DESARROLLO DE UNA LIBRERÍA DE FUNCIONES PARA SISTEMAS AUTÓNOMOS

Tutor: D. Jorge Chávez Orzáez

Autor: Juan Antonio Quintero del Toro

Escuela Superior de Ingenieros de Sevilla

ÍNDICE

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

AGRADECIMIENTOS

A mi tutor Jorge, por enseñarme el camino correcto que a veces no quería ver. Por los mil y un brindis que nos quedan.

A mi familia, por permitir que me labre un futuro sobre sus bases, y a mi abuelo, que me tuvo que dejar por el camino.

A la familia Feria por adoptarme y rezar por mí.

A mi otra familia, que se sienten y me hacen sentir como si lo fueran de verdad. Algunos viven conmigo y otros casi.

A los profesores que me han formado en cada uno de los eslabones de mi educación. En especial aquellos que han confiado en mí sin restricciones.

A Juan Alberto, por darme la oportunidad de valerme por mí mismo, y a David Canca, por contar conmigo para sus proyectos y hacerme ver el futuro un poco más claro.

A todos los que se me ha olvidado citar.

Dedicado a Eugenia.

Para siempre.

1 INTRODUCCIÓN A LOS SISTEMAS AUTÓNOMOS

Un sistema autónomo (a veces llamados “sistemas embebidos”, debido a una heterodoxa traducción de “embedded system”) consiste en un sistema compuesto por un hardware y, en ocasiones, un software, que desarrollan una función de forma independiente por sí mismos, sin necesidad de interactuar con ningún otro sistema para desarrollar su funcionalidad. Esto no quiere decir que no pueda comunicarse con otros sistemas para lectura o escritura de datos y variables, o que no pueda aplicar su funcionalidad o recibirla desde otro medio.

Un sistema autónomo, desde el punto de visto de la microelectrónica, se caracteriza por una unión hardware/software tal que uno de los factores pierde su sentido y funcionalidad sin el otro, ya que el código de la aplicación se encuentra “incrustado” en algún tipo de memoria no volátil dentro del conjunto o bien implementada mediante microprogramación, puertas lógicas, etc. Como ejemplo, podemos pensar en un reproductor mp3. Este dispositivo funciona por sí solo, ofreciéndonos el resultado de su funcionalidad, pero puede interactuar con un PC para cargar nuevas canciones o actualizar su firmware, lo que no se encuentra en contradicción con su condición autónoma. Además, puede recibir alimentación a través de la red eléctrica sin perjuicio de lo anteriormente comentado.

El auge de la tecnología dentro de la sociedad de la información y la comunicación ha dado lugar a numerosos avances que repercuten directamente en nuestra comodidad o seguridad, y los sistemas autónomos desempeñan un papel fundamental dentro de éste desarrollo. En la actualidad, teléfonos móviles, agendas personales electrónicas, controladores domóticos, sistemas de control de procesos y un largo etcétera son sistemas autónomos que nos facilitan numerosos quehaceres de nuestra vida diaria. También podemos ver reflejados estos avances en los robots capaces de desarrollar tareas peligrosas para el ser humano, ciertas máquinas que desarrollan tareas por sí mismas dentro de cadenas de producción, etc.

Este espectacular crecimiento ha desembocado, entre otras cosas, en la creación de arquitecturas estándares para las unidades centrales de proceso de los microprocesadores. De ésta forma, a partir de una funcionalidad tipo y de un juego de instrucciones básico estándar, cada una de las empresas de semiconductores puede desarrollar sus propios avances y mejorar los núcleos ofreciendo una amplia gama de dispositivos con periféricos y facilidades diferentes, pero que comparten un juego de

instrucciones, lo que facilita su programación y permite desarrollar para diferentes sistemas sin más que portar ligeramente el código generado. Éste es el caso de los microprocesadores basados en (o emulando a) núcleos 80C51, arquitectura ahora abierta de enorme simplicidad y realmente económica, que ya ofrece cientos de versiones del mismo de una enorme variedad de fabricantes.

También se han desarrollado arquitecturas que, sin ser abiertas y perteneciendo a un solo fabricante, debido a la simplicidad de su programación, los periféricos capaces de soportar e integrar en la misma pastilla, o bien las herramientas que la empresa comercializadora pone a nuestra disposición para interaccionar con el núcleo, se han ganado el beneplácito de los desarrolladores, como los microcontroladores de la serie 68xxx de Motorola, o la serie 16xxx de PIC.

Ilustración 1. Intel P8051[1]

Para aplicaciones de procesos rápidos en tiempo real, para grandes manejos de datos y cualquier aplicación especialmente sensible a la potencia de cálculo y a la velocidad, se dispone de procesadores digitales de señales (DSP), que aportan arquitecturas capaces de soportar micros en paralelo, con núcleos internos especialmente rápidos en las operaciones con números flotantes, debido a la capacidad de ejecutar varias instrucciones en paralelo. Texas Instruments y Motorola son fabricantes destacados de este tipo de máquinas.

En definitiva, numerosas son las opciones de las que disponemos atendiendo a las características de la aplicación que queramos desarrollar, la inversión del proyecto o, simplemente, la facilidad de interactuar con el microcontrolador.

[1] Imagen extraída de www.cpu-world.com

Una de las peculiaridades que nos pueden llevar a decidirnos por una u otra arquitectura es la disponibilidad de herramientas que faciliten la elaboración, desarrollo, depuración y posterior mantenimiento del código. En este sentido, si los requerimientos de potencia o velocidad son laxos, el uso de una arquitectura abierta como la 80C51 facilita en gran medida nuestro cometido. La estandarización del núcleo y del juego de instrucciones favorece el florecimiento de herramientas que, bajo el citado marco común, incluyen compiladores, ensambladores y traductores de y hacia otros lenguajes de alto nivel y ensambladores, y simuladores de núcleo y periféricos que permiten depurar código con facilidad.

2 OBJETIVO

El objeto de este proyecto fin de carrera es la realización de un conjunto de funciones que, enmarcadas dentro de una librería, permita comunicar un microcontrolador, que actúe como parte de un sistema autónomo, con algunos de los periféricos más comunes, y permitan controlar comunicaciones a través de diferentes interfaces y usando distintos protocolos. De esta forma se pretende facilitar la labor del programador en posteriores aplicaciones autosuficientes que requieran del uso de las funciones aquí descritas.

Usando estas funciones se realizó, a modo de caso de estudio, un ejemplo de funcionamiento de un sistema autónomo capaz de interactuar con un usuario y actuar sobre otro sistema genérico.

Con tal fin, se estratificaron las funciones de las librerías, de forma que se presentan al programador funciones con interfaces cómodas y que no dependen, en la medida de lo posible, del medio físico ni del dispositivo microcontrolador. Los casos en los que no hubo más remedio que depender de ellos se han resaltado explícitamente, y se ha informado de cómo ha de ser interpretado y, posteriormente, modificado el código del programa. Con esto se intentó obtener la máxima portabilidad posible, ofreciendo al programador la posibilidad de poder copiar la mayor parte del código y sólo tener que realizar ligeras modificaciones.

Atendiendo a esta división, se presentarán las funciones en este documento según accedan directamente al medio físico o sean independientes del mismo y sólo funcionen para el micro C51 del fabricante que hemos seleccionado.

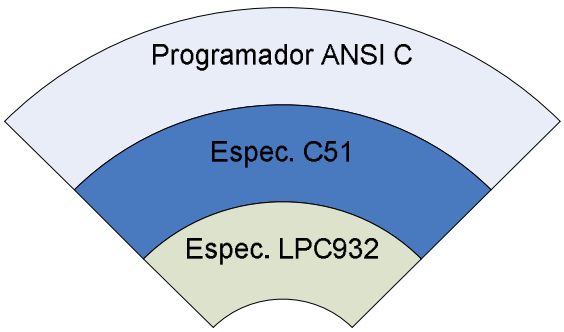

Ilustración 2. División por capas del software

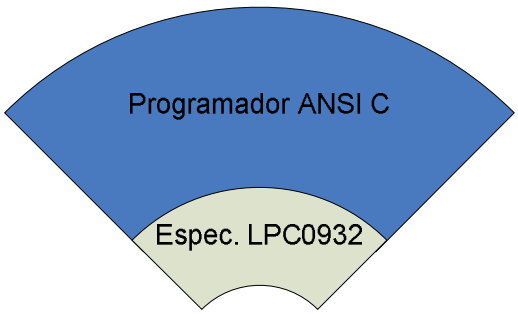

Ilustración 3. Acceso directo al hardware

En cuanto a la personalización del software para un dispositivo en concreto, se asumió que las funciones estaban implementadas sobre el micro P89LPC932 de Philips (hay un capítulo completo dedicado a este tema). De esta forma, las direcciones de los periféricos, puertos, etc., están referidas a la propia definición de los mismos dentro del citado micro.

Este microcontrolador se caracteriza, en principio, por ofrecer una solución muy económica para las aplicaciones en las que su uso sea recomendable, ya que se trata de un micro de muy bajo coste y que ofrece una más que suficiente configurabilidad y versatilidad.

El núcleo del mismo se basa en una implementación mejorada de una unidad central 80C51. Puesto que la arquitectura es libre, y cada fabricante de semiconductores la mejora, amplia e implementa libremente, este micro funciona al séxtuple de la velocidad del núcleo original.

Este microcontrolador se encuentra integrado en una pastilla en la que se incluyen numerosos periféricos que son de gran utilidad y que disminuyen sensiblemente el coste de las aplicaciones a realizar, amén de su ahorro en diseño, placa y cableado, ya que apenas necesitaremos agregar dispositivos a nuestro sistema. De entre estos periféricos podemos destacar una UART mejorada, un bus I2C, un bus SPI, puertos paralelo, capturadores de entradas, etc.

3 BREVE INTRODUCCIÓN A LA FAMILIA DE MICROPROCESADORES 80C51

Los micros C51 son unos microcontroladores de ocho bits desarrollados originalmente por Intel en 1980. Se trata, sin duda, del núcleo controlador más usado en todo el mundo, y un gran número de fabricantes han desarrollado sus propias versiones mejoradas del núcleo, por lo que actualmente se pueden encontrar en el mercado núcleos con representativas mejoras, tanto en la potencia de la unidad central de proceso, como en los periféricos que incluye.

La arquitectura original incluye, como se muestra en la siguiente ilustración, incluye:

- Una unidad central de proceso con un procesador booleano
- 6 fuentes de interrupción externa
- Temporizadores/contadores de 16 bits
- Puerto de comunicación serie con velocidad programable
- Ram de datos interna
- Memoria de programa, ROM, EPROM, EEPROM, FLASH

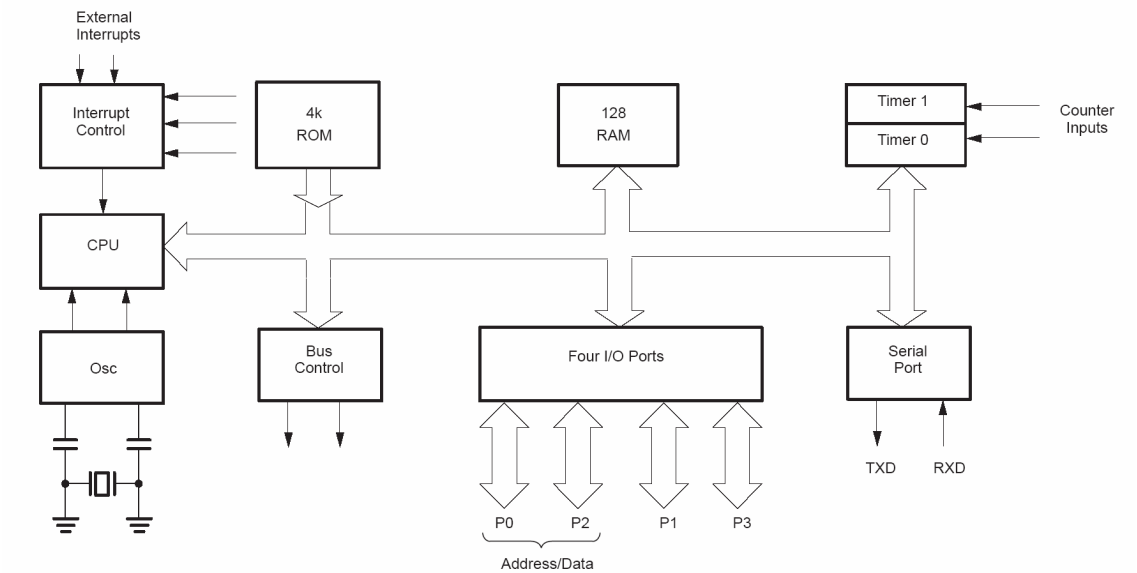

Ilustración 4. Arquitectura C51[1]

[1] Extraído del manual de Intel *MCs-51 Microcontroller Family User Manual*

El micro puede leer el programa bien desde la memoria externa, bien desde algún dispositivo de almacenamiento exterior al mismo. En cuanto al espacio de memoria a los que puede acceder el micro, nos encontramos con el siguiente diagrama:

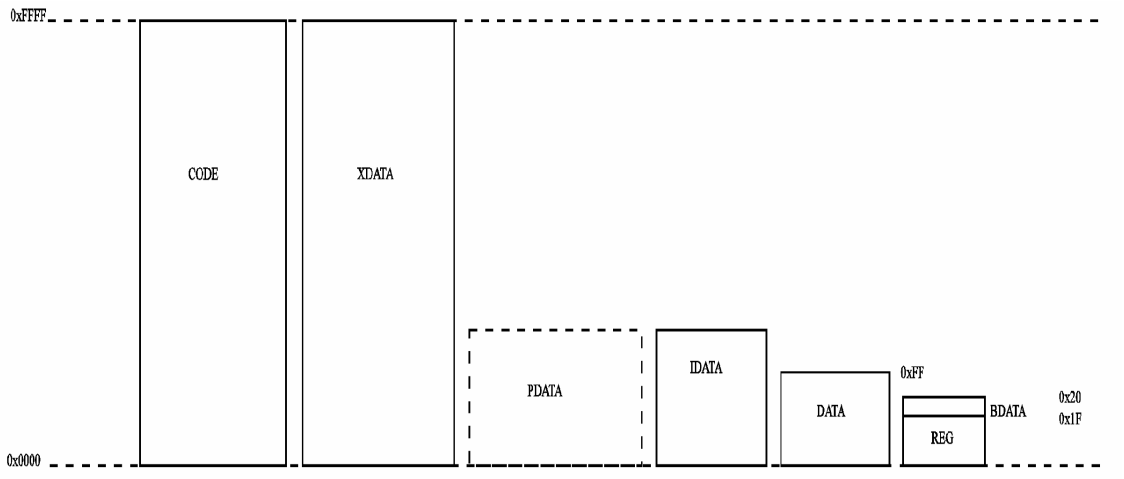

Ilustración 5. Espacio de memoria del C51

Puesto que diferentes tipos y trozos de memoria están solapados en un mismo espacio de direcciones, el micro dispone de instrucciones específicas, las cuales activan señales internas que se encargan de diferenciar los distintos tipos de acceso. Por ejemplo, a la clásica instrucción de ensamblador, MOV, se le añade MOVX, para acceder a la memoria externa; para la memoria de código, puesto que sólo es accesible desde el contador de programa, éste se encarga de colocar la señal oportuna que asegura una lectura correcta; etc.

Se definen modos de memoria en función de la zona a la que se va a asignar la memoria por defecto. Este proceso se explicará con detalle en el capítulo dedicado al compilador. La descripción detallada de todas las zonas de memoria, con sus limitaciones y posibles formas de acceso van a ser comentadas con cierto nivel de detalle, pero se recomienda la lectura de uno de los múltiples y gratuitos manuales de arquitectura 8051, que profundizan en totalidad y determinan las características exactas para cada fabricante.

En estos manuales se puede encontrar igualmente una descripción detalla de todos los periféricos que vienen incorporados en los núcleos de los microcontroladores tanto en general, como las modificaciones de cada marca comercial. No aporta nada a nuestra labor citar y parafrasear aquí cientos de páginas cuya información es gratuita y fácilmente disponible. En cualquier caso, las diferencias entre la arquitectura original

estándar y el microcontrolador elegido para la personalización del software van a ser descritas en sucesivos capítulos. Por último, destacar que todos los documentos relacionados en la bibliografía son gratuitos, y están a disposición de los visitantes en las páginas Web de los fabricantes citados.

4 MICROPROCESADOR PHILIPS P89LPC932

Como se ha comentado en el capítulo segundo, el desarrollo del nivel más bajo de las funciones de la librería se han implementado y personalizado para los microcontroladores de la serie P89LPC9xx de *Philips Semiconductors.*

Los micros de esta serie se caracterizan por una alta integración de periféricos en una pastilla en relación a su reducido precio. Estas características lo hacen ideal para la implementación de sistemas autónomos de reducidas dimensiones y bajo coste.

Los siguientes apartados desarrollan en mayor profundidad los periféricos y características del micro más sobresalientes que va a ser necesario conocer para la implementación de la librería.

