UAB

Proyecto Fin de Carrera

Ingeniería Técnica de Telecomunicaciones
Especialidad en Sistemas Electrónicos

Robot Móvil con EVK1100
Estudio y Aplicación

Francisco Manuel Muñoz Verdú

Director: Joan Oliver i Malagelada

Departamento de Microelectrónica y Sistemas Electrónicos

Escola d'Enginyeria (EE)
Universitat Autònoma de Barcelona (UAB)

Junio 2010

El tribunal d'avaluació d'aquest Treball Fi de Carrera, reunit el dia *7 de juliol de 2010*, ha acordat concedir la següent qualificació:

Tribunal: *Joan Oliver i Malagelada*

Pedro de Paco Sánchez

Núria Barniol Beumala

El sotasignant, Joan Oliver i Malagelada, Professor de l'Escola d'Enginyeria (EE) de la Universitat Autònoma de Barcelona (UAB),

CERTIFICA:

Que el projecte presentat en aquesta memòria de Projecte Fi de Carrera ha estat realitzat sota la seva direcció per l'alumne Francisco Manuel Muñoz Verdú.

I, perquè consti a tots els efectes, signa el present certificat.

Bellaterra, 21 de Juny del 2010.

Signatura:

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

En la actualidad, el microcontrolador forma parte fundamental del entorno que nos rodea, encontrándose presente en nuestro trabajo, casa y vida en general. La versatilidad que introducen los microcontroladores permite desarrollar sistemas complejos, que de haber sido desarrollados de forma tradicional, hubieran necesitado grandes cantidades de componentes electrónicos. Este hecho ha favorecido su uso en todo tipo de dispositivos como pueden ser pantallas LCD, dispositivos USB, dispositivos de comunicación Bus-CAN o Wireless, en vehículos, en elementos lumínicos como pueden ser bombillas o fluorescentes, baterías o en general en cualquier dispositivo electrónico que podamos encontrar en nuestro entorno.

La compañía Atmel puso en el mercado en el año 1997 los microcontroladores AVR8, el primer producto de arquitectura propia de Atmel. Los AVR8 son altamente utilizados en múltiples aplicaciones, ofreciendo un rendimiento muy bueno, debido a que fueron los primeros micros de 8 bits en implementar instrucciones RISC (aunque no todas las instrucciones tienen el mismo tamaño) y ofreciendo un diseño mucho más moderno que sus competidores, los PIC. En general, podemos decir que los AVR8 ofrecen un rendimiento excepcional, equiparable a otros microcontroladores de 16 bits, pero a un coste de uno de 8 bits.

Con la aparición de la arquitectura AVR32 (como evolución natural del AVR8), Atmel ha vuelto a revolucionar el mercado de los microcontroladores. Esta nueva arquitectura proporciona CPU's de altas prestaciones y bajo consumo energético, a un precio reducido. Comparados con sus competidores directos en precio y prestaciones, los ARM9 y ARM11, los AVR32 son un 35% más veloces y la densidad de código empleado es entre un 30% y un 50% más pequeña.

El estudio de los microcontroladores, los AVR32, representa la base del presente proyecto. Por ello, el proyecto se ha desglosado en dos partes: el estudio de la arquitectura AVR32 y el desarrollo de un robot móvil basado en la arquitectura AVR32.

La primera de estas dos partes, es un estudio enfocado a analizar la versatilidad de la arquitectura AVR32 frente a otros microcontroladores existentes en el mercado, como son el AVR8 o los ARM9 y ARM11.

En la segunda parte, se realiza un estudio de las posibilidades de la placa de evaluación EVK1100, la cual incluye un microcontrolador AVR32 con núcleo AT32UC3A0512. Esta placa nos proporcionará un completo entorno de desarrollo, equipado con un rico repertorio de periféricos y memorias, que permite obtener todo el potencial de estos microcontroladores de forma sencilla. Posteriormente, y mediante el uso de esta placa se desarrollará un robot móvil que permita poner en manifiesto todas las prestaciones y mejoras que estos núcleos ofrecen.

Los objetivos del proyecto son:

- Estudio y análisis de la arquitectura AVR32.
- Estudio de las prestaciones y características que ofrece la placa de evaluación EVK1100 para el diseño robótico, la cual incluye un microcontrolador con núcleo AVR32 UC3 AT32UC3A0512.
- Comparativa entre los microcontroladores AVR32 y sus antecesores, los AVR8.
- Comparativa entre la arquitectura AVR32 y la de ARM9 y ARM11, así como con otras arquitecturas existentes en el mercado.

- Construcción de un robot móvil mediante la placa de evaluación.

    - El robot deberá poder funcionar de forma autónoma.
    - Además, deberá permitirse el control remoto de forma inalámbrica desde un PC.

- Programación de un software capaz de controlar la plataforma robótica de forma remota desde un PC.

1.1 ESTRUCTURA DE LA MEMORIA

El presente documento ha sido estructurado de la siguiente forma:

1. **Introducción:** Se explica y detalla la motivación del proyecto así como se realiza una pequeña introducción al estado del arte en el momento que actualmente nos encontramos. Además, se detallan los objetivos que se pretende alcanzar con el desarrollo de este proyecto.

2. **Planificación, requisitos y herramientas de trabajo:** En este punto se ofrece una visión de cuál ha sido la planificación que se ha seguido durante la elaboración del proyecto, así como también se detallan los requisitos previos a la ejecución del proyecto y las herramientas de trabajo necesarias.

3. **Arquitectura AVR32:** Completo análisis de las principales características que muestran estos microcontroladores.

4. **Comparativa:** Este punto se centra en ver cuáles son las ventajas que presentan los AVR32 respecto a sus antecesores, los AVR8 y respecto a sus directos competidores, los ARM9 y ARM11. Además, se realiza un estudio comparativo entre los microcontroladores AVR32 de Atmel y el resto de alternativas presentes en el mercado.

5. **Uso del AVR32 en un Robot Móvil:** El siguiente capítulo, tiene como objetivos mostrar cuales son las principales características que ofrece el AVR32 para la robótica. Además y ya que todo el proyecto se realiza sobre la placa de evaluación EVK1100 la cual contiene un microcontrolador AVR32 UC3, se muestra cuáles son sus características y funciones principales. También se muestran las modificaciones que se han llevado a cabo sobre la placa de evaluación y se explicará cual es el proceso que se ha de seguir para poder programar estos microcontroladores sin la necesidad de hacer uso de un kit de evaluación. Por último, se detallan los resultados obtenidos de las pruebas realizadas al conjunto robótico.

6. **Conclusiones:** Una vez realizado todo el análisis y obtenidos los resultados, deseados o no, se procede a hacer una sintaxis final del proyecto valorando también las posibles problemáticas o incidencias surgidas durante el periodo de tiempo transcurrido entre el inicio y final del proyecto. Se hace una evaluación personal de éste y se añaden ideas de mejora de trabajo futuro.

7. **Bibliografía:** Lista y detalla la fuente de todas las referencias que aparecen en el informe así como los documentos utilizados en el proyecto. Con esto, se pretende dar veracidad a la información redactada y también facilitar al lector una serie de recursos,

con el fin de que éste entienda y tenga una compresión total de lo que se muestra y se quiere explicar en cada momento.

8. **Anexos:** Por último se incluirán otros documentos como son una guía de iniciación a la programación en AVR32 Studio o el código fuente que utiliza el microcontrolador para hacer mover el robot móvil.

2. PLANIFICACIÓN, REQUISITOS Y HERRAMIENTAS DE TRABAJO

En este segundo capítulo se pretende dar a conocer cuáles son los requisitos necesarios para la ejecución del proyecto, entre los cuales se encuentran los conocimientos previos de la tecnología a utilizar o las herramientas que será necesario utilizar para su elaboración. Además, se detallará cual ha sido la planificación que se ha llevado a cabo en el desarrollo del presente proyecto a lo largo de los 9 meses de duración de este.

2.1 REQUISITOS PREVIOS

Para la elaboración de este proyecto no es necesario disponer de conocimientos sobre el estado actual del mercado de los microcontroladores, ni tan si quiera conocer cuál es su funcionamiento. Será parte del desarrollo de este, el aprendizaje de los conocimientos necesarios para la correcta puesta en marcha del proyecto.

Sin embargo, sí que se predispone de conocimientos del lenguaje de programación C++ y de arquitectura de computadores, así como un nivel avanzado en cuanto a entendimiento de circuitos electrónicos analógicos y digitales se refiere. Estos conocimientos han sido adquiridos a lo largo de los tres años de duración de la carrera de Ingeniería Técnica de Telecomunicaciones, Especialidad en Sistemas Electrónicos, por lo que no se requerirá de una formación previa en el momento de iniciar el proyecto.

2.2 HERRAMIENTAS DE TRABAJO

Las herramientas de trabajo se diferencian entre las utilizadas para el desarrollo del software y las del hardware.

A nivel de desarrollo de software, la *Figura 1* muestra cual es el conjunto de herramientas necesarias para el desarrollo con microcontroladores AVR32. A continuación se ofrecen más detalles de cada una de ellas:

- **AVR32 Studio:** Se trata de un entorno de desarrollo integrado para desarrollar aplicaciones basadas en microcontroladores AVR32. Soporta todo tipo de procesadores AVR32 y dispone de herramientas suficientes para comenzar a desarrollar en lenguaje C++.

- **Compilador C/C++: "**AVR32 Studio" no dispone de compilador propio y es por eso que se debe hacer uso de uno externo, aunque se integra totalmente con "AVR32 Studio". Es por ello que Atmel ofrece de forma gratuita "AVR32 GNU Toolchain", que permitirá compilar, realizar debug y programar los microcontroladores AVR32.

- **AVR32 Software Framework:** Una de las grandes ventajas de hacer uso de AVR32 Studio es que dispone de ejemplos y librerías incluidos en el propio Framework que incluye. Todo el Framework está escrito en lenguaje C++, por lo que será el lenguaje elegido para la elaboración de todo el proyecto.

- **Starter Kit:** Tal y como se detallará en los próximos capítulos, para el desarrollo y testeo de los microcontroladores AVR32, se hará uso de una placa de evaluación EVK1100, la cual ofrece un conjunto de periféricos y memorias que permitirán demostrar todo el potencial de estos microcontroladores.

- **Debugger:** El debugger será necesario para poder programar la memoria Flash del microcontrolador. Para ello, se dispone de un JTAG MKII ICE, que ofrece una potente herramienta de programación y debug sobre el propio chip. No es la única solución existente para realizar esta tarea, pero para la elaboración del presente proyecto disponemos de esta herramienta.

Figura 1: Herramientas necesarias para el diseño del Software

A nivel de desarrollo de hardware, se hace uso de diferentes dispositivos electrónicos como son sensores, servomotores, motores DC, conversores RS232 a Bluetooth o algunos circuitos integrados como el L293D. Todos estos elementos han sido utilizados para la construcción de un robot móvil, tal y como se verá en los próximos capítulos, donde se detalla en profundidad, el uso y finalidad de cada uno de los elementos utilizados.

2.3 PLANIFICACIÓN

Inicialmente la planificación del proyecto fue pensada para llevarse a cabo en 3 meses, con fecha de inicio el 02/10/2009 (coincidiendo con la primera reunión realizada) y finalización el 12/02/2010.

A pesar de que estas fechas han sido estudiadas y analizadas en función de los requisitos que se creían convenientes, la planificación final no ha sido tal y como se esperaba. Este hecho es algo normal si tenemos en cuenta que se trata de un proyecto nuevo y por lo tanto, se desconocen los posibles retardos que puedan aparecer a lo largo de su ejecución. Es por ello que su finalización se pospone hasta el mes de Junio.

Los retrasos se deben principalmente, a la carencia de bibliografía especializada acerca de los microcontroladores AVR32, por lo que todo el aprendizaje se ha llevado a cabo a través del

estudio de los ejemplos y drivers disponibles en el Framework de Atmel, así como del análisis de los datasheets disponibles en la web oficial. La *Figura 2* muestra la planificación final seguida para la ejecución del proyecto.

	Nombre de tarea	Duración	Comienzo	Fin	Pred
1	**Robot móvil con EVK1100 - Estudio y Aplicación**	**198 días**	**vie 02/10/09**	**mar 06/07/10**	
2	Reunión con el tutor	1 día	vie 02/10/09	vie 02/10/09	
3	**Estudio y comprensión**	**44 días**	**lun 12/10/09**	**jue 10/12/09**	
4	Estudio y análisis del proyecto	2 días	lun 12/10/09	mar 13/10/09	
5	Estado del arte	3 días	mié 14/10/09	vie 16/10/09	4
6	Requisitos del proyecto	1 día	lun 19/10/09	lun 19/10/09	5
7	Recopilación de información y documentación	6 días	mar 20/10/09	mar 27/10/09	6
8	Estudio de la arquitectura AVR32	15 días	mié 28/10/09	mar 17/11/09	7
9	**Estudio de la placa EVK1100**	**12 días**	**mié 18/11/09**	**jue 03/12/09**	
10	Características técnicas principales, CPU y memoria	3 días	mié 18/11/09	vie 20/11/09	8
11	Interfaces de conexión	3 días	lun 23/11/09	mié 25/11/09	10
12	Programación mediante JTAGICE MKII	1 día	jue 26/11/09	jue 26/11/09	11
13	Software AVR32 Studio	5 días	vie 27/11/09	jue 03/12/09	12
14	Comunicación con el robot y funciones básicas de este	5 días	vie 04/12/09	jue 10/12/09	13
15	**Programación del robot**	**111 días**	**vie 11/12/09**	**vie 14/05/10**	
16	¿FreeRTOS o programación Standalone?	3 días	vie 11/12/09	mar 15/12/09	14
17	Programación del microcontrolador	108 días	mié 16/12/09	vie 14/05/10	16
18	**Programación de la aplicación de control**	**55 días**	**lun 01/03/10**	**vie 14/05/10**	
19	Estudio de la comunicación serie en Visual Basic 6.0	2 días	lun 01/03/10	mar 02/03/10	
20	Creación del programa	36 días	vie 26/03/10	vie 14/05/10	19
21	**Construcción del robot**	**7 días**	**vie 04/12/09**	**lun 14/12/09**	
22	**Definición de partes y componentes necesarios**	**3 días**	**vie 04/12/09**	**mar 08/12/09**	
23	Servomotores y sensores IR	2 días	vie 04/12/09	lun 07/12/09	13
24	Resto de elementos	1 día	mar 08/12/09	mar 08/12/09	23
25	Ensamblaje de las diferentes partes	4 días	mié 09/12/09	lun 14/12/09	24
26	**Pruebas y resultados**	**15 días**	**lun 17/05/10**	**vie 04/06/10**	
27	Testeo del sistema (aplicación de control + microcontrolador)	5 días	lun 17/05/10	vie 21/05/10	17
28	Análisis de errores y verificación	10 días	lun 24/05/10	vie 04/06/10	27
29	Documentación y memoria del PFC	126 días	vie 11/12/09	vie 04/06/10	14
30	Lectura del PFC	1 día	mar 06/07/10	mar 06/07/10	

Figura 2: Planificación del Proyecto

3. ARQUITECTURA AVR32

La compañía Atmel [1] opera en el mercado de los microcontroladores desde el año 1984 y su familia de dispositivos está compuesta entre otros, por microcontroladores basados en arquitectura ARM [2] y microcontroladores con arquitectura propia como los AVR8 y AVR32, utilizados en el proyecto.

Los microcontroladores AVR32, gracias a la arquitectura Harvard y a los múltiples buses de alta velocidad, garantizan un rendimiento excepcional y un bajo consumo energético gracias a los distintos modos de suspensión del MCU y del Escalado Dinámico de Frecuencias. Esta familia de microcontroladores se provee de herramientas de desarrollo gratuitas como el AVR32 Studio, que permiten empezar a desarrollar código C/C++ de forma sencilla.

A su vez, los AVR32 se descomponen en otros dos grupos:

- Los Microcontroladores **UC3 32-Bit Flash**, que disponen de instrucciones DSP y son capaces de alcanzar 91 DMIPS[(1)] a una frecuencia de 66 MHz, consiguiendo una eficiencia energética mejor que ningún otro chip de la competencia (1.3 mW / MHz).

- Los **AP7 32-Bit Application Processors**, que del mismo modo que los anteriores, disponen de instrucciones DSP y también SIMD[(2)]. A diferencia de la familia UC3 32-Bit, estos proporcionan un rendimiento de 210 DMIPS a una frecuencia de 150 MHz, y disponen de soporte total para Linux.

Para la realización del robot, se dispone del microcontrolador AT32UC3A0512, que pertenece a la familia de micros AVR32 UC3, construidos sobre los núcleos AVR32 UC, diseñados para el desarrollo de aplicaciones embebidas que requieran de alto rendimiento y memoria integrada en el propio chip, además de un comportamiento en tiempo real y bajo consumo energético.

Estos procesadores multiplican por un factor de dos las prestaciones de su competidor más directo para un mismo código (reduce entre un 5% y un 20% el código generado al compilar), y son comparables al ARM-Cortex M3 en número de puertas, aunque el AVR32 UC es el único núcleo de 32-Bit de su rango que incluye instrucciones DSP que son ejecutadas en un solo ciclo de reloj. Además, se trata del primer núcleo en integrar la memoria SRAM en el propio Pipeline, permitiendo la lectura y escritura en la memoria en un solo ciclo de reloj.

3.1 ARQUITECTURA

El núcleo AVR32UC está formado principalmente por un pipeline de 3 etapas por ciclo, diseñado especialmente para optimizar la entrega de instrucciones desde la memoria Flash. Sin la memoria Flash integrada en el propio chip, no sería posible hacer funcionar la CPU a máxima velocidad, sin que esta tuviera que esperar para recibir las próximas instrucciones. Además y entre otros, está formado por una unidad de protección de memoria (MPU), que se encargará de proteger las regiones de memoria protegidas, el acceso a la memoria y las declaraciones que se hacen en esta y que es indispensable para la implementación de sistemas operativos en tiempo real o por ejemplo el Power Manager, que es el encargado de controlar los osciladores o los PLLs y de generar las señales de reloj y reset del dispositivo. La

[1] Dhrystone-MIP (DMIP) es un test computacional sintético desarrollado en 1984 por Reinhold P. Weicker. La salida de este test, proporciona el número de iteraciones Dhrystone por segundo, ejecutadas en el microcontrolador.
[2] SIMD son un conjunto de instrucciones que aplican una misma operación sobre un conjunto de datos.

Figura 3 muestra el diagrama de bloques del núcleo AVR32UC, donde se pueden ver los ya indicados anteriormente además de otros.

El núcleo AVR32 UC, además de aceptar instrucciones DSP de un solo ciclo de reloj, da soporte a eventos como las interrupciones no enmascarables (NMI), a excepciones y a otros cuatro tipos de interrupciones con niveles de prioridad diferentes. De este modo, los eventos que tengan prioridades mayores a los eventos con menos, podrán avanzar en la cola de espera de forma automática. También puede operar en modo privilegiado o no. Este modo es especialmente usado en los sistemas operativos en tiempo real, permitiendo acceso a todos los recursos del sistema y usando una pila de sistema separada**.**

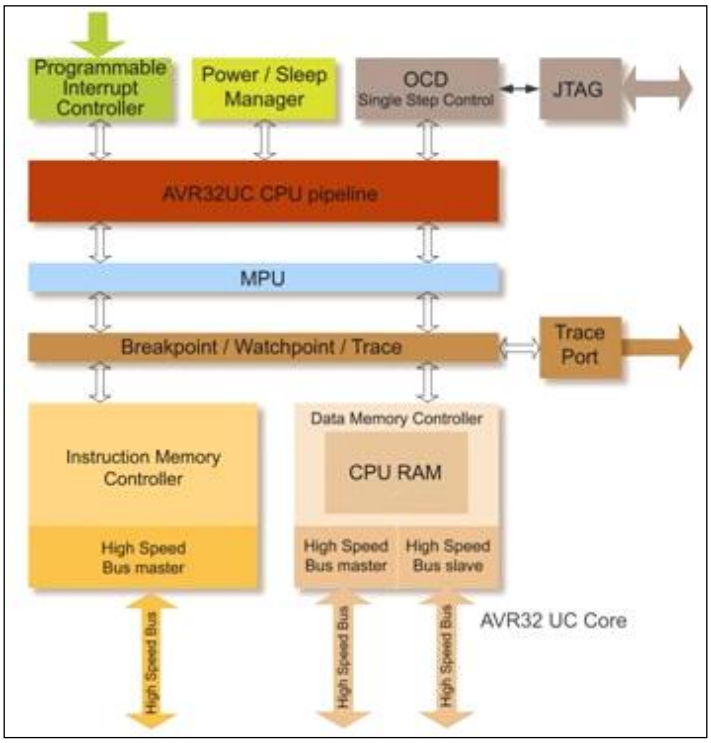

Figura 3: Diagrama de bloques del núcleo AVR32 UC3

3.2 CPU

El procesador de este microcontrolador AVR32UC está formado por un pipeline de tres etapas (*Figura 4*), que son: la etapa de **Instruction Fetch** (IF), **Instruction Decode** (ID) y la etapa **Execute** (EX), por lo que las instrucciones deben ser ejecutadas en ese orden (aunque algunas requerirán pasar varias veces por la etapa EX para ser completadas).

3.2.1 PREFETCH UNIT

La primera etapa del pipeline está compuesta por el módulo IF, y consiste en precargar una instrucción de 32 bits o 2 de 16 bits por ciclo de reloj en buffers FIFO internos, y de este modo alimentar a la etapa siguiente. Al mismo tiempo que se cargan las instrucciones, otras (ya sean RISC, extendidas o compactas), son entregadas a la etapa de descodificación.

3.2.2 DECODE UNIT

La segunda etapa se encarga de decodificar las instrucciones y generar las señales necesarias para la correcta ejecución de estas. Esta etapa acepta una instrucción por ciclo de reloj proveniente de la Prefetch Unit, de modo que la instrucción es decodificada y es entonces, cuando se generan las señales de control y las direcciones de los ficheros de registros. En el caso de que una instrucción no pueda ser decodificada, de que sea ilegal o que esté incompleta, una excepción es producida interrumpiendo la ejecución de esta.

3.2.3 THE EXECUTE

La tercera y última etapa es la encargada de realizar las lecturas, escrituras y operaciones sobre la memoria y los ficheros de registros. Esta etapa se subdivide en 3 sub-etapas: la ALU (Unidad Aritmético Lógica), la sub unidad de Multiplicación y las unidades de lectura y escritura.

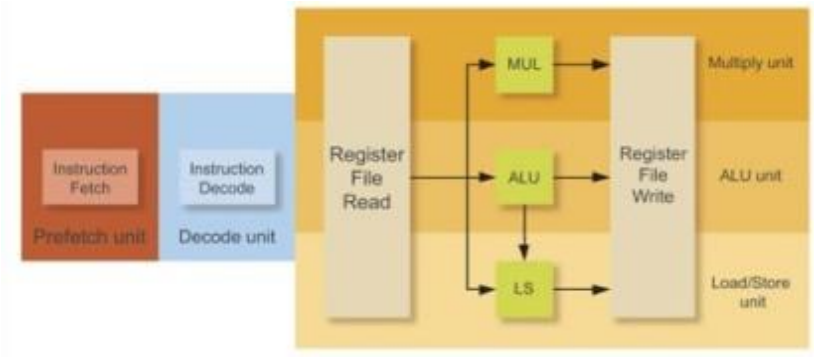

Figura 4: Etapas del Pipeline del AVR32 UC3

3.3 LA MEMORIA

La *Tabla 1* muestra las diferentes combinaciones disponibles para el núcleo AT32UC3A. En el caso a estudio, el microcontrolador utilizado es el AT32UC3A0512, que dispone de 512 KBytes de memoria Flash y 64 KBytes de SRAM.

Como se puede ver, las diferencias entre versiones de la familia AT32UC3A radican en la cantidad de memoria disponible. Para este proyecto, se dispone de una memoria interna de 512 KBytes y una SRAM de 64 KBytes. En caso de necesitar más cantidad de memoria, se puede acceder a las memorias externas que se encuentran en la placa EVK1100 o incluso almacenar datos en el lector de tarjetas SD/MMC que incorpora, de modo que la cantidad de memoria disponible se dispara.

Dispositivo	Flash	SRAM	Encapsulado
AT32UC3A0512	512 Kbytes	64 Kbytes	LQFP 144
AT32UC3A0256	256 Kbytes	64 Kbytes	LQFP 144
AT32UC3A0128	128 Kbytes	32 Kbytes	LQFP 144
AT32UC3A1512	512 Kbytes	64 Kbytes	TQFP 100
AT32UC3A1256	256 Kbytes	64 Kbytes	TQFP 100
AT32UC3A1128	128 Kbytes	32 Kbytes	TQFP 100

Tabla 1: Cantidad de memoria según el modelo

El espacio de memoria se divide en los segmentos definidos en la *Tabla 2.* Estos segmentos tienen direcciones de memoria pre-asignadas y no pueden ser modificadas, aunque como ya se verá más adelante, este no será un factor a tener en cuenta.

Device	Start Address	Size
Embedded SRAM	0x0000_0000	64 Kbytes
Embedded Flash	0x8000_0000	512 Kbytes
EBI SRAM CS0	0xC000_0000	16 Mbytes
EBI SRAM CS2	0xC800_0000	16 Mbytes
EBI SRAM CS3	0xCC00_0000	16 Mbytes
EBI SRAM CS1 / SDRAM CS0	0xD000_0000	128 Mbytes
USB Configuration	0xE000_0000	64 Kbytes
HSB-PB Bridge A	0xFFFE_0000	64 Kbytes
HSB-PB Bridge B	0xFFFF_0000	64 Kbytes

Tabla 2: Direcciones de memoria del AVR32 UC3

3.4 REGISTROS

El fichero de registros está organizado en 16 registros de 32 bits y que incluyen, entre otros, el Program Counter, el Link Register o el Stack Pointer. Adicionalmente, el registro R12 está diseñado para mantener los valores devueltos por las funciones que son llamadas desde la aplicación o cuando es usado implícitamente por algunas instrucciones.

Por otro lado, la arquitectura del AVR32UC no implementa hardware dedicado a los ficheros de registros de interrupciones ni tampoco registros para las direcciones de retorno o retorno de estados. A cambio, toda esta información se almacena en la pila del sistema (System Stack), permitiendo ahorrar en área del chip a costa de un tratamiento más lento de las interrupciones. La *Figura 5* muestra los diferentes ficheros de registros, aunque como ya se ha comentado, los registros de interrupciones, registros para las direcciones de retorno o registros para retorno de estados, no están implementadas a nivel hardware.

