envía un comando de configuración al dispositivo, en este caso no hay trama de respuesta.

Con el fin de cubrir todas las posibilidades del sensor inercial, y haciendo uso de las funciones que se acaban de describir, se implementa la interfaz ADIS16355 que contiene las siguientes funciones:

- **`void Init_ADIS16355 (void);`**

Es necesario inicializar la interfaz SPI antes de que ésta sea usada por primera vez, por eso antes de empezar a usar el sensor inercial debemos ejecutar esta función.

- **void ADIS16355_Bias_Null_Calibration (void);**

Es conveniente calibrar el dispositivo antes de usarlo, para que las medidas de velocidad y sobre todo de aceleración sean correctas. Es importante saber que se necesitan unos 30 segundos para conseguir hacer este calibrado. El tiempo de espera ya está incluido en la propia función.

- **void ADIS16355_Bias_Null_Restoring (void);**

Resetea los registros de la intensidad de *bias* que son modificados cada vez que se calibra el aparato con la función anterior.

- **float ADIS16355_Power_supply (void);**

Devuelve la potencia suministrada al equipo. Según el manual esta potencia debe estar comprendida en el rango 4.75-5.25 V.

- **void ADIS16355_Set_75_SensAVG (void);**

Establece el rango de velocidades angulares a +-75º/s.

- **void ADIS16355_Set_150SensAVG (void);**

Establece el rango de velocidades angulares a +-150º/s.

- **void ADIS16355_Set_300_SensAVG (void);**

Establece el rango de velocidades angulares a +-300º/s. En el caso de no utilizar ni ésta ni las otras dos funciones relacionadas con el rango de medida de velocidades, +-300º/s es el valor que trae por defecto el sensor. En el desarrollo de estas tres últimas funciones se accede al registro SENS/AVG, en el que se especifica el rango y las etapas de filtrado.

Funciones que miden la velocidad de giro en cada uno de los ejes en º/s:

- **float ADIS16355_X_Gyro_out (void);**
- **float ADIS16355_Y_Gyro_out (void);**
- **float ADIS16355_Z_Gyro_out (void);**

Funciones que devuelven el contenido del registro donde se calibra la intensidad de *bias* para medir la velocidad de giro en los tres ejes.

- **short ADIS16355_X_Gyro_off (void);**
- **short ADIS16355_Y_Gyro_off (void);**
- **short ADIS16355_Z_Gyro_off (void);**

Funciones que miden la aceleración en cada uno de los ejes respectivamente, medida en gravedades:

- **float ADIS16355_X_Accl_out (void);**
- **float ADIS16355_Y_Accl_out (void);**
- **float ADIS16355_Z_Accl_out (void);**

Funciones para medir la temperatura:

- **float ADIS16355_X_Temp_out (void);**
- **float ADIS16355_Y_Temp_out (void);**
- **float ADIS16355_Z_Temp_out (void);**

En el directorio del *driver* se encuentra el fichero `test_adis16355.c` donde hacemos uso de todas las funciones anteriormente descritas. El funcionamiento es el esperado, lo vemos en la siguiente figura 4.18. Y con esta prueba damos por finalizado el *driver*.

```
Testing tri-axis Inertial Sensor ADIS16355
----------------------------------------
Bias Null Calibration
Bias X gyro : 32
Bias Y gyro : 56
Bias Z gyro : 24

Power Supply: 4.924903 Volt

Establishing -+ 75 degrees/seconds
X Gyro out : 0.604395 grados/seg
Y Gyro out : -0.219780 grados/seg
Z Gyro out : 0.036630 grados/seg

X Accl out: -0.025220 g
Y Accl out: 0.005044 g
Z Accl out: -1.006278 g

X Temp out: 1.307700 grados C + Tambiente (25 grados C)
Y Temp out: 1.888900 grados C + Tambiente (25 grados C)
Z Temp out: 4.270100 grados C + Tambiente (25 grados C)

 _exit(0) called; rebooting...
Press a key to reboot
```

Figura 4.18: Captura del funcionamiento del sensor inercial ADIS16355

5. Conclusiones

El la sección 1.3 se expusieron los objetivos del proyecto, veamos ahora cuál ha sido el trabajo realizado:

- Se ha hecho un estudio sobre MaRTE OS y su estructura de *drivers*. Los conocimientos adquiridos nos han servido para saber cómo llevar a cabo el trabajo.
- Se ha conseguido desarrollar el manejador en MaRTE OS para la tarjeta de 8 puertos serie COM-1274-A1 de Parvus. Aumentando notoriamente el número de periféricos de este tipo que se pueden conectar ahora a nuestro sistema.