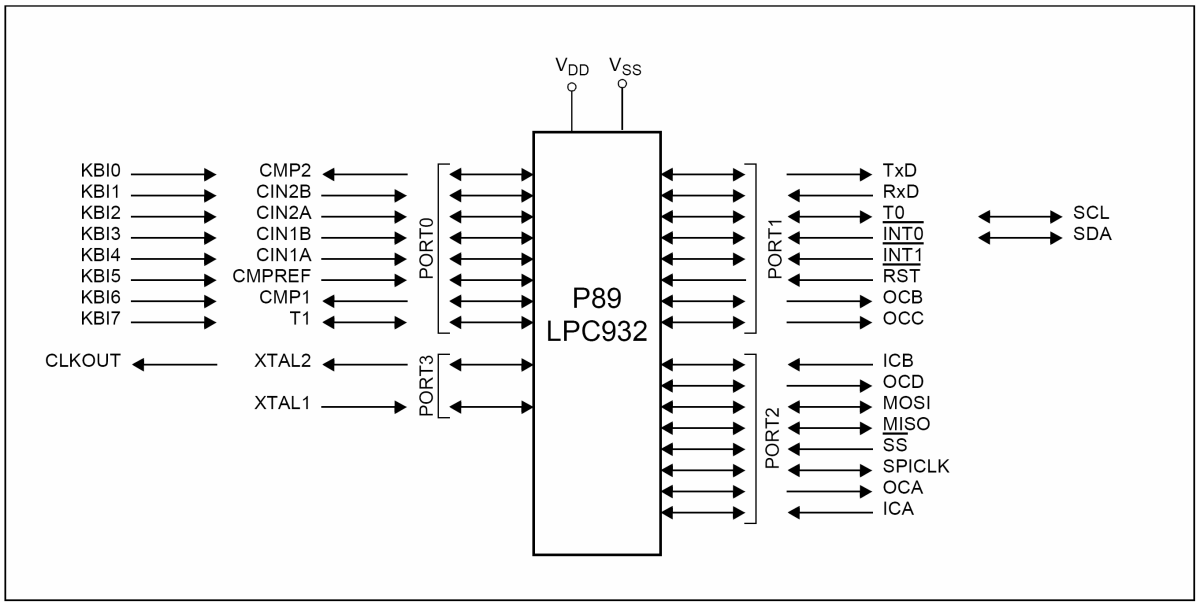

Ilustración 6. Descripción funcional de los puertos del microcontrolador[1]

4.1 Núcleo del microcontrolador

El microchip está diseñado a partir de la mejora de una unidad central de procesos 80C51, cuya arquitectura es libre y estándar, con lo que cada fabricante de microelectrónica puede realizar nuevas implementaciones que aporten avances tanto en su velocidad y potencia así como en su funcionalidad. El desarrollo de los nuevos núcleos debe ir parejo al respeto al juego de instrucciones original básico y a la funcionalidad primigenia, con lo que se favorece el nacimiento de herramientas que,

[1] Imagen extraída de *Philips P89LPC932 User Manual*

bajo el auspicio del citado marco, permiten simplificar y facilitar el desarrollo de aplicaciones basadas en chips que presenten dicha arquitectura.

Bajo el respeto al estándar que acabamos de citar, *Philips* desarrolla esta línea de microcontroladores, que multiplican por seis la velocidad del reloj del sistema y, por extensión, la velocidad del ciclo máquina de la arquitectura original, permitiendo al micro adaptarse a la mayoría de las necesidades de potencia que requieren las aplicaciones actuales.

El origen del reloj del sistema puede ser, bien externo, aceptando cristales o señales de reloj de hasta doce Mhz, o bien interno, debido a que posee dos osciladores integrados en la propia pastilla de silicio. Uno de ellos es el oscilador asociado al *Watchdog*, que corre a 400Khz, mientras que un segundo funciona a 7.3728 Mhz. El hecho de que esté integrado, con el ahorro de rutado y circuitería externa que esto conlleva, así como su idóneo periodo de oscilación, hacen que este último sea la opción elegida para la implementación de nuestro escenario y, en general, lo hacen adecuado para un gran volumen de aplicaciones y problemas cotidianos.

Todas las instrucciones del micro se ejecutan en uno o dos ciclos máquina, lo que equivale a dos o cuatro ciclos del reloj del sistema.

El código se carga dentro de los 8Kbytes de memoria flash interna, divididos en ocho sectores lógicos de 1Kbyte, y además se ponen a nuestra disposición 512 bytes de RAM. Una parte de estos bytes sólo es direccionable de forma indirecta, por lo que es perfectamente apta para alojar la pila de programa, de forma que podamos pasar parámetros a funciones y almacenar valores.

El último de los ocho sectores de la memoria flash contiene un programa de carga que permite reprogramar, siempre que esta opción se habilite por software o hardware, el microcontrolador dentro del mismo circuito e incluso dentro de la misma aplicación. Por tanto, no recomendamos la reprogramación de este último sector, puesto que, en caso contrario, sólo sería posible reprogramar el micro a través de un programador paralelo comercial de *Philips* siempre fuera del sistema.

4.2 UART

Tendremos a nuestra disposición, para las comunicaciones en serie, una UART mejorada, compatible con, aunque no igual a, las UART convencionales de cualquier otro microcontrolador basado en la arquitectura 80C51.

La principal diferencia con los dispositivos de los núcleos 80C51 tradicionales es que no está permitido usar el desborde de uno de los temporizadores del sistema (el marcado como TIMER2) con generador de baudios para las comunicaciones serie. Sin embargo, como contraprestación, podremos seleccionar numerosas fuentes generadores de señales.

El microcontrolador nos permite generarlas a partir del reloj del sistema, dándonos la posibilidad de dividir su ciclo por un valor constante, usar el desborde de uno de los temporizadores del sistema (TIMER1) o bien, quizás la opción que permite una mayor configurabilidad, es usar el propio generador de tasa de baudios (Baud Rate Generator) independiente. Con esta última opción, es posible programar dos registros de ocho bits que servirán como divisor del reloj del sistema, generando la señal a la tasa deseada.

La UART del micro puede operar en cuatro modos, y en todos ellos le transmisión comienza escribiendo en un registro especial del sistema, que funciona como buffer de transmisión. En todos ellos salvo en el primero, la recepción se inicia por la llegada del primer bit siempre y cuando esté habilitada la recepción a través de la UART, pero para el modo cero, es también necesario bajar una bandera adicional en la configuración. En cualquier caso, se pueden asociar interrupciones tanto a un evento como al otro.

- Modo 0: Todos los datos entran y salen por el Terminal de recepción (RxD). Por TxD sólo se envía de forma permanente el reloj del sistema. Se envían y reciben siempre ocho bits, y la velocidad esta fijada a un dieciseisavo del reloj de la CPU.
- Modo 1: Se envían o reciben diez bits, ya que al byte de información se le añaden el bit de arranque y el de parada. Cuando se actúa de receptor, el bit de parada se almacena en un uno de los bits de un registro de configuración, pudiendo generar, si así se configura, interrupción. En cuanto a la tasa de baudios, esta puede venir determinada por el desborde del contador TIMER1 o bien por el generador de baudios previamente programado, que divide el reloj del sistema por un valor de dieciséis bits.
- Modo 2: Se transmiten once bits, ya que a la trama se añade un noveno bit de información programable. Puede funcionar, y habitualmente esa es su función, como bit de paridad. En este caso el bit que se almacena en el registro de configuración no es el bit de parada sino el citado noveno bit. La tasa de baudios

puede generarse a partir de un dieciseisavo o un treintaidosavo del reloj del sistema.

- Modo 3: Este modo es básicamente el modo 2, salvo en lo que respecta a la tasa de baudios. En este caso no se fija a partir de una fracción determinada del reloj, sino que, como en el modo 1, se puede generar a partir del evento generado por el desborde del contador TIMER1 o bien programando el generador de baudios.

En la mayoría de las aplicaciones se usa el modo 1, ya que a las velocidades a las que se suele transmitir, y las condiciones en las que esta transmisión es llevada a cabo, la tasa de error hace innecesario un bit de paridad. Además, la configurabilidad del generador de baudios permite ajustar la transmisión para cada uno de los requerimientos particulares de cada problema, incluso variar la tasa dentro de la misma aplicación.

4.3 Puertos I/O paralelo

El microcontrolador está dotado de cuatro puertos de entrada/salida paralelo integrados en la misma pastilla. De estos cuatros, tres son de ocho bits y uno sólo tiene 2 bits. Sin embargo, a pesar de que pueda parecer un número suficientemente alto de puertos de comunicaciones, en éste, como en la mayoría de microcontroladores, casi todos los pines tienen funciones alternativas, relacionadas con los periféricos que se incorporan en el mismo microchip.

De esta forma, los puertos i/o paralelo deben ser los últimos en ser seleccionados, puesto que necesitamos saber, antes de otorgarles un espacio determinado, cuáles de los periféricos que aporta el micro vamos a utilizar, y así reservar los pines necesarios. También puede darse el caso contrario, y necesitar demasiados periféricos y no dar espacio a ningún puerto paralelo, en cuyo caso deberemos adquirir periféricos externos, reservar un puerto, y acceder a ellos mapeándolos convenientemente.

Los puertos de este micro se pueden configurar de cuatro formas diferentes, tal y como se advierte en el siguiente esquema extraído del manual de usuario del micro:

PxM1.y	PxM2.y	Port output mode
0	0	Quasi-bidirectional
0	1	Push-Pull
1	0	Input Only (High Impedance)
1	1	Open Drain

Tabla 1. Configuración del sentido de los pines[1]

De entre estas cuatro configuraciones, dos van a ser las que se recomienden debido a su versatilidad: *Quasi-bidirectional* para pines de entrada o entrada salida, y *Push-pull* para pines de salida.

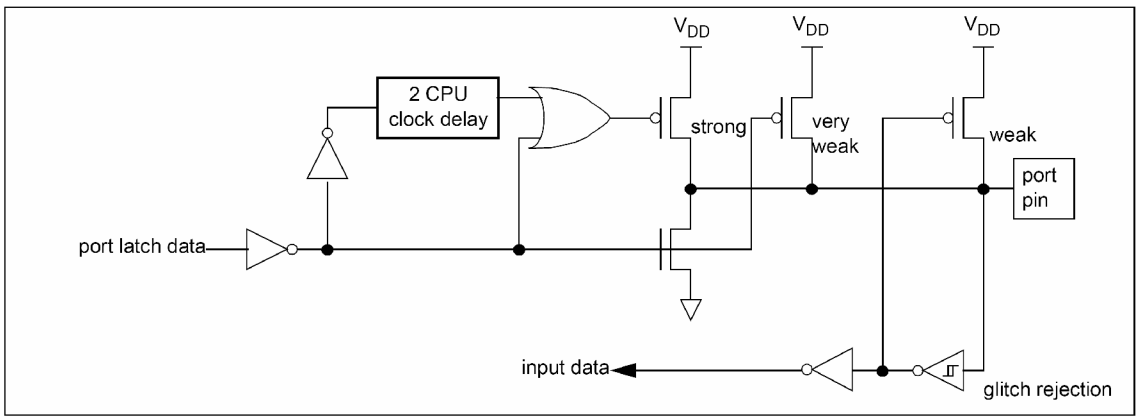

Ilustración 7. Configuración Quasi-bidirectional[2]

La ilustración ¿?¿?¿? nos muestra el esquema lógico de los pines configurados como *Quasi-bidirectional*, que permiten actuar a los pines como entradas y salidas de datos sin necesidad de reconfigurar el puerto cada vez que se necesite leer o escribir del mismo. Para conseguir esto, el circuito se encarga de cargar débilmente (*weakly driven*) las salidas lógicas a nivel alto, de forma que cualquier dispositivo externo es capaz de forzar un nivel bajo en el mismo pin. En el caso en el que el micro ponga un cero en el puerto, este va a ser cargado fuertemente, de forma que pueda bajar el nivel que cualquier dispositivo intente forzar en el puerto.

El transistor marcado como “muy débil” conduce cuando se desde el micro se escribe un nivel alto, y funciona a modo de *pull-up* si en el puerto se encuentra flotante.

[1] Imagen extraída de *Philips P89LPC932 User Manual*

[2] Ídem

El transistor nombrado “débil” se activa cuando en el *port-lach* tenemos un nivel alto y el pin ya tiene un nivel alto también. Si el pin es puesto a nivel bajo por algún dispositivo externo, el transistor deja de conducir, y sólo el transistor “*muy débil*” conduce. De esta forma, el dispositivo externo tiene que aportar la suficiente energía para vencer esta débil barrera y forzar un cero en el pin.

En cuanto al transistor “fuerte”, cuando el *port-latch* cambia de cero a uno, el transistor se enciende por dos ciclos de reloj, forzando un rápido cambio de nivel de la salida, y agilizando sus características dinámicas.

Otros aspectos importantes son el circuito de supresión de *glitch* cuando el puerto funciona como entrada, ya que evita transitorios indeseados; y además cabe destacar que, a pesar de trabajar con lógica de tres voltios, el puerto es capaz de recoger señales de cinco voltios. El único inconveniente es que causa un consumo extra de energía, ya que el exceso de voltaje es conducido hacia la fuente Vdd por los transistores.

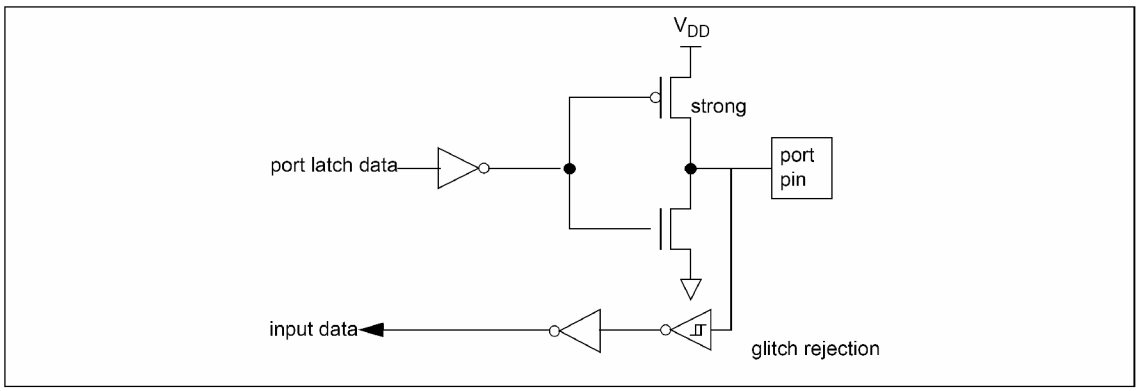

Ilustración 8. Configuración Push-pull[1]

La configuración *Push-pull* se recomienda cuando necesitamos configurar los pines como pines de salida. La estructura es similar al circuito *Quasi-bidirectional*, salvo que es capaz de forzar un nivel alto “fuerte” cuando en el *port-latch* tengamos un uno lógico escrito por el micro. En cualquier caso, el micro también podría recibir datos desde el exterior, ya que podemos acceder a el mediante una entrada con un comparador Schmitt-trigger y un circuito supresor de *glitch*.

El resto de configuraciones disponibles son menos usadas puesto que son menos versátiles. Se puede definir el puerto como sólo de entrada, pero si necesitáramos

[1] Imagen extraída de *Philips P89LPC932 User Manual*

escribir necesitaríamos modificar la configuración de los registros de configuración, por eso se suelen definir bidireccionales en estos casos. . Por otra parte, la salida a drenador abierto, que es la última de las opciones disponibles, puede tener interés en el caso de que quisiéramos atacar otro sistema a la salida del puerto y hacer uso de, por ejemplo, lógica cableada. En cualquier caso, cada uno de los pines del puerto es reconfigurable instantáneamente de forma individual e incluso dentro de la misma aplicación, sin más que variar el valor de un registro en concreto.

4.4 Temporización

El sistema microcontrolador posee dos contadores/temporizadores que son compatibles con los clásicos contadores TIMER0 y TIMER1 de un núcleo 80C51 clásico.

Ambos pueden ser configurados como temporizadores, actualizándose cada ciclo máquina, o como contadores de eventos, detectando un flanco en uno de dos pines externos dispuestos para tal fin. Estos pines son tanteados una vez cada ciclo máquina, y se chequea un cambio de valor entre cada ciclo, es decir, si en un ciclo máquina el pin se encuentra a valor alto, y en el siguiente período se encuentra a nivel bajo, la transición es detectada y el contador de eventos se actualiza, aunque esto será en el siguiente ciclo. Puesto que se necesitan dos ciclos máquina, que equivalen a cuatro períodos de reloj, para detectar un cambio de valor en el pin, la máxima frecuencia de captura de eventos es un cuarto de la frecuencia de reloj del sistema.

En cuanto a las diferentes formas de configurar cada uno de los contadores, se pueden disponer de hasta cinco formas diferentes, combinando la longitud de los registros que almacenan el calor de la cuenta, e incluso dividiendo el registro en dos para obtener dos cuentas simultáneas.

Por último, también podemos disponer de un sistema de cuenta en tiempo real, de forma que, a pesar de que el dispositivo se encuentre en cualquiera de los modos de ahorro de energía, el contador seguirá ejerciendo su función, y sólo será reseteado con una señal de arranque (*Power on Reset*).