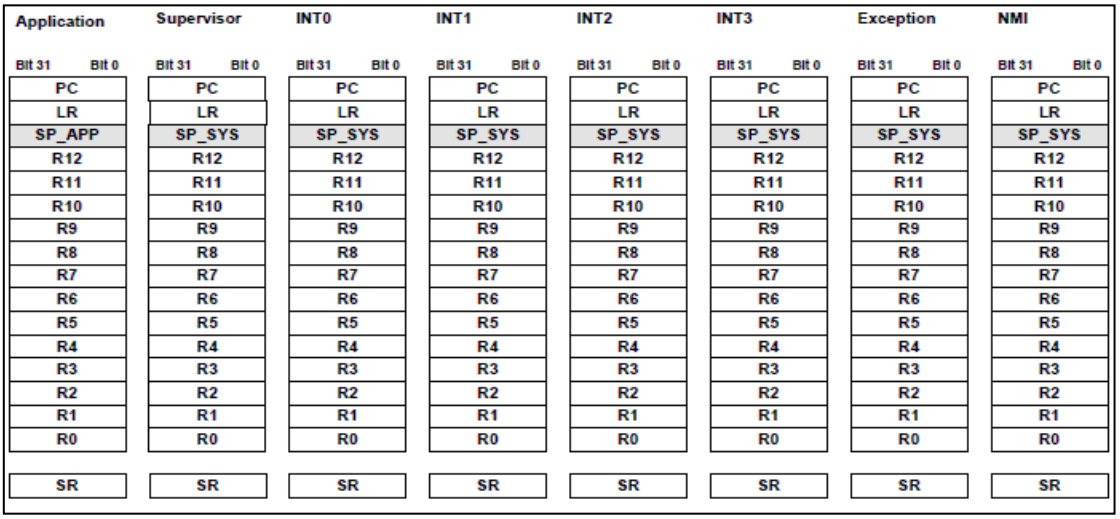

Figura 5: Ficheros de Registro del AVR32

El registro SR (Status Register) está dividido en dos partes, la primera superior y la segunda inferior que podemos verlas en la *Figura 6* (parte superior) y en el Datasheet [3] (parte inferior). Para más información acerca de los registros se debe consultar el manual del AVR32 [3].

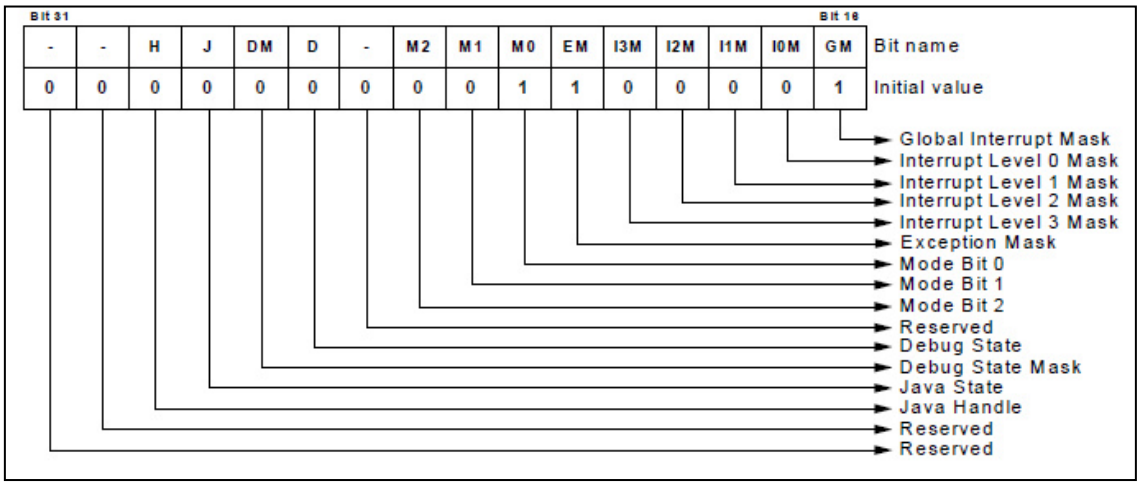

Figura 6: Parte superior del registro SR (Status Register)

Los registros de sistema están colocados fuera del espacio de memoria virtual del microcontrolador y sólo son accesibles utilizando instrucciones con permisos privilegiados como son la *mfsr* y *mtsr.* De este modo, es el programador el responsable de mantener la correcta secuencia de uso de las instrucciones anteriores y derivar en él las responsabilidades de su uso.

3.5 INTERRUPCIONES Y EXCEPCIONES

En ocasiones, la CPU se verá obligada a abortar la ejecución normal del programa para poder atender eventos especiales o que tenga mayores prioridades. Tradicionalmente se han llamado excepciones a los eventos generados internamente en la CPU y interrupciones a los eventos externos.

En este sentido, el AVR32 proporciona herramientas potentes para el control de eventos, de modo que los diferentes eventos que se produzcan tengan bien definidos sus niveles de prioridad para que no existan conflictos en caso de la recepción de múltiples eventos al mismo tiempo. Cuando uno de estos eventos aparece, la ejecución normal es “congelada” y se procede a tratar esta de forma separada. Una vez completada, se retoma la ejecución normal del programa.

Cada una de las etapas del pipeline posee un registro que mantiene el valor de la petición de excepción asociada a una instrucción en esa etapa del pipeline, que permitirá más tarde continuar con la ejecución normal de la instrucción “contaminada”. Las excepciones son detectadas en dos etapas del pipeline. La etapa EX (3.2.3) detecta todas las excepciones relacionadas con las direcciones de datos. Por otro lado, todas las otras excepciones incluidas las interrupciones, son detectadas por la etapa ID.

3.5.1 SUPERVISOR CALLS

La arquitectura AVR32 tiene definida una instrucción que permite ejecutar instrucciones en modo supervisor. Esta instrucción, llamada *scall*, está diseñada específicamente para poder

ejecutarse en cualquier contexto y que esta pueda ejecutar rutinas que requieran de privilegios de supervisor.

3.5.2 DEBUG REQUESTS

Por otro lado, esta arquitectura dispone además de interrupciones dedicadas al modo debug. Cuando una de estas peticiones es recibida por el núcleo, todo él pasa a modo debug.

3.6 MEMORY PROTECTION UNIT

La arquitectura del AVR32 define como opción la inserción de una Unidad de Protección de Memoria (MPU). De hecho, se trata de una simple alternativa a la inserción de una MMU (Memory Management Unit) completa, pero que permite proteger del mismo modo la memoria. Esta unidad permite al usuario dividir la memoria en diferentes espacios protegidos (con un máximo de 8), de modo que su espacio está definido y tiene comienzo en la dirección de memoria que el usuario especifica. A su vez, cada región es dividida en 16 subregiones, las cuales pueden ser definidas con 1 o 2 series de permisos diferentes. El número de regiones protegidas implementadas se almacena en el campo *DMMU SZ* del registro de sistema *CONFIG1*.

La MPU es la responsable de chequear que todas las transferencias de datos en la memoria tienen los permisos correctos para que estas puedan completarse. Por ejemplo, si un acceso a memoria es realizado con permisos incorrectos o se intenta acceder a una dirección de memoria que no reside en ninguna región protegida, una excepción es generada y el acceso es cancelado. Por supuesto y como ya se ha dicho, el usuario tiene a su disposición crear diferentes regiones de acceso a memoria con los permisos que él desee, de modo que todos los accesos a memoria (protegida) se produzcan sin ningún tipo de problema.

El espacio de las regiones protegidas puede variar desde los 4 Kbytes hasta los 4 Gbytes (siempre que dispongamos de esa cantidad de espacio), y siempre debe corresponder a una potencia de dos. Cuando un acceso es realizado a una región de memoria seleccionada por la MPU, el propio hardware procede a determinar que subregión es la más apropiada para almacenar los datos. Por el contrario, si se accede a otro espacio de memoria, la transferencia es abortada inmediatamente.

Si desea activar el uso de la MPU, deberá activar el *bit E* en el registro *MPUCR*. En caso de no activarlo, los accesos a memoria se producirán sin ningún tipo de violación en el acceso. Para definir una región de memoria protegida, se deberá hacer uso del registro *MPUARn* (MPU Address Register), donde se define (*Figura 7*) la dirección de inicio de memoria y el tamaño de la región.

Figura 7: Memory Protection Unit Address Register

- **Base Address:** Este campo indica el inicio de la región de memoria. Para definir las direcciones de memoria estas deben ser alineadas con su correspondiente tamaño. Es decir, como el tamaño mínimo de una región de memoria es de 4 KB, únicamente se hace uso de los 20 bits más significativos del campo “Base Address”. El resto de bits,

simplemente deberán ser puestos a 0 y en caso de ser un tamaño distinto, hacerlo acordemente, ya que en caso contrario, la memoria quedará indefinida.

- **Size:** Indicará el tamaño de memoria (siempre en potencias de 2).
- **V:** Siempre que la región protegida sea válida, este flag será marcado a 1. En caso contrario o cuando se realice un reset, este campo será marcado como 0 y por lo tanto, no será considerada la región protegida.

Por otro lado, el registro MPUAPR (MPU Access Permission Register A) (*Figura 8*) indica cuales son los permisos que tendrá cada región. Cada vez que se haga un reset del sistema, este campo será puesto a 0, con lo que los permisos desaparecerán.

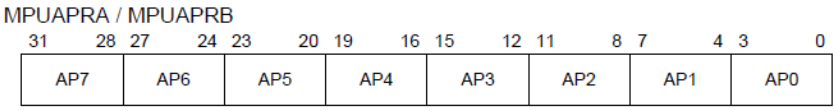

Figura 8: Memory Protection Unit Access Permission Register

3.7 HSB (MATRIZ DE BUSES)

Todos los buses utilizados en el microcontrolador están integrados dentro de la matriz de buses de alta velocidad. Esta matriz, implementa una estructura que permite el acceso paralelo entre múltiples buses de alta velocidad (hasta 16 maestros o 16 esclavos, *Figura 9*) del sistema, incrementando notablemente el ancho de banda global. Además, esta matriz incorpora 16 registros para funciones especiales que dan soporte a las aplicaciones para hacer uso de características especiales y proporciona un decodificador por cada interfaz maestra del bus, permitiendo que cada bus pueda mapear la memoria de forma distinta.

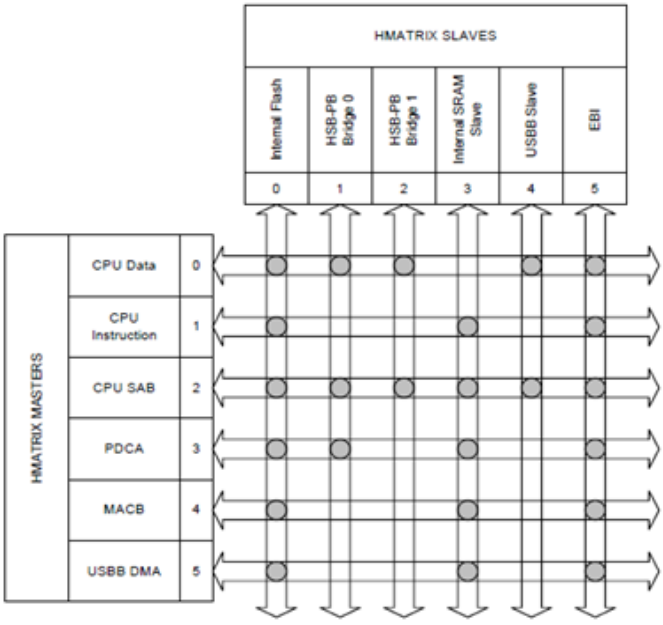

Figura 9: La matriz de buses (HSB)

Todos los módulos conectados a un mismo bus usan el mismo reloj, aunque este reloj puede ser modificado por el Power Manager.

La matriz HSB, proporciona al AVR32UC tres interfaces de memoria. La primera de ellas se encuentran conectada a un bus maestro para la etapa de Instruction Fetch del pipeline, otra para el acceso a los datos y una tercera permite a otros buses maestros acceder a la memoria RAM interna de la CPU. Esto permite mantener la memoria RAM en el interior de la CPU mejorando el acceso a esta, reduciendo las latencias y garantizando un tiempo determinista. Además, el consumo energético es reducido al no necesitar un bus completo para acceder a la memoria.

3.8 SISTEMA OCD

Los microcontroladores AVR32 están orientados a abarcar un gran número de aplicaciones distintas. Es por eso que, ya que se espera de estos dispositivos una gran velocidad y flexibilidad, también se espera que sus posibilidades de testeo sean lo más altas posibles. El AVR32 incluye el sistema OCD (On Chip Debugging) que proporciona una herramienta flexible y poderosa para realizar debug sobre el propio chip. Esta interfaz de testeo, proporciona al debugger externo acceso a la lógica del chip a través del puerto JTAG.

3.9 POWER MANAGER

El Power Manager es el encargado de controlar los osciladores, PLLs y de generar las señales de reloj y reset del dispositivo. Como controla los dos osciladores de cristal junto con los dos PLLs, estos pueden ser usados para multiplicar la frecuencia del reloj consiguiendo frecuencias de funcionamiento mayores, lo que se traduce en más velocidad de cálculo.

Los relojes proporcionados son divididos en dos grupos: los relojes síncronos y los genéricos. El primero de ellos es usado como reloj principal para la lógica digital del dispositivo, mientras que el reloj genérico puede ser utilizado para la conexión de periféricos que requieran frecuencias específicas como pueden ser “Timers” o módulos de comunicación. Además, los relojes síncronos están divididos en tres dominios que permiten habilitarlos o deshabilitarlos o incluso hacerlos que funcionen a frecuencias distintas, lo que permite ahorrar energía, haciendo funcionar los periféricos a una frecuencia baja, mientras que la CPU seguirá trabajando a plena carga. Esta capacidad que presenta el microcontrolador, puede ser efectuada “en caliente”, es decir, las frecuencias del dispositivo pueden ser modificadas, una vez este se encuentre en marcha y sin que por ello, el sistema o los dispositivos se vean afectados.

3.10 REAL TIME COUNTER

El Real Time Counter (RTC) permite activar interrupciones que sean lanzadas después de largos intérvalos de tiempo (hasta 272 años reales) o también, medir con precisión (resolución máxima de 16 KHz), secuencias de tiempo real. Está formado un prescaler de 16 bit, el cual está conectado a un oscilador RC de 32 KHz. El prescaler puede ser programado como se desee, de modo que permitirá escoger mayores o menores resolución de tiempos.

3.11 TIMER/COUNTER

El Timer Counter (TC) incluye 3 canales idénticos de 16 bits. Cada canal puede ser programado de forma independiente para realizar una amplia variedad de funciones que incluyen medidas de frecuencia, contador de eventos, medidas de intérvalos de tiempo, generación de pulsos, retrasos temporales y pulsos PWM.

Cada canal dispone de 3 entradas de relojes externos, 5 de relojes internos y 2 entradas de propósito general (tanto de entrada como de salida) que pueden ser configuradas por el usuario. Además, disponen de interrupciones internas que pueden ser programadas de forma independiente para cada canal. Sus principales características son:

3.12 INTERRUPT CONTROLLER

El INTC (Interrupt Controller) recoge todas las interrupciones generadas por los periféricos, priorizándolas y entrega una petición de interrupción a la CPU. La arquitectura del AVR32 soporta hasta 4 niveles de prioridades para las interrupciones, donde estas se dividen en hasta 64 grupos de interrupciones diferentes. Cada grupo dispone de 32 líneas de petición de interrupción. Si varios grupos tienen pendientes interrupciones del mismo nivel, el grupo con el número menor es el que toma la prioridad.

3.13 EXTERNAL INTERRUPTS CONTROLLER

El módulo de Interrupciones Externas, permite a los diferentes pines del MCU actuar como pines para recibir interrupciones externas. Estas interrupciones pueden ser generadas a nivel bajo o alto de la señal, o en el flanco de subida o de bajada, pero para evitar interrupciones “falsas”, cada línea tiene un filtro configurable que permite eliminar posibles glitches que aparezcan en la línea. Este tipo de controlador admite la conexión de un teclado externo, de modo que cada vez que se presione una tecla, esta generará una interrupción que será identificada por el módulo.

3.14 INTERFACES DE ENTRADA Y SALIDA

Además de las características antes mencionadas, interesa ver cuáles son las capacidades a nivel de comunicación que el AVR32 ofrece. Estas interfaces permitirán la comunicación entre el microcontrolador y los diferentes periféricos que se desee utilizar. A continuación se detallan algunas de las interfaces más importantes que este dispositivo ofrece.

3.14.1 GPIO

El controlador GPIO (General-Purpose Input/Output Controller) es el responsable de controlar todos los pines de entrada y salida del microcontrolador. Cada una de las líneas del MCU puede ser utilizada como un puerto de propósito general (tanto de entrada como de salida) o puede asignarse una de las funciones disponibles en ese PIN. De este modo, se asegura la optimización de todos los pines del producto, permitiendo utilizar los 109 pines del microcontrolador a modo de propósito general (encender un LED, activar un Relé…).

Cada puerto es capaz de multiplexar hasta 4 funciones periféricas y además, todos ellos disponen de un filtro anti-glitch, de modo que los pulsos que sean más cortos de que un ciclo de reloj, serán rechazados.

3.14.2 ADC

Un conversor ADC puede convertir un voltaje en un número binario digital. Los conversores A/D son utilizados en cualquier lugar donde sea necesario procesar una señal, almacenarla o transportarla en forma digital. Los puertos ADC (Analog-to-Digital Converter) incluidos en el AVR32, están basados en Conversores Analógicos – Digital por Registros de Aproximación Sucesiva (SAR) de 10 bits.

El ADC soporta dos modos de resolución, 8 bits o 10 bits, dando como resultado una conversión que es reportada a un registro común para todos los canales. Además, incorpora un modo “Sleep” que reduce el consumo de potencia del MCU.

La resolución del conversor indica el número de valores discretos que se pueden obtener dependiendo del rango del voltaje de entrada. Esta resolución se traduce en 256 valores para el caso de los 8 bits o para 1024 para el caso de los 10 bits.

3.14.3 PWM

Una canal PWM (Pulse Width Modulation Controller) permite generar señales cuadradas que pueden ser configuradas según se desee, permitiendo modificar características como el periodo, el duty-cicle y la polaridad de la señal. Estas celdas son capaces de controlar varios canales independientemente, donde cada canal, controla una salida que proporciona una señal cuadrada.

El AVR32 dispone de 7 canales independientes con contadores de 20 bits cada uno. Cada canal puede seleccionar de forma independiente, un clock diferente (de 13 posibles a escoger), un periodo o un duty-cycle. Además, la polaridad y la situación de la señal puede ser programada.

3.14.4 UNIVERSAL SYNC/ASYNC RECEIVER/TRANSMITTER

Los USART permiten la transmisión de datos a través de un canal full dúplex universal a través de un puerto serie. El formato de los datos es ampliamente programable de forma que admite una gran variedad de estándares. El receptor implementa un código detector de errores por paridad y además, permite la transmisión con dispositivos más “lentos” gracias al módulo time-out, que permite la detección de trazas de datos de longitud variable.

Además, el USART posee tres modos de test que son: el loopback remoto, el loopback local y el “echo” automático. También soporta la conexión de periféricos con controladores DMA, permitiendo la transferencia de datos desde el transmisor al receptor.

3.14.5 TWO WIRE INTERFACE

El AVR32 dispone de un canal TWI. Este canal, permite interconectar componentes a través de un bus de dos cables. El primero de estos cables permite transmitir la señal de reloj, mientras

que el segundo transmite y recibe datos a una velocidad de hasta 400 Kbits por segundo. Es compatible con el estándar I2C y programable como maestro o esclavo.

4. COMPARATIVA

En el mercado existen muchas alternativas a los microcontroladores de AVR32. Una sola compañía puede disponer de varios modelos, que sumado al elevado número de casas fabricantes de microcontroladores que hay, dan como resultado un amplio abanico de opciones para escoger. Es por ello que determinar que microcontrolador es el más idóneo para la tarea que se va realizar, debe ser parte de un estudio exhaustivo por parte del desarrollador.

Este capítulo pretende hacer una introducción al estado actual del mercado de los microcontroladores y realizar una pequeña comparativa entre las arquitecturas del AVR8 y AVR32 y con la de sus competidores directos los ARM9 y ARM11. Además, se presenta de forma breve, cual es el estado actual del mercado de los microcontroladores.

4.1 AVR32 VS AVR8

La arquitectura AVR32 apareció recientemente en el mercado de MCUs, pero eso no quiere decir que Atmel no se encontrara en el mercado anteriormente. El microcontrolador AVR8 (que a continuación se presenta) es altamente utilizado, disponiendo de una de las relaciones consumo/potencia más bajas del mercado. Estos microcontroladores disponen de 8 o 16 bits (según modelo) y son capaces de ejecutar instrucciones RISC en un solo ciclo de reloj.

A continuación se hace un breve resumen de las características más importantes de las que disponen estos microcontroladores y una comparativa respecto a los nuevos AVR32.

4.1.1 AVR 8-BIT RISC

La arquitectura AVR8 es una arquitectura pensada para ofrecer altas prestaciones minimizando el consumo energético. Haciendo uso de una arquitectura tipo RISC, ejecuta sus instrucciones en un solo ciclo de reloj, gracias al pipeline de una sola fase que incluye, ofreciendo un rendimiento de 1 MIPS por MHz. Algunas de sus características más importantes son:

- Arquitectura RISC capaz de ejecutar instrucciones en 1 sólo ciclo de reloj.
- Velocidad de funcionamiento de hasta 32 MHz, ofreciendo hasta 1 MIPS por MHz.
- Uso de la arquitectura Harvard.
- 32 registros de propósito general.

Gracias a estos 32 registros, el AVR8 ofrece una gran flexibilidad, especialmente cuando se programa en lenguajes de alto nivel como pueden ser C, Pascal o Basic.

Su consumo energético es realmente bajo debido a que es capaz de operar en niveles de tensión tan bajos como 1.8V y además de disponer de hasta 6 modos de reposo, los cuales son muy útiles cuando se realizan aplicaciones que ahorren energía ya que el microcontrolador es capaz de volver a su estado de funcionamiento normal de forma muy rápida ante un evento externo.

Además y siempre en vista de mejorar el consumo de estos microcontroladores, la frecuencia a la que opera el MCU puede ser controlada mediante el software que diseñemos.

Tal y como se puede ver en la *Figura 10*, la arquitectura AVR8 la componen 3 familias de microcontroladores:

- **TinyAVR:** Microcontroladores de propósito general con hasta 16 KBytes de memoria Flash programable y 512 Bytes de memoria SRAM.
- **MegaAVR:** Altas prestaciones gracias al multiplicador hardware que implementa. Además, dispone de 256 KBytes de memoria Flash, 4 KBytes de memoria EEPROM y 8 KBytes de SRAM.
- **XMEGA:** Los XMEGA son microcontroladores de 8 o 16 bits, que disponen de una serie de periféricos que incrementan el rendimiento de estos circuitos respecto a los MegaAVR, como pueden ser las controladoras DMA.

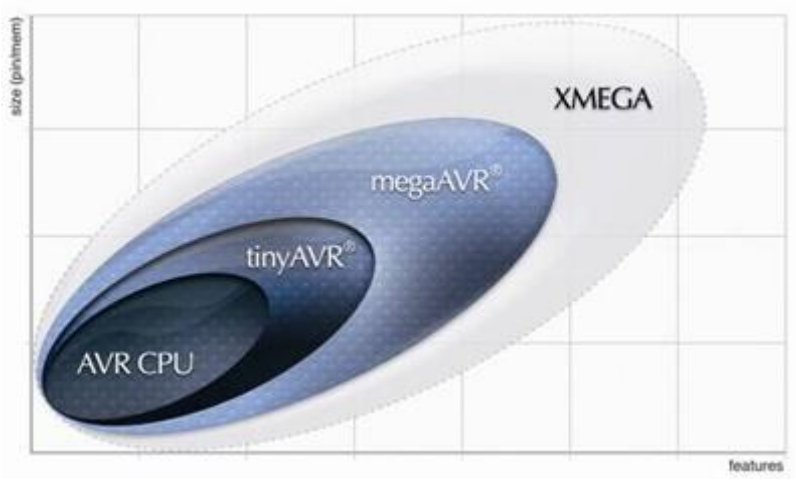

Figura 10: Escalabilidad de los microcontroladores AVR8

Escalabilidad: Una de las ventajas y características más importantes que aportan estos microcontroladores es la escalabilidad que ofrecen. Es decir, se puede reutilizar código para que sea utilizado en los distintos dispositivos de la familia AVR8 sin realizar grandes cambios. Por ejemplo, si se está utilizando un microcontrolador de características limitadas y para la realización del proyecto es preciso aumentar el rendimiento de este, puede ser sustituido por uno de la misma familia y que disponga de mejores características, sin realizar cambios importantes en el código del programa y usando siempre el mismo software de desarrollo.

4.1.2 PRINCIPALES DIFERENCIAS

Ya que el rendimiento de los nuevos microcontroladores AVR32 se sitúa más cerca de los modelos XMEGA que de los tinyAVR, la comparativa entre ambos productos se hará utilizando el microcontrolador AT32UC3A0512 y los ATxmega384A1. La *Figura 11* muestra las diferencias entre arquitecturas y a continuación se detallan algunos de los aspectos más importantes a tener en cuenta entre un dispositivo y el otro:

Número de bits: Para conocer la importancia que tienen el número de bits en el funcionamiento de los microcontroladores, primero se ha de comprender algunos conceptos de arquitectura de microcontroladores como los explicados a continuación. Los microcontroladores AVR hacen uso de un juego de instrucciones tipo RISC (Reduced Instruction Set Computer), de modo que, todas las instrucciones están limitadas por tamaño al número de bits que pueda manejar el microcontrolador. En el caso de los AVR8, el número de bits se limita a 8/16, de modo que las instrucciones han de tener como máximo 8/16 bits de longitud. Para los AVR32 esto se duplica hasta los 32 bits, que sumado al hecho de que estas instrucciones son ejecutadas en un ciclo de reloj, permite hacer uso de instrucciones más complejas ganando en velocidad de procesado.

Además de velocidad de procesado, el número de bits indica la cantidad de memoria que puede direccionar el microcontrolador. Para el AVR32 tenemos un total de 2^{32} bits, mientras que para el AVR8 en el mejor de los casos la cantidad de memoria se reduce a 2^{16} bits.

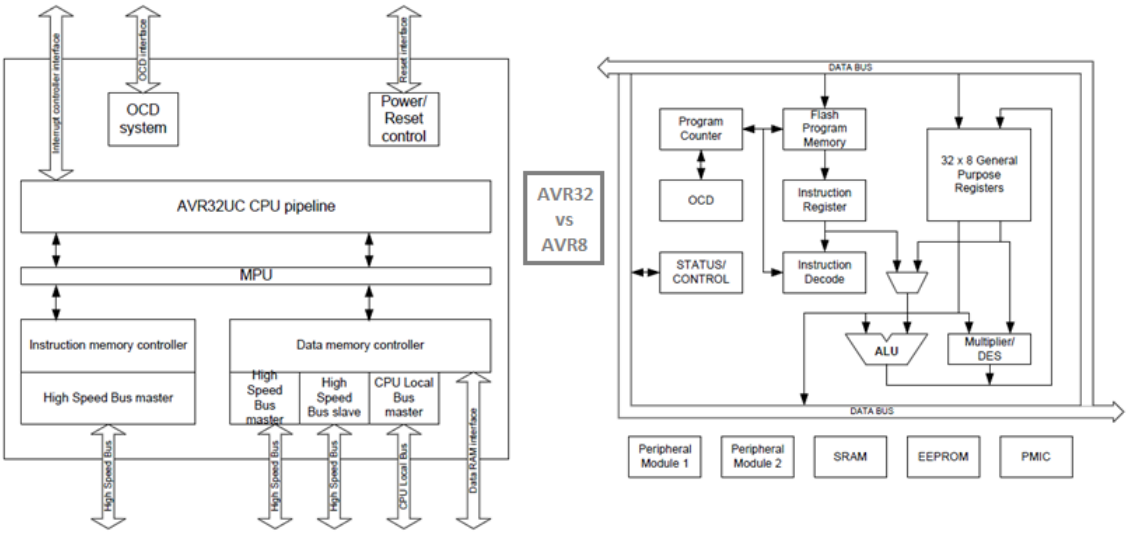

Figura 11: Diagrama de bloques de la arquitectura AVR32 (derecha) vs AVR8 (Izquierda)

Velocidad de reloj: Las frecuencias de reloj utilizadas en los AVR8 se encontraban como máximo en valores de 32 MHz, ofreciendo un rendimiento máximo de 1 DMIPS por cada MHz de funcionamiento del reloj. Esta velocidad se aumenta hasta los 66 MHz del AT32UC3A0512, consiguiendo un rendimiento de 1.49 DMIPS por MHz utilizado, con un máximo de 91 DMIPS a plena potencia.