  Recordar que, desafortunadamente, no se ha podido crear un *software* de configuración de las direcciones de memoria de los puertos serie de la tarjeta, ya que no se disponía de la información para hacerlo.
- Gracias al estudio del bus SPI, y del subsistema I2C ya existente en MaRTE OS, se ha podido desarrollar una nueva interfaz SPI.
- Se ha implementado el manejador de un dispositivo real que se comunica a través del protocolo SPI. Se trata del sensor inercial de tres ejes ADIS-16355, y gracias a su correcto funcionamiento queda probado también satisfactoriamente el subsistema SPI anterior.

Además de haber cumplido los objetivos que nos propusimos inicialmente en el desarrollo de este trabajo, hemos contribuido a aumentar las posibilidades de interacción con el exterior de un equipo que utilice el sistema operativo MaRTE OS.

5.1. Trabajo futuro

Sería interesante conseguir solventar el problema que tenemos al no poder configurar las direcciones de memoria en la tarjeta COM-1274-A1 de Parvus. Para ello deberíamos ponernos en contacto con el fabricante con el fin de conocer el protocolo de comunicación con la memoria EEPROM de la tarjeta.

El mercado de periféricos con interfaz SPI es muy amplio. Mediante este subsistema se abre enormemente la capacidad de incorporar nuevos artilugios de utilidad a nuestro equipo. Sería interesante entonces la creación de nuevos *drivers* para dispositivos con esta interfaz.

En el subsistema SPI se necesitan realizar nuevos adaptadores para otros tipos de controladores del bus SPI que hay en el mercado, ya que sólo hemos desarrollado adaptadores tipo *bit-banging*, concretamente para la tarjeta de I/O digital Acces 104-AIO12-8-DC. El diseño del subsistema ha sido pensado lo más general posible, con la intención de facilitar la creación de nuevos adaptadores y nuevos *drivers*. Los ficheros ya existentes servirán de guía a la hora de realizarlos.

Bibliografía

[1] Jane W.S. Liu, “Real-Time Systems”

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

[3] IEEE Standard 1003.13:1998 “Standard for Information Technology -Standardized Application Enviroment Profile- POSIX Realtime Application Support (AEP)” TheInstitute of Electrical and Electronics Engineers, 1998.

[4] Código fuente de MaRTE OS:
http://marte.unican.es.

[5] Cargando MaRTE OS: http://marte.unican.es/documentation.htm

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

[7] GEME-3000, ADLINK: http://www.adlinktech.com/PD/web/PD_detail.php?pid=261#

[8] Manual COM-1274,
Parvus: http://www.parvus.com/product/overview.aspx?prod=COM-1274.
Eurotech:http://www.eurotech.com/en/pb.aspx?tab=download&pg=COM-1274&pid=254&pc=67&tt=Manual

[9] Drake J.M. y Mediavilla E.: Técnicas de entrada y salida de tiempo real”.

[10] Plug and Play standard: http://es.wikipedia.org/wiki/Plug-and-play

[11] Legacy ISA:
http://es.wikipedia.org/wiki/IBM_Personal_Computer/AT
http://es.wikipedia.org/wiki/Bus_ISA

[12] Francisco Guerreira. “Entorno para la instalación y utilización de manejadores de dispositivos en MaRTE OS”. 2003.

[13] Daniel Sangorrín, “Gestión de Dispositivos de Entrada/Salida Analógica, Digital y por el bus serie I2C”, Trabajo fin de carrera UC, 2006.
http://marte.unican.es/memoria_drivers.pdf.

[14] Protocolo SPI:
Sección 8 del manual del microcontrolador M68HC11 de motorola en donde se especifica la comunicación SPI:
http://www.eee.metu.edu.tr/~balper/HC11%20Reference%20Manual.pdf
http://www.i-micro.com/pdf/articulos/spi.pdf

[15] Tarjeta Acces 104-AIO12-8
http://www.accesio.com/go.cgi?p=../104/104-aio12-8.html
http://www.accesio.com/MANUALS/104-AIO12-8.PDF

[16] Bit banging: http://en.wikipedia.org/wiki/Bit-banging

[17] Analog Devices ADIS16355:
http://www.analog.com/static/imported-files/Data_Sheets/ADIS16350_16355.pdf

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

[19] Analog Devices:
http://www.analog.com/static/imported-files/eval_boards/ADIS1635x-PCB.pdf