Este contador en tiempo real puede usarse, dentro del código cargado en el micro, como fuente de interrupciones, al igual que cualquiera de los demás contadores; sin embargo, la característica que lo hace diferente del resto es que, mientras cuenta cuando el resto del dispositivo está apagado, puede generar señales que hagan salir al núcleo de cualquiera de los modos de ahorro de energía y “despierten” el sistema.

5 DISPOSITIVO LCD ESTÁNDAR KS0066 (HD44780)

5.1 Descripción

Las funciones de biblioteca se personalizarán para los displays lcd basados en el controlador HD44780, que es un dispositivo CMOS estándar que permite no sólo leer y escribir en la matriz de puntos, sino que además proporciona un reducido pero potente juego de instrucciones de control. La mayoría de las matrices lcd disponibles en el mercado son compartibles o bien integran este tipo de instrucciones.

En general, los dispositivos basados en este tipo de controladores pueden representar hasta 189 caracteres y símbolos diferentes, 8 de los cuales son definibles por el usuario, y son totalmente compatibles con el estándar de representación ASCII, lo que los hace óptimos para aplicaciones de terminales de datos, instrumentos médicos, cualquier tipo de instrumento portátil de medida, y en general, cualquier aplicación que, de forma compacta y ocupando el menor espacio posible, necesite mostrar información o interactuar con un usuario. Además, el desarrollo de las tecnologías del cristal líquido pone a nuestra disposición instrumentos cada vez más pequeños, con una relación de contraste bastante alta y con ángulos de visión cada vez mayores.

5.2 Interfaz de comunicaciones

En general, se van a disponer de dos tipos de comunicación entre el sistema microcontrolador, que actuará siempre como maestro, y el controlador del display. Se usarán siempre tres señales de control para gobernar el protocolo de comunicación y los datos podrán presentarse en formato de char (8 bits) o bien en dos paquetes de cuatro bits, comúnmente llamados *Nibbles*. Ésta última opción es la más común cuando se trabaja con sistemas autónomos, ya que ahorra espacio de cableado y libera cuatro pines del microcontrolador, que, tal y como se comentó en anteriores capítulos, puede ser un recurso crítico en función de los componentes que necesitemos integrar.

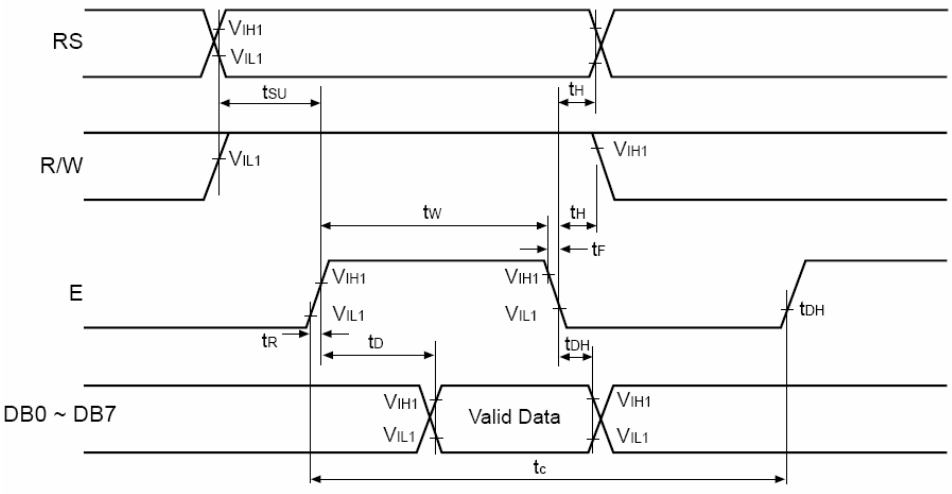

Ilustración 9. Cronograma de lectura[1]

Los ciclos de lectura y escritura son, por lo general idénticos al que mostramos a continuación, perteneciente a un 329-0341 de *Theale Components*. Estos diagramas están representados para un interfaz de ocho bits, aunque, puesto que los bits de información se escriben y leen en paralelo, es idéntico al cronograma para cuatro bits. En éste último caso, se necesitaría escribir o leer en dos ciclos consecutivos.

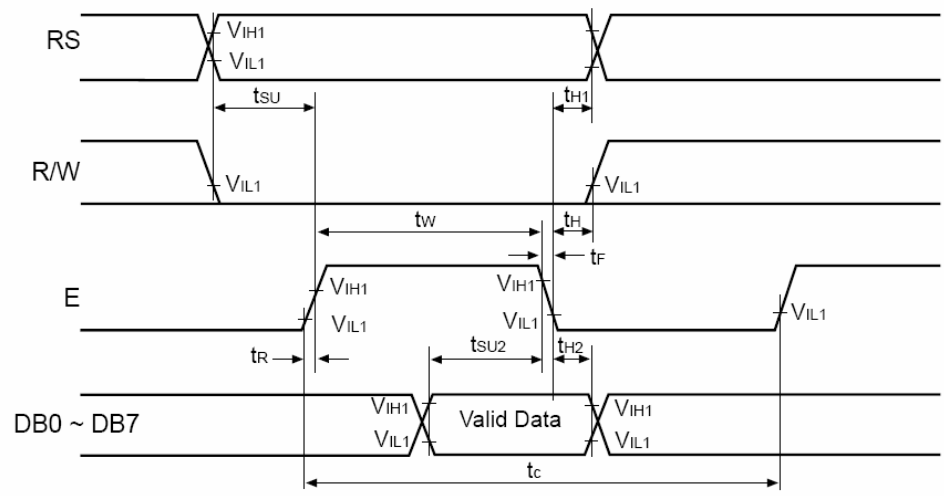

Ilustración 10. Cronograma de escritura[2]

El siguiente cronograma muestra un ejemplo de una secuencia temporal de una escritura en el registro de instrucción, usando una interfaz de cuatro bits de datos. Se observa como se realizan de forma consecutiva las dos escrituras (las dos ocasiones en

[1] Imagen extraída del Datasheet del display 329-0341 de *Theale Components.*

[2] Ídem

las que la señal E –Enable- se pone a nivel alto) sin comprobar, ya que no es necesario, que el display no se encuentre ocupado, ya que permanece a la espera de la segunda parte de la instrucción.

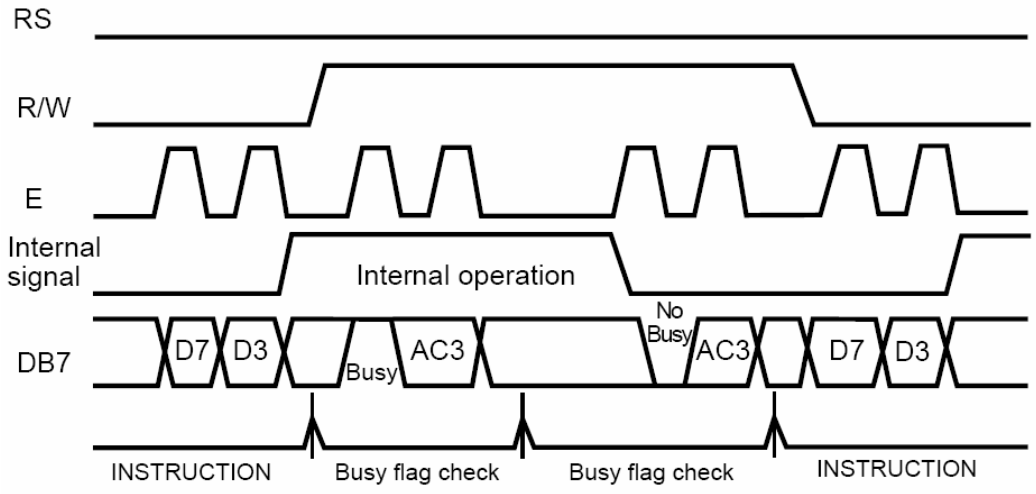

Ilustración 11. Dos instrucciones usando cuatro bits[1]

Debemos tener en cuenta, como observamos en los ciclos de espera entre las dos instrucciones, que para chequear la bandera de ocupado (bit D7) tenemos que efectuar los dos ciclos completos, de forma que el lcd ha de poner en el bus el *Nibble* completo. Esta bandera nos servirá para poder ejecutar bucles de espera activa antes de poner ningún dato en el bus, ya que corren el peligro de perderse o ser malinterpretados por el controlador de la matriz LCD.

En cuanto a la duración de estos ciclos representados, dependerá en todo momento del fabricante del display, al igual que la duración del tiempo de ejecución de las instrucciones que enviamos al display.

5.3 Pasos para el uso de la matriz lcd en un programa

1. Definir el puerto de comunicaciones del micro en el que va a estar conectado. Para que funcione adecuadamente, el cableado debe ser coherente con lo implementado en la librería. El nibble más significativo del puerto debe ir cableado con los bits de datos del lcd en el mismo orden. En cuanto a las líneas de control, pueden pertenecer a cualquier puerto. El hecho de pertenecer a puertos diferentes no interviene para nada en la temporización, puesto que todas las instrucciones que modifican algún puerto tarda un ciclo máquina.

[1] Imagen extraida del Datasheet del display 329-0341 de *Theale Components.*

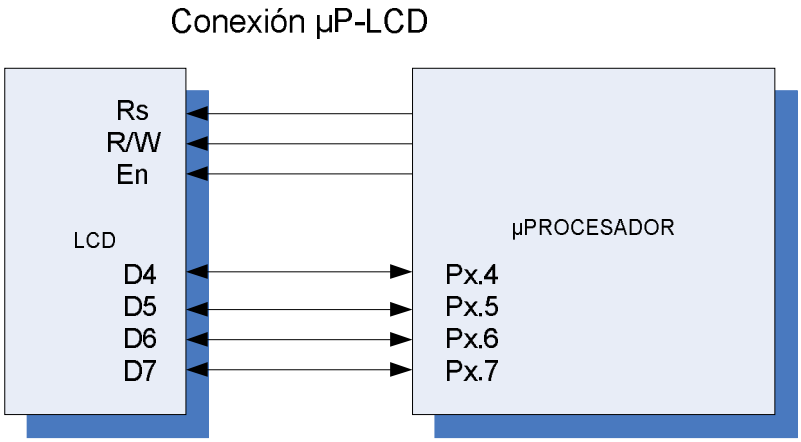

Ilustración 12. Cableado del display

2. Definir correctamente los registros que hacen referencia a los puertos que van a ser accedidos para el intercambio de datos y señales entre ambos componentes. Esto se comentará más adelante cuando se haga referencia a la forma en la que se ha definido en nuestra librería.
3. Inicializar los valores de estos registros (puertos) de forma que se especifiquen claramente qué pines son de salida y cuáles bidireccionales para evitar posibles colisiones o conflictos.
4. Inicializar el display enviando las instrucciones adecuadas. Para el tipo de controlador del display que nos ocupa, estas instrucciones son estándar y se pueden encontrar en cualquier hoja de dato de cualquier dispositivo. Durante esta inicialización, tendremos bits dedicados que debemos especificar, como los que determinan el encendido de la matriz gráfica, el mostrar o no el cursor, el parpadeo, etc. [9]

6 HERRAMIENTAS PARA LA IMPLEMENTACIÓN SOFTWARE

6.1 Introducción a µVisión2

Para el desarrollo del caso de estudio, y para la implementación y simulación de las funciones en general, se usó la suite de desarrollo, para procesadores basados en arquitectura 80C51, *µVision2* de *Keil Software*.

µVision2 es un plataforma de desarrollo de software diseñado para Windows, que combina un robusto e inteligente editor de textos, que incorpora su propio código de colores para diferenciar diferentes ítems del código, y un editor de proyectos, que aporta su funcionalidad a una herramienta que, a modo de *makefile*, compila los módulos de forma inteligente en función de las modificaciones que se realicen. Incorpora, por debajo de estas funciones de gestión y edición, numerosas herramientas software de la misma compañía, como un compilador de C, un intérprete de ensamblador para C51 (A51), linkador, etc.

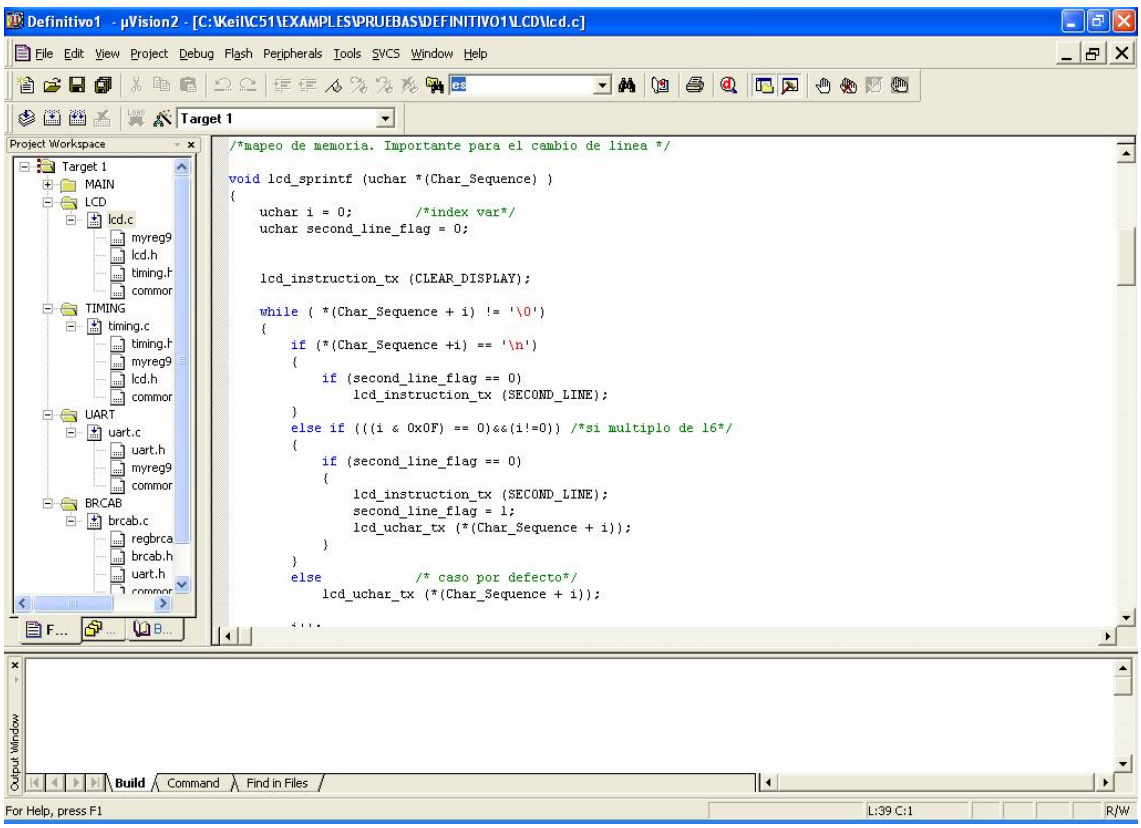

Ilustración 13. Ventana principal de la aplicación

Las ventajas de usar *µVision2* frente a una programación estándar es ensamblador son:

- Uso de un potente editor para el código fuente
- Proporciona una base de datos con las características de configuración de la mayor parte de los dispositivos basados en esta arquitectura existentes en el mercado.
- Incorpora un editor de proyectos, lo que permite un mantenimiento de los mismos mucho más eficiente.
- Integra facilidades de compilación inteligente.
- Es totalmente configurable, ya que se pueden modificar, editar o visualizar todas las características del micro desde ventanas.
- Incorpora un depurador de código capaz de simular los periféricos incluidos en la pastilla del microcontrolador.
- Contiene enlaces a numerosos archivos de ayuda, no sólo de uso y manejo del programa, sino también a hojas de datos y características de los dispositivos, permitiendo gestionar los documentos asociados a un proyecto, incluyendo los documentos y hojas de datos que incluya o cree el usuario de forma externa a la aplicación.

6.2 Compilador ANSI C Keil

6.2.1 Descripción general. Ventajas de la programación en lenguaje C frente a ensamblador

El compilador ANSI C de *Keil* está diseñado, según asegura la empresa fabricante, para generar un código rápido y compacto para los microprocesadores de la familia C51, de forma que se aproxime, en la medida de lo posible, a la velocidad de ejecución de un código óptimo equivalente escrito directamente en ensamblador.

El hecho de escribir directamente en un lenguaje de alto nivel como C, y puesto que se va a generar un código tremendamente optimizado, nos va a permitir sacar partido de numerosas circunstancias:

- En primer lugar, no tendremos la necesidad de aprendernos el juego de instrucciones del ensamblador A51, y podremos empezar a escribir con pocos

conocimientos sobre la disposición interna de la memoria del microprocesador. Podremos ir formándonos a medida que generemos códigos que cumplan con requisitos cada vez más complejos y necesiten un uso de recursos cada vez más refinado.