En términos de potencia, los nuevos microcontroladores están multiplicando por más de 4 veces la capacidad que tenían sus antecesores, de modo que pueden ser utilizados para aplicaciones que requieran de una mayor potencia y donde los AVR8 se queden atrás. Entre otros, permitirá hacer correr un sistema operativo en tiempo real, que coordine y gestione todas las funciones que tenga que desarrollar el dispositivo.

A pesar de las velocidades, ambos microcontroladores son capaces de ejecutar instrucciones RISC en un solo ciclo de reloj, lo cual los sitúa muy por delante de algunos de sus competidores, como pueden ser los microcontroladores PIC, que necesitan hasta 4 ciclos para realizar la misma instrucción.

Memoria: A nivel de memoria Flash, se dobla la cantidad respecto del AVR8, llegando a los 512 KBytes de memoria interna en el MCU, frente los 256 KBytes del AVR8. Si bien, la memoria es un factor variable según el modelo escogido dentro de una misma familia de microcontroladores, en el mejor de los casos es el expuesto anteriormente. Una nueva ventaja que se ofrece para estas nuevas memorias, es poder acceder a ellas en 1 sólo ciclo de reloj siempre y cuando la velocidad de operación del MCU no sobrepase los 33 MHz, mejorando el rendimiento global.

La *Figura 12* [10] muestra en un gráfico, donde se encontraría cada uno de los microcontroladores de AVR, en función de la potencia y el consumo energético. Como se puede ver, los AVR32 UC3 consiguen unos consumos muy parecidos a los del MegaAVR, pero es superado por los AVR32 AP7, los cuales disponen de la mejor relación de potencia y consumo energético.

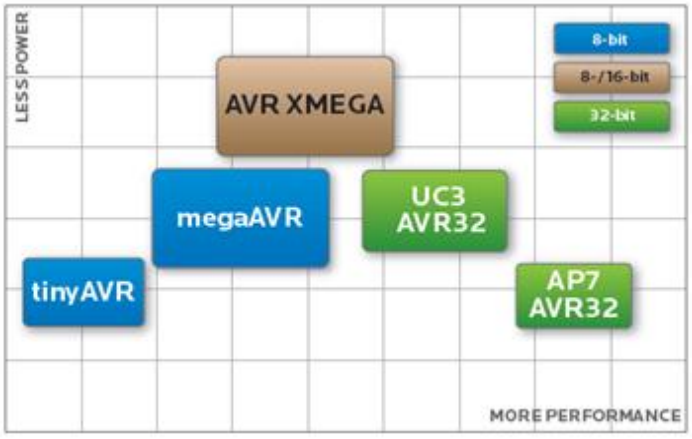

Figura 12: Relación Consumo / Potencia de los microcontroladores AVR8 y AVR32

Además, y ya que la diferencia no radica únicamente en los 5 puntos explicados anteriormente, a continuación se muestra la *Tabla 3* donde se detallan todas las diferencias que existen entre estos dos microchips.

Característica	AVR8 (ATxmega256A3B)	AVR32 (AT32UC3A0512)
Encapsulado	100 Pines	144 Pines
Velocidad (MHz)	32 MHz	66 MHz
DMIPS	32 DMIPS	91 DMIPS
Memoria Flash	384 KBytes	512 KBytes
Memoria RAM	32 KBytes	64 KBytes
IO	78	109
Timers / Counters	8, 16 Bits	3, 16 Bits
Canales ADC	12 Bits, 16 Canales	10 Bits, 8 Canales
Canales DAC	12 Bits, 4 Canales	Ninguno
Comunicación	8 USART, 4 SPI	4USARTs, USB2.0, Ethernet…

Tabla 3: Tabla paramétrica del AVR8 y el AVR32 [14]

4.2 AVR32 vs ARM

ARM (Advanced RISC Machines), es la familia de microprocesadores RISC más utilizada en el mundo (se calcula que cerca del 75% de los microprocesadores del mundo poseen un núcleo ARM). El diseño de estos microprocesadores comenzó en el año 1983 como un proyecto dentro de la empresa Acorn Computers, que años más tarde derivó en la creación de la empresa con el mismo nombre ARM. Entre sus más famosos microcontroladores se encuentran los de las familias ARM9, ARM11 y ARM Cortex, y todos soportan un gran rango de periféricos (ADC, DAC, USB, SPI, UART, I2C,…), lo que los sitúan en una de las mejores opciones del mercado. En general se puede afirmar, que ARM no se trata de una marca si no de una arquitectura y como tal, esta empresa licencia sus arquitecturas a otras.

De todos los modelos que ofrece ARM, los ARM9, ARM11 o Cortex M3, son los competidores directos de los AVR32 (en cuanto a prestaciones y precio). Estos núcleos presentan las siguientes características técnicas:

ARM9: Estos microcontroladores, poseen un núcleo de 32 bits RISC que incluye un pipeline de 5 etapas, logrando hasta 300 MIPS de potencia. Son soportados por una gran variedad de sistemas operativos entre los que se incluyen WindowsCE, Symbian OS o Linux.

ARM11: Los ARM11 son capaces de ofrecer hasta 740 MIPS, lo que los hacen perfectos para ser utilizados en PDA's, teléfonos móviles, videoconsolas o automoción, entre otros. Estos

microprocesadores ofrecen un consumo realmente bajo, llegando a los 0.6mW/MHz, además de disponer de modos de ahorro energético. A diferencia del ARM9, el pipeline en este caso es de 8 fases, lo que contribuye al incremento de potencia.

Cortex-M: La familia de microcontroladores Cortex-M está especialmente diseñada para su uso en aplicaciones que requieran de grandes prestaciones y número de puertas. Esta familia se divide en tres tipos de microcontroladores que son:

- Cortex M3 diseñado para su uso en microcontroladores.
- Cortex M1, para su implementación en FPGAs
- Cortex M0, que se trata del procesador de ultra bajo consumo más pequeño jamás creado por ARM

.

Es por ello, que comparativamente, nos interesa el estudio del Cortex M3. Entre otras sus características más importantes son:

- Núcleo de 32 bits de alto rendimiento
- Basado en un pipeline de 3 etapas y arquitectura Harvard
- Capaz de realizar instrucciones de multiplicación/división en un solo ciclo de reloj
- Hasta 1.25 DMIPS por MHz
- Dispone de las nuevas instrucciones Thumb-2 licenciadas por ARM

A priori, estos núcleos pueden parecer superiores en potencia respecto los AVR32, pero la realidad es bien distinta, ya que a igualdad de frecuencias, los AVR32 son superiores. Este hecho, se está haciendo más presente debido al uso de algoritmos complejos, donde se ve como la arquitectura AVR32 es mejor que la de ARM. Actualmente, el uso de sistemas de compresión de datos, codificación de señales, decodificación de datos o video, Transformadas de Fourier (FFTs) o Transformaciones de Cosenos Discretas, por ejemplo, ha provocado la necesidad de usar algoritmos DSP muy exigentes computacionalmente.

Históricamente, estos problemas se solventaban aumentando la velocidad del procesador o mediante la inclusión de varios núcleos en un mismo encapsulado. Sin embargo, las aplicaciones que hacen uso de complejos algoritmos DSP han aumentado, por lo que esta no es una vía eficiente de mejorar el rendimiento de los microcontroladores. Es por ello que se pueden llevar a cabo otras mejoras para aumentar el rendimiento computacional sin afectar al consumo energético, como pueden ser:

- Reducir el número de ciclos utilizados para los procesos de carga y lectura. Más del 30% de instrucciones utilizadas son de este tipo, por lo que su reducción implicará una mejora sustancial de rendimiento.
- Coordinar las operaciones repetitivas para que estas puedan ser ejecutas de forma múltiple.
- Maximizar la utilización de los recursos del Pipeline.
- Minimizar las latencias que se producen en las distintas instrucciones de salto (algunas pueden consumir hasta 5 ciclos de reloj cada una).
- Mejorar la densidad de código. Si este es pequeño, más instrucciones pueden ser almacenadas en la cache del micro y por lo tanto, se reduce el tráfico con las memorias externas.

Los microcontroladores AVR32 mejoran todos estos aspectos, viendo incrementado su rendimiento de forma excelente sin que se vea afectado su consumo energético. Las pruebas

(*Figura 13* y *Figura 14)* realizadas por Atmel [7], demuestran como la arquitectura AVR32 es notablemente mejor que la de ARM:

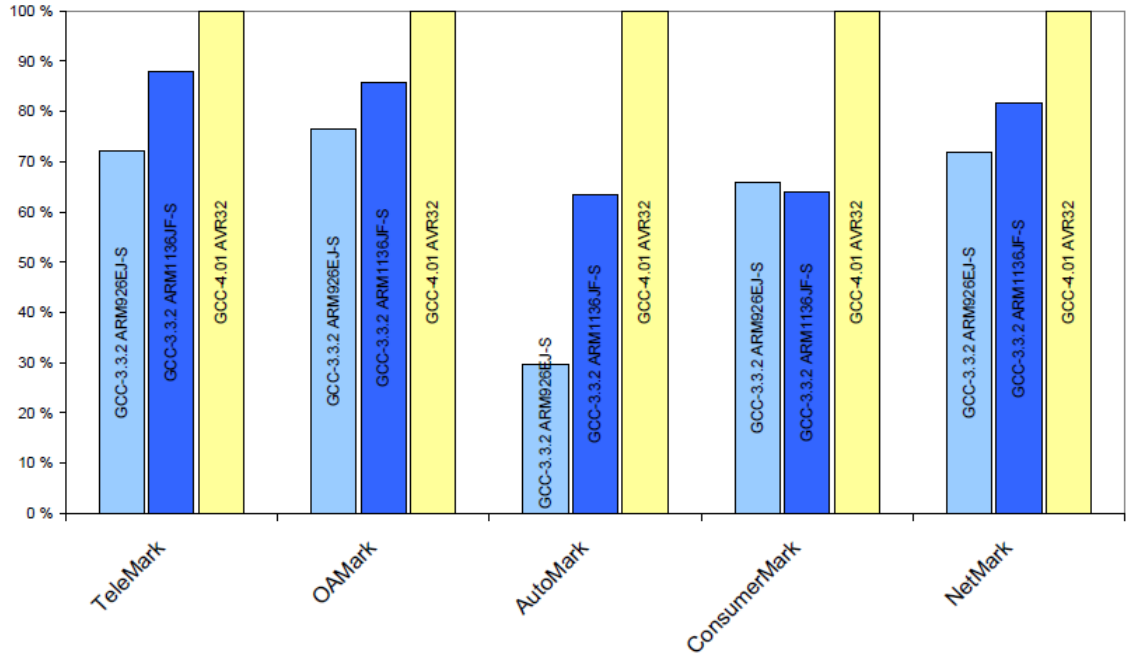

Figura 13: Pruebas de rendimiento normalizadas para AVR32, ARM9 y ARM11.

La primera de estas gráficas (*Figura 13*) muestra una serie de tests de rendimiento hechos sobre los microcontroladores AVR32, ARM9 y ARM11. Los datos ofrecidos en estas pruebas han sido obtenidos normalizando las frecuencias de todos los microcontroladores, de modo que se puedan realizar en las mismas condiciones. Como se puede ver, AVR32 es superior (alrededor de un 35%) a las otras dos arquitecturas en todas las pruebas realizadas: TeleMark™, OAMark™, AutoMark™, ConsumerMark™ y NetMark™.

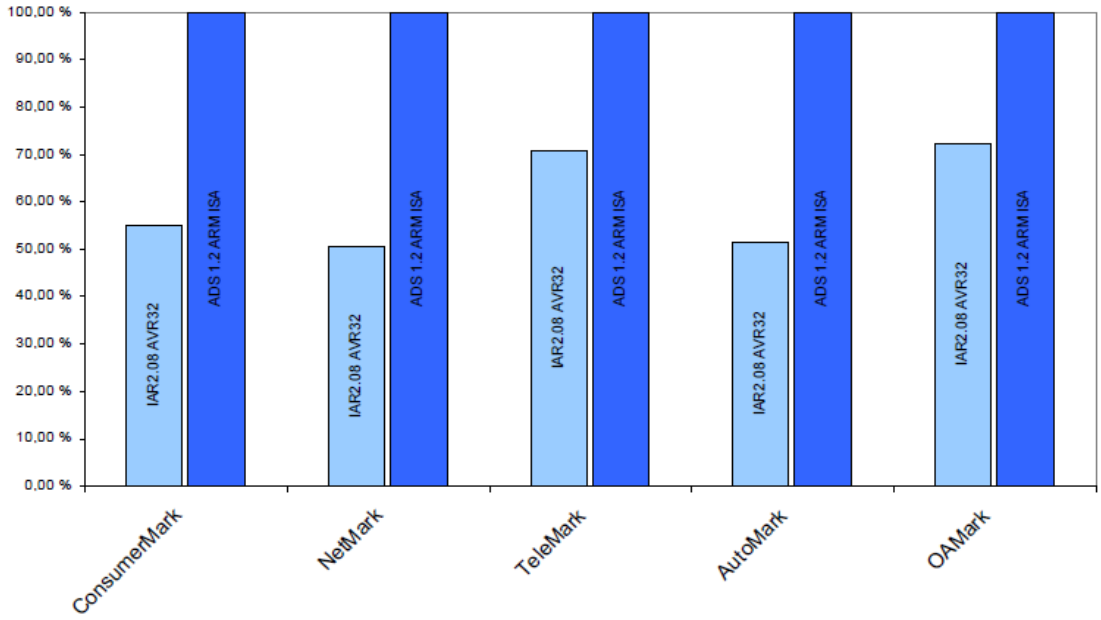

Figura 14: Pruebas de tamaño de código optimizado para la velocidad.

La *Figura 14*, muestra el tamaño de código utilizado para cada uno de las pruebas realizadas. En general, se puede afirmar que AVR32 requiere entre un 5% y un 20% menos de código que el empleado por ARM, para ejecutar las mismas instrucciones, incluso se puede decir que en

aplicaciones de alto rendimiento, cuando el código se encuentra optimizado para obtener una máxima velocidad de ejecución, el código de AVR32 es entre un 30% y un 50% más pequeño que el de la competencia.

Por lo tanto, para aplicaciones que requieran de alto rendimiento, un consumo energético bajo y de uso de instrucciones DSP, la arquitectura AVR32 se presenta superior en velocidad de procesado, densidad de código y consumo energético que la de su competencia.

4.3 OTRAS ALTERNATIVAS

Para finalizar con la comparativa entre microcontroladores, es interesante ver como son los diferentes productos que se pueden encontrar en el mercado. Es por eso que, todo y que ya se ha optado por la variante AT32UC3A0512 de la familia AVR32 UC3, a continuación se mostrarán otras alternativas existentes en el mercado:

4.3.1 FUJITSU

Su familia está formada por microcontroladores de 8, 16 y 32 bits, aunque principalmente orientada a la automoción, con modelos que incluyen el protocolo de comunicación FlexRay el cual pretende ser el sustituto del bus CAN. En general, no se trata de un competidor directo de la arquitectura AVR32, ya que todos sus modelos están claramente orientados a la automoción y no es el caso del MCU utilizado en este proyecto.

F^2MC: Disponibles en versiones de 8 y 16 bits, estos microcontroladores están enfocados a aplicaciones de audio digital, productos del hogar y de oficina o para instrumentación en automoción.

- Funcionamiento hasta 64 MHz
- Disponibles en versiones de 144 pines
- Todos los modelos disponen de memoria Flash en el propio MCU
- Orientados al bajo consumo energético

FR: Esta familia está compuesta por microcontroladores tipo RISC de 32 bits con arquitectura propia de Fujitsu y están especialmente orientados a aplicaciones de control. Actualmente existen en desarrollo varios modelos de alta velocidad para atender a aquellas aplicaciones que requieran de una elevada velocidad de procesado.

4.3.2 ALTERA

Altera es una empresa pionera en dispositivos programables lógicos. En este sentido, Altera ofrece una gran variedad de FPGAs, que si bien, no son microcontroladores, pueden ser utilizadas como tal. A grandes rasgos, una FPGA es un dispositivo que puede ser programado después de ser fabricado (del mismo modo que un microcontrolador) y que permite programar cualquier tipo de función o aplicación lógica que deseemos en él. En realidad, se programa el comportamiento del hardware y no un software, de modo que su velocidad de ejecución es mucho más rápida de lo que sería en un microcontrolador, aunque no son tan versátiles. A pesar de estar centrada en la producción de FPGAs, Altera pone en el mercado una serie de procesadores que, como el Nios II, ofrecen un rendimiento de hasta 340 MIPs y unas posibilidades de configuración que no se encuentran en otros dispositivos del mercado.

Nios II: Como ya se ha comentado, el Nios II es un microcontrolador de 32 bits diseñado específicamente para la familia de FPGAs de Altera. Dispone de un pipeline de 6 etápas para conseguir el mayor rendimiento posible (hasta 340 MIPS y 1.18 MIPs por MHz). Es un directo competidor con el microcontrolador MicroBlaze de Xilinx. Entre los modelos de Nios II se pueden encontrar opciones orientadas al rendimiento, a soluciones económicas y a alternativas de bajo consumo energético.

4.3.3 FREESCALE

Freescale es una empresa de reciente creación (2004), que nació de la división de semiconductores de Motorola. Actualmente se encuentra entre las 20 primeras empresas mundiales de semiconductores. Entre sus microcontroladores y del mismo modo que la gran mayoría de compañías, encontramos opciones de 8, 16 y 32 bits. Dispone de una arquitectura propia como es la HCXX (donde XX varía en función del dispositivo) o los ColdFire, y de otros modelos de microcontroladores basados en tecnología de ARM.

8 Bits: Entre los modelos de 8 bits se puede encontrar los modelos RS08, HC05, HCS08 o HC11 entre otros. Este último por ejemplo, es un dispositivo con 1 Kbyte de memoria RAM, 640 bytes de memoria EEPROM y 32 Kbytes de EPROM. Además dispone de canales ADC y PWM, así como modos de operación de bajo consumo energético. Su funcionamiento es de cómo mucho, 4 MHz, por lo que lo sitúan en uno de los microcontroladores con características más limitadas de los que se han visto.

16 Bits: La familia de microcontroladores de 16 bits está formada por los modelos S12 y HC16. Los basados en el modelo S12, tienen frecuencias de operación de hasta 80 MHz, disponen de entre 1 y 64 Kbytes de memoria RAM y entre 32 y 1000 Kbytes de memoria Flash. Estos microcontroladores pueden llegar a tener hasta 152 pines, con lo que las posibilidades aumentan considerablemente respecto sus antecesores, los de 8 bits. Por lo que respecta a los modelos HC16, disponen de unas características más reducidas que los S12 en todos los aspectos.

32 Bits: Por último, dentro de la familia de 32 bits, se encuentran los 68K/ColdFire, que son uno de los procesadores más extendidos en el mercado de microcontroladores, gracias a su bajo coste y su alto número de periféricos integrados. Los modelos correspondientes a los Coldfire V4e, proporcionan un rendimiento de 308 MIPS a una frecuencia de 200 MHz, un alto número de pines (388 como máximo) y opciones de conectividad como pueden ser los puertos PCI, Ethernet, CAN o I2C.

4.3.4 LUMINARY MICRO

Luminary hace uso del núcleo Cortex M3 de ARM para fabricar sus propios microcontroladores. Sus MCU corren a una frecuencia de 80 MHz, disponiendo de memoria Flash y SRAM integrada, una controladora DMA de 32 canales y una gran variedad de periféricos integrados como los puertos Ethernec, CAN, USB, SPI o I2c. Estos microcontroladores están específicamente diseñados para su uso en control de motores industriales

- 32 Bits de rendimiento con un coste de un microcontrolador de 8/16 bits
- Disponen de hasta 32 canales DMA y funcionan a una velocidad máxima de 80 MHz
- Incluyen periféricos integrados como pueden ser los puertos 10/100 Ethernet MAC+PHY, CAN, USB On-The-Go, USB Host/Device, SSI/SPI, UARTs, y I2C

- Requieren de la mitad del espacio de memoria para el código respecto de los núcleos ARM7

4.3.5 MICROCHIP

Los microcontroladores fabricados por Microchip son conocidos mundialmente como PIC. Esta familia de microcontroladores RISC dispone de modelos de 8, 16 y 32 bits, siendo los modelos de 8 bits uno de los más vendidos alrededor del mundo, con más de 400 tipos distintos.

PIC8: La arquitectura PIC está basada en una modificación de la arquitectura RISC de Harvard, que permite mejorar el número de pines originales de 6 a 80 y la memoria programable de 384 bytes a 128 kbytes.

- Compatibles con instrucciones de 12, 14 y 16 bits para mejorar la eficiencia y rendimiento
- Las instrucciones y los datos son transmitidos por buses separados, evitando los cuellos de botella
- Disponen de un pipeline de 2 etapas

PIC16: Dentro de los dispositivos de 16 bits de Microchip se encuentran los modelos PIC24 que ofrecen una gran variedad de periféricos, tamaños de memoria o tipos de encapsulado. Se ofrecen dos versiones, las de alto consumo y las de bajo consumo, donde la potencia varía desde los 16 MIPS hasta los 40 MIPS del modelo PIC24H.

- Disponibles en versiones de hasta 256 kbytes de memoria Flash y 16 kbytes de SRAM
- Encapsulado de hasta 100 pines
- Ejecución de instrucciones y multiplicaciones en un solo ciclo de reloj

PIC32: El modelo de 32 bits (PIC32), ofrece un rendimiento de 1.56 MIPS por MHz, llegando a los 183.762 MIPS a una frecuencia de 80 MHz. Además y del mismo modo que el resto de microcontroladores, ofrecen soporte para una gran variedad de periféricos integrados.

4.3.6 NXP

Esta empresa fabricante de microcontroladores hace uso de núcleos de otras casas. En este caso, NXP hace uso de los procesadores 8051, ARM7, ARM9 y Cortex. Por lo tanto, sus procesadores no presentan ninguna novedad reseñable respecto los explicados anteriormente.

4.3.7 RENESAS

El mercado de Renesas está orientado a los microcontroladores de 16 y 32 bits, aunque deja espacio en su catálogo para un modelo de 8 bits en el que su principal característica es el bajo consumo energético.

SuperH: El microcontrolador tipo RISC SuperH es el buque insignia de la empresa Renesas. Se trata de un microcontrolador orientado a conseguir el máximo rendimiento y miniaturización posible. En este aspecto, su funcionamiento llega a los 600 MHz y dispone de modelos con dos cores e incluso puertos de expansión PCI Express.

4.3.8 TEXAS INSTRUMENTS

Desde los microcontroladores de ultra bajo consumo como los MSP430 hasta los de alto rendimiento como los TMS320C2000™, pasando por los microcontroladores de 32 bits de propósito general basados en procesadores ARM y Cortex, Texas Instrument ofrece una gran oferta de microcontroladores que abarcan todos los mercados.

MSP430: El microcontrolador MSP430 de ultra bajo consumo, es un microcontrolador RISC de 16 bits orientado especialmente a los dispositivos móviles. Su CPU está optimizada para lenguajes de programación C y Ensamblador y ofrece 16 registros de propósito general. Su rendimiento es discreto comparado con la competencia, llegando en el mejor de los casos a 25 MIPS, aunque se ha de tener en cuenta de que se trata de un microcontrolador orientado al bajo consumo. Su precio de salida comienza en los 0.49$ lo que hace que sean una muy buena opción.

TMS320C2000: Los microcontroladores de 32 bits C2000 están diseñados para alcanzar un gran rendimiento en aplicaciones de control en tiempo real. Estos microcontroladores están construidos sobre los existentes F2833x (de alto rendimiento), y ofrecen velocidades de hasta 300 MHz consiguiendo 600 MFLOPS. Además ofrecen 516 Kbytes de memoria RAM y canales PWM con una resolución de 65 picosegundos.

4.4 TABLA COMPARATIVA

La *Tabla 3* muestra de forma paramétrica, cuales son las principales características técnicas de varios microcontroladores de las compañías antes mencionadas. Como se ha podido ver, existen muchas alternativas al microcontrolador AVR UC3 usado en nuestro proyecto, algunas aportan mejoras, otras no, pero si bien es cierto que el mercado de microcontroladores es muy amplio y el hecho de decantarse por una opción o por otra puede ser debida únicamente a algunas pequeñas diferencias que se presentan entre estos.

Freescale o Microchip, son alternativas perfectas para nuestro microcontrolador e incluso en el caso de Freescale, existen modelos que superan con creces las características técnicas de los AVR32, aunque hay que tener en cuenta que un salto tan grande en potencia como es el caso de la familia ColdFire de Freescale, siempre vendrá acompañado de un aumento más que considerable en cuanto a consumo energético y por lo tanto, se puede decir, que están en sectores diferentes.