- La gestión de registros internos de la CPU va a correr a cargo del compilador, con lo que no necesitamos asociarlos a cada una de las variables de las funciones de nuestro código.
- El código fuente puede adquirir una estructura formal, impuesta por el propio lenguaje, y puede ser divido en tareas usando funciones. De esta forma nos evitamos ilegibles páginas de código en ensamblador, y fabricamos código fácilmente ampliable, ya que bastará con entender una función con parámetros y valores de retorno, que al fin y al cabo, es el punto de vista que el programador usuario debe tener de la librería.
- También mejorarán la legibilidad otros aspectos como la capacidad de usar expresiones complejas, tipos de datos diferentes o estructuras de datos.
- Las funciones y otras partes del programa pueden ser fácilmente incluidas en nuevos programas, sin necesidad de proceder a ningún tipo de reajuste de memoria, registros, manejo de direcciones de retorno, etc. ya que se pueden codificar funciones que no dependan de la estructura interna del micro; de hecho se han elaborado así para una mejor comprensión de la librería.
- También pueden ser fácilmente incluidas en el código de otras aplicaciones diseñadas para otros microprocesadores, debido a la portabilidad del lenguaje, sin más que hacer algunas pequeñas modificaciones que evitarán tener que reescribir el código.
- Al disponer de un compilador basado en el estándar ANSI, tendremos a nuestra disposición un amplio juego de funciones de librerías incluidas en él, y que permitirán de forma sencilla tener acceso a conversiones de tipos, operaciones aritméticas flotantes, manejo de cadenas, gestión de zonas de memoria, etc.

Esta serie de ventajas favorece de forma significativa el trabajo del programador, ya que se reducen drásticamente los tiempos empleados para la programación y la depuración de los programas generados.

6.2.2 Soporte para las peculiaridades de la arquitectura C51

El compilador que tenemos a nuestra disposición, como se comentó en el apartado anterior, está basado en el estándar ANSI C, pero no cumple con él. Esto es debido a que, puesto que la programación en estos dispositivos exige elevadas cotas de optimización, se incluyen extensiones propias de la arquitectura, de forma que resulte más sencillo realizar un código preciso.

6.2.2.1 Tipos de datos

El compilador soporta los clásicos tipos escalares de datos:

- signed y unsigned char (8 bits).
- signed y unsigned short (16 bits).
- signed y unsigned int. (16 bits).
- signed y unsigned long (32 bits).
- enum (16 bits).
- float (32 bits).

Para la personalización del C51, se incluyen los siguientes tipos escalares de datos:

- bit
- sbit (1 bit).
- sfr (8 bits).
- sfr16 (16 bits).

Estos tipos no son parte de ANSI C, y no pueden ser accedidos mediante punteros. Esto tiene como consecuencia que tampoco van a poder ser pasados como parámetro a ninguna función.

Una variable de tipo bit puede ser creada al principio de cualquiera de las funciones del programa, pero las otras tres van a tener un carácter especial y completamente diferente. Tanto sbit como sfr y sfr16 van a permitirnos acceder a los registros especiales de funcionamiento de la CPU, pero las variables han de ser inicializadas con los valores de su posición en el mapeo de memoria. Posteriormente, se podrán usar de una forma sencilla leyendo y escribiendo de ellos. Sólo se pueden definir fuera de cualquier función, por lo que su uso puede ser similar al de una directiva de

preprocesado que asocia un valor (en este caso, su dirección) a una cadena de caracteres que lo representa.

Por ejemplo:

```
sfr P0 = 0x80;
…
void main (void){
…
uchar valor;
…
P0 = valor;
```

En la primera sentencia, se define la dirección en la que se accede al registro P0. Esta definición no puede ser modificada ni referenciada mediante punteros de ahí en adelante. Dentro de la función, podemos darle valores a ese tipo de datos, con la certeza de que lo que asignamos no es ya una dirección, como en su definición, sino el valor que queremos escribir en el registro.

El compilador realiza, de forma automática, todas las conversiones de datos necesarias para poder interactuar con este tipo de datos especiales de forma sencilla y natural, sin forzar estructuras ni declaraciones extrañas. Así, podremos escribir con integers, chars, etc. en uno de estos registros, y viceversa, el valor de estos registros puede ser almacenado en dichos valores, porque las promociones de datos son automáticas y "lógicas", manteniendo la coherencia e incluso el signo, si procediera.

6.2.2.2 Tipos de memoria del C51

Desde el programa escrito en C se debe poder hacer referencia a datos almacenados en las diferentes zonas de memoria que la arquitectura C51 pone a nuestra disposición. De este modo, una variable puede ser explícitamente asignada a una de estas zonas mediante las siguientes palabras reservadas:

- *code*. Hace referencia a una zona dentro de la memoria de programa. Puesto que esta memoria es, en tiempo de ejecución, de sólo lectura, el valor almacenado en la variable definida como *code* queda fijada al programar el microcontrolador, no pudiendo ser modificada de ninguna otra manera que no sea reprogramando el dispositivo. Debido a la ausencia de pila real y ya que la memoria RAM de estos dispositivos es bastante limitada, las hemos usado para guardar por ejemplo, cadenas de caracteres que van a ser fijos en una aplicación, tablas y estructuras de datos, etc., pero siempre teniendo en cuenta que se resuelve en

tiempo de compilación. Sólo es accesible desde la dirección que almacene el contador de programa.

- *data*. Esta es una zona de memoria direccionable de forma directa. Es la más rápida de todas, por lo que su uso es recomendable para todas las variables que no van a poder almacenarse en registros, y que han de ser usadas frecuentemente. También para las interrupciones en las que el uso del tiempo de procesador es crítico.
- *idata*. Es una zona de memoria accesible de forma indirecta. Esta característica la hace adecuada para albergar la pila de programa, así como cualquier estructura de datos y tabla de pequeñas dimensiones.
- *bdata*. La arquitectura interna permite direccional una pequeña zona (16 bytes) de la memoria interna bit a bit. Existen instrucciones específicas en ensamblador, y por lo tanto, han de tener su equivalente en este compilador de alto nivel.
- *xdata*. Si la variable es definida así, el compilador la situará fuera del microcontrolador, en algún dispositivo externo que permita leer y escribir. La dirección a partir de la cual está mapeada esta memoria se guarda en registros internos. Los accesos son lentos, por lo que no son recomendables para el uso dentro de aplicaciones sensibles al tiempo ni para variables cuyo valor sea actualizado o usado con relativa frecuencia. Sin embargo, su uso está justificado en circunstancias en las que necesitamos almacenar tablas o estructuras de datos grandes, o simplemente, necesitemos conocer de forma externa, y en tiempo real, el valor de variables, usando un emulador
- *pdata*. Las variables *pdata* se encuentran mapeadas en una zona de memoria externa de 256 bytes dentro de *xdata*. Se diferencia de éstas últimas en que son accesibles de forma paginada. Esta característica la hacen apropiada para albergar tablas de mediano o pequeño tamaño, acelerando los accesos al ser accesibles por página, y, al igual que el tipo anterior, para ser observadas en tiempo real mediante algún kit emulador.

También es posible dejar que el compilador escoja por defecto un tipo de almacenamiento a una variable, en el caso de que no lo especifiquemos en su definición. En el siguiente punto se analiza esta circunstancia con detenimiento.

6.2.2.3 Modelos de memoria del C51

El modelo de memoria seleccionado determina el tipo de memoria que se usa por defecto en los parámetros pasados a funciones, las variables generadas interna y automáticamente, y las variables declaradas sin especificar el tipo de memoria. Este tipo de memoria elegido se puede sobrescribir siempre y cuando se especifique explícitamente en la definición.

El compilador determina el tipo de memoria a partir de la primera sentencia del programa principal escrito en C, que ha de tener la forma:

```
#pragma MODEL
```

donde MODEL puede ser una de las tres palabras reservadas que se detallan a continuación:

- SMALL: Todas las variables se ubican, por defecto, en la memoria interna de datos del C51 (equivale a declarar todas las variables como *data*). Éste es el modelo más eficiente desde el punto de vista de los accesos a las variables, ya que son lo más rápidos que pueden llegar a ser. Sin embargo, puesto que no se dispone de memoria externa, todos los objetos, incluida la pila de programa, se ubica en la escasa memoria interna. Esto limita y condiciona fuertemente la programación, ya que se disponen de pocos bytes para la pila, con lo que el paso de parámetros a funciones queda bastante limitado. Además no van a poder manejarse tablas o estructuras de gran tamaño, a no ser que se hagan estáticas incrustándolas en el código de programa mediante *code*. También va a condicionar, haciéndolas prácticamente prohibitivas, las anidaciones de varias funciones o las funciones recursivas, que deben ser evitadas a toda costa. Este es el modelo que el programa asigna por defecto a todos nuestros proyectos, ya que, generalmente, los microcontroladores basados en C51 vienen en pastillas con bastantes periféricos integrados, de forma que puedan funcionar si ningún tipo de dispositivo externo.
- COMPACT: Todas las variables se ubican por defecto en una página (256 bytes) de memoria externa. En este modelo, se han de declarar en el código de arranque del programa el sentido de los pines por los cuales se establecerá el bus de direcciones. Un programa definido así equivale a tener todas las variables definidas como *pdata*. Como limitación, solo puede acoger 256 bytes de

variables. Este modelo no es tan eficiente ni rápido como el anterior, pero sí puede establecerse como termino medio entre SMALL y LARGE.

- LARGE: Este modelo sitúa todas las variables en memoria externa, por lo que es equivalente a definir todas las variables como *xdata*. Para direccionar se usa un registro especial llamado DPTR (Data Pointer), pero el direccionamiento con 8 bits es altamente ineficiente, sobre todo para variables de más de un octeto.

Analizados todos los modelos, queda patente que el último es el más lento de todos y, además, es el que más código en ensamblador genera al compilar. Sin embargo, va a ser necesario cuando se manejen estructuras de datos de gran tamaño.

Como conclusión y, a la vez, recomendación general, puesto que el modelo SMALL es el que genera el código más eficiente, es el que se debe usar siempre, modificando las variables en las que fuera necesario redefinir su tipo para acomodarlo a las circunstancias y ubicarlas en memoria externa, código, etc. Si con esto no fuera suficiente para cumplir todas las especificaciones del programa (o el compilador avise de la falta de memoria), entonces, y sólo entonces, debemos pasar al siguiente modelo (COMPACT), pero intentando albergar todas las variables posibles como *data*. Sólo en última instancia, cuando no tengamos otra alternativa, usaríamos el modelo LARGE, pero siempre, como en el anterior caso, alojando las variables en la memoria interna para acelerar en lo posible la ejecución del programa.

El programa nos situará por defecto dentro del recomendado tipo de memoria SMALL, si bien podremos cambiarlo de varias formas. La primera de ellas es, como se muestra al principio de este punto, incluyendo la sentencia #pragma en la primera línea de nuestro programa en C; y otra posibilidad es actuar directamente sobre la línea de comandos que pone el compilador a nuestra disposición. Sin embargo, lo más recomendable es ajustar todos los parámetros relativos a la compilación al definir el proyecto, dentro de las opciones que el programa pone a nuestra disposición. Así, en la pestaña de opciones para cada objetivo de nuestro software, podemos definir el modelo en el menú desplegable señalado en la ilustración 14.

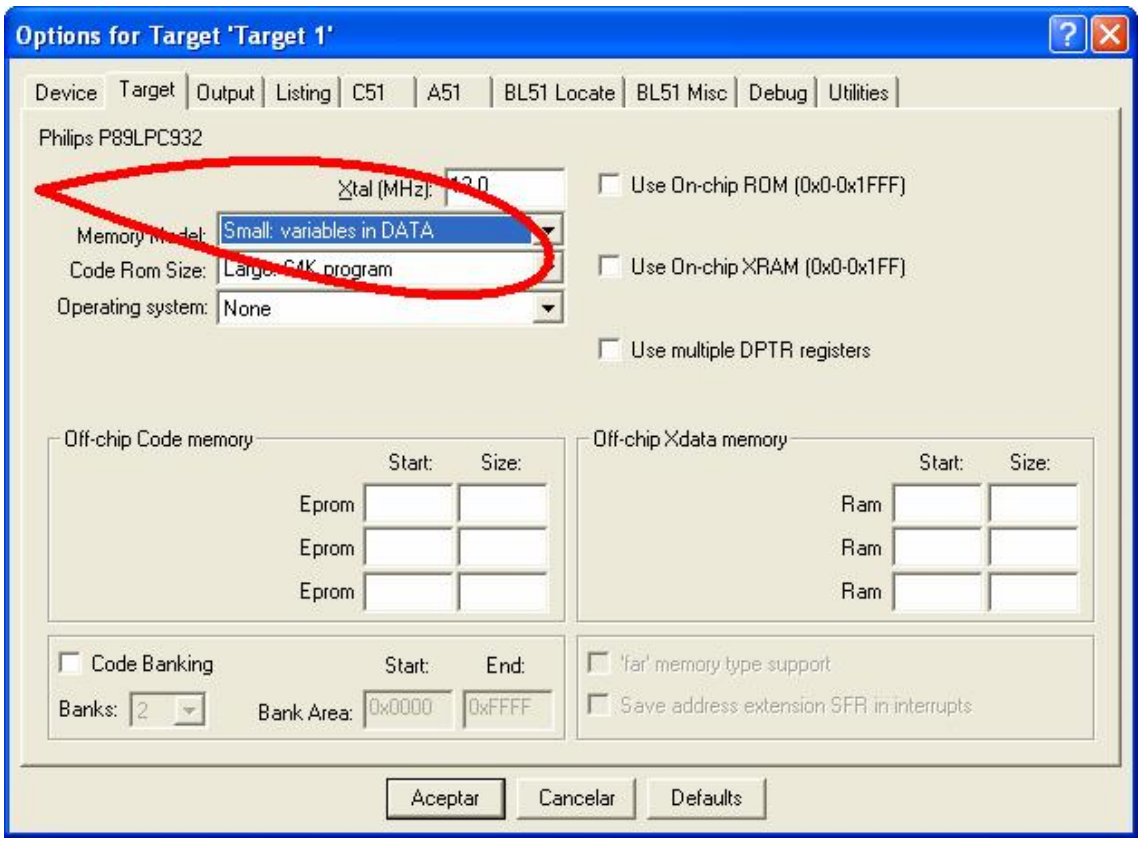

Ilustración 14. Opciones para el programa objetivo

6.2.2.4 Uso de punteros a zonas de memoria específica

El compilador pone a nuestra disposición, debido a la peculiar arquitectura de los núcleos C51, dos tipos diferentes de punteros: punteros genéricos y punteros a zona de memoria específica.

Los punteros genéricos se declaran sin especificar una zona concreta de memoria, es decir, de la misma manera que si estuviéramos trabajando con C estándar.

```
char *s         /* puntero a carácter. Cadena*/
```

Este puntero puede referenciar, en cualquier punto del programa, una instancia del tipo char sea cual sea su ubicación en memoria y el modelo de datos. El código que genera un puntero de este tipo en ensamblador consta de tres bytes; el primero de ellos contendrá información sobre el tipo de memoria, mientras que los dos siguientes contendrán el byte alto del offset y el byte bajo respectivamente. A pesar de que son lentos y se consume mucha memoria para generarlos, este tipo de punteros favorece la escritura de un código cómodo, ya que no tenemos que preocuparnos de cuál es la memoria que el programador o el compilador le ha asignado a la variable a la que

estamos referenciando. Asimismo, la legibilidad y comprensión del código se ve gratamente favorecida, y se genera un código portable.

En el extremo opuesto, tenemos la opción de definir a qué tipo de memoria va a apuntar permanentemente el puntero que acabamos de definir.

```
char xdata *s
```

Este tipo de declaraciones repercute en un ahorro de memoria, ya que, como el tipo de memoria se asigna en tiempo de compilación, no es necesario usar un byte que nos indique un tipo como en el caso de los punteros genéricos. Así, el tamaño del puntero almacenado sólo dependerá de lo grande que sea el tipo de memoria al que esté apuntando, por lo que sólo se necesitará un byte si se apunta a una variable *data*, *idata*, *bdata* o *pdata*, o bien dos bytes si apuntamos a *code* o *xdata*.

Los punteros así definidos son más rápidos y generan un código más eficiente, pero perdemos la capacidad de poder apuntar cualquier tipo de variable. De igual forma, el código escrito con estos puntero se vuelve más complicado de comprender, amén de no portable.

En resumidas cuentas, para facilitar nuestra tarea, a pesar de que en cuanto a memoria y velocidad no sea lo más recomendable, se han usado punteros genéricos, aunque, según los resultados mostrados por el fabricante, puesto que existe una zona de la memoria interna que sólo es referenciable mediante indirección, los punteros específicos pueden sernos de gran ayuda en caso de necesitar referenciar variables muy sensibles a la velocidad de ejecución, ya que el código generado es extremadamente sencillo y veloz.

6.2.2.5 Simulación de pila para funciones “reentrant”

Las funciones *reentrant* son aquellas funciones compartidas por varios procesos, de forma que mientras se está ejecutando la función, otro proceso, interrupción, etc., puede detenerla y llamar y ejecutar una nueva instancia de la misma. También puede aplicarse este concepto a las funciones recursivas. Se han comentado en capítulos anteriores los perjuicios que esta forma de programar puede tener en nuestra aplicación, ya que se puede llenar la pila con facilidad. Pero hay otra peculiaridad que también debemos tener en cuenta: los argumentos que se pasan a las funciones, así como las variables locales se envían, al compilar, a posiciones fijas dentro de la memoria que corresponda (según el modelo), por lo que sus valores se perderían.

Para ello, una característica del compilador apartada del estándar permite diferenciar funciones como *reentrant*. Las funciones que sean definidas de esta manera van a poder ser llamadas desde varios procesos o ejecutarse de forma recursiva, ya que, para ellas, se simula dentro de memoria interna o externa, un trozo de pila que permita pasar sus parámetros sin temor a perderlos.

En cualquier caso, deberían usarse sólo en las situaciones en las que fueran imprescindibles, ya que se genera un código mucho más lento, además del gasto de un elemento generalmente crítico como la memoria.

6.2.2.6 Interrupciones

El compilador nos permite crear de una forma sencilla las rutinas de atención a las interrupciones del C51. Sólo necesitaremos saber qué número de interrupción es la que estamos implementando.

```
void isr_function(void) interrupt n [using m]
```

Esta declaración genera una interrupción ya que se añade la etiqueta *interrupt* al final de la declaración de la función. Para generar el valor correcto en el vector de interrupciones del micro, el compilador usará el número de interrupción *n*, mediante la fórmula $8 \cdot n + 3$, puesto que la primera interrupción alberga su vector en la posición 0003h. El segundo y opcional modificador de la declaración [using m] obliga al compilador a utilizar un banco de registros determinado de los cuatro que posee el núcleo del C51. Como normal general, este parámetro no debe ser especificado, dejando el banco de registros 0, el que se usa por defecto, activo para la interrupción, o que el compilador sea el que lo modifique si lo considera oportuno. Las circunstancias en las que debe usarse un nuevo banco de registros son:

- Interrupciones sensibles al tiempo de ejecución, y que deben estar activas tan rápido como sea posible, a pesar de que esté usada la mayor parte de la memoria RAM.
- Interrupciones que tengan llamadas a otras funciones, evitando de esta manera que se reescriban variables locales en la pila o registros.
- Para proteger los datos de una interrupción frente a interrupciones de mayor nivel y queramos definirlo nosotros explícitamente, en lugar de dejar que el compilador decida.

6.2.2.7 Paso de parámetros y valores de retorno

Los registros internos del microcontrolador van a servir de variables para el paso de parámetros en la mayoría de los casos, puesto que se trata de la forma más fácil y rápida de mantener la información y hacerla disponible para la siguiente función en ejecutarse, así no tienen que ser escritas ni leídas de ninguna memoria.

En total, se pueden pasar por registro hasta un máximo de tres argumentos de tipo char o int, o punteros específicos de unos o dos bytes; o uno sólo de los tipos de datos de cuatro bytes (long o float) o un puntero genérico de 3 bytes.

Si nuestras funciones necesitasen pasar más parámetros de los que acabamos de citar, el compilador los ubicaría en alguna de memoria fijada para esos argumentos. Este hecho facilita la programación, pero genero un código más lento.

En cuanto a los valores de retorno, el compilador los asigna siempre a algún registro o registros del micro. La siguiente ilustración muestra las equivalencias definidas por el fabricante.

Return Type	Register	Description
bit	Carry Flag	
char, unsigned char, 1-byte pointer	R7	
int, unsigned int, 2-byte pointer	R6 & R7	MSB in R6, LSB in R7
long, unsigned long	R4 — R7	MSB in R4, LSB in R7
float	R4 — R7	32-Bit IEEE format
generic pointer	R1 — R3	Memory type in R3, MSB R2, LSB R1

Ilustración 15. Registros que almacenan valores de retorno[1]

6.2.2.8 Optimizador de código

El optimizador de código del C51 ejecuta numerosos pasos antes de generar el código a la salida. Todos esos pasos son transparentes para el programador, por lo que no queda más remedio que acudir al manual del software y creer a pies juntillas lo afirmado por el fabricante.

El compilador analiza y crea un código tan eficiente en velocidad como en espacio, si bien una de estas dos antagónicas características puede hacerse prevalecer

[1] Imagen extraída del manual *Gettin started with C51* de *Keil Software*.

mediante instrucciones en línea de comandos al compilador. Las optimizaciones que lleva a cabo el compilador son:

- Compresión de constantes. Las constantes que aparezcan en una dirección o expresión son combinadas como un único valor constante.
- Gestión de saltos en el código.
- Eliminación de código que no puede ser alcanzado dentro del programa (código muerto)
- Gestión de registros. Siempre que sea posible, las variables se almacenarán en registros, por lo que no se tendrá que usar memoria para su almacenamiento y su uso será más rápido y eficiente.
- Paso de parámetros por registros. Hasta un máximo de tres.
- Eliminación de expresiones comunes. Todas las expresiones recurrentes o idénticas que se ejecutan varias veces en una función son reconocidas y calculadas una sola vez.
- Fusión de código. Los bloques de instrucciones comunes se funden en uno sólo, y se hace accesible mediante una instrucción de salto.
- Reutilización de código común de entrada a las funciones.
- Las secuencias de instrucciones múltiples son empaquetadas dentro de subrutinas. Las instrucciones son reordenadas para hacer este paquete lo más grande posible.

Estas optimizaciones son llevadas a cabo por el compilador en su tarea de "traducir" a ensamblador, pero, de acuerdo con las características del ensamblador y de la arquitectura de un C51, también realiza las siguientes:

- Optimización del flujo de datos. Operaciones complejas son sustituidas por operaciones más simples de forma que se disminuya el tiempo de ejecución.
- Las constantes y variables son computadas e incluidas directamente en las operaciones.
- Se optimiza el acceso a la memoria extendida, ya que el registro DPTR se usa como puntero a datos externos, por lo que se ahorran instrucciones y se incrementa la densidad del código.

- Los datos sufren un proceso de “overlay”, que consiste en que la zona de memoria que ocupa puede ser reutilizada y reasignada para otros datos y funciones, de forma que funciona a modo de capas.
- Las estructuras *switch* y *case* pueden transformadas, dependiendo de su número y la localización dentro del código, en una tabla de saltos o en una cadena de saltos para intentar generar el código más pequeño posible.

El optimizador puede recibir cuatro instrucciones diferentes que pueden alterar y determinar su comportamiento final frente al código, haciendo que se decante, en numerosas situaciones, por unas estructuras u otras.

- OPTIMIZE(SIZE). Esta instrucción reemplaza instrucciones comunes de C por subprogramas, de forma que son capaces de reducir al máximo el código del programa, pero siempre a costa de la velocidad.
- OPTIMIZE(SPEED). Las operaciones comunes de C son expandidas en una serie de órdenes de forma que se ejecutan de la manera más rápida posible. Como perjuicio, el tamaño del programa se incrementará.
- NOAREGS. No se usan accesos absolutos a los registros. De esta forma, el código no depende del banco de registros que se esté usando.
- NOREGPARMS. Este comando hace que el paso de parámetros sea en secciones locales de memoria en lugar de a través de los registros. Su uso sólo está justificado en circunstancias en las que se necesite que el código sea compatible con versiones anteriores del compilador y ensamblador.

Durante la fase de simulación y pruebas de las funciones de las librerías, una vez analizado el código generado en diferentes circunstancias, se llegó a la conclusión de que el proceso de optimizado de código es lo suficientemente potente como para dejarle actuar sin modificarle ninguno de sus parámetros ni forzar ningún comportamiento. Sólo en algunas ocasiones, y debido al tamaño de los datos que se manejaban, se hubo de forzar al compilador a incrustar la parte estática de diferentes tablas dentro de la memoria de programa.

6.2.3 Generación e inclusión de código ensamblador

El compilador puede generar código en ensamblador en lugar de un objeto en código máquina. Si se compila usando la directiva SRC se genera un código listo para

ser interpretado y pasado a código máquina por el ensamblador A51. Éste código es importante ya que puede ser analizado para comprobar qué está haciendo el compilador con el código fuente que acabamos de escribir. Analizándolo, podríamos emplear el código generado y modificarlo para que funciones siguiendo unos patrones fijos a nuestro parecer.

Así, podemos acceder desde el código c a rutinas generadas en ensamblador por el compilador, ya que las variables que pasamos como parámetros, la dirección de retorno, y el valor de retorno de la función van a estar generalmente almacenados en registros o en posiciones fijas de memoria. Podemos saber en qué registros se guardan y en qué posiciones sin más que interpretar el código generado por la llamada a la función y las últimas líneas, en las que se devuelve el flujo de programa a la función llamante.

```
    28:          lock_us_delay (6701);
C:0x025A    7F2D     MOV      R7,#0x2D
C:0x025C    7E1A     MOV      R6,#0x1A
C:0x025E    1203E0   LCALL    lock_us_delay(C:03E0)
```

Como ejemplo, vemos el código generado por una llamada a una instrucción que se encuentra en la línea 28 del archivo fuente en C. Recibe como parámetro un entero de 16 bits y no devuelve nada en el nombre. Interpretando el código vemos como el parámetro se ha enviado en los registros R6(el byte alto) y R7(el byte bajo). De esta forma podremos hacer uso de esta rutina en ensamblador o bien inspirarnos en ella para crear nuestros propios códigos óptimos.

Para insertar código ensamblador dentro del archivo fuente en C, debemos incluirlo en las directivas de preprocesado entre las sentencias:

```
#pragma asm
#pragma endasm
```

6.2.4 Generación de información de depuración

Los objetos generados por el compilador siguen el formato de objetos de Intel OMF51 e incorporan completa información acerca de los símbolos que contiene. Adicionalmente el compilador puede incluir información sobre nombre de variables, nombres de funciones, números de línea para el control de la ejecución, etc., para permitir un depurado detallado con el depurador incorporado en µVision2, aunque también es compatible con depuradores de terceras partes.

El módulo ensamblador A51 también genera información similar, de forma que el depurado de las secciones escritas en ensamblador sigue las mismas directrices, ya que se dispone de la misma información.

6.3 Resto de componentes

6.3.1 Ensamblador A51

El ensamblador A51 es capaz de transformar instrucciones simbólicas en ensamblador en código máquina. Debe ser usado cuando necesitemos tener un control total y exacto sobre el hardware, y no queremos dejar nada al arbitrio del intérprete compilador de C. Además, al escribir en ensamblador nos aseguramos el generar un código pequeño, ya que se va a hacer única y exactamente lo que ordenemos al micro.

6.3.2 Linkador BL51

El linkador BL51 genera código ejecutable en formato 8051 a partir de ficheros de tipo objeto. Resuelve además todas las referencias a variables y funciones externas y asigna direccionamiento absoluto a todos los segmentos que sean reubicables dentro del código, para conseguir una secuencia de código compacta y homogénea. Asimismo, interpreta el entorno (modelo de memoria y aritmética necesaria) y selecciona la librería de tiempo de ejecución (*run-time library*) apropiada.

Sin embargo, su función más sobresaliente es la de reutilizar a modo de capas las zonas de memoria que están siendo usadas (overlay), estableciendo una gestión de la limitada cuantía de memoria existente en estos dispositivos., logrando así reducir el consumo total de memoria usada. Podremos usar la directiva OVERLAY para controlar de forma manual las referencias a funciones que el linkador usa para asignar zonas exclusivas en la memoria.

Sin embargo, también existe la posibilidad de usar la directiva NOOVERLAY, que deshabilita esta función por completo. Esto sólo está recomendado en el caso de que se esté generando un código exclusivamente para su depurado, o bien, caso no recomendable debido a la cantidad de memoria que se está usando, cuando se tengan en el código llamadas a funciones de forma indirecta mediante punteros a función.

El linkador también tiene constancia de los registros que son modificados por funciones externas a cada objeto. De esta forma, se genera, para cada proyecto, un archivo con ésta y otra información sobre los registros y que, mediante realimentación,

puede ser leído e interpretado por el compilador de C, de forma que se tiene control sobre los registros que están siendo usados en cada momento y puede generar un código que optimice su uso.

Una última característica que proporciona, aunque no vamos a necesitar disponer de ella, es el *code banking*. El *code banking* consiste en albergar el código en memoria externa, en un dispositivo que permita crear bancos de 256 bytes (paginar) y poder alternar entre ellos (*bank switching*). Así, podremos manejar hasta 32 bancos pudiendo tener un código que alcance los 2megabytes.

Necesitamos tener, para su correcto funcionamiento, una zona de memoria accesible sea cual sea el banco en el que se esté, por lo que esta memoria ha de estar repetida en todos los bancos o bien alojada en otro dispositivo accesible con el mismo mapeo de memoria. En esta zona de memoria se debe albergar todo el código que ha de estar disponible sea cual sea la zona de memoria que se está ejecutando: constantes, vectores de interrupción, las rutinas de atención a las interrupciones, la tabla de saltos entre bancos, etc.…

6.3.3 Gestor de librerías

Esta utilidad nos permite crear y mantener librerías de una forma fácil, ya que es capaz de generar una librería (*.lib) en lugar de un archivo ejecutable en la compilación. La creación de una librería en este formato permite su distribución de una forma fácil, permitiendo compartir las funciones generadas, y lo único que tenemos que hacer es seleccionar la opción correcta en la pestaña Output, como se muestra en la siguiente ilustración.

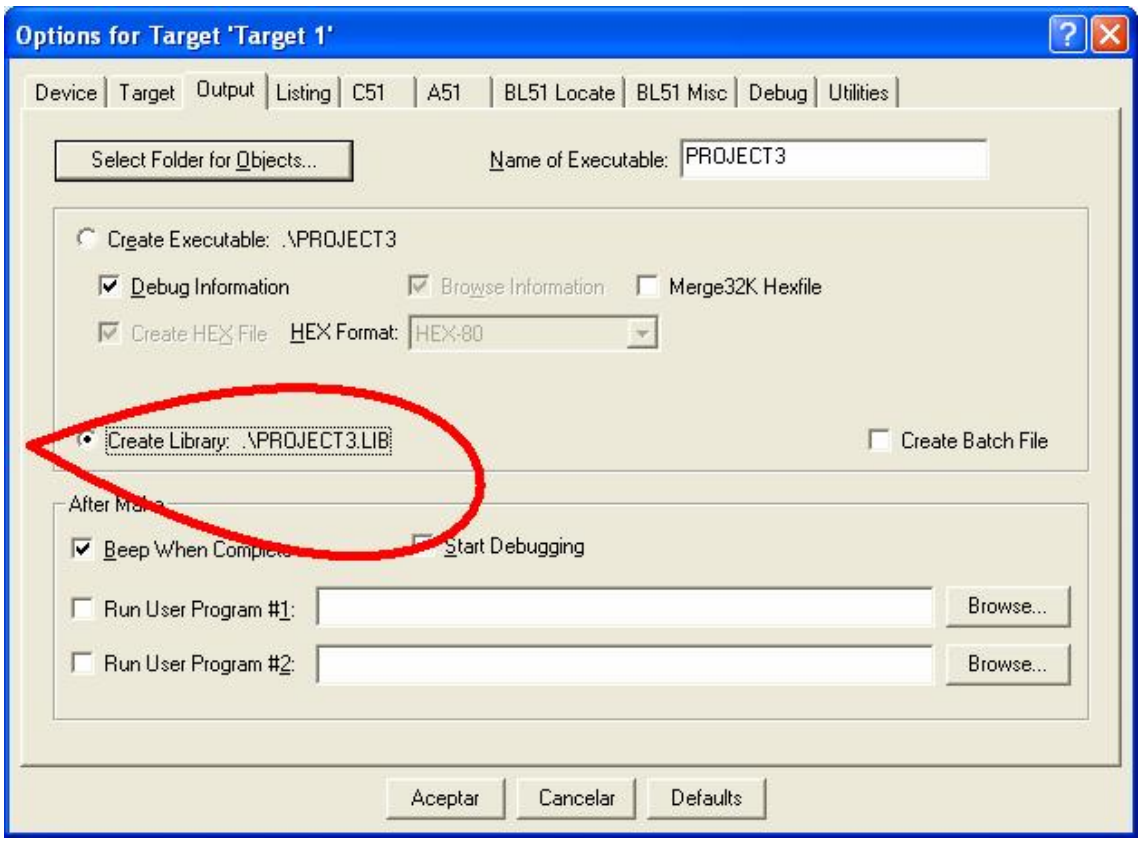

Ilustración 16. Menú de opciones de salida

6.4 Empezar a trabajar con µVision2

Todas estas características comentadas proporcionan una suite de programación sencilla de usar y con todas las características fácilmente accesibles puesto que se basa en una interfaz intuitiva.

Cuando se genera un proyecto, se selecciona el dispositivo para el cual va a generarse la programación. Una vez seleccionada, podremos copiar el código de inicialización de forma automática a nuestro directorio de trabajo y añadirlo al proyecto como uno de los ficheros a compilar. Este código de inicialización genera una situación inicial del dispositivo de acuerdo con el valor por defecto que se especifica en las hojas de datos, y puede ser editable y, por ende, modificable.