Familia	Dispositivo	Flash (kBytes)	SRAM (kBytes)	DMA Ch.	I/O	F.Max (MHz)
AVR8	megaAVR	256	16	-	86	20
AVR8	tinyAVR	8	0,5	-	28	20
AVR8	xmegaAVR	384	32	4	78	32
AVR ARM	AT91SAM	2048	256	24	160	400
AVR32	AVR UC3	512	64	15	109	66
AVR32	AVR AP7	-	32	-	160	150
AVR 8051	AVR 8051	128	8	-	44	60
Fujitsu F2MC	16 Bits	832	32	16	144	56
Fujitsu F2MC	8 Bits	60	18	-	100	16
Fujitsu FR	32 Bits	2112	128	8	320	100
Altera Nios II	32 Bits	-	-	-	-	-
Freescale 8	8 Bits	64	4	-	69	25
Freescale 16	16 Bits	128	12	-	91	32
Freescale 32	ColdFire	4000	2576	96	388	1700
Luminary M.	Stellaris	256	96	32	72	80
PIC 8 Bits	PIC18	128	3,96	-	70	64
PIC 16 Bits	PIC24H	256	16	8	85	-
PIC 32 Bits	PIC32	512	128	8	100	80
NXP ARM9	ARM9	-	256	-	160	266
NXP Cortex	Cortex M3	512	64	-	70	100
Renesas	SuperH	2048	1600	-	-	600
TI	MSP430	256	16	3	87	25
TI	C2000	512	516	6	88	300

Familia	VCC (V)	Timers	PWM Ch.	RTC	SPI	TWI	USART	AD/DA Ch.
AVR8	1.8 – 5.5	6	16	Si	2	Si	4	16
AVR8	0.7 – 5.5	2	6	-	Si	Si	1	11
AVR8	1.6 – 3.6	8	24	Si	4	4	8	20
AVR ARM	1.6 – 5.5	9	8 (Control.)	Si	5	2	3	18
AVR32	1.8 – 3.3	3	7	Si	2	1	4	8
AVR32	-	-	-	-	-	-	4	-
AVR 8051	2.4 – 6.0	3	-	-	Si	Si	2	Si
Fujitsu F2MC	3.3 – 5.5	-	-	-	-	-	1	40
Fujitsu F2MC	1.8 – 5.5	-	-	-	-	-	1	12
Fujitsu FR	2.7 – 5.5	-	-	-	-	-	3	40
Altera Nios II	-	-	-	-	-	-	-	-
Freescale 8	1.8 – 5.5	2	-	-	-	-	-	12
Freescale 16	4.5 – 5.5	2	-	-	Si	-	-	8
Freescale 32	1.8 – 5.5	-	-	-	Si	-	-	64
Luminary M.	-	4	8	-	2	-	3	16
PIC 8 Bits	1.8 – 5.5	7	-	-	Si	-	2	28
PIC 16 Bits	3.0 – 3.6	13	8	Si	2	-	2	-
PIC 32 Bits	-	7	5	Si	4	-	6	16
NXP ARM9	1.2 - ¿?	6	11	-	2	-	7	9
NXP Cortex	3.3 - ¿?	4	6	-	1	-	4	9
Renesas	1.2 - ¿?	-	-	Si	Si	-	-	-
TI	-	33	10	-	8	-	4	16
TI	-	24	24	-	4	-	3	16

Tabla 4: Características paramétricas de los diferentes microcontroladores

5. USO DEL AVR32 EN UN ROBOT MÓVIL

Se desea diseñar y construir una plataforma que permita probar la versatilidad de la placa EVK1100 y su microcontrolador AVR32 UC3, en un proyecto de aplicación en tiempo real. Para ello, se ha escogido la construcción de un robot móvil, que dispondrá de funciones similares a los robots que se pueden encontrar a nivel comercial, por lo que servirá para testear las características y opciones más interesantes que el AVR32 UC3 nos ofrezca.

A continuación se presenta el diseño y construcción de esta plataforma móvil con fines académicos y de investigación, pero que servirá para poder testear y validar arquitecturas de control de robots o para poder probar algoritmos de navegación autónoma, con diferentes sensores o dispositivos electrónicos. Para ello, se hará un seguimiento de las características que el AVR32 ofrece a la robótica junto a su placa de evaluación EVK1100, un estudio de las herramientas y procesos necesarios para poder comenzar a trabajar con los microcontroladores AVR32 sin la necesidad de una placa de evaluación y el diseño de un software de control que permita realizar un control manual y autónomo del robot móvil.

5.1 CARACTERÍSTICAS DEL AVR32 PARA LA ROBÓTICA

Sin duda, el microcontrolador AVR32 presenta grandes mejoras respecto al AVR8. Es por ello que a continuación se revisan los puntos del microcontrolador AVR32 más importantes cuando se desea realizar un robot.

Número de pines GPIO: En concreto, el encapsulado utilizado de 144Pines LQFP, proporciona hasta 109 pines de propósito general (GPIO). Este dato es realmente importante para el manejo de robots, ya que a mayor número de pines utilizables a modo general, mayor cantidad de periféricos y funciones podrá realizar este robot.

Velocidad de Procesado: Para la realización de nuestra plataforma móvil, no se requiere de una gran velocidad de cálculo. Así mismo, tampoco es imprescindible que la frecuencia del MCU sea elevada, premiando la duración de la batería sobre la potencia. En este caso, vemos que el microcontrolador es capaz de alcanzar las siguientes velocidades:

- Hasta 91 DMIPS corriendo a 66 MHz
- Hasta 49 DMIPS corriendo a 33 MHz (con acceso a la memoria en el mismo ciclo de reloj)

Todo y que el rendimiento que ofrece el MCU es realmente bueno comparado con otros micros del mismo segmento, la velocidad de funcionamiento se verá reducida hasta los 12 MHz, siendo suficiente para la gestión del robot y de todos los dispositivos que a este se encuentren conectados.

Ejecución de instrucciones en un solo ciclo de reloj: Esta característica puede presentar grandes mejoras a un robot que realice funciones complejas, como pueden ser el tratamiento digital de imágenes. En el caso de dotarlo de una cámara que fuera capaz de captar imágenes, estas podrían ser procesadas más rápido gracias al conjunto de instrucciones DSP que incluyen estos microcontroladores y que son ejecutadas en un solo ciclo de reloj, ganando en velocidad de cálculo.

Memoria interna: La cantidad de código a cargar en el microcontrolador variará mucho en función del programa que se realice. Sin embargo, la elaboración de una plataforma de testeo del microcontrolador y de su placa de evaluación, provoca que el código se haga extenso

debido a la gran inclusión de pruebas y ejemplos de uso de la plataforma móvil. En el caso del microcontrolador en uso, se dispone de:

- 512 KBytes de memoria interna Flash de alta velocidad
- Capaz de realizar accesos a esta en un solo ciclo de reloj

Para programas como el utilizado y otros mucho más complejos, a mayor cantidad de memoria, mejor. Sin embargo, esto puede provocar una des-optimización del código empleado, ya que el hecho de no preocuparse por el espacio consumido, puede hacer que la programación se haga de una forma menos óptima.

Memoria RAM: En el caso de necesitar mover una gran cantidad de variables y datos, el microcontrolador nos ofrece 64 KBytes de memoria RAM. Tal cantidad de memoria no es necesaria para la realización de un proyecto de robot móvil como el que aquí se plantea, ya que únicamente se moverán datos muy básicos sobre distancias y posicionamiento de modo que sólo se hará uso de una pequeña cantidad de toda esta memoria disponible. De todos modos, parte del programa es cargado en memoria una vez iniciamos el micro, así que, a mayor cantidad de memoria, mayor fragmento de programa se podrá cargar en la memoria RAM.

Además, en el caso de necesitar hacer uso de más cantidad de memoria RAM, la arquitectura AVR32 permite ampliar esta cifra conectando memoria externa a través de su interfaz de memoria externa. De este modo, las posibilidades crecen enormemente, ya que es capaz de hacer uso de buses de 24 bits, con lo que se puede direccionar 2^{24} bits de memoria.

Control de Interrupciones: El control de interrupciones es parte fundamental en el momento de programar robots y cualquier otro dispositivo. Permitirán aceptar peticiones internas, externas de otros periféricos o controlar sensores sin la necesidad de estar pendientes en todo momento de estos dispositivos. El AVR32 permite hacer uso de hasta 2048 interrupciones diferentes, con 4 niveles de prioridad, un número suficiente para cumplir con las expectativas de cualquier entorno robótico.

En el caso de nuestro proyecto, únicamente se hará uso de interrupciones para la comunicación serie a través del USART del microcontrolador. Para la programación de los sensores, no se hará uso, ya que interesará activar los sensores en momentos específicos a modo de ahorrar batería.

Power Manager y Watchdog Timer: Este microcontrolador dispone de dos funciones muy interesantes como son el Power Manager y el Watchdog Timer. La primera de ellas, permitirá entrar en modo de suspensión del MCU, de modo que el consumo energético se reduce casi a 0 y además, permite hacer uso de diferentes frecuencias de funcionamiento y que estas puedan ser modificadas y cambiadas en “caliente”. Mientras que el segundo, el Watchdog Timer, ofrecerá la posibilidad de realizar un control del robot de modo que este no se quede “colgado” y en caso de que esto ocurra, poder realizar una recuperación o reset del sistema sin la necesidad de intervenir.

Ambos elementos son importantes ya que permitirán ahorrar energía y dotar de mayor autonomía al microcontrolador (en el caso del Watchdog Timer, lo que permitirá es no tener que manipular la placa en caso de cuelgue), de modo que si se busca premiar la duración de batería de un robot así como poder variar el rendimiento de este en función del estado en que se encuentre, se deberán implementar ambas funciones.

Canales PWM y ADC: Gracias a los 7 canales PWM de 16 bits que incluye el microcontrolador AVR32 UC3, se podrá hacer uso de hasta 7 dispositivos que necesiten señales cuadradas para su funcionamiento. Este es el caso de los motores o servomotores, por ejemplo, los cuales

variarán su posición en función del periodo que tenga la señal PWM. También pueden usarse para modificar la velocidad a la que un motor DC se desplaza, tal y como se ha realizado en este proyecto, además de controlar el servomotor que dirige al sensor IR.

Por otro lado, se dispone de 8 canales ADC que permitirán conectar hasta 8 dispositivos que proporcionen una salida de tensión analógica al microcontrolador. Este es el caso de la mayoría de sensores infrarrojos que se encuentran en el mercado o potenciómetros.

3 Timer/Counters: En este aspecto, los Timers disponibles en el microcontrolador pueden llegar a ser insuficientes para la elaboración de proyectos complejos. Únicamente con 3 posibles canales que han de alternarse las funciones de Timer o Contador, serán suficientes para la realización de una plataforma móvil simple, pero posiblemente los usuarios más avanzados exijan mayor cantidad de contadores.

Usart, SPI, TWI, USB y Ethernet: En el aspecto de comunicación del microcontrolador, se puede decir que sus características son idóneas para realizar proyectos de robótica. Con 4 canales USART que pueden funcionar a modo de Modem, 2 SPI, 1 TWI compatible con el estándar I2C y posibilidad de hacer uso de USB y Ethernet, se cumple con todos los requisitos del robot móvil que se está realizando, y de bien seguro que se cumplirán el del resto de usuarios.

5.2 LA PLACA EVK1100 EN UN ROBOT AUTÓNOMO

Para la realización del robot móvil, se dispone de la placa de evaluación EVK1100 (*Figura 15*). Esta placa de evaluación, ofrece un entorno de desarrollo para el microcontrolador AVR32 AT32UC3A0512 y está equipada con una serie de memorias y periféricos, que permitirán experimentar todo el potencial y características de estos microcontroladores.

La ventaja de trabajar con una placa de evaluación como la EVK1100 es que proporciona una serie de elementos ya pre-instalados y configurados para su utilización. Todos los periféricos y opciones de los que dispone, pueden ser analizados y estudiados a través de los múltiples ejemplos que se encuentran en el entorno de programación. A continuación se listan las características técnicas más importantes de la placa de evaluación, aunque todas ellas serán analizadas en los siguientes puntos:

- Microcontrolador de 32 Bits AT32UC3A0512 con encapsulado QFP144.
    - 512 kBytes de memoria Flash y 64 kBytes de memoria RAM.
- LCD color azul de 4 líneas por 20 caracteres de ancho, con luz de fondo ajustable.
- Conector USB (2.0) y conexión Ethernet RJ45.
- Listo para usar sensores de: Luz, Temperatura y Potenciómetro.
- 3 pulsadores y 1 Joystick.
- 6 LEDs (4 mono color verde y 2 bicolor verde/rojo).
- 8 Mbytes de Atmel DataFlash y 32 Mbytes de SDRAM.
- 2 USARTs y 1 conector SPI.
- 1 conector JTAG y 1 conector Nexus.
- 1 slot SD/MMC.
- 1 conector TWI compatible con I2C.
- Área de conexionado.
- Alimentación a través de USB o de conector externo 8-20V DC.

Todos estos elementos se interconectan con el microcontrolador y disponen de pines visibles en la placa para su fácil utilización. A continuación se describen algunos de los elementos más importantes que componen la placa EVK1100, así como su uso dentro del contexto del robot móvil que se está realizando.

Figura 15: Visión general de la placa de evaluación EVK1100

5.2.1 ALIMENTATION

La placa de evaluación EVK1100 permite alimentarla a través del puerto USB o de un conector Jack externo de 2.1mm, siendo los niveles de tensión utilizados en este último caso, los comprendidos entre los 8 y los 20V DC. Tal y como muestra la **¡Error! No se encuentra el rigen de la referencia.**, para poder seleccionar entre un tipo de alimentación u otra, se dispone de un interruptor externo de dos posiciones (una para la alimentación a través de USB y otra para la alimentación a través del Jack).

Este sistema de alimentación está formado por tres reguladores de tensión que proporcionan niveles de tensión de 5, 3.3 y 1.8V, acompañados de un diodo de corriente encargado de que la polarización de la corriente sea la correcta.

El sistema de reguladores está formado de modo que el regulador de 5V es el encargado de alimentar el regulador de 3V y este a su vez, el de 1.8V. Las funciones de los reguladores son:

- El regulador de 5V es el regulador de tensión maestro y como tal, el encargado de proporcionar la potencia necesaria al resto de reguladores.
- El de 3.3V proporciona niveles de tensión aptos a todos los periféricos de la placa. Es por ello que, todos los dispositivos que conectemos a esta, tienen que ser compatibles con niveles de tensión mínimos de 3.3V.
- Y por último, el regulador de 1.8V es el encargado de proporcionar la tensión necesaria al núcleo del microcontrolador.

Otro aspecto a tener en cuenta, es el nivel de corriente que pueden soportar estos reguladores, de modo que, el máximo nivel de corriente recomendado será de 1A para el regulador de 5V. Si se sobrepasan estos niveles, se corre el riesgo de quemarlos, por lo que la placa se volvería inservible ya que no pueden ser sustituidos fácilmente.

En este aspecto, el uso de una placa como es la EVK1100, puede suponer un lastre a nuestro robot debido a sus grandes dimensiones y la necesidad de alimentarlo a más de 8V. Es de agradecer, que esta incluya un puerto USB a través de la cual también puede ser alimentada de modo que durante todo el desarrollo y testeo del robot, no será necesario el uso de baterías para alimentarla.

5.2.2 MEMORIA EXTERNA

A pesar de que la placa proporciona una cantidad de memoria externa considerable (8 Mbytes de Atmel DataFlash y 32 Mbytes de SDRAM), no será necesaria su utilización para la realización del proyecto, ya que la cantidad de datos que se manejarán no superarán en ningún momento los 512 KBytes de memoria Flash que se incluyen en el encapsulado del microcontrolador. Si fuera necesario mover un gran volumen de datos, sería imprescindible su uso. Además, el puerto de expansión para tarjetas SD aumenta las posibilidades de almacenamiento, pudiendo utilizar la placa a modo de memoria USB, por ejemplo.

5.2.3 OSCILADORES

La placa de evaluación EVK1100 permite el funcionamiento del microcontrolador a una frecuencia de 66 MHz, aunque para la realización del proyecto, se configurará a 12 MHz. En las pruebas realizadas, se ha observado como esta velocidad es suficiente para que el sistema garantice un correcto funcionamiento, con el ahorro energético que implica trabajar a menor frecuencia. Los 12 MHz son obtenidos de un oscilador externo situado sobre la placa de evaluación, pero aparte de este, existen otros:

- Un oscilador (XC1) principal conectado a la entrada OSC0 del microcontrolador, y que funciona a 12 MHz.
- Otro oscilador externo (XC2) conectado al OSC1 del microcontrolador, y que funciona también a 12 MHz.
- Un oscilador RTC (Real Time Counter) (XC5) que funciona a 32,768 KHz.

5.2.4 USARTS

Tal y como se indicó en las características técnicas antes listadas, la placa de evaluación EVK1100 dispone de dos conectores USART: USART 0 y USART 1 (*Figura 16*). El primero de ellos es un conector RS232 estándar, mientras que para el segundo, se añade la capacidad de poder funcionar a modo de modem. Ambos conectores hacen uso de un puerto DB9.

Gracias a estos conectores, se puede establecer una comunicación con el microcontrolador a través del puerto RS232 de un ordenador cualquiera. Abriendo un “Terminal” o ejecutando un programa dedicado, podremos enviar y recibir órdenes de arranque o parada a nuestro robot.

Es importante tener en cuenta que la placa de evaluación permite seleccionar valores de tensión TTL o RS232, ya que si por error se conecta un dispositivo que funcione con valores TTL a través del puerto RS232, este se volvería inservible.

5.2.5 SPI

El bus SPI (Serial Peripheral Interface) es usado para la transferencia de información en modo serie entre dispositivos compatibles con este protocolo (por ejemplo: conectar dos microcontroladores para que se comuniquen entre ellos, o conectar periféricos que tengan soporte para bus SPI).

La placa de evaluación proporciona un área de 6 pines (*Figura 16*), que permitirán conectar directamente sobre esta un dispositivo SPI. Sin embargo, no se hará uso de estos pines, ya que internamente la placa conecta el interfaz SPI al LCD, de modo que se podrá hacer uso de este dispositivo sin necesidad de realizar ningún tipo de conexión.

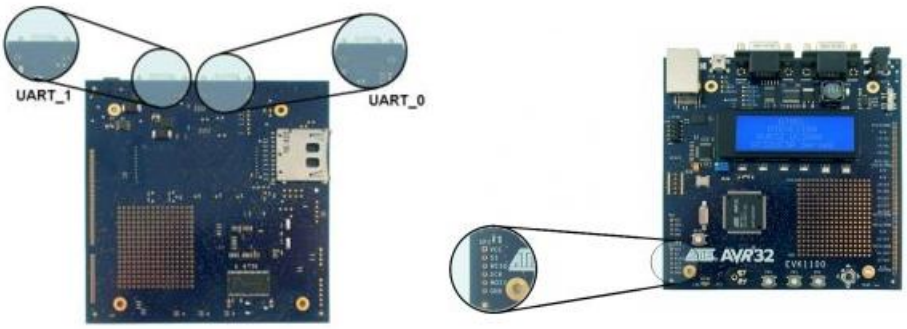

Figura 16: Conectores USART (izquierda) y interfaz SPI (derecha)

5.2.6 TWI

Del mismo modo que ya ocurría para la interfaz SPI, la placa de evaluación proporciona un área de conexionado directo para el uso del protocolo TWI (hay que recordar que es compatible con el modo I2C). Gracias a este interfaz de conexionado directo, se podrá hacer uso de memorias externas EEPROM o de sensores que sean capaces de funcionar mediante el protocolo I2C (en el mercado existen sensores infrarrojos o por ultrasonidos que permiten ser conectados a este tipo de puertos directamente).

Únicamente dispone de 4 pines, de los cuales dos se corresponden a las líneas de alimentación para los dispositivos que conectemos, y los otros dos para las líneas SDA y SCL (siempre recordando que la alimentación es de 3.3V debido a los reguladores internos de la placa).

Para la realización de nuestro proyecto no es necesaria de su utilización, ya que los sensores utilizados serán conectados todos directamente a los puertos ADC.

5.2.7 ETHERNET Y USB

A través del puerto Ethernet que incluye el kit de evaluación, se pueden testear las capacidades que este protocolo nos ofrece sobre el microcontrolador, asignando una IP a la placa para realizar transferencia de información o incluso, realizar un servidor web, como el que viene instalado por defecto en la propia placa. Únicamente es necesario conectar la placa a un ordenador y configurar la conexión de red, de modo que se podrá acceder a todas las funcionalidades que a modo de ejemplo trae por defecto.

El robot móvil no hace uso de estos puertos, ya que toda la comunicación será realizada a través del puerto serie. A pesar de esto, si fuera necesario transmitir información a grandes distancias y alta velocidad, esta sería una de las mejores formas para hacerlo.

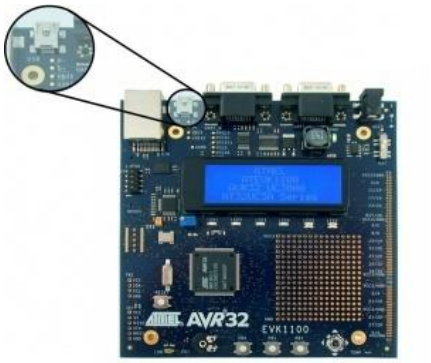

Figura 17: Visión del conector USB

Por otro lado, el USB integrado en la placa EVK1100, proporciona una interfaz que habilita a la placa para actuar como un host USB o como un dispositivo USB (a modo de memoria Flash). A pesar de esto, para la realización del proyecto tampoco será necesario su uso, ya que únicamente será utilizado para alimentar el microcontrolador mientras se encuentra conectado al PC y es programado a través del puerto JTAG.

5.2.8 JTAG

El interfaz JTAG permitirá conectar la placa de evaluación al programador JTAGICE MKII de Atmel, de modo que toda la programación del dispositivo se llevará a cabo a través de este interfaz serie. Además, el JTAG junto al sistema OCD (incluido en el propio microcontrolador), proporciona una excelente herramienta de debug y testeo sobre el propio chip.

5.2.9 LCD

El LCD instalado proporciona una pantalla de 4x20 líneas para mostrar los mensajes que creamos convenientes a través de ella. Puede ser usada a modo de consola de debug si se cree conveniente, o simplemente para mostrar datos del robot en todo momento. En concreto este LCD es un EA-DIP204B-4NLW, con una pantalla de color Blanca/Azul, que incluye un potenciómetro ADJ2 que permite cambiar la intensidad de la luz de fondo.

Figura 18: Visión de la pantalla LCD

A pesar de que el microcontrolador utilizado es de bajo consumo, el consumo de potencia por parte de todo el conjunto de la placa sí que es elevado y sumando el de la pantalla aun lo es más. Teniendo en cuenta este factor, se hará uso de él, aun sabiendo que perjudicará en la duración de la batería.

Internamente se encuentra conectado al puerto SPI del microcontrolador, siendo este el encargado de gestionar toda la información que fluye hacia la pantalla.

5.2.10 LEDs

En la placa se pueden encontrar 6 LED's, que permitirán interactuar con esta y servirán para mostrar códigos de errores visuales o lo que se desee. De los 6 LED's proporcionados, 4 LED's son monocromos (de color verde) y 2 bicolor (rojo y verde). Estos LED's serán de gran utilidad para poder ver cómo evoluciona el código programado en la placa así mismo como la activación de algunos canales PWM y de otros GPIO.

5.2.11 Push Buttons y Joystick

Del mismo modo que con los LED's, gracias a los pulsadores y al joystick incluidos en la placa, se podrá interactuar con el microcontrolador, y así poder probar funciones o controlar en un futuro nuestro robot.

En concreto, la EVK1100 proporciona 3 pulsadores y 1 joystick de 4 direcciones, tal y como se observa en la *Figura 19*. Hacen uso de lógica inversa, es decir, mientras están pulsados, su valor lógico es 0, al contrario de lo que se podría pensar en un comienzo.

Su programación es sencilla, siendo únicamente necesario recoger en una variable el valor de tensión del puerto al que se encuentra asociado. De este modo, se podrán utilizar como se crea conveniente.

Figura 19: Pulsadores y Joystick de la EVK1100

5.2.12 AREA DE CONEXIONADO

Un aspecto importante a tener en cuenta en el momento de trabajar con una placa de evaluación como lo es la EVK1100, son los puertos de expansión que esta tenga. En este caso, Atmel ha dotado a su placa de toda una serie de pines preparados para conectar lo que sea necesario. Estos pines serán de gran utilidad para poder hacer un uso directo a través de ellos de los múltiples pines necesarios del microcontrolador (por ejemplo, conectar directamente los servomotores a los canales PWM).

A través de estos puertos de expansión también se puede acceder a funciones tales como la interfaz SSC, los canales PWM, canales ADC, Timers, USART, TWI o todos los puertos GPIO entre otros (todos con nivel de tensión de 3.3V).

Además, dispone de un área de prototipado que permitirá incluir otros elementos sobre la propia placa, como si de una placa perforada se tratara. De este modo, el conexionado se optimiza, ya que la dimensión de las conexiones se reducen al estar sobre la propia placa.

5.2.13 POTENCIÓMETRO Y SENSOR DE TEMPERATURA Y LUZ

Además de todos los elementos ya comentados, la placa de evaluación proporciona otros tres dispositivos que pueden ser utilizados según se crea conveniente (aunque no serán de ninguna utilidad para la construcción del robot móvil). En la parte inferior de la placa, se encuentra un potenciómetro junto con un sensor de luz y temperatura. Estos tres elementos se encuentran conectados internamente a los canales ADC, de modo que únicamente se debe realizar la programación que corresponda para activar dichos canales ADC.

5.3 PERSONALIZANDO LA PLACA

A parte del uso de la placa de evaluación EVKK1100, para la elaboración de un robot será necesario personalizar la placa. Para ello, dispone de puertos de expansión donde se puede trabajar y soldar directamente los componentes o dispositivos que se crean necesarios.

En concreto, los elementos que deben ser conectados en la placa para poder realizar nuestro robot móvil son los que detallados en la *Figura 20*. A continuación, se detalla con más profundidad cada una de estas modificaciones o adiciones que se han de llevar a cabo sobre la placa, así como la relación que estos tendrán con cada uno de los canales que se utilicen en la placa de evaluación EVK1100. Si lo que se desea es obtener más detalle acerca de las

conexiones reales que se han de realizar en la placa de evaluación y el microcontrolador AVR32, debe dirigirse a la *Tabla 5*, donde se encuentran todos especificados.

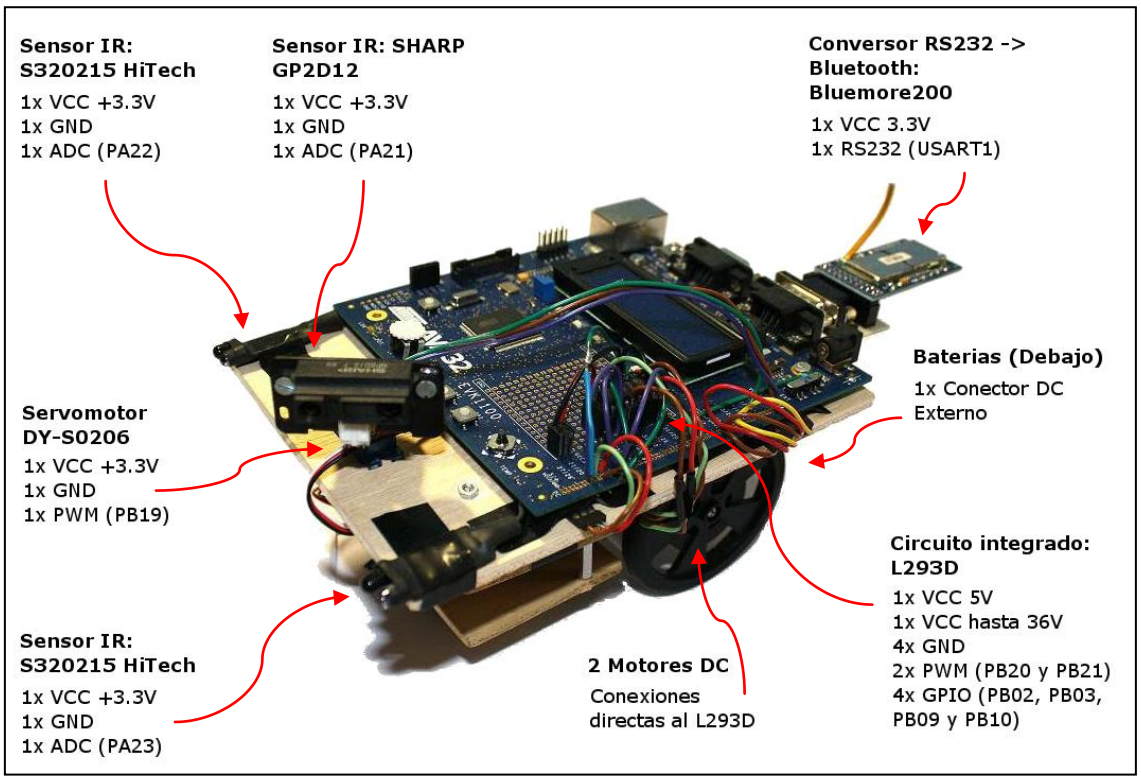

Figura 20: Elementos necesarios para la construcción del Robot

5.3.1 MOTORES DC

Para la construcción del robot móvil, se hace uso de 2 motores DC que proporcionarán el movimiento y control del robot. Estos motores no están conectados directamente a la placa de evaluación, si no que disponen de un circuito previo formado por un controlador de motor por PWM. Este circuito permite controlar tanto la dirección como la velocidad del motor DC y para su funcionamiento será necesario realizar las conexiones mostradas en la *Tabla 5*.