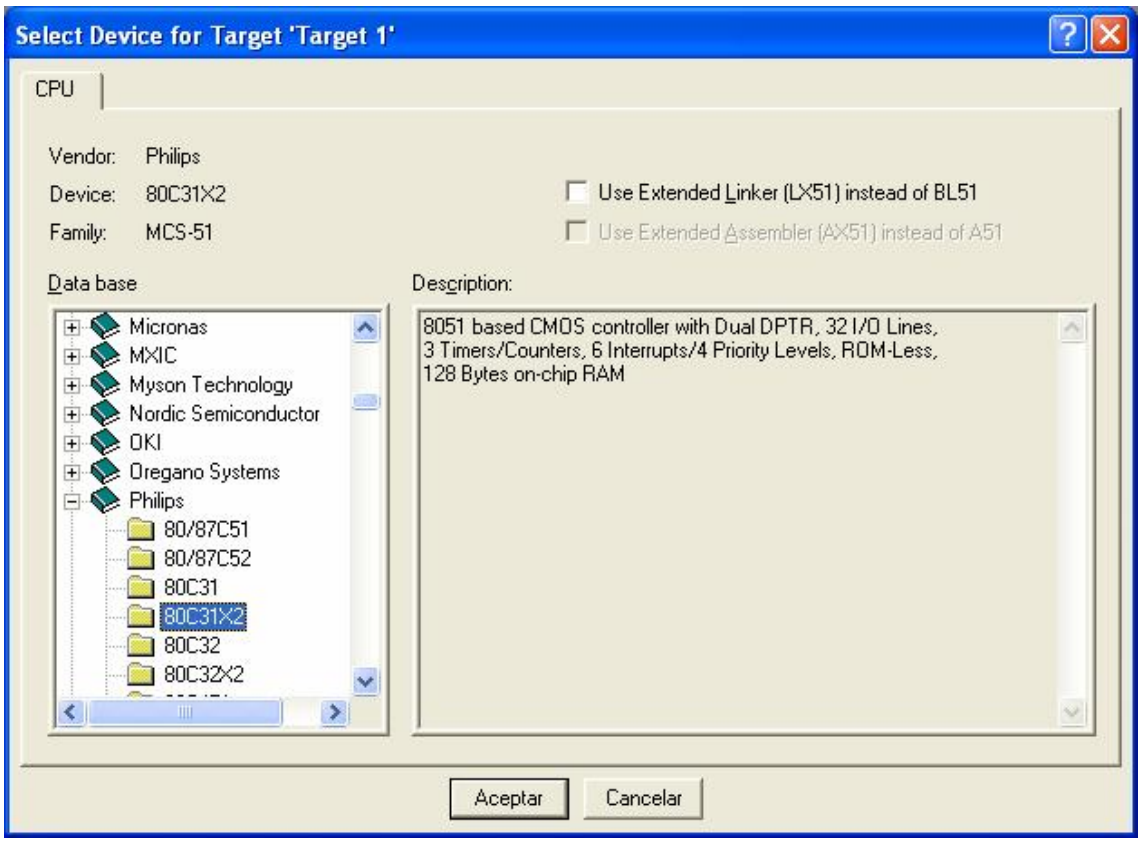

Ilustración 17. Selección de dispositivo objetivo

Posteriormente, mediante el comando *manage components* podremos añadir objetivos al proyecto, y dentro de cada uno de ellos, manejar y organizar los grupos de archivos, dentro de los cuales se incluyen archivos fuente. A partir de ese momento, tan sólo consiste en escribir código y compilar, como cualquier entorno de programación estándar.

Para programar debemos seguir ciertas reglas que se desprenden de todo lo comentado acerca de la estructura interna del 80C51 y acerca de las características del compilador de Keil.

- Siempre se usarán, salvo que sea estrictamente necesario no hacerlo, variables de ocho bits. El 8051 es un micro de ocho bits, y no tiene ningún tipo de instrucción de 16 bits. Por tanto las variables *char* son mucho más eficientes que las *int*.
- Siempre se usarán, salvo que sea estrictamente necesario no hacerlo, variables del tipo *unsigned*, ya que el micro no posee instrucciones capaces de comparar, multiplicar, etc. con signo. De hecho, las instrucciones que involucran variables

signadas son reemplazadas al compilar por una secuencia de instrucciones que sustituyen el comportamiento del signo.

- Las únicas divisiones que el micro soporta son divisiones de valores de ocho bits. Cualquier otra división da lugar a numerosas líneas de ensamblador para emular su comportamiento.
- El fabricante del compilador recomienda evitar las estructuras de bits, ya que no todas las versiones (sólo las más recientes) del compilador las soportan tal y como las define ANSI. La explicación es que generan un código altamente ineficiente. En su lugar, se recomienda usar el tipo definido *bit* para cada uno de los valores que necesitemos, y pasar sus valores a funciones escritas por el usuario que gestionen esa información.
- El estándar ANSI determina que el producto de dos palabras de 8 bits es otra palabra de 8 bits. Eso significa que cualquier *unsigned char* que deba ser multiplicado ha de ser promocionado a *unsigned int* si pensamos que existe la posibilidad de obtener un valor que no quepa en 8 bits. Luego, podemos almacenar sus ocho bits menos significativos en otra variable de 8 bits. El fabricante recomienda:

```
unsigned char z ;
unsigned char x ;
unsigned char y ;
z = ((unsigned int) y * (unsigned int) x) >> 8 ;
```

- Operaciones de entero que producen un resultado almacenado en ocho bits van a funcionar siempre, ya que, siguiendo el estándar ANSI, se van a usar sólo los ocho bits menos significativos del resultado. De esta manera, el compilador accede sólo a uno de los bytes del resultado con una sola instrucción en ensamblador, ahorrando código.
- Todos los operandos en una expresión son almacenados en una pila. Para modelos de memoria SMALL, donde la pila es interna, deben estar prohibidas las operaciones con números flotantes, ya que consumen una enorme cantidad de bytes que apilar en memoria. Aunque se pierda precisión, siempre se han de usar operandos en coma fija. Una comparación entre números de 8 bits sin signo tarda en ejecutarse 5 ciclos máquina, mientras que para flotantes consume 302; una división de números de 32 bits con signo tarda 1624 ciclos.

6.5 Depuración de código

El código generado puede ser testeado y depurado mediante el *μVision2 debugger* incorporado en la suite que estamos manejando. Nos ofrece dos modos de operación diferentes.

- Podemos simular con algunos modelos comerciales de placas de pruebas y depuración conectadas al ordenador mediante el puerto serie. El depurador lleva incorporados algunos drivers para darles soporte directamente desde el computador.
- Su uso más común será el de un producto de simulación puramente software, que es capaz de simular la mayoría de dispositivos que están incluidos en las versiones comerciales de los microcontroladores sin necesidad de ningún hardware real conectado.

La simulación del código es capaz de controlar hasta 16 megabytes de posiciones de memoria, mapeadas para lectura, escritura o área de código, detectando los accesos ilegales si se produjeran. Asimismo, posee las características del microcontrolador para ofrecernos los periféricos *on-chip* extraídas de la base de datos desde la que elegimos el dispositivo cuando empezamos a desarrollar un proyecto.

La depuración puede llevarse a cabo bien a partir del código fuente, o bien a partir de uno de los elementos más socorridos par averiguar qué es lo que está pasando dentro del microprocesador, la ventana *Disassembly Window*, en la que se muestran mezclados el código fuente en C y la secuencia de instrucciones en ensamblador que han sido generadas para cada línea. Si decidimos ejecutar el programa paso a paso siendo ésta la ventana activa, cada uno de los pasos de ejecución va a ser una instrucción en ensamblador, y no una línea de código C. Esta secuencia de instrucciones se puede modificar temporalmente sobre la marcha y ver el efecto que producen sobre la ejecución del programa.

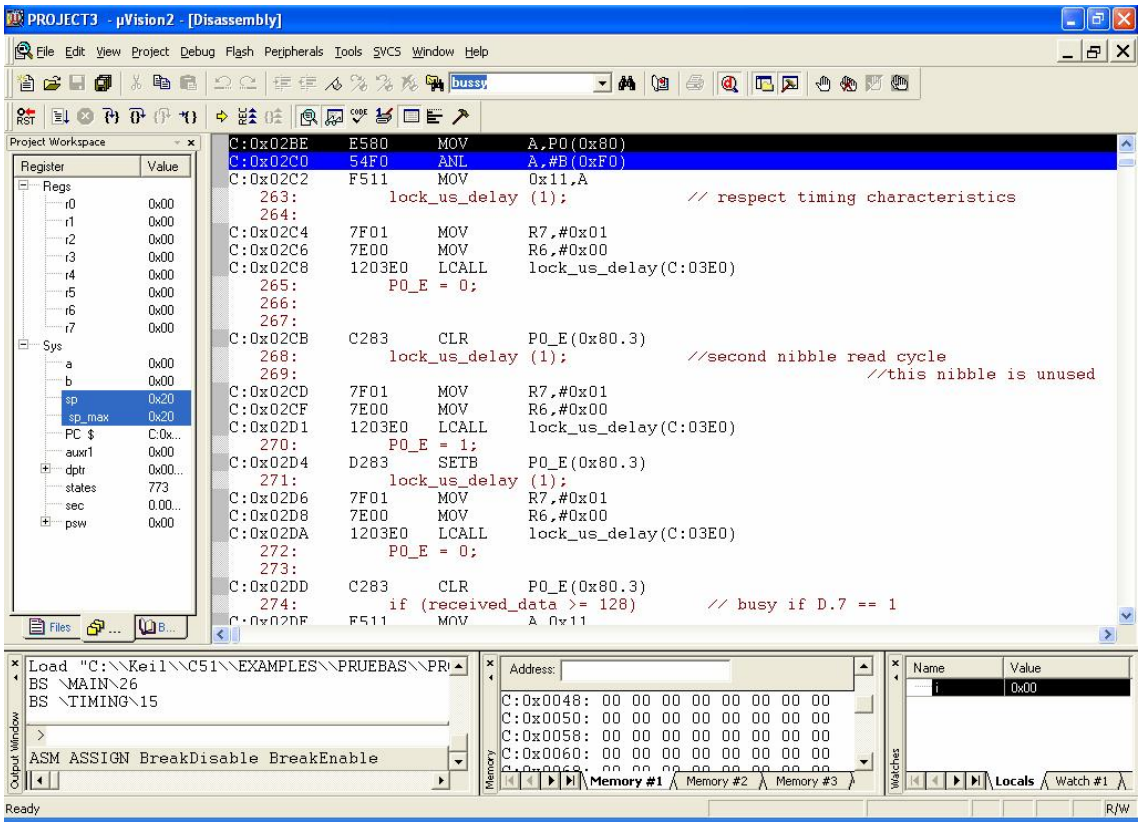

Ilustración 18. Dissasembly

Por lo demás, tenemos a nuestra disposición las herramientas clásicas de depuración que, tal como se aprecian en la ilustración son el mapeo de memoria, la disponibilidad del valor instantáneo de cualquier expresión o variable, e incluso podemos hacer uso de un editor en el que se pueden escribir comandos en modo consola.

7 DESCRIPCIÓN DE LA LIBRERÍA

La librería desarrollada va a estar dividida, en primera aproximación, según cada uno de los periféricos o partes del sistema a los que haga referencia. Para cada una de las partes se ha definido, tanto el archivo fuente de las funciones, como un archivo de cabecera del mismo nombre y, obviamente, distinta extensión. En este archivo de definiciones se encuentran, tanto los prototipos de las funciones como las definiciones de tipos de datos y constantes que permitan dar más legibilidad al código.

En concreto, las secciones que la forman, y que recogen tanto funciones de manejo y control como implementación de protocolos de comunicaciones cuando son necesarios, son:

- Temporización y funciones para TIMERS. Se encuentran implementadas en *timming_functions.c* y *timming_functions.h*.
- Funciones de entrada/salida y gestión de las comunicaciones en paralelo, que están en *io_functions.c* y *io_functions.h*.
- Funciones de comunicación a través de protocolo serie UART, incluidas en *serial_com.c* y *serial_com.h*.
- Funciones de control y comunicaciones con displays lcd basados en HD44780, que pertenecen a los archivos *lcd_com.c* y *lcd_com.h*.

En cuanto a la división por capas de la librería, dentro de cada una de las partes de las que se compone, pueden encontrarse funciones independientes del medio físico y funciones que tienen acceso directo al hardware de la aplicación.

Estos últimos, por propia definición, serán dependientes del hardware de la aplicación que estemos desarrollando en ese momento. Aún así, se han codificado siguiendo una estructura sencilla, presentando una serie de recomendaciones para la personalización en función de microprocesador sobre el cual estemos implementando la programación.

8 FUNCIONES DEPENDIENTES DEL MEDIO FÍSICO. PERSONALIZACIÓN PARA PHILIPS P89LPC32

Las funciones dependientes del nivel físico se personalizaron para este microcontrolador de *Philips*. Se necesitaron para tal meta una serie de definiciones que se van a describir a lo largo de este punto, y que ponen a nuestra disposición las características principales del micro en forma de registros *sfr*, un tipo de datos especial de μVision2 que permite referenciar fácilmente los registros. De esta forma, se puede escribir y leer de ellos con instrucciones simples, como se comentó en la descripción del compilador.

Las funciones que aquí se muestran dependen, en casi todos los casos, de las definiciones previas de cada módulo, por lo que se habrán de modificar estas definiciones si se quiere personalizar esta librería para cualquier otro microcontrolador basado en arquitectura 80C51. Uno de los objetivos a la hora de implementar la librería que nos ocupa, es que su posterior mantenimiento y personalización sea llevada a cabo de la forma más sencilla posible. Por este motivo, las funciones sólo accederán, siempre que sea posible, a los registros, funciones, constantes, etc. que tengan que ver con el hardware a través de las definiciones previas de la cabecera. Para los casos en los que lo no se usó el citado método, bien por simplicidad, o bien porque la velocidad de ejecución así lo recomendaba, se comentan de forma clara para que el mantenimiento sea hacedero.

8.1 Funciones de temporización

La temporización no sólo va a depender del dispositivo microcontrolador que estemos usando, sino que también va a depender, para un mismo procesador, del reloj del sistema que se esté usando en este momento.

En nuestro caso, el microprocesador funciona con un oscilador integrado en la misma pastilla de frecuencia $f_{clk} = 7{,}3728 Mhz$, y puesto que funciona seis veces más rápido que las unidades estándar C51, la instrucción más rápida sólo consume dos ciclos de reloj por cada ciclo máquina. Así, un ciclo máquina consume, cualquiera que sea la

fuente de reloj de sistema, un tiempo igual a $T_{MC} = \frac{2}{f_{clk}} = 2 \cdot T_{clk}$, y particularizando para el reloj integrado en el sistema, $T_{MC} = \frac{2}{7{,}3728} \cdot 10^{-6} \approx 2 \cdot 0{,}13563 ms = 0{,}27126 ms$.

Bajo este precepto inicial se han realizado todos los cálculos de tiempo, por lo que el uso de cualquier otra señal de reloj u oscilador para dicho micro, así como el uso de cualquier otro micro que no tenga las mismas características, producirá impredecibles resultados a priori.

Recordemos que la mayoría de las instrucciones de ensamblador se ejecutan en un ciclo de reloj, salvo algunas instrucciones, como las de salto, que se ejecutan en dos, pero siempre trabajando con datos de ocho bits sin información de signo dentro de la propia memoria integrada en el microchip. Para la programación en C, dependerá del tipo de datos, de las estructuras, de la memoria o los registros que queden libres, y de un sin fin más de variables, con lo que sólo se puede saber cuánto va a tardar una instrucción a posteriori, y por supuesto, dentro de un contexto determinado.

8.1.1 lock_cycles_delay

La función *lock_cycles_delay* mantiene el procesador ocupado mediante un bucle ajustable en tiempo.

```
void lock_cycles_delay (uint N_Cycles);
```

El tiempo durante el cual el procesador va a estar bloqueado viene determinado por el parámetro de 16 bits *N_Cycles* que no tiene signo, y se basa en ejecutar un bucle *'for'* durante un número de ciclos establecidos por el parámetro.

El sencillo núcleo de la función se muestra a continuación:

```
    uint i;
    for (i = 0; i<N_Cycles; i++);
```

Por tanto, a priori, tan sólo se tiene que controlar el tiempo que tarda por cada iteración el código generado por el compilador, y ajustar unos valores de ciclos apropiados para que el micro se bloquee durante un intervalo de tiempo que pueda ser determinado.

Sin embargo, nos encontramos con varios problemas. En principio, y siguiendo todas las recomendaciones que se vierten en los manuales y que han sido recogidas en

este documento, se intenta codificar esta función usando datos sin signo de ocho bits, pero con estos datos, sólo es posible iterar dentro del bucle 256 veces.

Con este número de iteraciones, se puede dejar el micro ocupado como máximo, un valor de tiempo cercano al medio milisegundo, que puede parecer un valor razonable cuando se trata de programaciones orientadas enteramente a las comunicaciones y protocolos entre dispositivos microelectrónicos, pero que es, a todas luces, insuficiente en el caso en el que se tenga que interactuar con dispositivos mecánicos y, el peor caso de todos, con humanos. La primera tentativa puede ser anidar esta función en un bucle que itere x veces, y multiplicar el tiempo máximo por el número de veces que se ha llamado a la función. Pero esto es rotundamente erróneo, puesto que no sólo la función *lock_cycles_delay* va a consumir ciclos de reloj, sino que los bucles que programemos y que se sean capaces de repetir la llamada a la función también van a disponer de un número determinado de ciclos de reloj para llevar a cabo sus tareas de comparaciones e incrementos. Además el paso de parámetros a las funciones no es gratuito. Por lo tanto, la solución tenía que llegar de alguna otra manera.

Se codificó entonces la función usando valores de 16 bits, a pesar de que, a priori, es aún más difícil de predecir qué tiempo va a tardar en ejecutarse cada parte de la función, puesto que los parámetros se han de dividir en dos paquetes de ocho bits, trabajar con ellos, cambiar de byte cuando se desborde cierto contador, etc.… Este trabajo lo podemos hacer de forma manual, con lo que el compilador va a tener que volver a interpretar cada una de nuestras instrucciones, o dejar que genere su propio código. En este caso, puesto que la función es bastante sencilla, el código generado va a ser casi con toda probabilidad el código más rápido, así que se optó por dejar que el compilador decidiese, y mediante la traducción a ensamblador, comprobar qué código se ha generado y cuántos ciclos se van a consumir por cada llamada a la función.

Así se realizó el experimento, y se obtuvo un código que generaba, de forma fija, una sección de 29 ciclos máquina ($7{,}87ms$ con nuestro oscilador) y con cada iteración, 12 ciclos máquina adicionales ($2{,}98ms$). Además, cada vez que se supere un múltiplo de 256, el sistema ha de incrementar algún registro donde tenga almacenado el byte alto del índice de la función, consumiendo un ciclo extra.

Con estos datos, y de forma general, podemos establecer que la conversión de microsegundos a número de ciclos necesarios viene determinada por la fórmula:

$$7{,}87 + 2{,}98 \cdot N + \left(\lceil N / 256 \rceil \cdot 0{,}2713\right)$$

donde N representa el número que se ha pasado como parámetro.

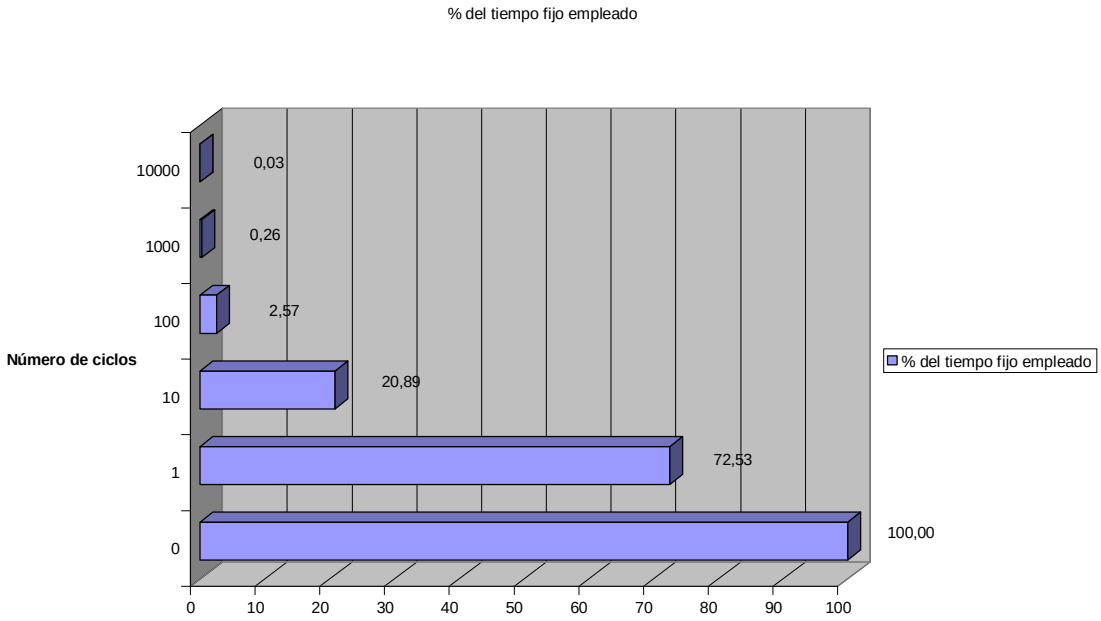

Tabla 2. Porcentaje de tiempo fijo

Esta fórmula no es fiable al ciento por ciento, por que si el parámetro que se pasa es constante, o se puede resolver en tiempo de compilación, el optimizador de código puede hacer esta función más corta si, por ejemplo, el parámetro tiene un valor que cabe en 8 bits, o si se prevé que pueda ser llamada desde una función que tenga consumidos los registros, y se tiene que almacenar los datos de manera diferente, consumiendo algún ciclo adicional; pero, en cualquier caso, los valores no van a diferir en exceso de los resultados esperados.

Una última propuesta, aparentemente lógica, es realizar una función que convierta un valor de tiempo, pasado como parámetro, al número de ciclos aproximados necesarios para bloquear el procesador durante ese intervalo. Sin embargo, el tener que operar repetidas veces con números flotantes podría dar lugar a funciones que tarden en ejecutarse algún orden de magnitud más del tiempo durante el que pretendíamos congelar el micro[1].

La solución adoptada es realizar los cálculos de los ciclos a mano, y definirlos como constantes simbólicas dentro del archivo de cabecera *io_lib.h*. Se añaden, por tanto, después de calcularlos:

[1] C51 Primer, pág. 99. Estadísticas de ciclos máquina en operaciones matemáticas básicas.

```
//Time-Machine cycles equivalence constant

#define MS_100  33500
#define MS_20 6701
#define MS_5    1550
#define US_4100 1375
#define US_100  30
```

Así podemos acceder a las definiciones de una forma fácil si hubiera que modificarlas.

Esta función va a servir como base para la definición de estructuras de bloqueo del procesador basadas en segundos y milisegundos pasados directamente como parámetros, sin ninguna conversión.

8.1.2 timer1_init

```
void timer1_init (uint value, uchar enable_irq);
```

Esta función arranca el temporizador marcado como *timer1*, de forma que comienza a funcionar en el modo más genérico posible, contando con dos registros de 8 bits. Este temporizador realiza una cuenta ascendente con cada ciclo del reloj PCLK, es decir cada dos ciclos del reloj del sistema, lo que equivale a un ciclo máquina.

La función realiza una precarga de los registros contadores al valor *value* de 16 bits, con lo que se podrá decidir cuántos ciclos se precisa que se cuente hasta que el bit de desborde se active.

El segundo parámetro de la función determinará si se establece la interrupción asociada al desborde o no. En el caso de que no se permita, se recuerda al usuario la necesidad de limpiar el bit de desborde, ya que no se realiza de forma automática. Para limpiarlo, se puede usar la siguiente instrucción:

```
TCON &= 0x7F;
```

Esta instrucción fuerza un cero en el octavo bit (el más significativo) del registro TCON, que es el bit de desborde.

En el caso de que se opte por una gestión del bit mediante interrupciones, se debe definir en el código una rutina con el siguiente prototipo:

```
static void timer1_isr (void) interrupt 3
```

8.1.3 timer1_expired

```
uchar timer1_expired (void);
```

Esta rutina devuelve en el nombre de su función el valor del bit de desborde. De esta forma, se puede usar para realizar polling cada cierto tiempo y ver si se ha consumido un cierto número de ciclos precargado en el contador. También es posible su uso para establecer esperas activas de la forma:

```
while(¡timer1_expired);
```

8.2 Funciones entrada/salida

De acuerdo con las características de hardware de los sistemas basados en C51, y debido a las limitaciones de programación del compilador, será necesario definir tablas con los pines que definen los puertos del microcontrolador, ya que, al estar definidos como *sfr* (*special function register*) no pueden ser pasados como parámetro ni ser referenciados de forma indirecta. Sólo podremos darle valores tratándolos como si fuera una variable estándar.

Las tablas se van a incluir dentro de la memoria correspondiente al código de programa, por eso van a llevar el modificador *code* al principio de su definición. Esto es así debido a que no podemos desaprovechar la escasa memoria RAM disponibles y porque, si se incluyeran ésta y el resto de tablas necesarias no se dispondría de RAM suficiente. El compilador, en cualquier caso, avisaría al usuario de esa circunstancia, ya que tiene en todo momento control de los trozos de memoria que se tienen ocupados y los que se reservan para las variables.

Las funciones que necesiten acceder a los pines de forma individual, o aquellas que necesiten saber si un pin pertenece o no a alguno de los puertos en concreto, puede hacer uso de estas tablas, que se definen en el módulo *io_functions.c*, y que no deben ser modificados a menos que se cambie el modelo de micro y cambien los pines.

```
//Philips LPC932 port pin locations
//Read the user manual before editing it.
//DO NOT EDIT unless you change the uc model

code uchar hw_port0_pins[] = {3, 26, 25, 24, 23, 22, 20, 19};
code uchar hw_port1_pins[] = {18, 17, 12, 11, 10, 6, 5, 4};
code uchar hw_port2_pins[] = {1, 2, 13, 14, 15, 16, 27, 28};
code uchar hw_port3_pins[] = {9,8};
```

También se permite la posibilidad de que el usuario se defina un conjunto de pines como su propio puerto de 8 ó 16 bits de salida. Son perfectamente configurables

por el usuario, y no será necesario que se guarde ningún tipo de orden ni que pertenezcan a un puerto en concreto. Si no se van a usar, se recomienda que se dejen los que se encuentran detallados por defecto, que se corresponden a los puertos uno y dos, pero que, en ningún caso, se dejen vacías las tablas

```
//user defined byte and int ports
//fill the arrays in spite of you didn't need it

code uchar my_byte_port1_pins[8]={3, 26, 25, 24, 23, 22, 20, 19};
code uchar my_byte_port2_pins[8]={18, 17, 12, 11, 10, 6, 5, 4};

code uchar my_int_port_pins[16]={3, 26, 25, 24, 23, 22, 20, 19,
                                 18, 17, 12, 11, 10, 6, 5, 4};
```

La última facilidad que se ofrece al usuario en este sentido, es poder configurar los pines de entrada y salida de forma individual, y que se puedan agrupar en un conjunto que no tiene por qué ser de ocho ni dieciséis bits. Por ejemplo, si necesitamos tres líneas de salida y tres de entrada, o cualquier otro número arbitrario hasta veintiséis, para cualquier fin, podríamos modificar las siguientes definiciones de tablas que están dentro de *io_functions.c* y que, al igual que las tablas anteriores, y por los mismos motivos por los que se han comentado previamente, se han incrustado dentro de la memoria.

```
//for defined ports for undetermined number of pins
//my_num_outs must be the number of elements of my_out_pins
//my_num_ins must be the number of elements of my_in_pins
//fill this data in spite of you didn't need it

code uchar my_num_outs = 3;
code uchar my_out_pins [] = {1,2,6};

code uchar my_num_ins = 3;
code uchar my_in_pins [] = {3,4,7};
```

Para que todas las funciones descritas a partir de este punto funcionen de manera correcta, es obligatorio que se rellene la tabla *my_pin_io*. Esta tabla representa, con cada valor, la dirección de las comunicaciones en cada pin, asignándoles un carácter determinado de acuerdo con el código que se detalla. Sólo se consideran tres casos: entrada, salida y bidireccional, ya que son los tres casos comunes y que, con toda seguridad, se van a encontrar en los pines de todos los micros basados en C51. También se podría haber tenido en cuenta un cuarto caso, que es la salida a drenador abierto. Para ello, se habría de añadir un cuarto caso posible a la tabla y modificar la función *set_pins*, que se comenta en apartado siguiente.

```
//define the i/o pins
//fill this array with you desired defaults is mandatory!
//Beware of the funcionalities asociated to the pins

//values  0->i/o
//              1->input
//              2->output
//index+1 represents pin number

code uchar my_pin_io[]={2, //pin1      =     P2^0;
                       2,    //pin2      =     P2^1;
                       2,    //pin3      =     P0^0;
                       0,    //pin4      =     P1^7;
                       0,    //pin5      =     P1^6;
                       1,    //pin6      =     P1^5; EXT RESET
                       1,    //pin7 Vss -ignored value-
                       0,    //pin8      =     P3^1; XTAL1
                       0,    //pin9      =     P3^0; XTAL2//CLK OUT
                       0,    //pin10     =     P1^4; EXT IRQ
                       0,    //pin11     =     P1^3; I2C DATA
                       0,    //pin12     =     P1^2; I2C CLOCK
                       2,    //pin13     =     P2^2; SPI MOSI
                       1,    //pin14     =     P2^3; SPI MISO
                       0,    //pin15     =     P2^4; SPI ^SS
                       0,    //pin16     =     P2^5; SPI CLK
                       1,    //pin17     =     P1^1; UART RXD
                       2,    //pin18     =     P1^0; UART TXD
                       0,    //pin19     =     P0^7;
                       0,    //pin20     =     P0^6;
                       1,    //pin21     Vdd -ignored value-
                       0,    //pin22     =     P0^5;
                       0,    //pin23     =     P0^4;
                       2,    //pin24     =     P0^3;
                       2,    //pin25     =     P0^2;
                       2,    //pin26     =     P0^1;
                       0,    //pin27     =     P2^6;
                       0};   //pin28     =     P2^7;
```

8.2.1 set_pins

Es casi imprescindible ejecutar esta función al principio de cualquier programa que queramos hacer funcionar en este micro, y que use las estructuras de entrada/salida personalizadas que esta librería pone a disposición del usuario programador. La función se define como:

```
void set_pins (void);
```

Esta función interpreta la tabla *my_pin_io* que debe encontrarse incrustada en la memoria de código. Mediante la leyenda que se especifica en la definición de la tabla, se especifica el carácter de las comunicaciones en cada micro.

El algoritmo de la función reconoce los pines que se definen en las tablas *hw_portx_pins* y los busca de forma secuencial en nuestra tabla de definiciones para leer su valor, y así modificar los registros de modo de cada puerto PxM1 y PxM2[1].

El usuario debe ser consciente de que, al estar la mayoría de microcontroladores orientados a la máxima integración, y por tanto, teniendo la mayoría de los pines funciones alternativas asociadas a periféricos integrados en la pastilla, la modificación de esta tabla puede tener efectos secundarios no contemplados en un principio, por lo que se recomienda una lectura detallada del manual para evitarlos.

8.2.2 write_out_value

Esta función lee los pines definidos como salida en la tabla *my_out_pins*, que pueden llegar a ser hasta veintiséis, y les asigna el valor Out_Val. La correspondencia entre el valor y los pines definidos es la lógica: la lista de pines comienza con el bit menos significativo (my_out_pins[0]), y se le dará el valor del bit menos significativo de Out_Val. El segundo de los parámetros es la máscara de transmisión.

```
void write_out_value (ulong Out_Val, ulong Mask);
```

El tiempo que tarda en ejecutarse esta función es variable. Como primer parámetro, el número de pines va a ser determinante en el tiempo total de ejecución, ya que hay que buscarlos uno a uno. Otro factor determinante es el orden, ya que la búsqueda de los pines es secuencial.

En primer lugar, se busca el primero de los valores de la tabla my_out_pins[0] por todos los puertos, comenzando por el primer pin del primer puerto, luego el segundo, etc. Posteriormente se busca en el segundo pin de cada puerto, y así hasta encontrarlo. Con el siguiente pin definido se vuelve a ejecutar el proceso desde el principio.

Por tanto, podemos concluir que la ejecución de la función será más rápida cuanto más cercana a los pines más bajos de cada pin esté la tabla que hemos definido. Es decir, los cuatro pines más rápidos en encontrarse serán el primer pin de cada uno de los cuatro puertos disponibles.

[1] Ésta es la nomenclatura usada por Philips y otros importantes fabricantes de semiconductores, y, como tal, está recogido en la mayoría de manuales acerca de la arquitectura 8051. Se recomienda al usuario acceder al manual de cada microcontrolador para el caso de que las nomenclaturas sean diferentes.

De todas formas, el carácter de los argumentos de esta función le hace ser especialmente lenta, ya que tarda más de mil ciclos en darle el formato adecuado a los valores, prepararlos para entrar en el algoritmo interno de búsqueda, y volver a componer los resultados para devolver el flujo de la función de forma adecuada.