5.3.2 SERVOMOTOR

El servomotor permitirá realizar un barrido a modo de radar/sonda para poder detectar los obstáculos que se encuentren en la trayectoria del robot. Estos dispositivos son capaces de moverse en función de la amplitud de la señal cuadrada (*Figura 21*) que se le inyecte. Por ejemplo, para señales con una amplitud de 1.25 ms, el servomotor se situará en una posición. Si cambiamos a 2 ms, se centrará en otra, de modo que variando este parámetro, se podrá hacer girar el servomotor.

Este dispositivo dispone de 3 entradas que deberán ser conectadas directamente sobre la placa de evaluación: VCC, GND y la entrada para la señal PWM. El funcionamiento se hará a una tensión de 3.3V por lo que, cualquier toma de VCC sobre la placa será apta para alimentar el dispositivo.

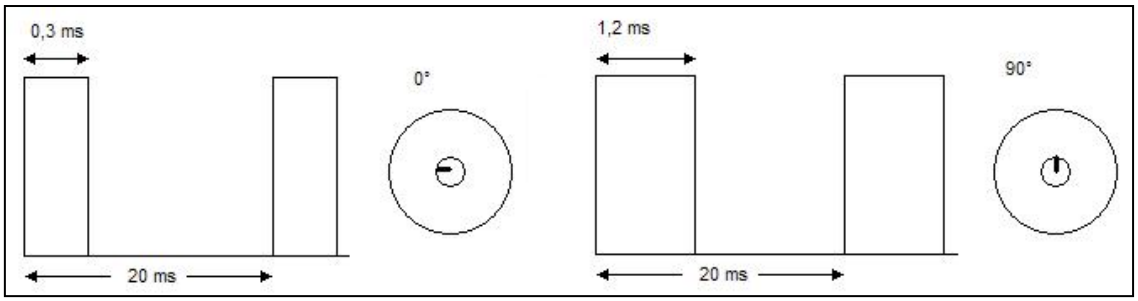

Figura 21: Funcionamiento de un servomotor

5.3.3 SENSORES IR

Los sensores infrarrojos se responsabilizarán de medir la distancia a la que se encuentre el robot de un objeto que pueda obstaculizar su trayectoria. Mediante estas medidas, se variará la dirección y movimiento del robot de modo que en ningún momento este colisione.

Se hace uso de 3 sensores infrarrojos, uno de los cuales está montado sobre el servomotor y será el que tome las medidas a modo de radar o sonda. Los otros dos, situados en los laterales, permitirán detectar las colisiones cuando la distancia del objeto sea demasiado próxima, para que el radar pueda detectarla.

A pesar de tratarse de dispositivos físicamente diferentes, el funcionamiento es exactamente el mismo, disponiendo de 1 entrada VCC, 1 GND y otra de Output, que dará un valor de tensión proporcional a la distancia a la que se encuentre el objeto a evitar. Es por eso, que estos elementos hacen uso de los canales ADC, para transformar ese voltaje de la salida, en un valor binario que pueda ser tratado por el microcontrolador.

5.3.4 BLUEMORE200

Por último, disponemos de un dispositivo que permitirá la comunicación vía Bluetooth con un PC que actuará como controlador del robot. Se trata de un elemento que es capaz de transformar una señal serie que obtiene a través de un puerto RS232, en una señal apta para ser transmitida vía Bluetooth. Este proceso es totalmente transparente para el usuario y no requiere de ningún tipo de programación especial, únicamente conectarlo a través del puerto COM de la placa de evaluación y realizar toda la comunicación a través del USART del microcontrolador.

5.3.5 TABLA DE CONEXIONADO

La *Tabla 5* muestra cuales han de ser todas las conexiones necesarias que se han de realizar sobre la placa de evaluación EVK1100. En ella, no se especifican las conexiones a VCC o GND ya que por lo general todos los elementos las necesitan y la placa dispone de muchos puertos de expansión que pueden ser utilizados con este fin.

Función	Puerto EVK1100	PIN AVR32
1,2 Enable Motor	PB20 – 1x PWM	3
1A Motor	PB02 – 1x GPIO	96
1Y Motor	-	-
2Y Motor	-	-
2A Motor	PB03 – 1x GPIO	98
4A Motor	PB10 – 1x GPIO	115
4Y Motor	-	-

3Y Motor	-	-
3A Motor	PB09 – 1x GPIO	113
3,4 Enable Motor	PB21 – 1x PWM	5
Bluemore200	1x RS232	Ver Datasheet
IR GP2D12	PA21 – 1x ADC	73
IR S320215	PA22 – 1x ADC	74
IR S320215	PA23 – 1x ADC	75
Servo DY-S0206	PB19 – 1x PWM	143

Tabla 5: Conexiones necesarias hacia la placa de evaluación EVK1100

5.4 PROGRAMACIÓN DEL AVR32

Como ya se ha visto, para la realización del robot móvil se hará uso de la placa de evaluación EVK1100. Pero, ¿qué pasaría si se quisiera programar microcontroladores AVR32 sin disponer de una de estas placas de evaluación?

Siguiendo con la temática del presente capítulo, a continuación se realiza un estudio de los pasos y las herramientas que se han de seguir para poder realizar la programación de un microcontrolador en el caso de no disponer de una placa de evaluación.

La *Figura 22* enumera todas las interfaces disponibles, aunque el uso de cada una dependerá del programador del que se disponga. Esta es la primera cuestión que el desarrollador debe plantear cuando se programan microcontroladores, ya que en función de la interfaz por la que se opte, cambiará la herramienta a utilizar.

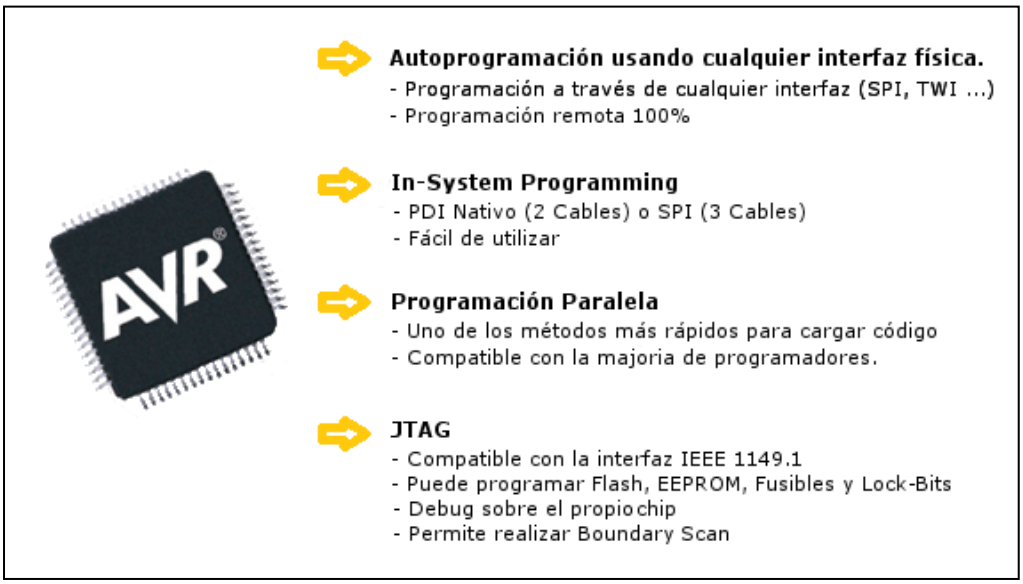

Figura 22: Diferentes interfaces de programación del AVR32

En este caso, se hace uso del puerto JTAG ya que la herramienta disponible es el JTAG MKII ICE, el cual tal y como se verá a continuación (*Tabla 6*), dispone de potentes herramientas para la programación y el debug sobre el propio microcontrolador. Pero además de este programador, existen otros diferentes en el mercado [9], todos ellos distribuidos por Atmel y que se describen a continuación:

	**AVR Dragon:** Por menos de 50$ Atmel pone a nuestra disposición un programador que permite todos los modos de programación disponibles para la familia AVR: • In-System Programming (ISP) • High Voltage Serial Programming (HVSP) • Programación Paralela • Programación JTAG Además, permite realizar debug del microcontrolador.
	**AVRISP MKII:** Esta herramienta de programación tiene soporte para todos los chips de 8 Bits de AVR, por lo tanto, no nos será de utilidad para la programación de los AVR32. A pesar de ello, a continuación se listan algunas de sus principales características: • In-System Programming en todos los AVR8 • Totalmente compatible con AVR Studio • Soporta todos los niveles de tensión de AVR • Velocidad rápida de programación
	**JTAGICE MKII:** Esta herramienta será la utilizada para la realización del presente proyecto. Tal y como se está viendo, no es la única existente en el mercado, pero sí que es una de las mejores opciones: • Permite la programación a través de puerto JTAG y ISP • Dispone de capacidades para realizar debug sobre el microcontrolador • Posibilidad de hacer uso de la interfaz “debugWire”. • Soporte para “Program-Breakpoints, Data-Breakpoints” y para control total de la ejecución del programa.
	**AVR ONE:** El AVR ONE es una poderosa herramienta de desarrollo y debug sobre el propio chip para cualquier dispositivo AVR. • Soporte para realizar debug a través de las interfaces JTAG, debugWire, PDI y Nexus. • Soporte para programar a través de la interfaz ISP, JTG y PDI.

Tabla 6: Programadores disponibles en el mercado

Todo y que existen varias opciones para programar los microcontroladores AVR32, vamos a centrarnos en la interfaz JTAG, ya que es la que se utilizará para la programación tal y como se ha dicho anteriormente.

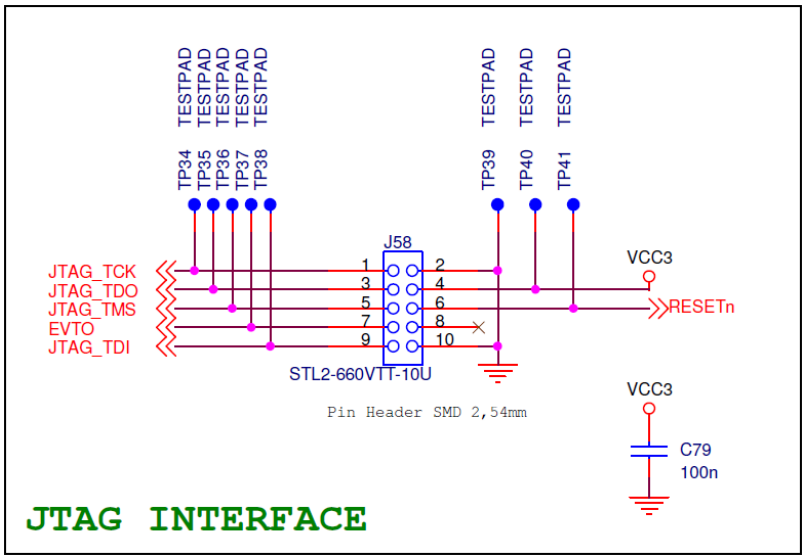

Figura 23: Conexiones necesarias para la programación a través de JTAG

Esta interfaz dispone de 10 pines, tal y como se puede ver en la *Figura 23*, que irán conectados al microcontrolador. Todas las conexiones necesarias se encuentran en la documentación [4] disponible en la propia web de Atmel.

Además de las conexiones del puerto JTAG, también se deberán realizar las conexiones necesarias sobre el microcontrolador. La *Figura 24* muestra el esquemático con las conexiones que serán necesarias para el funcionamiento del circuito.

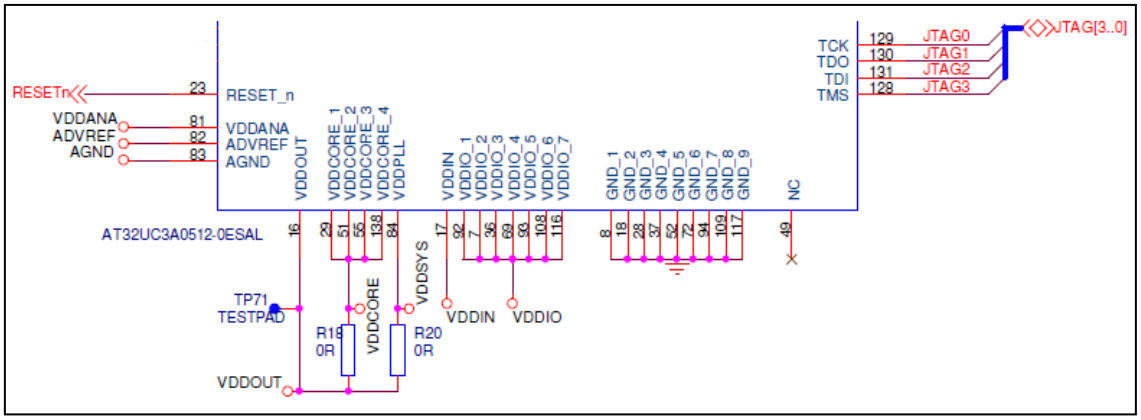

Figura 24: Conexiones necesarias para la programación JTAG

En la parte derecha, se pueden ver los 4 pines que irán conectados al puerto JTAG, mientras que en la parte inferior, se ven todas las conexiones de alimentación que serán necesarias. Sin embargo, en esta figura no se especifica cómo deben realizarse estas conexiones, por lo que nuevamente tendremos que dirigirnos a la documentación existente en la web oficial [5].

Por último y una vez que se tengan claras las herramientas y interfaces que van a ser utilizadas en la programación del robot, así como las conexiones que serán necesarias, se procederá a abrir la suite AVR32 Studio, que permitirá hacer uso de todas las nombradas y por supuesto, enviar el código de programa a la memoria interna del microcontrolador.

5.5 SOFTWARE DE CONTROL

El software del robot está dividido en dos partes: el software propio del microcontrolador y una aplicación de control que servirá para poder enviar datos al robot desde un PC con conexión Bluetooth. A continuación se ofrecen más detalles de ambos programas así como pequeñas indicaciones sobre su programación.

5.5.1 APLICACIÓN DE CONTROL

Mediante este programa realizado con Visual Basic 6.0, se pretende crear una pequeña aplicación que permita gobernar el robot desde un PC cualquiera que disponga de conexión Bluetooth. Su diseño es sencillo, pues no es el objetivo del proyecto, pero como se puede ver en la *Figura 25* dispone de funciones suficientes para testear y gobernar el robot, permitiendo por ejemplo, activar el modo autónomo o el modo de control manual.

Toda la programación de la comunicación se realiza a través del puerto serie del PC, de modo que únicamente se debe crear un pequeño interfaz que permita seleccionar el puerto COM a través del que se establecerá la conexión y pulsar sobre "Conectar". A partir de este momento, si la conexión ha sido satisfactoria, se habilitan los botones de control del robot.

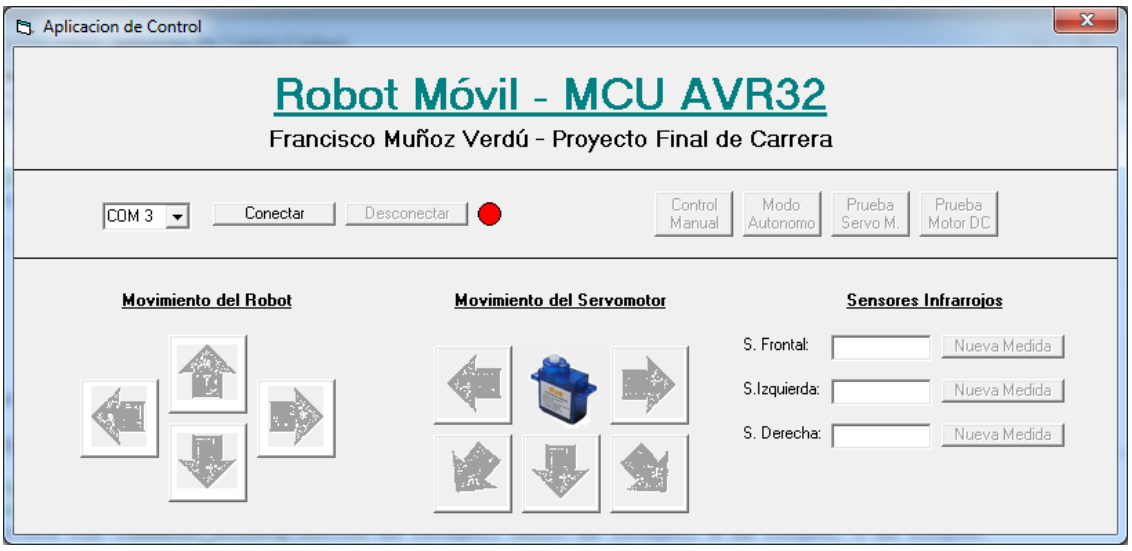

Figura 25: Aplicación de Control

Para la comunicación a través del puerto serie se hace uso del componente "MSComm" que se encuentra disponible en las librerías de Visual Basic 6.0, por lo que en caso de hacer uso de esta aplicación, primero se deberá comprobar que se dispone de dicho componente instalado en el sistema. Todo el código utilizado para la generación de este programa se podrá encontrar en el capítulo [**¡Error! No se encuentra el origen de la referencia.**] de este documento.

5.5.2 SOFTWARE DEL AVR32

Tal y como se ha comentado en el punto anterior, la aplicación de control permitirá comunicarse con el robot. Esta comunicación será gestionada por el microcontrolador, en el extremo del robot móvil, por lo que este deberá disponer de un programa que dote de autonomía e inteligencia al robot.

Toda la programación del microcontrolador se lleva cabo en lenguaje C++. Así están escritos todos los ejemplos que se encuentran en el entorno de programación, además de ser un

lenguaje ya conocido y por lo tanto el aprendizaje de este se limitará al uso de las funciones o variables específicas de estos microcontroladores. Dicho software, permitirá la interacción con la Aplicación de Control antes detallada y además se responsabilizará de dotar al robot de la inteligencia necesaria para que este no colisione con ningún objeto.

En cualquier caso y si lo que se desea es obtener más información acerca del código utilizado en este, deberá dirigirse al capítulo [C] del Anexo.

5.6 Pruebas y Resultados

Una vez se dispone de todo el conjunto correctamente montado, conectado y programado, es necesario realizar una serie de pruebas para comprobar el buen comportamiento de este. Estas pruebas consisten en la comprobación de que toda la comunicación se establece de forma correcta y ambas partes (microcontrolador y PC) se entienden e interactúan sin problemas. Es por ello que a continuación se muestra cual es el funcionamiento de todo el sistema y cuál debería ser su comportamiento.

5.6.1 Pruebas

Las pruebas realizadas se pueden separar en dos partes: la comprobación de la comunicación e interacción entre el microcontrolador AVR32 y el PC, y el correcto funcionamiento del robot en modo autónomo.

La *Figura 26*, muestra el diagrama de bloques de la comunicación entre el AVR32 y la Aplicación de Control. Tal y como se observa, la Aplicación de Control actúa como maestro de la comunicación, siendo la encargada de gestionar la transmisión y de dictar cuál es la siguiente función a realizar, según lo indicado por el usuario.

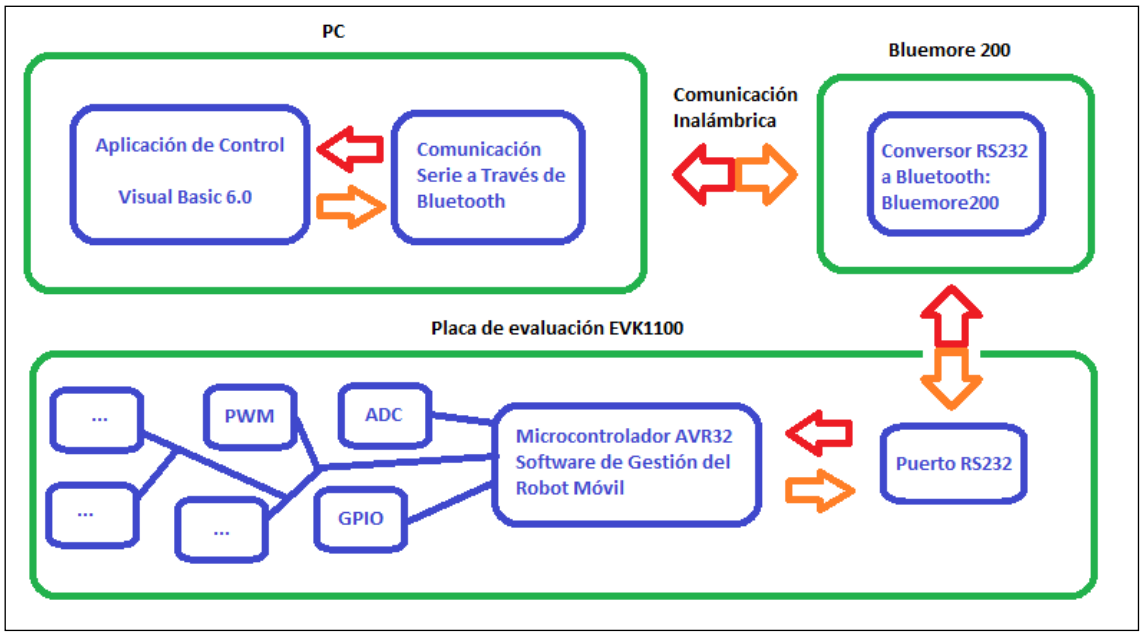

Figura 26: Diagrama de bloques del conexionado del sistema

El funcionamiento es sencillo, basándose únicamente en el intercambio de caracteres ASCII entre el PC y el microcontrolador. A continuación se detalla a modo de ejemplo y para entender su funcionamiento, un posible caso de comunicación:

- A través de la Aplicación de Control, se debe seleccionar el puerto COM asociado al Bluetooth del PC. Una vez elegido, se procederá a realizar la conexión.
- Si la comunicación se produce, se enviará un carácter ASCII a través del puerto serie. Para hacerlo, bastará con pulsar sobre una de las opciones disponibles en el programa. Por ejemplo, para habilitar el control manual, deberá pulsarse el botón correspondiente y este enviará una "f".
- Este carácter, será enviado de forma inalámbrica desde el PC hasta el microcontrolador, gracias al conversor RS232 – Bluetooth disponible.
- Si los datos llegan correctamente, el microcontrolador entrará en modo manual y a su vez, devolverá otro carácter ASCII al PC, de modo que la Aplicación de Control podrá entender que la comunicación ha sido satisfactoria.
- En caso de querer detener la ejecución del programa en curso, se pulsará sobre el botón correspondiente y este, enviará una "q", la cual volverá a ser entendida por el microcontrolador.
- Nuevamente, si la comunicación se produce de forma satisfactoria, el microcontrolador devolverá el control de la ejecución a la Aplicación de Control y se podrá volver a lanzar otra función.

La segunda prueba, consiste en comprobar que el robot es capaz de comportarse de forma autónoma. Para ello y tal y como ya se detalló en capítulos anteriores, este está dotado de una serie de sensores infrarrojos que deberán evitar cualquier tipo de colisión.

El algoritmo utilizado finalmente, es relativamente sencillo, habiendo descartado otros más complejos y con más comprobaciones, por ser toscos y contener demasiadas instrucciones repetitivas, que finalmente no aportaban nada al comportamiento global. A continuación, se muestra cual ha sido el algoritmo utilizado:

- El primer paso a realizar una vez se entra en modo autónomo, es realizar una comprobación íntegra de los tres sensores infrarrojos instalados.
- En caso de que ninguno de los sensores detecte una colisión, el robot por defecto iniciará su movimiento hacia delante durante 1 segundo.
- En caso contrario, si se detecta un obstáculo, el robot detendrá su marcha e iniciará un movimiento que permita esquivarlo (retrocederá y girará hacia un costado).
- Si se detecta más de un obstáculo al mismo tiempo, se ha definido una serie de niveles, que dan prioridad a los sensores laterales, ya que estos detectarán cuando un objeto se encuentra más cerca.

La velocidad de movimiento del robot es fija y para cambiarla deberá reprogramarse este. Es por ello, que se ha optado por escoger una velocidad no demasiado rápida y unos tiempos máximos de ejecución de 1 segundo. De este modo, cada segundo se realizan comprobaciones del entorno y se pueden detectar objetos de forma satisfactoria.

5.6.2 RESULTADOS

La ejecución de las pruebas ha sido correcta y el funcionamiento final es el esperado. A continuación se muestran los resultados desglosados en dos partes: los resultados obtenidos respecto a la arquitectura AVR32 y los obtenidos respecto a la placa de evalución EVK1100.

Arquitectura AVR32: El rendimiento de la arquitectura AVR32 ha demostrado ser excelente tras las pruebas y el estudio realizado. Las mejoras en la arquitectura desarrolladas por Atmel han hecho que el AVR32 sea un micro de excelentes prestaciones y con un consumo energético realmente bajo. A continuación se muestran algunos de los resultados obtenidos para la arquitectura AVR32:

- **Dificultad para medir las prestaciones de la arquitectura AVR32:** Comprobar todo el potencial de esta arquitectura, es una ardua tarea, ya que la aplicación construida no llega a explotar todas sus capacidades. A pesar de esto, a través del estudio de su arquitectura se ha demostrado como los AVR32 son idóneos para la robótica móvil y además, se ha podido ver la superioridad frente a otras alternativas existentes en el mercado.

- **Drivers del Framework de Atmel:** La suite de desarrollo AVR32 Studio dispone de un completo Framework donde se puede encontrar multitud de ejemplos y drivers para estos microcontroladores. Se trata de una ayuda importante para desarrolladores noveles, ya que estos ejemplos y drivers están listos para ser utilizados y se encuentran perfectamente detallados.

Placa de evaluación EVK1100: La placa EVK1100 ha demostrado ser una excelente herramienta para el desarrollo de aplicaciones, debido a la gran cantidad de periféricos disponibles y a la facilidad de uso que estos presentan gracias a los drivers del framework de Atmel. A continuación se detallan algunos de los resultados más importantes que se han obtenido de la construcción de un robot móvil con ella:

- **Facilidad de uso de la placa EVK1100:** Todas las pruebas realizadas han sido llevadas a cabo con éxito gracias a la facilidad de uso que presenta esta placa de evaluación. Por ejemplo, la disponibilidad de puertos RS232 (donde conectar directamente un cable serie y realizar debug a través de un PC) facilita el desarrollo de aplicaciones, o también la disponibilidad de una pantalla LCD ya instalada o botones totalmente configurables.

- **Entorno de pruebas del robot móvil:** Las pruebas finales realizadas al robot móvil han sido satisfactorias, aunque ha sido preciso realizar múltiples modificaciones de código hasta lograr el comportamiento deseado del conjunto. Las pruebas pueden dividirse en dos:

    - Funcionamiento del modo autónomo: Tras muchas modificaciones del código, se ha podido comprobar cómo el robot funciona de una forma totalmente autónoma, evitando colisionar con cualquier objeto que se le presente delante. Ha sido necesaria reducir la velocidad de funcionamiento de los motores, así como parar todo el conjunto cada vez que se realiza una medición, para evitar colisiones no deseadas.
    - Control manual remoto: La comunicación vía Bluetooth con el PC se realiza de forma satisfactoria, no detectando ninguna incidencia en su uso.

- **Escasa duración de la batería:** Durante las pruebas realizadas sobre el conjunto del robot, se ha apreciado como la duración de la batería es muy corta. A priori la alimentación de este se ha realizado únicamente con un conjunto de 6 pilas de 1.5V, proporcionando un total de 9V. Esto es debido principalmente a 2 factores:

    - Aunque se trabaja con un microcontrolador de bajo consumo, el conjunto de la placa de evaluación no lo es. El uso de la pantalla LCD, varios canales PWM, ADC, GPIO, dispositivos infrarrojos o la inserción del módulo Bluemore200, provocan que el consumo energético crezca de forma desmesurada.
    - El hecho de compartir una única fuente de alimentación entre la placa de evaluación EVK1100 y los dos motores de corriente continua, provoca una

caída de tensión demasiado grande ante la activación de los segundos. Este hecho tiene como consecuencia continuos reinicios del sistema.

Para solventar este último problema, se ha optado por separar los motores DC de la placa de evaluación EVK1100 mediante el uso de 2 baterías independientes. Así, se ha conseguido que los reinicios no se produzcan, aunque el consumo global sigue siendo excesivo.

- **Una única toma de 5V:** Aunque no es un impedimento para la elaboración del robot móvil, el hecho de incluir más de una toma que proporcione valores de 5V hubiera sido bueno. En este caso, la toma existente ha sido utilizada para alimentar el integrado L293D, aunque hubiera sido útil disponer de alguna más para poder utilizarlas en el Bluemore200 o en los sensores infrarrojos.

- **Área de conexionado poco eficiente:** La soldadura de elementos en la cuadrícula disponible para la conexión de dispositivos puede ser complicada, debido a que para la interconexión de puntos deben utilizarse puentes cableados, ya que de otro modo, se podría dañar la placa de evaluación. Además, todos los pines preparados para ser utilizados en el lateral (canales PWM, ADC, GPIO…), se encuentran demasiado juntos y puede provocar que se hagan contactos no deseados, cortocircuitándo la placa en ocasiones. Este problema se soluciona realizando soldaduras sobre la placa, pero provoca situaciones problemáticas ante la aparición de errores de diseño.

- **Tamaño de la placa:** Las dimensiones de la placa de evaluación EVK1100 hacen que su uso no sea el mejor para la construcción de robot móviles pequeños.

6. CONCLUSIONES

Los objetivos principales desarrollados en el proyecto han sido:

- El análisis de la arquitectura AVR32.
- La comparativa con otras alternativas existentes en el mercado (AVR8 o ARM)
- Estudio de la placa de evaluación EVK1100.
- Construcción de un robot móvil haciendo uso de esta.
- Creación de una aplicación de control, capaz de gestionar un robot de forma remota.

Los microcontroladores AVR32 han supuesto una evolución respecto a otros dispositivos existentes en el mercado. Tal y como se ha detallado en capítulos anteriores, el mercado de microcontroladores es muy amplio y abarca una enorme cantidad de dispositivos y aplicaciones distintas. En la medida de lo posible, cada una de las empresas productoras ha ido evolucionando sus modelos a otros más rápidos y potentes, con más memoria y periféricos, pero en su mayoría, sin aportar nada nuevo a la arquitectura de estos. Atmel, además de aumentar la frecuencia de sus microcontroladores, ha optado por la mejora de la arquitectura ampliando el potencial de sus micros sin ningún tipo de coste energético, a través de:

- La reducción del número de ciclos de carga y lectura de la CPU.
- La ejecución múltiple de tareas repetitivas.
- Maximizar la utilización de los recursos del pipeline.
- Minimizar las latencias en las instrucciones de salto.
- Mejoras en la densidad de código.

La arquitectura AVR32 ha demostrado ser óptima para la elaboración de aplicaciones móviles, gracias al eficiente uso energético que esta realiza y al nivel de potencia que ofrece. Además, si sumamos a las mejoras antes citadas, las más de 2000 interrupciones que puede controlar, junto con la integración de periféricos como el I2C, Ethernet o USB, hacen que sea una de las mejores opciones cuando se busca movilidad.

Al contrario, la placa de evaluación EVK1100 es poco eficiente cuando se desea personalizar para la realización de robots móviles. Los objetivos del proyecto han sido cumplidos con éxito, pero su gran tamaño provoca que la aplicación construida sea pesada. Además, el elevado número de componentes que integra, incrementa el gasto energético del conjunto de forma considerable.

La carencia de bibliografía sobre la arquitectura AVR32, ha sido un factor importante a lo largo del desarrollo del proyecto y puede ser determinante para que un usuario se decante por el uso de otra arquitectura. Los datos necesarios para realizar el estudio y la programación del AVR32, se han obtenido a través del análisis de la documentación del dispositivo y de los drivers disponibles en el entorno de programación. A pesar de que los objetivos han sido alcanzados, Atmel debería esforzarse en mejorar este aspecto.

Concluyendo este trabajo, pienso que los AVR32 son una de las mejores opciones existentes en el mercado, pero a pesar de que estos microcontroladores llevan varios años en el mercado, es difícil encontrar aplicaciones comerciales que hagan uso ellos, debido en gran parte a la influencia que tiene ARM en el mercado global. Personalmente opino que Atmel debería hacerse un hueco a nivel mundial, haciendo virtud de las mejoras que presentan estos microcontroladores e incluso, poner en circulación productos propios que hagan uso de los AVR32. De esta forma, se podría comenzar a ver productos bajo la marca de Atmel, del mismo modo que por ejemplo, ARM lo hace en el mercado de los dispositivos móviles.

7. BIBLIOGRAFÍA

[1] www.atmel.com. “Atmel Corporation”, página web oficial.

[2] www.arm.com. “ARM Ltd.”, página web oficial.

[3] www.atmel.com/dyn/resources/prod_documents/doc32000.pdf. “AVR32 Architecture Document”, Atmel Corp (11/2007).

[4] www.atmel.com/dyn/resources/prod_documents/doc2562.pdf. “Connecting to a target board with the AVR JTAGICE MKII”, Atmel Corp (07/2006).

[5] www.atmel.com/dyn/resources/prod_documents/doc32090.pdf. “UC3A schematic checklist”, Atmel Corp (12/2008).

[6] http://www.atmel.com/dyn/resources/prod_documents/doc32058.pdf. “AVR32 32-Bit Microcontroller Datasheet”, Atmel Corp (11/2009).

[7] www.atmel.org/dyn/resources/prod_documents/doc4092.pdf. “MCU Architectures for Compute-Intensive Embedded Applications”, Atmel Corp (12/2005).

[8] www.atmel.com/dyn/resources/prod_documents/doc32103.pdf. “Quick-start Guide. EVK1100 + Windows”, Atmel Corp (02/2010).

[9] www.atmel.com/dyn/resources/prod_documents/doc4064.pdf. “Quick Reference Guide”, Atmel Corp (02/2009).

[10] www.atmel.com/dyn/resources/prod_documents/doc7919.pdf. “AVR32 UC3 Flash Microcontrollers”, Atmel Corp (07/2009).

[11] www.atmel.com/dyn/resources/prod_documents/doc32002.pdf. “AVR32UC: Technical Reference Manual”, Atmel Corp (03/2010).

[12] www.avrfreaks.com. AVR Freaks, Comunidad oficial de AVR.

[13] www.freertos.org. “The FreeRTOS Project”, página web official.

[14] Kai Qian, David den Haring, Li Cao (2009). “Embedded Software Development with C”, Springer.

ANEXOS

A. PROGRAMACIÓN

Este punto pretende realizar una introducción a la programación de los microcontroladores AVR32 mediante la suite de desarrollo de Atmel. Esta herramienta, el AVR32 Studio, es gratuita y permitirá desarrollar y obtener multitud de ejemplos.

A continuación se analizan cuales son las dos opciones más comunes para desarrollar microcontroladores y se detalla el porqué decantarse por la programación Standalone frente a la implementación de un sistema operativo en tiempo real.

Por último se muestran fragmentos de código utilizados en el desarrollo del proyecto, detallando el funcionamiento de los diferentes módulos utilizados. De esta forma se pretende hacer una introducción a la programación en código C++, mostrando ejemplos del funcionamiento de diversos dispositivos.

A.1 ENTORNO DE PROGRAMACIÓN

El entorno de desarrollo utilizado para la programación del robot, es el que proporciona la propia compañía Atmel, el AVR32 Studio. Esta herramienta está basada en Eclipse, un popular entorno de desarrollo integrado y de código abierto.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para VisualAge. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

El funcionamiento de AVR32 Studio es sencillo y como ya se ha comentado, quien esté familiarizado con Eclipse no deberá tener ningún problema para comprender su funcionamiento. A pesar de ello, el siguiente punto describe el funcionamiento de este entorno y algunas de sus funciones más interesantes.

A.2 AVR32 STUDIO

Esta herramienta de trabajo puede ser descargada de forma gratuita de la propia página web de Atmel [1]. Es por ello que, antes de poder comenzar a programar el microcontrolador, será necesario instalar el entorno de programación como tal. Para ello [8], se debe instalar el GNUToolchain (proporciona los componentes necesarios para poder realizar la programación del MCU) y el AVR32 Studio.

El siguiente paso es conectar el programador JTAGICE MKII al puerto USB del PC y este a su vez a la placa de evaluación EVK1100 a través del puerto JTAG. Si todo funciona correctamente, AVR32 Studio reconocerá de forma automática estos dos dispositivos, aunque la mejor prueba es realizar una lectura de los registros internos del microcontrolador y así ver si este responde de forma correcta. Para ello, se seguirán los siguientes pasos:

1. Hacer “click” con el botón derecho del ratón sobre “AVR32 Targets” y seleccionar “Scan Targets”. Deberá aparecer el dispositivo tal y como se ve en la imagen siguiente:

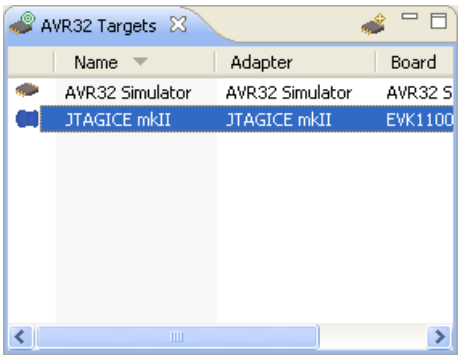

Figura 27: Ventana de dispositivos en AVR32 Studio

2. Ahora que AVR32 Studio ya detecta el JTAGICE MKII, se ha de configurar para indicarle al programa cual es la configuración exacta que se desea utilizar. Para ello, pulsamos encima del JTAGICE MKII y rellenamos los campos de la pestaña "Properties" tal y como se indica a continuación:

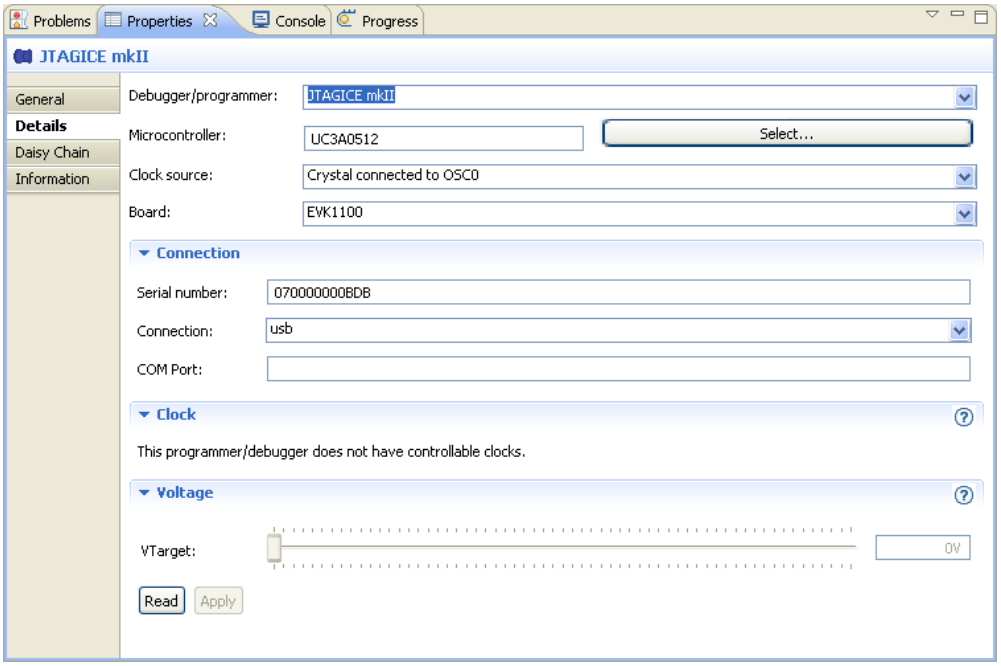

Figura 28: Configuración del microcontrolador en AVR32 Studio

3. A continuación, para leer los registros internos del microcontrolador, se ha de hacer "click" derecho con el ratón sobre el JTAGICE MKII que se encuentra situado en la parte inferior derecha y posteriormente, pulsar sobre "Read General Registers". Ahora se debería ver una nueva ventana con 16 registros y sus correspondientes valores en ese instante.

4. Por último, si se desea borrar todos los datos que se encuentran grabados sobre el microcontrolador, se debe proceder del mismo modo que en el punto 3, pero esta vez pulsando sobre "Chip Erase". Ahora, si se leen los valores de los registros, estos se encuentran con valor 0x00.

Antes de realizar el punto número 4 (borrado del Chip), se puede conectar la placa de evaluación EVK1100 mediante un cable Ethernet a un PC, y ver la aplicación que lleva implementada de fábrica: un servidor web que proporciona datos en tiempo real de la placa.

Aunque en caso de no hacerlo, siempre se puede recuperar esta aplicación del propio AVR32 Studio, ya que está incluida en él.

Estos ejemplos han sido de gran utilidad para la realización del proyecto. La bibliografía acerca del AVR32 es escasa (al contrario que para el AVR8), y por lo tanto, los ejemplos que se pueden encontrar también lo son. A pesar de este hecho, gracias al Framework incluido, se ha podido avanzar en el aprendizaje de este microcontrolador sin problemas. Entre los ejemplos incluidos en el entorno de programación se encuentran los siguientes (entre otros):

- **Control Panel Demo:** Se trata del servidor web que se incluye por defecto en la placa de evaluación EVK1100. En caso de haber hecho un *erase* del chip y desear recuperarlo, este es el programa a cargar.
- **CPU MPU Example:** Ejemplo sobre la configuración de la MPU.
- **External Interrupt Controller:** Como indica su nombre, proporciona 3 ejemplos distitnos del funcionamiento de las interrupciones externas.
- **General Purpose I/O (GPIO):** Ofrece dos ejemplos sobre cómo hacer uso de los puertos GPIO. Desde el punto de vista del proyecto son muy interesantes ya que la activación de LEDs y motores, se realiza a través de estos puertos.
- **Interrupt Controller (INTC):** Ejemplo sobre el uso de las interrupciones internas.
- **Power Manager (PM):** Indica cómo hacer uso del Power Manager, que entre otras cosas permitirá seleccionar diferentes velocidades de reloj.
- **Pulse Width Modulation (PWM):** Importante ejemplo sobre cómo utilizar los canales PWM que serán vitales para el movimiento de los servomotores.
- **USART Example:** Detalla el funcionamiento de los USART de la placa. Imprescindible para la comunicación serie con el robot.
- **DIP204 Example:** Completo ejemplo que muestra datos a través del LCD integrado y además hace uso de interrupciones externas que son activadas a través de los botones incluidos en la placa.
- **FreeRTOS Example:** Ejemplo de uso de este sistema operativo en tiempo real. Se trata de un SO gratuito y ampliamente extendido a muchos otros microcontroladores.

En caso de querer utilizar alguno de estos ejemplos, se deberá pulsar sobre “File – New – AVR32 Example Project” y aquí seleccionar la placa de evaluación. Una vez seleccionado uno de los ejemplos a utilizar, pulsar sobre “Finish” y en la barra lateral izquierda aparecerá listado como un nuevo proyecto.

Una vez llegados a este paso, lo único que queda es identificar donde se encuentran los ficheros con el código fuente. Bajo el directorio “Proyecto / src /” se encontrarán todos los ficheros fuentes y es aquí, donde se podrán añadir otras fuentes o ficheros de cabeceras, en caso de ser necesario.

AVR32 Studio dispone de una herramienta que automáticamente instalará sobre el proyecto todas las dependencias y librerías necesarias para el uso de cualquier componente de nuestro microcontrolador. De este modo, únicamente se deberá seleccionar los drivers que se considere necesario y ya se podrá comenzar a trabajar con estos nuevos componentes. Para acceder a este menú de configuración, se debe hacer “click” en “Framework – Select Drivers/Components/Services” y seleccionar los que se vallan a utilizar.

A.3 ¿FREERTOS O PROGRAMACIÓN "STANDALONE"?

En el momento de iniciar un proyecto en AVR32 Studio, el propio programa hace hincapié en que tipo de proyecto se desea, y entre otros, una de las opciones que muestra es la de crear un proyecto standalone. El concepto de standalone, históricamente, no es otro que el de una aplicación que no necesita ningún tipo de sistema operativo para funcionar. En el caso a estudio, parece ser claro que no se hará uso de SO, así que los primeros contactos con el entorno de programación y código se hacen bajo estas condiciones.

Pero a pesar de lo mencionado, una de las ventajas de disponer de un microcontrolador de altas prestaciones, es la de poder instar en él un sistema operativo en tiempo real, que ayude a la planificación y coordinación de las tareas que se implementen. Es por ello, que a pesar de que la programación "standalone" puede ser más sencilla, podría ser interesante el decantarse por implementar uno de estos sistemas.

Como sistema operativo, AVR32 Studio nos permite hacer uso de FreeRTOS, un sistema operativo en tiempo real para dispositivos embebidos y que ha sido portado a la mayoría de microcontroladores. Este SO está distribuido bajo la GPL y está diseñado para ser pequeño y simple. De hecho, el núcleo del SO está formado únicamente por 3 o 4 ficheros. El lenguaje utilizado en su programación es C, lo que permite que sea legible, fácil de portar a otras plataformas y fácil también, de mantener.

Este sistema, se puede descargar de forma totalmente gratuita desde la web de FreeRTOS [13], y dispone de varios ejemplos específicos para los diferentes microcontroladores que lo soportan. Algunas de sus características técnicas más importantes son:

- Soporte oficial para 23 arquitecturas de microcontroladores.
- Diseñado para ser pequeño, simple y fácil de utilizar.
- Portable a otras arquitecturas gracias a que su código está escrito en C.
- Soporte para tareas y rutinas.
- No hay límite para el número de tareas que pueden ser creadas y lanzadas.
- No hay límite para el número de prioridades que pueden ser usados.
- Más de una tarea puede tener el mismo nivel de prioridad.
- Soporte para colas, semáforos y mutexes para las comunicaciones y sincronización entre tareas e interrupciones.

Debido a que las exigencias de nuestro proyecto no requieren su uso, toda la programación se llevará a cabo de forma Standalone, aunque es conveniente saber de su existencia y uso.

A.4 PROGRAMACIÓN DEL AVR32

A continuación se mostrarán fragmentos de código utilizado para la configuración de los diferentes elementos necesarios en la realización del proyecto, aunque para la obtención del código fuente completo deben dirigirse al punto B y C del Anexo.

Power Manager:

El Power Manager es el encargado de gestionar, entre otras cosas, el funcionamiento de los buses internos del microcontrolador. Es por ello que se utilizará para proporcionar un reloj a todo el sistema, que podrá variar en función de las necesidades. Se ha de tener en cuenta que a mayor velocidad de reloj, mayor potencia de cálculo, pero también conllevará un aumento del consumo energético.

Esta velocidad se puede variar mediante la inicialización de relojes externos/internos o mediante el uso de los PLL's, de modo que su velocidad de funcionamiento puede oscilar entre los 112 KHz hasta los 66 MHz máximos. A continuación se muestra el fragmento de código utilizado para la inicialización en los siguientes casos:

12 MHz: Únicamente es necesario incluir la siguiente línea al comienzo del programa principal:

```
pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP);
```

Para el resto de velocidades de reloj, será necesario la modificación de los divisores del PLL, aunque el proyecto se llevará a cabo a una frecuencia de 12 MHz ya que se trata de una velocidad suficiente para el funcionamiento del conjunto.

66 MHz:

```
pm_switch_to_osc0( &AVR32_PM, FOSC0, OSC0_STARTUP );
pm_pll_setup( &AVR32_PM, 0,
          10,
          1,
          0,
          16);
pm_pll_set_option(&AVR32_PM, 0,
           1,
           1,
           0);
pm_pll_enable(&AVR32_PM, 0);
pm_wait_for_pll0_locked(&AVR32_PM);
pm_cksel(&AVR32_PM,
           0,
           0,
           0,
           0,
           0,
           0);
flashc_set_wait_state(1);
pm_switch_to_clock(&AVR32_PM, AVR32_PM_MCCTRL_MCSEL_PLL0);
```

Si bien, para poder hacer uso de estas funciones, primero es necesario incluir en nuestras librerías las correspondientes al Power Manager a través del menú “Framework – Select Drivers”. Posteriormente, deberá ser declarada al inicio del programa.

GPIO:

Como ya se ha comentado anteriormente, los canales GPIO permitirán activar/desactivar señales o leer valores digitales de dispositivos externos. Para el proyecto, su uso gira entorno a la activación de las ruedas motrices.

En primer lugar, es necesario que del mismo modo que para el Power Manager, se incluyan las librerías necesarias en el proyecto a través del selector de drivers incluido en AVR32 Studio. Para este caso y para todos los posteriores deberá realizarse del mismo modo, por lo que de ahora en adelante este paso será omitido. Una vez incluidas las librerías necesarias, deben definir las variables que sean acordes a los elementos que se desea controlar. En este caso, cada uno de los motores dispone de 3 controles:

- El PIN Enable, que activará o desactivará el motor y que será definido para que haga uso del PIN PB01 del microcontrolador.
- Los interruptores A y B, que serán los encargados de ofrecer las señales necesarias para poder gobernar el robot.

```
#define ENABLE_PIN_MOTOR_DER        AVR32_PIN_PB01
#define A_PIN_MOTOR_DER             AVR32_PIN_PB02
#define B_PIN_MOTOR_DER             AVR32_PIN_PB03
```

Una vez realizado este paso, se debe inicializar el Bus GPIO y este ya se encontrará listo para utilizarse. Además, AVR32 permite activar el filtro anti-glitch, de modo que cualquier pulso de duración menor a un ciclo de reloj será descartado y no afectará al funcionamiento del robot. En este caso, se hace uso de este filtro en todas las salidas/entradas del motor:

```
gpio_local_init();
gpio_enable_pin_glitch_filter(ENABLE_PIN_MOTOR_DER);
gpio_enable_pin_glitch_filter(A_PIN_MOTOR_DER);
gpio_enable_pin_glitch_filter(B_PIN_MOTOR_DER);
```

Por último, solo queda hacer uso de estos canales mediante la habilitación o deshabilitación de estos. Si por el contrario, se desea obtener un valor lógico desde un evento externo, este puede ser leido mediante el último de los comandos, aunque para la realización de esta tarea, los canales ADC son los más apropiados, por ofrecer niveles analógicos y no valores de 0 o 1 como ocurre con el GPIO.

```
gpio_set_gpio_pin(ENABLE_PIN_MOTOR_DER);
gpio_set_gpio_pin(A_PIN_MOTOR_DER);
gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

gpio_get_pin_value(GPIO_PUSH_BUTTON_0);
```

ADC:

Este tipo de canales serán los encargados de realizar la lectura de los sensores externos utilizados para la detección de colisiones del robot. Serán utilizados estos y no los GPIO, ya que permitirán obtener un valor "analógico", que nos aportará datos sobre la distancia a la que se encuentren los obstáculos más precisos que en el caso de hacer uso del GPIO. Para su uso, el primer paso es declarar las variables necesarias tal y como se muestra en el siguiente fragmento. Existen 7 canales posibles para utilizar, aunque en este ejemplo se especifica su uso para el canal 0.

```
#define     DETECTAR_COLISION_CHANNEL   0
#define     DETECTAR_COLISION_PIN       AVR32_ADC_AD_0_PIN
#define     DETECTAR_COLISION_FUNCTION  AVR32_ADC_AD_0_FUNCTION

volatile avr32_adc_t *adc = &AVR32_ADC;
signed short adc_value_ir = -1;
unsigned short adc_channel_sensor = DETECTAR_COLISION_CHANNEL;
```

Una vez se han declarado estas variables es necesario mapear todos los puertos que se vallan a utilizar para que funcionen como canales ADC y no como canales de propósito general. Tal y como se dijo, todos los puertos disponen de varias funciones y es por eso que antes de hacer uso de ellos se debe especificar en nuestro programa que función han de realizar. Por último, sólo quedará hacer uso de las funciones que se muestran para poder realizar medidas a través de estos puertos.

```
static const gpio_map_t ADC_GPIO_MAP =
{
{DETECTAR_COLISION_PIN, DETECTAR_COLISION_FUNCTION},
{DETECTAR_COLISION_PIN_1, DETECTAR_COLISION_FUNCTION_1},
{DETECTAR_COLISION_PIN_2, DETECTAR_COLISION_FUNCTION_2},
};

gpio_enable_module(ADC_GPIO_MAP, sizeof(ADC_GPIO_MAP) /
sizeof(ADC_GPIO_MAP[0]));

adc_configure(adc);

adc_enable(adc,adc_channel_sensor);
adc_enable(adc,adc_channel_sensor_1);
adc_enable(adc,adc_channel_sensor_2);

adc_start(adc);
adc_value_ir = adc_get_value(adc, adc_channel_sensor);
```

PWM:

La programación de los canales PWM es sencilla y a diferencia de otros microcontroladores, no es necesario gestionar ningún tipo de interrupción ni contador. Con la configuración mostrada se pueden generar pulsos cuadrados del periodo que deseemos a través del canal 0. Como existen 7 canales, sólo hay que modificar el valor 0 y sustituirlo por un número entre el 0 y el 6.

```
int status_pwm_0 = -1;
pwm_opt_t pwm_opt_channel_0;
avr32_pwm_channel_t pwm_channel_0 = { .ccnt = 0 };
unsigned int channel_id_0;
```

Como siempre, el primer paso a realizar es una declaración de todas las variables implicadas en la generación de señales PWM, para posteriormente seguir con la generación de la onda. Este microcontrolador permite modificar muchas de las características de la señal así como lo son la polaridad esta, su posición, el periodo o el duty-cycle a utilizar.

```
channel_id_0 = 0;

gpio_enable_module_pin(AVR32_PWM_0_PIN, AVR32_PWM_0_FUNCTION);

pwm_opt_channel_0.diva = AVR32_PWM_DIVA_CLK_OFF;
pwm_opt_channel_0.divb = AVR32_PWM_DIVB_CLK_OFF;
pwm_opt_channel_0.prea = AVR32_PWM_PREA_MCK;
pwm_opt_channel_0.preb = AVR32_PWM_PREB_MCK;

pwm_init(&pwm_opt_channel_0);

pwm_channel_0.CMR.calg = PWM_MODE_LEFT_ALIGNED;
pwm_channel_0.CMR.cpol = PWM_POLARITY_LOW;
pwm_channel_0.CMR.cpd = PWM_UPDATE_DUTY;
pwm_channel_0.CMR.cpre = AVR32_PWM_CPRE_MCK_DIV_1024
pwm_channel_0.cdty = 210;
pwm_channel_0.cprd = 234;
pwm_channel_0.cupd = 0;
```

Las 4 últimas líneas de código son las que permitirán seleccionar el periodo del pulso PWM. Mediante la siguiente fórmula se puede calcular el valor de estas variables para generar la frecuencia deseada.

(115200/256)/20 == 22.5Hz == (Clock Frequency / Prescaler) / Period

Por último, sólo quedará iniciar o parar el canal PWM según sea necesario. Para ello, se hará uso de las instrucciones “pwm_stop_channels” y “pwm_start_channels” tal y como se puede ver en el siguiente fragmento de código.

```
pwm_channel_init(channel_id_0, &pwm_channel_0);
pwm_stop_channels(1 << channel_id_0;
pwm_start_channels(1 << channel_id_0);
```

Ahora que ya se encuentran configurados y funcionando los canales PWM, se podrá controlar el servomotor según convenga o proporcionar a los motores DC, una velocidad variable conectando uno de sus terminales (por ejemplo el Enable), a uno de los canales PWM.

Delay:

Nuevamente nos encontramos que gracias a las librerías existentes en el entorno de programación, será muy sencillo poder generar delays temporales únicamente llamando al comando “delay_ms()”. Como su nombre indica, nos generará un retraso de tantos milisegundos como le especifiquemos entre los paréntesis.

Este tipo de función es imprescindible si queremos hacer uso del LCD, ya que con él podremos limitar el refresco de la pantalla para que esta sea legible (aunque también podríamos hacer uso de un simple bucle for).

```
delay_init(FOSC0);
delay_ms(500);
```

USART:

El uso que se hará del USART en este proyecto, será el poder comunicarse con el robot desde un PC, mandándole instrucciones y poder realizar un debug a través de un terminal y un puerto serie. También será útil para mostrar por pantalla el valor de los sensores y de este modo comprobar si están funcionando de forma correcta. Para ello, una vez más deberán declararse todas las variables iniciales necesarias para el funcionamiento de este dispositivo, y mapear los puertos para que funcionen en modo USART.

```
#define COMM_ROBOT_USART              (&AVR32_USART1)
#define COMM_ROBOT_USART_RX_PIN    AVR32_USART1_RXD_0_0_PIN
#define COMM_ROBOT_USART_RX_FUNCTION
AVR32_USART1_RXD_0_0_FUNCTION
#define COMM_ROBOT_USART_TX_PIN     AVR32_USART1_TXD_0_0_PIN
#define COMM_ROBOT_USART_TX_FUNCTION
AVR32_USART1_TXD_0_0_FUNCTION

int status_usart = -1;

static const gpio_map_t USART_GPIO_MAP =
{
  {COMM_ROBOT_USART_RX_PIN, COMM_ROBOT_USART_RX_FUNCTION},
  {COMM_ROBOT_USART_TX_PIN, COMM_ROBOT_USART_TX_FUNCTION}
};
```

Ahora llega el momento de definir las características básicas de funcionamiento del USART como lo son la tasa de transferencia, la longitud del carácter o el tipo de paridad que va a utilizarse. Para conocer mejor estas características y su uso, se puede revisar la librería correspondiente al USART, ya que es aquí donde se encuentran las diferentes opciones y parámetros a utilizar, así como una breve explicación de estos.

```
static const usart_options_t USART_OPTIONS =
{ .baudrate     = 57600,
  .charlength   = 8,
  .paritytype   = USART_NO_PARITY,
  .stopbits     = USART_2_STOPBITS,
  .channelmode  = USART_NORMAL_CHMODE };
```

```
gpio_enable_module(USART_GPIO_MAP, sizeof(USART_GPIO_MAP) / sizeof(USART_GPIO_MAP[0]));

usart_init_rs232(COMM_ROBOT_USART, &USART_OPTIONS, FOSC0);
```

Ahora, sólo restará hacer uso de este canal de comunicación ya sea para realizar debug a través del terminal serie o para comunicarse con el robot.

```
usart_getchar(COMM_ROBOT_USART);
usart_putchar(COMM_ROBOT_USART, opcion);

print_dbg("\033[22;30m");
print_dgb_hex(opcion);
```

B. Aplicación de Control en Visual Basic 6.0

```
// ##############################################################################
// # Programa para el control remoto del robot móvil                            #
// # Por Francisco Muñoz Verdú. UAB.                                            #
// ##############################################################################

// Variables para la comunicación con el microcontrolador AVR32.
Dim i As Integer, Letra(99)

// Buffer donde se almacenarán los datos de la comunicación RS232.
Dim InBuff As String

// Función para Cambiar / Quitar el control manual del Robot.
Private Sub Command1_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

// Deshabilitamos todos los controles a excepción del botón 1, que será el del Control Manual.
If Command1.Caption = "Control Manual" Then

// Guardamos la letra "f" y la enviaremos a través del puerto serie mediante MSComm1.Output
   Letra(0) = "f"
   MSComm1.Output = Letra(0)

// Deshabilitamos los botones, para que no puedan ser utilizados
   Command2.Enabled = False
   Command5.Enabled = False
   Command6.Enabled = False
   Command1.Caption = "Fin Control"

// Si volvemos a pulsar sobre el botón 1, habilitaremos todos los botones de nuevo
ElseIf Command1.Caption = "Fin Control" Then

// Enviamos una "q" al microcontrolador para salir del programa del modo manual.
   Letra(0) = "q"
   MSComm1.Output = Letra(0)

// Y volvemos a deshabilitar los controles manuales.
   Command2.Enabled = True
   Command5.Enabled = True
   Command6.Enabled = True
   Command1.Caption = "Control Manual"
   Command3.Enabled = False
   Command4.Enabled = False
   Command7.Enabled = False
   Command8.Enabled = False
   Command9.Enabled = False
   Command10.Enabled = False
   Command11.Enabled = False
   Command12.Enabled = False
   Command13.Enabled = False
   Command23.Enabled = False
   Command24.Enabled = False
   Command16.Enabled = False
```

```
    Command17.Enabled = False
    Command18.Enabled = False

End If
End Sub

// Función para activar el modo autónomo. El proceso de envío de datos se hará igual para
// todas las funciones del programa mediante el uso de MSComm1.
Private Sub Command2_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

// Enviamos al micro la orden para entrar en Modo Autónomo
If Command2.Caption = "Modo Autonomo" Then
    Letra(0) = "e"
    MSComm1.Output = Letra(0)
    Command1.Enabled = False
    Command5.Enabled = False
    Command6.Enabled = False
    Command2.Caption = "Fin Modo"

// Enviamos al micro la orden para salir del modo autónomo
ElseIf Command2.Caption = "Fin Modo" Then
    Letra(0) = "t"
    MSComm1.Output = Letra(0)
    Command1.Enabled = True
    Command5.Enabled = True
    Command6.Enabled = True
    Command2.Caption = "Modo Autonomo"
End If
End Sub

// Función que lanzará una prueba sobre el servomotor de forma automática.
Private Sub Command5_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)
    Letra(0) = "a"
    MSComm1.Output = Letra(0)

// Inicialmente deshabilitamos todos los botones y más tarde, en el control de comunicación se
// volverán a activar.
    Command1.Enabled = False
    Command2.Enabled = False
    Command5.Enabled = False
    Command6.Enabled = False
    Command5.Caption = "Ejecutando..."
End Sub

// Función que lanzará una prueba sobre el motor DC de forma automática.
Private Sub Command6_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

    Letra(0) = "b"
    MSComm1.Output = Letra(0)
```

```
// Inicialmente deshabilitamos todos los botones y más tarde, en el control de comunicación se
// volverán a activar.
   Command1.Enabled = False
   Command2.Enabled = False
   Command5.Enabled = False
   Command6.Enabled = False
   Command6.Caption = "Ejecutando..."
End Sub

// Enviamos una "s" para que el robot se mueva hacia detrás
Private Sub Command10_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "s"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "k" para que el robot realice la medida del sensor frontal
Private Sub Command11_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "k"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "l" para que el robot realice la medida del sensor derecho
Private Sub Command12_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "l"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "j" para que el robot realice la medida del sensor izquierdo
Private Sub Command13_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "j"
MSComm1.Output = Letra(1)
End Sub

// Control para la conexión con los puertos COM. Debemos seleccionar el que tengamos
// configurado en nuestro PC
Private Sub Command14_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

// Realizamos un control de errores para evitar que el programa se cierre al seleccionar un
// puerto que esté cerrado
On Error GoTo control_errores

If Combo1.Text = "COM 1" Then
   Letra(2) = 1
End If
```

```
If Combo1.Text = "COM 2" Then
   Letra(2) = 2
End If

If Combo1.Text = "COM 3" Then
   Letra(2) = 3
End If

If Combo1.Text = "COM 4" Then
   Letra(2) = 4
End If

If Combo1.Text = "COM 5" Then
   Letra(2) = 5
End If

If Combo1.Text = "COM 6" Then
   Letra(2) = 6
End If

// Configuración necesaria para el funcionamiento del módulo MSComm1
With MSComm1
.CommPort = Letra(2)
.RThreshold = 1
.RTSEnable = True
.Settings = "57600,n,8,1"
.SThreshold = 1
.PortOpen = True
End With

Text1.Text = ""

// Si la conexión se establece, ponemos en color verde el círculo
Shape5.BackColor = &HFF00&
Command14.Enabled = False
Command15.Enabled = True

// Y además, habilitamos los botones de control
If MSComm1.PortOpen = True Then
   Command1.Enabled = True
   Command2.Enabled = True
   Command5.Enabled = True
   Command6.Enabled = True
End If
Exit Sub

// Mensaje de error en el caso de no seleccionar un puerto correcto
control_errores:
MsgBox "El Puerto COM indicado no es correcto. Por favor, seleccione otro.", vbExclamation,
"Error de conexión"

End Sub
```

```
// Función que cerrará la comunicación serie y además, pondrá en rojo el circulo. También se
// deshabilitarán los botones de control.
Private Sub Command15_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

If MSComm1.PortOpen = True Then
   MSComm1.PortOpen = False
End If

Shape5.BackColor = &HFF&
Command15.Enabled = False
Command14.Enabled = True

If MSComm1.PortOpen = False Then
   Command1.Enabled = False
   Command2.Enabled = False
   Command5.Enabled = False
   Command6.Enabled = False
End If
End Sub

// Enviamos una "a" para que el robot se mueva hacia la izquierda
Private Sub Command7_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "a"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "w" para que el robot se mueva hacia delante
Private Sub Command8_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "w"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "d" para que el robot se mueva hacia la derecha
Private Sub Command9_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "d"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "y" para mover el servomotor
Private Sub Command24_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As
Single)

Letra(1) = "y"
MSComm1.Output = Letra(1)
End Sub
```

```
// Enviamos una "u" para mover el servomotor
Private Sub Command16_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)

Letra(1) = "u"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "i" para mover el servomotor
Private Sub Command17_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)

Letra(1) = "i"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "o" para mover el servomotor
Private Sub Command18_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)

Letra(1) = "o"
MSComm1.Output = Letra(1)
End Sub

// Enviamos una "p" para mover el servomotor
Private Sub Command23_MouseUp(Button As Integer, Shift As Integer, X As Single, Y As Single)

Letra(1) = "p"
MSComm1.Output = Letra(1)
End Sub

// Cuando cerremos el formulario, también se cerrará la comunicación con el puerto serie.
Private Sub Form_Unload(Cancel As Integer)

If MSComm1.PortOpen = True Then
   MSComm1.PortOpen = False
End If
End Sub

// Ante la recepción de información a través del puerto COM, establecemos unas reglas para
// que actúe en consecuencia.
Private Sub MSComm1_OnComm()

// Guardamos los datos recibidos en un buffer llamado InBuff, declarado al inicio del programa.
InBuff = MSComm1.Input
Text1.Text = InBuff

// Guardamos el valor del sensor frontal dentro de una casilla de texto
If Letra(1) = "k" Then
   If Text1.Text <> "q" Then
      Text2.Text = InBuff
   End If
```

```
End If

// Guardamos el valor del sensor izquierdo dentro de una casilla de texto
If Letra(1) = "j" Then
   If Text1.Text <> "q" Then
      Text4.Text = InBuff
   End If
End If

// Guardamos el valor del sensor derecho dentro de una casilla de texto
If Letra(1) = "l" Then
   If Text1.Text <> "q" Then
      Text3.Text = InBuff
   End If
End If

// Variable que recibiremos ante la ejecución del modo manual. Se habilitarán todos los
// botones ‘de control
If Text1.Text = "8" Then
   Command3.Enabled = True
   Command4.Enabled = True
   Command7.Enabled = True
   Command8.Enabled = True
   Command9.Enabled = True
   Command10.Enabled = True
   Command11.Enabled = True
   Command12.Enabled = True
   Command13.Enabled = True
   Command23.Enabled = True
   Command24.Enabled = True
   Command16.Enabled = True
   Command17.Enabled = True
   Command18.Enabled = True
End If

// Con esta condición, evaluaremos para que función se envía una sentencia de Quit.
If Text1.Text = "q" Then
   If Command5.Caption = "Ejecutando..." Then
      Command1.Enabled = True
      Command2.Enabled = True
      Command5.Enabled = True
      Command6.Enabled = True
      Command5.Caption = "Prueba Servo M."
   End If
   If Command6.Caption = "Ejecutando..." Then
      Command1.Enabled = True
      Command2.Enabled = True
      Command5.Enabled = True
      Command6.Enabled = True
      Command6.Caption = "Prueba Motor DC"
   End If
End If
End Sub
```

C. Software de Control del Microcontrolador

```
/*****************************************************************************************************
*                                    Robot Móvil. Proyecto Final de Carrera                          *
*****************************************************************************************************/

// Incluimos todas las librerias necesarias. Previamente tenemos que haber añadido los ficheros
// necesarios a nuestro proyecto. Para hacer eso nos vamos a "Framework – Select Drivers /
// Components/Services" y marcamos los que deseemos.

#include "board.h"
#include "print_funcs.h"
#include "gpio.h"
#include "pm.h"
#include "adc.h"
#include <avr32/io.h>
#include "compiler.h"
#include "usart.h"
#include "delay.h"
#include "pwm.h"
#include "spi.h"
#include "dip204.h"
#include "intc.h"

// Defines necesarios para el funcionamiento de la comunicación USART. Siempre son
// necesarios dos del mismo tipo, ya que el PIN indica el PIN físico del microcontrolador y la
// función nos dirá que función realizará ese PIN. En la parte de configuración del USART se
// mapearán los pines para que funcionen como nosotros deseamos.

#define COMM_ROBOT_USART                (&AVR32_USART1)
#define COMM_ROBOT_USART_RX_PIN         AVR32_USART1_RXD_0_0_PIN
#define COMM_ROBOT_USART_RX_FUNCTION    AVR32_USART1_RXD_0_0_FUNCTION
#define COMM_ROBOT_USART_TX_PIN         AVR32_USART1_TXD_0_0_PIN
#define COMM_ROBOT_USART_TX_FUNCTION    AVR32_USART1_TXD_0_0_FUNCTION
#define COMM_ROBOT_USART_IRQ            AVR32_USART1_IRQ

// Defines necesarios para el funcionamiento de los motores DC. Cada motor tiene 3 entradas,
// el enable y los pines a y b que servirán para controlar el sentido. El pin enable es definido
// como un canal PWM.

#define A_PIN_MOTOR_DER                 AVR32_PIN_PB02
#define B_PIN_MOTOR_DER                 AVR32_PIN_PB03
#define A_PIN_MOTOR_IZQ                 AVR32_PIN_PB09
#define B_PIN_MOTOR_IZQ                 AVR32_PIN_PB10

// Defines necesarios para el funcionamiento de los canales ADC y los sensores IR. Del mismo
// modo que con los USART, hay que indicar el pin y la función que va a realizar para
// posteriormente poder realizar el mapeo de los GPIO. Además, se definen algunas variables
// para poder almacenar los datos obtenidos, entre otras.

// El canal 0 estará definido para el sensor Frontal. El canal 1 para el sensor derecho y el canal
// 2 para el sensor izquierdo.

#define     DETECTAR_COLISION_CHANNEL     0
#define     DETECTAR_COLISION_PIN         AVR32_ADC_AD_0_PIN
#define     DETECTAR_COLISION_FUNCTION    AVR32_ADC_AD_0_FUNCTION
#define     DETECTAR_COLISION_CHANNEL_1   1
#define     DETECTAR_COLISION_PIN_1       AVR32_ADC_AD_1_PIN
#define     DETECTAR_COLISION_FUNCTION_1  AVR32_ADC_AD_1_FUNCTION
#define     DETECTAR_COLISION_CHANNEL_2   2
```

```
#define		DETECTAR_COLISION_PIN_2		AVR32_ADC_AD_2_PIN
#define		DETECTAR_COLISION_FUNCTION_2	AVR32_ADC_AD_2_FUNCTION

volatile avr32_adc_t *adc = &AVR32_ADC; // Variable para definir el canal ADC en MEM.
signed short adc_value_ir = -1; // Lo usaremos para medir los valores del sensor frontal
unsigned short adc_channel_sensor = DETECTAR_COLISION_CHANNEL;

signed short adc_value_ir_1 = -1; // Lo usaremos para medir los valores del sensor derecho
unsigned short adc_channel_sensor_1 = DETECTAR_COLISION_CHANNEL_1;

signed short adc_value_ir_2 = -1; // Lo usaremos para medir los valores del sensor izquierdo
unsigned short adc_channel_sensor_2 = DETECTAR_COLISION_CHANNEL_2;

// Variables necesarias para el funcionamiento de los canales PWM. Estos nos ayudaran
// a poder controlar el movimiento del servomotor y ajustar la velocidad de los motores DC.

int status_pwm_0 = -1;
pwm_opt_t pwm_opt_channel_0; // PWM configurar opciones para el Channel 0.
avr32_pwm_channel_t pwm_channel_0 = { .ccnt = 0 };  // Configuración para un único canal.
unsigned int channel_id_0; // Generamos una variable para que pueda ser utilizada en
referencia al número del canal, a lo largo del programa.

int status_pwm_1 = -1;
pwm_opt_t pwm_opt_channel_1; // PWM configurar opciones para el Channel 3.
avr32_pwm_channel_t pwm_channel_1 = { .ccnt = 0 };  // Configuración para un único canal.
unsigned int channel_id_1; // Generamos una variable para que pueda ser utilizada en
referencia al número del canal, a lo largo del programa.

int status_pwm_2 = -1;
pwm_opt_t pwm_opt_channel_2; // PWM configurar opciones para el Channel 2.
avr32_pwm_channel_t pwm_channel_2 = { .ccnt = 0 };  // Configuración para un único canal.
unsigned int channel_id_2; // Generamos una variable para que pueda ser utilizada en
referencia al número del canal, a lo largo del programa.

int duty_servo_motor = 216;

// Y por último antes del main principal, declararemos todas las funciones que hemos
// ido creando y que son necesarias para la ejecución del programa.

int configurar_usart(void); // Función que configurara todos los parámetros del USART
int configurar_canal_pwm_0(void); // Función que configurara todos los parámetros del PWM0
int configurar_canal_pwm_1(void); // Función que configurara todos los parámetros del PWM1
int configurar_canal_pwm_2(void); // Función que configurara todos los parámetros del PWM2
void configurar_puertos_gpio(void); // Función que configura los puertos GPIO utilizados
void configurar_dip204(void); // Función que configura el display dip204
void configurar_canal_adc(void); // Función para configurar los puertos ADC
void prueba_servomotor(void); // Función que servirá para testear el servomotor
void modo_automata(void); // Función que hará que el robot se mueva de forma autónoma
void modo_manual(void); // Función para que el robot funcione controlado desde el PC
int comprobacion_frontal(void); // Comprobar si hay obstáculo en el frontal derecho
int comprobacion_lateral_derecho(void); // Comprobar si hay obstáculo en el lateral derecho
int comprobacion_lateral_izquierdo(void); // Comprobar si hay obstáculo en el lateral izquierdo
void prueba_motores_dc(void); // Prueba automática de los motores DC


/*****************************************************************************************************
*                                          Función Main                                             *
*****************************************************************************************************/
```

```
int main()
{
	int status = -1;   // Variable para guardar el retorno de las funciones (SUCCES o FAIL)
	int opcion = -1;  // Variable para guardar la opción seleccionada en menú

	pm_switch_to_osc0(&AVR32_PM, FOSC0, OSC0_STARTUP); // Osc. 0 a 12 MHz
	gpio_local_init(); // Habilita el bus local para la interfaz GPIO.

	delay_init(FOSC0); // Habilita los delays

	configurar_dip204(); // Configuramos el display dip204

	delay_ms(1500);

	// Mostraremos por la pantalla LCD, un mensaje personal
	dip204_clear_display();
	dip204_set_cursor_position(1,1);
	dip204_write_string("Proyecto Final de C.");
	dip204_set_cursor_position(1,2);
	dip204_write_string("Francisco Munoz V.");
	dip204_set_cursor_position(1,3);
	dip204_write_string("Escola d'Enginyeria");
	dip204_set_cursor_position(1,4);
	dip204_write_string("UAB");
	dip204_hide_cursor();

	delay_ms(1500);

	// Configuramos la interfaz del USART1 para Debug y transmisión/recepción
	status = configurar_usart();

	dip204_clear_display();
	dip204_set_cursor_position(1,1);
	dip204_write_string("Configurando: USART");
	dip204_hide_cursor();

	delay_ms(1000); // Para que le de tiempo al Bluetooth a configurarse

	dip204_set_cursor_position(1,2);
	dip204_write_string("Configurando: GPIO");
	dip204_hide_cursor();

	configurar_puertos_gpio(); // Configuramos los puertos GPIO

	delay_ms(500);

	dip204_set_cursor_position(1,3);
	dip204_write_string("Configurando: PWM");
	dip204_hide_cursor();

	status = configurar_canal_pwm_0(); // Configuramos el canal PWM0
	status = configurar_canal_pwm_1(); // Configuramos el canal PWM1
	status = configurar_canal_pwm_2(); // Configuramos el canal PWM2

	delay_ms(500);

	dip204_set_cursor_position(1,4);
	dip204_write_string("Configurando: ADC");
	dip204_hide_cursor();
```

```
	configurar_canal_adc(); // Configuramos el canal ADC para ser usado con el sensor IR

	delay_ms(500);

	// Este será el bucle principal del programa de testeo. Se incluyen 5 funciones posibles
	// a realizar y que tendremos que escoger abriendo una sesión de hyperterminal y
	// enviando los caracteres correspondientes a la función a realizar. Una vez realizada la
	// función volveremos al menú de selección.

	for(;;) // Bucle infinito
	{
		// Existen muchas combinaciones para poder realizar en la comunicación serie
		// a través de sesiones de hyperterminal. Si queremos tabular, borrar la pantalla
		// o cambiar el color del texto entre otras, deberemos dirigirnos a la librería
		// print_funcs.h, donde están todas descritas.

		// Función que esperará recibir un sólo carácter a través del terminal y lo
		// almacena en opción.
		opcion = usart_getchar(COMM_ROBOT_USART);

		// Función que cogerá el valor de opción y lo imprimirá por pantalla. Sirve para
		// entre otras cosas, comprobar que la comunicación entre el robot y el PC se
		// está realizando correctamente.

		delay_ms(1250); // Para hacer mas lento el refresco del terminal

		// Switch que nos enviará a una función u otra en función del valor de opción.
		// Una vez se complete la función realizada volveremos al switch y el bucle for
		// (infinito) volverá a comenzar.

		switch(opcion)
		{
		  case 'a':
			prueba_servomotor();
			break;

		  case 'b':
			prueba_motores_dc();
			break;

		  case 'e':
			modo_automata();
			break;

		  case 'f':
			modo_manual();
			break;

		  default:
			break;
		}
	}

	return 1; // Devolvemos un uno al finalizar el bucle, aunque nunca saldremos de él...
}

int configurar_usart(void)
{
	// Esta función sólo servirá para iniciar la configuración de los módulos USART. Una
	// vez completada, devolverá un valor indicando si la ejecución ha sido correcta o no.
```

```
	// Después ya podremos empezar a hacer uso de la comunicación por puerto serie

	int status_usart = -1; // Variable para guardar el valor del retorno del usart_init_rs232

	// Como hemos explicado al comienzo (en el punto de los Defines), tenemos que
	// mapear en el mapa de puertos GPIO las funciones que van a realizar los pines del
	// USART. Si no lo hacemos, por defecto funcionarán como puertos GPIO.

	static const gpio_map_t USART_GPIO_MAP =
	{
		{COMM_ROBOT_USART_RX_PIN,
		COMM_ROBOT_USART_RX_FUNCTION},
		{COMM_ROBOT_USART_TX_PIN, COMM_ROBOT_USART_TX_FUNCTION}
	};

	// En este punto se especificarán las características de las que deberá hacer uso el
	// USART. En el caso de querer realizar una comunicación con el PC, tenemos que
	// asegurarnos que nuestro terminal está configurado con los mismos parámetros.

	// Nuevamente, si queremos conocer más opciones debemos ir a las librerías y ver las
	// opciones existentes.

	static const usart_options_t USART_OPTIONS =
	{
		.baudrate     = 57600,
		.charlength   = 8,
		.paritytype   = USART_NO_PARITY,
		.stopbits     = USART_1_STOPBIT,
		.channelmode  = USART_NORMAL_CHMODE
	};

	// Habilitaremos todos los pines que sean necesarios para su uso
	gpio_enable_module( USART_GPIO_MAP, sizeof(USART_GPIO_MAP) /
	sizeof(USART_GPIO_MAP[0]) );

	// Iniciamos el USART en modo RS232. Para debug usamos las funciones print.
	status_usart = usart_init_rs232(COMM_ROBOT_USART, &USART_OPTIONS,
	FOSC0);

	return status_usart; // Devolvemos el valor del status_usart (FAIL O SUCCESS)
}

void configurar_puertos_gpio(void)
{
	// Mediante la función gpio_enable_pin_glitch_filter() estaremos habilitando el filtro anti
	// glitch en todos los puertos GPIO que creamos conveniente. Este hará que si se
	// recibe un pulso de duración menor a un ciclo de reloj, sea omitido.

	gpio_enable_pin_glitch_filter(A_PIN_MOTOR_DER);
	gpio_enable_pin_glitch_filter(B_PIN_MOTOR_DER);
	gpio_enable_pin_glitch_filter(A_PIN_MOTOR_IZQ);
	gpio_enable_pin_glitch_filter(B_PIN_MOTOR_IZQ);
	gpio_enable_pin_glitch_filter(GPIO_PUSH_BUTTON_0);
	gpio_enable_pin_glitch_filter(GPIO_PUSH_BUTTON_1);

	// Ahora se ponen a 0 (valor lógico) todos los pines que vamos a utilizar en el
	// movimiento de los motores DC, a la espera de recibir órdenes.

	gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
	gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
```

```
	gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
	gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
}

int configurar_canal_pwm_0(void)
{
	// El canal 0 será el utilizado para el funcionamiento del servomotor. Por eso, la
	// configuración de los periodos y dutycycle será distinta a la de los canales 1 y 2.

	channel_id_0 = 0; //Usamos el canal 0, definido al comienzo.

	// Habilitamos el puerto GPIO para que funcione como canal PWM.
	gpio_enable_module_pin(AVR32_PWM_0_PIN, AVR32_PWM_0_FUNCTION);

	// PWM controller configuration.
	pwm_opt_channel_0.diva = AVR32_PWM_DIVA_CLK_OFF;
	pwm_opt_channel_0.divb = AVR32_PWM_DIVB_CLK_OFF;
	pwm_opt_channel_0.prea = AVR32_PWM_PREA_MCK;
	pwm_opt_channel_0.preb = AVR32_PWM_PREB_MCK;

	// Inicializamos el canal 0 con la configuración anterior.
	pwm_init(&pwm_opt_channel_0);

	pwm_channel_0.CMR.calg = PWM_MODE_LEFT_ALIGNED;    // Channel mode.
	pwm_channel_0.CMR.cpol = PWM_POLARITY_LOW;         // Channel polarity.
	pwm_channel_0.CMR.cpd =  PWM_UPDATE_DUTY;          // No utilizado
	pwm_channel_0.CMR.cpre = AVR32_PWM_CPRE_MCK_DIV_1024;    // Prescaler.
	pwm_channel_0.cdty = 221;      // Channel duty cycle, should be < CPRD.
	pwm_channel_0.cprd = 234;      // Channel period.
	pwm_channel_0.cupd = 0;        // Channel update is not used here.

	// Para calcular los valores del periodo o del duty cycle tenemos que seguir los
	// siguientes pasos. Primero de todo tenemos que saber cual es la frecuencia que
	// deseamos. Una vez hecho, modificamos los valores en la siguiente fórmula:
	// (115200/256)/20 == 22.5Hz == (MCK/prescaler)/period, con MCK == 115200Hz,
	// prescaler == 256, period == 20.

	// Inicializa el canal 0 pero no iniciamos la señal.
	status_pwm_0 = pwm_channel_init(channel_id_0, &pwm_channel_0);

	return status_pwm_0; // Devolvemos el valor del status_pwm_0 (FAIL O SUCCESS)
}

int configurar_canal_pwm_1(void)
{
	// El funcionamiento es idéntico al del canal 0. Únicamente se variará el valor del
	// periodo y del dutycycle para dejarlo acorde a las necesidades del motor DC.

	channel_id_1 = 1; //Usamos el canal 1, definido al comienzo.

	// Habilitamos el puerto GPIO para que funcione como canal PWM.
	gpio_enable_module_pin(AVR32_PWM_1_PIN, AVR32_PWM_1_FUNCTION);

	// PWM controller configuration.
	pwm_opt_channel_1.diva = AVR32_PWM_DIVA_CLK_OFF;
	pwm_opt_channel_1.divb = AVR32_PWM_DIVB_CLK_OFF;
	pwm_opt_channel_1.prea = AVR32_PWM_PREA_MCK
	pwm_opt_channel_1.preb = AVR32_PWM_PREB_MCK;

	pwm_init(&pwm_opt_channel_1); // Inicializamos el canal 1
```

```
	pwm_channel_1.CMR.calg = PWM_MODE_LEFT_ALIGNED;    // Channel mode.
	pwm_channel_1.CMR.cpol = PWM_POLARITY_LOW;         // Channel polarity.
	pwm_channel_1.CMR.cpd =  PWM_UPDATE_DUTY;          // No utilizado
	pwm_channel_1.CMR.cpre = AVR32_PWM_CPRE_MCK_DIV_1024; // Prescaler.
	pwm_channel_1.cdty = 1000;    // Channel duty cycle, should be < CPRD.
	pwm_channel_1.cprd = 1000;    // Channel period.
	pwm_channel_1.cupd = 0;       // Channel update is not used here.

	// Inicializa el canal 1.
	status_pwm_1 = pwm_channel_init(channel_id_1, &pwm_channel_1);

	return status_pwm_1; // Devolvemos el valor del status_pwm_1 (FAIL O SUCCESS)
}

int configurar_canal_pwm_2(void)
{
	// El funcionamiento es idéntico al del canal 0. Únicamente se variará el valor del
	// periodo y del dutycycle para dejarlo acorde a las necesidades del motor DC.

	channel_id_2 = 2; //Usamos el canal 2, definido al comienzo.

	// Habilitamos el puerto GPIO para que funcione como canal PWM.
	gpio_enable_module_pin(AVR32_PWM_2_PIN, AVR32_PWM_2_FUNCTION);

	// PWM controller configuration.
	pwm_opt_channel_2.diva = AVR32_PWM_DIVA_CLK_OFF;
	pwm_opt_channel_2.divb = AVR32_PWM_DIVB_CLK_OFF;
	pwm_opt_channel_2.prea = AVR32_PWM_PREA_MCK;
	pwm_opt_channel_2.preb = AVR32_PWM_PREB_MCK;

	pwm_init(&pwm_opt_channel_2); // Inicializamos el canal 3
	pwm_channel_2.CMR.calg = PWM_MODE_LEFT_ALIGNED;    // Channel mode.
	pwm_channel_2.CMR.cpol = PWM_POLARITY_LOW;         // Channel polarity.
	pwm_channel_2.CMR.cpd =  PWM_UPDATE_DUTY;          // No utilizado
	pwm_channel_2.CMR.cpre = AVR32_PWM_CPRE_MCK_DIV_1024; // Prescaler
	pwm_channel_2.cdty = 1000;    // Channel duty cycle, should be < CPRD.
	pwm_channel_2.cprd = 1000;    // Channel period.
	pwm_channel_2.cupd = 0;       // Channel update is not used here.

	// Inicializa el canal 2.
	status_pwm_2 = pwm_channel_init(channel_id_2, &pwm_channel_2);

	return status_pwm_2; // Devolvemos el valor del status_pwm_2 (FAIL O SUCCESS)
}

void configurar_dip204(void)
{
	// Esta función será la encargada de realizar todas las configuraciones necesarias
	// para el correcto funcionamiento del display LCD. Por defecto está desactivada al
	// inicio del main para poder ahorrar energía.

	// Mapeamos todos los puertos necesarios. En el caso del LCD, hace uso de la
	// comunicación a través del protocolo SPI que funciona sobre los canales USART.

	static const gpio_map_t DIP204_SPI_GPIO_MAP =
	{
		{DIP204_SPI_SCK_PIN,  DIP204_SPI_SCK_FUNCTION },    // SPI Clock.
		{DIP204_SPI_MISO_PIN, DIP204_SPI_MISO_FUNCTION},  // MISO.
		{DIP204_SPI_MOSI_PIN, DIP204_SPI_MOSI_FUNCTION},  // MOSI.
		{DIP204_SPI_NPCS_PIN, DIP204_SPI_NPCS_FUNCTION} // Chip S. NPCS.
```

```
        };

        // El LCD hará uso de interrupciones, por lo que para poder configurarlo correctamente,
        // primero las deberemos deshabilitar y volver a arrancar.

        Disable_global_interrupt(); // Disable all interrupts.

        INTC_init_interrupts(); // Init the interrupts

        Enable_global_interrupt(); // Enable all interrupts.

        // Opciones necesarias del SPI para el funcionamiento del display DIP204.
        spi_options_t spiOptions =
        {
                .reg          = DIP204_SPI_NPCS,
                .baudrate     = 1000000,
                .bits         = 8,
                .spck_delay   = 0,
                .trans_delay  = 0,
                .stay_act     = 1,
                .spi_mode     = 0,
                .modfdis      = 1
        };

        // Asignamos todos los puertos GPIO necesarios al SPI.
        gpio_enable_module(DIP204_SPI_GPIO_MAP, sizeof(DIP204_SPI_GPIO_MAP) /
        sizeof(DIP204_SPI_GPIO_MAP[0]));

        spi_initMaster(DIP204_SPI, &spiOptions); // Inicializamos el SPI en modo maestro

        // Set selection mode: variable_ps, pcs_decode, delay
        spi_selectionMode(DIP204_SPI, 0, 0, 0);

        spi_enable(DIP204_SPI); // Habilitamos el SPI

        spi_setupChipReg(DIP204_SPI, &spiOptions, FOSC0); // Setup chip registers

        // Con esta función encendemos la pantalla LCD. Es importante tener en cuenta,
        // que el uso de esta pantalla provocará un alto coste de energía, por eso debemos
        // evitar tener encendida cuando no sea necesario.

        dip204_init(backlight_PWM, TRUE);

        // Mostraremos un mensaje por pantalla a modo de ejemplo. También se puede hacer
        // uso del LCD para realizar debug sin necesidad de conectar el robot al PC, pero como
        // ya hemos comentado, es más costoso a nivel energético.

        dip204_set_cursor_position(8,1);
        dip204_write_string("ATMEL");
        dip204_set_cursor_position(7,2);
        dip204_write_string("EVK1100");
        dip204_set_cursor_position(6,3);
        dip204_write_string("AVR32 UC3");
        dip204_set_cursor_position(3,4);
        dip204_write_string("AT32UC3A Series");
        dip204_hide_cursor();
}

void configurar_canal_adc(void)
{
```

```
	// Para la configuración de los canales ADC procederemos en primera instancia, del
	// mismo modo que para los GPIO y será habilitando el filtro anti glitch en los puertos
	// que vallamos a utilizar.

	gpio_enable_pin_glitch_filter(AVR32_PIN_PA21);
	gpio_enable_pin_glitch_filter(AVR32_PIN_PA22);
	gpio_enable_pin_glitch_filter(AVR32_PIN_PA23);

	// Mapeamos los puertos GPIO para que realicen las funciones de canales ADC
	static const gpio_map_t ADC_GPIO_MAP =
	{
	{DETECTAR_COLISION_PIN, DETECTAR_COLISION_FUNCTION},
	{DETECTAR_COLISION_PIN_1, DETECTAR_COLISION_FUNCTION_1},
	{DETECTAR_COLISION_PIN_2, DETECTAR_COLISION_FUNCTION_2},
	};

	gpio_enable_module(ADC_GPIO_MAP, sizeof(ADC_GPIO_MAP) /
	sizeof(ADC_GPIO_MAP[0]));

	// Función que configurará los parámetros necesarios de los canales ADC
	adc_configure(adc);

	// Con estas 3 funciones estaremos habilitando los 3 canales que se van a utilizar
	adc_enable(adc,adc_channel_sensor);
	adc_enable(adc,adc_channel_sensor_1);
	adc_enable(adc,adc_channel_sensor_2);

	// Hasta el momento no hemos realizado ninguna medición, se trata únicamente de la
	// configuración inicial (necesaria y suficiente), para el uso de estos canales.
}

void prueba_servomotor(void)
{
	// Esta función realizará un testeo del servomotor. Para ello hace un barrido de
	// izquierda a derecha. Servirá para comprobar si el servomotor está bien centrado y si
	// no es así, volver a colocarlo correctamente.

	// La función cambiar_canal_pwm0 parará, arrancará y configurará el canal pwm 0, con
	// el periodo que le estemos pasando a la función.

	int prueba_servo = 0;
	unsigned long duty = 200;

	pwm_start_channels(1 << channel_id_0);  // Start channel 0.

	// Creamos un bucle que hará aproximadamente 3 iteraciones, que realizará 3 barridos
	// del servomotor, actualizando la frecuencia que tiene la señal.
	while(prueba_servo < 90)
	{
	  pwm_channel_0.cupd = duty;
	  pwm_sync_update_channel(channel_id_0, &pwm_channel_0);
	  delay_ms(50);
	  duty = duty + 1;

	  prueba_servo = prueba_servo + 1;

	  if (duty==228)
	  {
		duty = 200;
		delay_ms(150);
```

```
            }
        }

        prueba_servo = 0;
        pwm_stop_channels(1 << channel_id_0);
        print_dbg("q");
}

void prueba_motores_dc(void)
{
        // Esta función hará una demostración del movimiento de las ruedas. Primero se
        // moverá hacia delante, después hacia atrás y finalmente realizará un pequeño giro.

        // Primero nos aseguramos que los canales están bien parados…
        pwm_stop_channels(1 << channel_id_1);
        pwm_stop_channels(1 << channel_id_2);

        // Para arrancarlos de Nuevo y que cojan la correcta configuración.
        pwm_start_channels(1 << channel_id_1);
        pwm_start_channels(1 << channel_id_2);

        delay_ms(100);

        // Inicialmente el valor del duty es igual al del periodo, por eso las ruedas no actúan. Si
        // queremos modificarlo, deberemos actualizarlo de la siguiente forma:
        pwm_channel_1.cupd = 600;
        pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
        pwm_channel_2.cupd = 600;
        pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

        // Movimiento de ambas ruedas hacia delante
        gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
        gpio_set_gpio_pin(B_PIN_MOTOR_DER);
        gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
        gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);

        delay_ms(1250);

        // Movimiento de ambas ruedas hacia detrás
        gpio_set_gpio_pin(A_PIN_MOTOR_DER);
        gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
        gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
        gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);

        delay_ms(1250);

        // Giro hacia la derecha
        gpio_set_gpio_pin(A_PIN_MOTOR_DER);
        gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
        gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
        gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);

        delay_ms(1250);

        // Ahora pararemos los canales PWM y además pondremos a nivel bajo las señales de
        // control del integrado L293D
        pwm_stop_channels(1 << channel_id_1);
        pwm_stop_channels(1 << channel_id_2);
        gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
        gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
```

```
	gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
	gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);

	// Enviamos una “q” para que la aplicación de control entienda que hemos llegado al
	// final de la ejecución.
	print_dbg("q");
}

void modo_manual(void)
{
	// Esta función permitirá hacer un control manual del robot, en parte, hacienda uso de
	// todas las funciones antes especificadas. Para ello, inicialmente envía un “8” al
	// programa de control, que lo entenderá como una señal para activar todos los mandos
	// de control.

	print_dbg("8"); // Escribimos a través del Puerto serie un “8”

	int variable_control = -1; // Variables utilizadas para controlar la ejecución del programa.
	unsigned long valor_duty = 600;

	// Del mismo modo que para la demo de los motores DC, inicialmente los desactivamos
	// para volver a arrancarlos después y que estos se configuren correctamente.
	pwm_stop_channels(1 << channel_id_0);
	pwm_stop_channels(1 << channel_id_1);
	pwm_stop_channels(1 << channel_id_2);

	gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
	gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
	gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
	gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

	delay_ms(200);

	// Comenzamos a emitir por los 3 canales PWM utilizados (DC + Servomotor)
	pwm_start_channels(1 << channel_id_0);  // Comienza a emitir el canal 1.
	pwm_start_channels(1 << channel_id_1);  // Comienza a emitir el canal 1.
	pwm_start_channels(1 << channel_id_2);  // Comienza a emitir el canal 2.

	// Movemos el servomotor a la posición central
	pwm_channel_0.cupd = 218;
	pwm_sync_update_channel(channel_id_0, &pwm_channel_0);

	// El siguiente bucle se encargará de recibir todas las peticiones que se realicen desde
	// la aplicación de control. No se saldrá del bucle hasta recibir una “q”.
	while (variable_control != 'q')
	{
		// Recogerá un valor a través del USART1
		variable_control = usart_getchar(COMM_ROBOT_USART);

		switch(variable_control)
		{
		  case 'a':
				// Esta función moverá las ruedas del motor hacia izq.

				// Primero de todo activaremos los PWM y después GPIO.
				pwm_channel_1.cupd = valor_duty;
				pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
				pwm_channel_2.cupd = valor_duty;
				pwm_sync_update_channel(channel_id_1, &pwm_channel_2);
```

```
                    // Con esta combinación, los motores se mueven hacia izq.
                    gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                    gpio_set_gpio_pin(B_PIN_MOTOR_DER);

                    delay_ms(1000);

                    // Paramos los motores después de moverlos durante 1 seg.
                    pwm_channel_1.cupd = 1000;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                    pwm_channel_2.cupd = 1000;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                    gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                    break;

case 'd':
                    // Esta función hace exactamente lo mismo que el caso “a” pero
                    // moverá el robot hacia la derecha.

                    pwm_channel_1.cupd = valor_duty;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                    pwm_channel_2.cupd = valor_duty;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                    gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                    gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);
                    gpio_set_gpio_pin(A_PIN_MOTOR_DER);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                    delay_ms(1000);

                    pwm_channel_1.cupd = 1000;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                    pwm_channel_2.cupd = 1000;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                    gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                    gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                    gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                    break;

case 'w':
                    // Esta función hace exactamente lo mismo que el caso “a” pero
                    // moverá el robot hacia delante.

                    pwm_channel_1.cupd = valor_duty;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                    pwm_channel_2.cupd = valor_duty;
                    pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                    gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                    gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);
```

```
                gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                gpio_set_gpio_pin(B_PIN_MOTOR_DER);

                delay_ms(1000);

                pwm_channel_1.cupd = 1000;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                pwm_channel_2.cupd = 1000;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                break;

        case 's':
                // Esta función hace exactamente lo mismo que el caso “a” pero
                // Moverá el robot hacia detrás.

                pwm_channel_1.cupd = valor_duty;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                pwm_channel_2.cupd = valor_duty;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
                gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                gpio_set_gpio_pin(A_PIN_MOTOR_DER);
                gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                delay_ms(1000);

                pwm_channel_1.cupd = 1000;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
                pwm_channel_2.cupd = 1000;
                pwm_sync_update_channel(channel_id_1, &pwm_channel_2);

                gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
                gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
                gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
                gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

                break;

        case 'j':
                // Esta función realizará una medición del sensor lateral izq.

                // Para ello, primero iniciamos el canal ADC y realizamos una medición
                // guardando el valor para enviarlo a la aplicación de control.
                adc_start(adc);
                adc_value_ir_2 = adc_get_value(adc, adc_channel_sensor_2);

                // Transmitimos la medición hacia el PC.
                print_dbg_hex(adc_value_ir_2);
                delay_ms(1500);

                break;

        case 'k':
```

```
		// Esta función realizará una medición del sensor frontal

		// Para ello, primero iniciamos el canal ADC y realizamos una medición
		// guardando el valor para enviarlo a la aplicación de control.
		adc_start(adc);
		adc_value_ir = adc_get_value(adc, adc_channel_sensor);

		// Transmitimos la medición hacia el PC.
		print_dbg_hex(adc_value_ir);
		delay_ms(1500);

		break;

	case 'l':
		// Esta función realizará una medición del sensor frontal

		// Para ello, primero iniciamos el canal ADC y realizamos una medición
		// guardando el valor para enviarlo a la aplicación de control.
		adc_start(adc);
		adc_value_ir_1 = adc_get_value(adc, adc_channel_sensor_1);

		// Transmitimos la medición hacia el PC.
		print_dbg_hex(adc_value_ir_1);
		delay_ms(1500);

		break;

	case 'z':
		// Función para mover el servomotor
		  if (pwm_channel_0.cupd > 200)
		  {
			pwm_channel_0.cupd = pwm_channel_0.cupd - 1;
			pwm_sync_update_channel(channel_id_0, &pwm_channel_0);
		  }

		  delay_ms(200);

		  break;

	case 'x':
		// Función para mover el servomotor
		  if (pwm_channel_0.cupd < 230)
		  {
			pwm_channel_0.cupd = pwm_channel_0.cupd + 1;
			pwm_sync_update_channel(channel_id_0, &pwm_channel_0);
		  }

		  delay_ms(200);

		  break;

	case 'q':
		// Función que parará todos los canales PWM y parará la ejecución del
		// modo manual.

		pwm_stop_channels(1 << channel_id_0);
		pwm_stop_channels(1 << channel_id_1);
		pwm_stop_channels(1 << channel_id_2);

		return;
```

```
                default: // Función default por si el comando que enviamos no existe.

                        break;
                }
        }
}

void modo_automata(void)
{
        // El propósito del modo autómata es dotar al robot de movimiento y "inteligencia"
        // suficiente para no topar con ningún objeto que se encuentre en su camino. Es por ello
        // que se han una serie de condiciones para que en caso de detectar una colisión sepa
        // actuar en consecuencia.

        // En el programa también se hace uso de algunas funciones específicas que han sido
        // definidas a continuación de esta función y que son necesarias para el cálculo de
        // distancias y el movimiento de las ruedas.

        int distancia_frontal_derecho = 6;
        int distancia_frontal_frente = 6;
        int distancia_frontal_izquierdo = 6;
        int distancia_lateral_izquierdo = 1;
        int distancia_lateral_derecho = 1;
        int seleccion = -1;
        int var_case = -1;
        int prueba_servo = 0;
        unsigned long duty = 200;

        pwm_start_channels(1 << channel_id_1);  // Comienza a emitir el canal 1.
        pwm_start_channels(1 << channel_id_2);  // Comienza a emitir el canal 2.
        pwm_start_channels(1 << channel_id_0);  // Comienza a emitir el canal 0.

        // Activamos el movimiento de las ruedas.
        pwm_channel_1.cupd = 600;
        pwm_sync_update_channel(channel_id_1, &pwm_channel_1);
        pwm_channel_2.cupd = 600;
        pwm_sync_update_channel(channel_id_2, &pwm_channel_2);

        // Inicialmente y antes de comenzar, realizamos un testeo del servomotor que permitirá
        // comprobar si su funcionamiento es correcto o no. El funcionamiento es el mismo que
        // el utilizado para la función prueba_servomotor().
        while(prueba_servo < 60)
        {
          pwm_channel_0.cupd = duty;
          pwm_sync_update_channel(channel_id_0, &pwm_channel_0);
          delay_ms(50);
          duty = duty + 1;
          prueba_servo = prueba_servo + 1;

          if (duty==228)
          {
                  duty = 200;
                  delay_ms(150);
          }
        }

        prueba_servo = 0;

        delay_ms(200); // Hacemos delays periódicos para dar tiempo al sistema
```

```
for(;;) // Bucle infinito para gestionar el modo autónomo.
{
        // El siguiente bucle realice un barrido del servomotor, parándose en tres
        // posiciones para realizar medidas con los sensors infrarrojos.
        while(duty <= 226)
        {
          pwm_channel_0.cupd = duty;
          pwm_sync_update_channel(channel_id_0, &pwm_channel_0);

          delay_ms(50);

          duty = duty + 1;
          prueba_servo = prueba_servo + 1;

          // Estos “if” llamarán a la función comprobación_frontal() en los tres casos
          if (duty==200)
          {
                distancia_frontal_derecho = comprobacion_frontal();
                delay_ms(50);
          }
          if (duty==219)
          {
                distancia_frontal_frente = comprobacion_frontal();
                delay_ms(50);
          }
          if (duty==226)
          {
                distancia_frontal_izquierdo = comprobacion_frontal();
                delay_ms(50);
          }
        }

        duty = 200; // Iniciamos la variable duty para la siguiente comprobación

        // Comprobación del sensor lateral derecho
        distancia_lateral_derecho = comprobacion_lateral_derecho();
        delay_ms(50);

        // Comprobación del sensor lateral izquierdo
        distancia_lateral_izquierdo = comprobacion_lateral_izquierdo();
        delay_ms(50);

        // Generamos unas condiciones que darán prioridad a los sensores laterales
        // frente al sensor delantero.
        if ((distancia_lateral_derecho == 1)&&(distancia_lateral_izquierdo == 1))
                var_case = 1;
        else if (distancia_lateral_derecho == 1)
                var_case = 3;
        else if (distancia_lateral_izquierdo == 1)
                var_case = 2;
        else if (distancia_frontal_frente == 10)
                var_case = 1;
        else if (distancia_frontal_izquierdo == 10)
                var_case = 2;
        else if (distancia_frontal_derecho == 10)
                var_case = 3;
        else
                var_case = -1;
```

```
// El siguiente switch podrá hacer 4 cosas, mover el robot hacia delante, detrás,
// izquierda o derecha en función de las condiciones antes seleccionadas
switch(var_case)
{
	case 1: // Obstáculo delante

		gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
		gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
		gpio_set_gpio_pin(A_PIN_MOTOR_DER);
		gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

		delay_ms(750);
		seleccion = rand() % 2;

		// Aleatoriamente decidimos que camino seguir
		// izquierda o derecha

		if (seleccion == 0)
		{
			// Giro hacia la izquierda
			gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
			gpio_set_gpio_pin(B_PIN_MOTOR_DER);

			delay_ms(500);

			// Paramos las ruedas
			gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
			gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
		}
		else if (seleccion == 1)
		{
			// Giro hacia la derecha
			gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
			gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);
			gpio_set_gpio_pin(A_PIN_MOTOR_DER);
			gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

			delay_ms(500);

			// Paramos las ruedas
			gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
			gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
			gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
		}

		break;

	case 2: // Obstáculo izquierdo

		// Movemos hacia atrás
		gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
		gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
		gpio_set_gpio_pin(A_PIN_MOTOR_DER);
		gpio_clr_gpio_pin(B_PIN_MOTOR_DER);
```

```
            delay_ms(750);

            // Giro hacia la derecha
            gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_set_gpio_pin(A_PIN_MOTOR_DER);
            gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

            delay_ms(500);

            // Paramos las ruedas
            gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
            gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

            break;

        case 3: // Obstáculo derecho

            // Movemos hacia atrás
            gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_set_gpio_pin(A_PIN_MOTOR_DER);
            gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

            delay_ms(750);

            // Giro hacia la izquierda
            gpio_set_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
            gpio_set_gpio_pin(B_PIN_MOTOR_DER);

            delay_ms(500);

            // Paramos las ruedas
            gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
            gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

            break;

        default:
            // Movimiento hacia delante.
            gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_set_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
            gpio_set_gpio_pin(B_PIN_MOTOR_DER);

            delay_ms(750);

            // Paramos las ruedas
            gpio_clr_gpio_pin(A_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(B_PIN_MOTOR_IZQ);
            gpio_clr_gpio_pin(A_PIN_MOTOR_DER);
            gpio_clr_gpio_pin(B_PIN_MOTOR_DER);

            break;
```

```
                }
        }
}

int comprobacion_frontal(void)
{       // Esta función lanzará una medición para el sensor frontal.
        delay_ms(200);
        adc_start(adc); // Inicializa el módulo ADC
        adc_value_ir = adc_get_value(adc, adc_channel_sensor); // Toma la medida

        if (adc_value_ir >= 0x260) // Si el valor es mayor que uno prefijado, enviamos un 1
                return 1; // Este uno indica que se ha encontrado un obstáculo
        else
                return -1;
}

int comprobacion_lateral_derecho(void)
{       // Esta función lanzará una medición para el sensor lateral derecho.
        adc_start(adc); // Inicializa el módulo ADC
        adc_value_ir_2 = adc_get_value(adc, adc_channel_sensor_2); // Toma la medida

        if (adc_value_ir_2 < 0x200) // Si el valor es mayor que uno prefijado, enviamos un 1
                return 1; // Este uno indica que se ha encontrado un obstáculo
        else
                return -1;
}

int comprobacion_lateral_izquierdo(void)
{       // Esta función lanzará una medición para el sensor lateral izquierdo.
        adc_start(adc); // Inicializa el módulo ADC
        adc_value_ir_1 = adc_get_value(adc, adc_channel_sensor_1); // Toma la medida

        if (adc_value_ir_1 < 0x200) // Si el valor es mayor que uno prefijado, enviamos un 1
                return 1; // Este uno indica que se ha encontrado un obstáculo
        else
                return -1;
}
```