Sirva como ejemplo: si la tabla de pines de salida tiene cero argumentos, con lo que ni siquiera se llega a entrar en el algoritmo, la función consume 1582 ciclos máquina. Si definimos un solo pin de salida (el más rápido de localizar, P0.0) La ejecución tarda 1635 ciclos si el bit está enmascarado y 1676 si no lo está y, finalmente, se escribe en el puerto. Los cuatro bits más rápidos de localizar hacen que la función tarde 2036 ciclos (1845 si están enmascarados) y localizar los 8 bits del puerto (232 entradas en el algoritmo) son sólo 2237 si los bits se enmascaran y 3600 si se escriben los ocho. Por tanto, vemos que al escribir 4 bits, el 90% del tiempo se dedica al formateo de datos, y al escribir 8, aproximadamente el 60%.

bits en my_out_pins	Ciclos si bits enmascarados	Ciclos si no enmascarados	% de tiempo en escritura	Observaciones
0	1582	1582	0%	Tiempo de paso de parámetros y preparación de variables.
1	1635	1676	2,44%	bit más rápido de localizar
4	1845	2036	9,38%	4 bits más rápidos de localizar
8	2237	3600	37,86%	8 bits del puerto cero
16	2894	5658	48,85%	16 bits fruto de la unión del puerto cero y el puerto 1

Tabla 3. Ciclos según número de bits

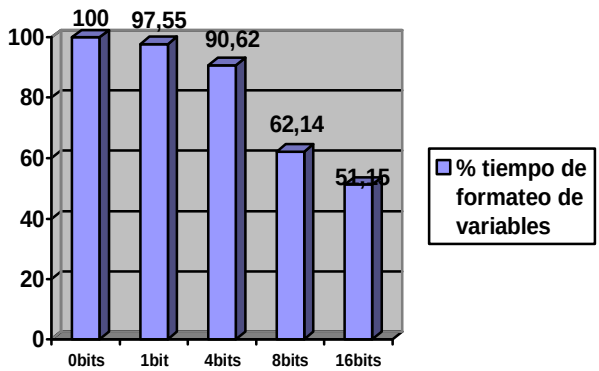

Tabla 4. Tiempo invertido en preparar datos

Por tanto, atendiendo a los resultados, esta función sólo debe usarse cuando no podamos usar cualquiera de las otras que usan los puertos tal y como están predefinidos en el hardware, o bien, si tenemos un cableado que así lo fuerza, las que nos permiten definir puertos de tamaño fijo. El mayor gasto se efectúa mientras que se pasa del formato de 32 bits a 8 bits, a pesar de que la función se encarga en trocearlos y jugar con cada una de las 4 partes del entero largo por separado.

Un ejemplo de uso es el siguiente:

```
…
code uchar my_num_outs = 4;
code uchar my_out_pins [] = {3,18,1,9};
…
void main (void)
{
…
    ulong Out_Data;
…
    write_out_value  (Out_Data, 0x000000FF);
…
```

Es importante destacar que my_num_outs determina el número de salidas, aunque el parámetro que se le pase tenga 32 bits o la máscara tenga más bits a uno, sólo valdrán los 4 menos significativos.

8.2.3 read_in_value

Ésta función es la complementaria a la comentada anteriormente, ya que realiza misma función, pero leyendo en lugar de escribir en los puertos.

```
ulong read_in_value (ulong Mask);
```

Se necesita, para su correcto funcionamiento, tener definido una tabla, al estilo de la necesaria en el punto anterior, pero esta vez que defina el número de los pines que vamos a utilizar como entrada (*my_in_pins*) y un valor que almacene el tamaño de la tabla (*my_num_ins*). Estos valores han de ser definidos previamente por el usuario, ya que están incrustadas en el código, por lo que se resuelve en tiempo de compilación.

En la llamada a la función se pasa como parámetro la máscara de lectura, que indicará cual de los valores del puerto van a ser leídos, y se devuelve el valor leído en un tipo long, de forma que no se limite el número de bits del puerto.

La función funciona internamente, de forma similar a la anterior, por lo que los pines se deben ir buscando uno a uno por los pines de los puertos predefinidos por el hardware. Puesto que el algoritmo interno es una adaptación y se basa en los mismos

principios que el anterior, se leerán antes los pines pertenecientes a los bits bajos de los puertos, siendo primero el P0.0, luego el P1.0 etc…

De esta forma, el test de eficiencia al que se sometió la anterior función es válido también para este, puesto que la función va a tardar lo mismo salvo por pequeñas diferencias que, en cualquier caso, permanecen constantes. Esta función puede usarse del siguiente modo:

```
…
code uchar my_num_ins = 2;
code uchar my_in_pins [] = {3,4};  // P0.1, P1.7
…
void main (void)
{
…
    uchar Input_data;
    Input_data = (uchar) read_in_value (0x0000000F);
…
```

En este caso, sólo los dos bits menos significativos van a ser relevantes en el valor de salida, aunque se asegura que los demás se van a poner a cero. El casting lo realiza el compilador automáticamente, pero se incluye en el ejemplo para aportar claridad.

8.2.4 write_my_int_port_value

```
void write_my_int_port_value (uint Out_Val, uint Mask);
```

Esta función está diseñada para poder escribir en un bus de 16 bits. La particularidad de la misma reside en que, al igual que en las funciones comentadas anteriormente, se permite elegir de forma arbitraria los bits que van a formar parte del puerto sin tener por qué tener ningún tipo de orden ni disposición prefijada.

Para el correcto funcionamiento de la rutina, es necesario que estén definidas todas las tablas que se comentaron al principio de la presente sección, y tener en cuenta que los dieciséis puertos definidos en la tabla *my_in_port_pins*, definidos desde el pin 0 (menos significativo) en adelante son los que se usan para definir el puerto paralelo.

Al igual que ocurriera en las funciones que manejaban datos de 32 bits, la mayor parte del tiempo de ejecución de las funciones se basa en el formateo de datos, tanto a la entrada como a la salida de la función (recordemos que el micro sólo trabaja “cómodamente” con datos de 8 bits).

Suponiendo que se han definido los siguientes pines como puerto, que coinciden con el puerto 0 (el byte bajo) y el puerto 1 (el byte alto):

```
code uchar my_int_port_pins[16] = {3, 26, 25, 24, 23, 22, 20, 19,
                                   18, 17, 12, 11, 10, 6, 5, 4};
```

se realizan una serie de pruebas donde se pretende demostrar, de nuevo, el riesgo para la eficiencia del programa de trabajar con datos distintos de los ocho bits con signo.

Si se enmascaran los ocho bits, una llamada como la siguiente:

```
write_my_int_port_value (0xFFFF, 0x0000);
```

produce un código ensamblador que tarda en ejecutarse la friolera de 1,219 ciclos, que son todos de formateo y preparado de datos, ya que al estar los bits enmascarados, no se procede a buscarlos, por lo que no se consumen ciclos en el algoritmo de búsqueda. Esto se traduce en $0{,}33ms$ con el reloj incorporado en la pastilla de $7{,}37Mhz$. Si por el contrario decidimos escribir todos y cada uno de los dieciséis bits:

```
write_my_int_port_value (0xFFFF, 0xFFFF);
```

se consumen 3606 ciclos de reloj, es decir, casi el triple que en el caso anterior, a pesar de que al cambiar de byte hay, de nuevo, que reorganizar los datos para un tratamiento adecuado. Esto, de nuevo, deja de manifiesto que estas funciones en las que no se utiliza el micro de forma "natural" no son recomendables a no ser que sean de uso estrictamente obligado.

Máscara	Ciclos	Tiempo (µs)	%tiempo de formateo
0x0000	1219	331,27	100%
0xFFFF	3606	1079,55	33,80%
0x00FF	2571	697,47	47,41%
0xFF00	2627	712,60	46,40%

Tabla 5. Escritura de 16 bits libres

La escritura en uno de sólo de los dos bytes que contiene el entero se realiza en unos tiempos muy similares, debido a la forma del algoritmo de búsqueda, que ya ha sido comentado con anterioridad.

Nótese que en ningún caso se asegura la sincronía de los datos en la salida, puesto que se van sacando a medida que el algoritmo de localización de pines los va encontrando.

8.2.5 read_my_int_port_value

Esta función, en los mismos términos que la anterior, realiza la lectura de dieciséis bits definidos en *my_int_port_pins*. Al igual que su homóloga para la escritura, debe tenerse en cuenta el carácter de los pines que se están usando, y deben ser correctamente definidos en la tabla global de pines a través del uso de *set_pins*. En cualquier caso, no se accede a los pines que estén enmascarados, por lo que también es posible jugar con las máscaras para escribir y leer de este puerto definido por el usuario.

```
uint read_my_int_port_value (uint Mask);
```

La llamada a la función necesita que se le pase una secuencia de 16 bits de máscara, que seré el único parámetro que reciba. Asimismo, se devuelve en el nombre de la función el valor leído en un dato de tipo *unsigned int*.

En cuanto al uso de la función, el compilador recurrirá de forma automática a forzar la promoción de datos, por lo que se puede asignar el valor de la función a cualquier tipo de datos, tanto de 4, 16 como 32 bits con o sin signo. En el caso de que la máscara pasada como parámetro sea de ocho bits, el compilador también forzará un byte alto lleno de ceros, puesto que se trata de un tipo de datos sin signo.

8.2.6 write_my_byte1_port_value

En la definición de tablas de este módulo nos encontramos con las dos definiciones siguientes:

```
code uchar my_byte1_port_pins[8]={3, 26, 25, 24, 23, 22, 20, 19};
code uchar my_byte2_port_pins[8]={18, 17, 12, 11, 10, 6, 5, 4};
```

Estas dos tablas definen dos puertos de ocho bits que han sido definidos de forma arbitraria (los valores asignados en la definición son sólo a modo de ejemplo, y se corresponden con los pines de los puerto 0 y 1). Estos puertos pueden ser útiles en el caso de que el cableado o la disposición física del micro en la placa de circuito que lo alberga no permitan acceder directamente a los pines de los puertos definidos en el hardware.

```
void write_my_byte1_port_value (uchar Out_Val, uchar Mask)
```

Siguiendo el modelo de función habitual, se reciben dos parámetros en la llamada a la rutina y no se devuelve ningún valor. Los parámetros recibidos siguen el

orden habitual y tienen el significado usual: el primero de ellos nos da el valor a escribir en el puerto, y el segundo la máscara de transmisión, esta vez en datos de ocho bits.

Una llamada del tipo:

```
write_my_byte1_port_value (0xFF, 0x00);
```

produce un código que consume para ejecutarse 521 ciclos máquina, que se corresponden con 141,33*ms*, con lo que se pueden ver claramente la diferencia que supone el trabajar con un tipo de datos adecuado u otro. En este caso, tan sólo el 5% de los ciclos de la función anterior han sido necesarios para dar formato a los datos al entrar y salir de la rutina.

Para el caso en que todos los bits son buscados y no se enmascara ninguno:

```
write_my_byte1_port_value (0xFF, 0xFF);
```

Esta llamada produce un código de 1873 ciclos, lo que supone que algo más de un 27,8% del tiempo de ejecución (508,07*ms*) se dedica al tratamiento de datos. Esto es ya algo más razonable que los casos anteriores, ya que en medio milisegundo tendremos el valor correcto a la salida de los pines definidos.

A pesar de todo esto, siempre que se puedan usar los puertos predefinidos por el fabricante, capaces de ser escritos con una sólo instrucción, se debe hacerlo.

	8 bits enmascarados	8bits sin enmascarar	tiempo en escritura
ciclos	521	1873	
µs	141,33	508,07	

Tabla 6. Escritura de 8 bits libres

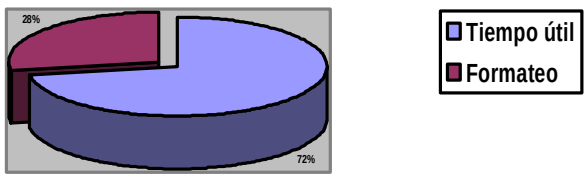

Tabla 7. Distribución de tiempos

8.2.7 read_my_byte1_port_value

Esta función ofrece la correspondiente versión de lectura de la función anterior, y su definición es similar a las anteriores, ya que sigue el mismo convenio de nombrado:

```
uchar read_my_byte1_port_value (uchar Mask);
```

Se le entrega la máscara de recepción en la llegada y se devuelve el valor leído y enmascarado en el nombre de la función.

El algoritmo de búsqueda de pines funciona de forma idéntica, por lo que la descripción de los tiempos de ejecución y la relación y descripción de los mismos no merece mayor comentario.

8.2.8 write_my_byte2_port_value

Esta función es la misma que *write_my_byte1_port_value* salvo que se define el puerto dentro de la tabla *my_byte2_port_pins*.

8.2.9 read_my_byte2_port_value

Se trata del mismo caso que el anterior.

8.2.10 write_hw_port

En esta función se usan directamente los tres puertos de ocho bits que el microcontrolador pone a disposición del diseñador, por lo que aquí no cabe personalización de pines posibles.

```
void write_hw_port(uchar Port_Address, uchar Out_Val, uchar Mask);
```

Como se ha comentado en numerosas ocasiones, el microcontrolador integra tres puertos paralelo de ocho bits, numerados del 0 al 2, además de un puerto de dos bits, que está reconocido como PORT3. Para poder tener una función única para los tres puertos y no necesitar ninguna otra que complique la legibilidad de los programas, se debe pasar como parámetro el puerto en el que queremos realizar el proceso de escritura.

Sin embargo, también hemos comentado que los puertos se definen con la estructura de datos *sfr*, con lo que no podremos pasarlo como parámetro, ni pasarlo como referencia a una función, ni asociarle un puntero.

La solución a este problema es, de nuevo, el uso de tablas estáticas incrustada en la memoria de código. Recordemos que, a pesar de que sólo se puede acceder a ellas a través del contador de programa, con lo que se ralentiza la ejecución de los programas, las variables que se encuentran en la memoria de programa conllevan un gasto cero de la escasa memoria RAM disponible. Es por esta razón que encontramos, en *io_lib.h* ls siguientes líneas:

```
#define P0_ADD  0x80
#define P1_ADD  0x90
#define P2_ADD  0xA0
#define P3_ADD   0xB0

sfr P0   = P0_ADD;  //0x80;
sfr P1   = P1_ADD;  //0x90;
sfr P2   = P2_ADD;  //0xA0;
sfr P3   = P3_ADD;  //0xB0;
```

Por un lado, tenemos definidas, como constantes simbólicas, las direcciones que en el mapa de memoria interno tienen los cuatro puertos a los que podemos acceder. Del mismo modo, y puesto que es imprescindible para poder escribir en ellos, se han definidos los registros especiales (*sfr*) que hacen referencia a dichas direcciones, de modo que podemos leer y escribir de los puertos mediante instrucciones simples de ensamblador, y en lenguaje C como si de una variable normal y corriente se tratara.

Así solucionamos el problema y matamos dos pájaros de un tiro, ya que, como decimos, se puede leer y escribir de los puertos en un solo ciclo máquina, y además podremos pasar las direcciones como parámetros mediante estas constantes sin necesitar de que el usuario sepa las direcciones de los puertos.

Estas direcciones no deberían ser modificadas, puesto que los puertos están recogidos dentro del estándar 80C51, y como tal deberían ser comunes a todos los modelos de todos los fabricantes. En cualquier caso, la modificación es tan sencilla como cambiar el valor de las constantes definidas.

De esta forma, ya tenemos razonada la presencia del primero de los parámetros, la dirección del puerto en el que queremos escribir. Los demás parámetros son los de costumbre: el valor que queremos transmitir al puerto y la máscara de transmisión como último parámetro. En este caso, puesto que así lo obliga la disposición del micro, todos los parámetros son de ocho bits.

Para este caso, en el que los parámetros se acomodan de forma natural a la arquitectura del micro, la diferencia de ciclos gastados en cada escritura es abismal. Una instrucción del tipo

```
write_hw_port(P0_ADD,0xFF, 0xFF);
```

consume tan sólo doce ciclos de reloj, incluyendo los ciclos que se tardan en pasar a registros las variables y ceder y devolver el control de la función. En realidad, las instrucciones de escritura y enmascarado sólo consumen dos ciclos, pero el poder agruparlo en una función, y hacerlo prácticamente independiente del medio físico, salvo por la declaración de unas constantes, bien vale 10 ciclos máquina.

En el peor de los casos, que es pasarle como parámetro el puerto P2, que es la última comparación que realiza la rutina antes de escribir el dato en el puerto, tan sólo se consumen 16 ciclos ($4{,}34ms$). En este caso el enmascarado se hace en una sola instrucción y no evaluando bit a bit como en las anteriores funciones, con lo que el valor de la máscara no influye en la duración de la ejecución de la rutina, que, en ningún caso, llega al 3% del mejor de los casos posibles en los que el puerto está libremente definido por el usuario.

	Tiempo de enmascarado(ms)	Ciclos máquina	Tiempo total (ms)
Puerto 0	0,27	10	2,71
Puerto 1	0,27	13	3,53
Puerto 2	0,27	16	4,34

Tabla 8. Escritura en puertos hardware

La última gran ventaja que presenta esta función es la sincronía, ya que, al poder escribirse directamente sobre un solo registro, el micro pone los valores adecuados en todos los pines de forma simultánea, con lo que podemos tratar con precisión sistemas síncronos de mediana velocidad. Según la hoja de datos, para evitar interferencias y EMI, los flancos tanto de subida como de bajada nunca van a ser menor de $10ns$ [1]. Por tanto, ya tenemos perfectamente definidos los límites a los que se va a tener que ajustar nuestros sistemas que lleven este micro como unidad central de procesos, al definir los tiempos de subida y los tiempos que se tarda en poner un valor en el puerto.

[1] Datasheet P89LPC932 Rev. 04

8.2.11 read_hw_port

Esta rutina, como su propio nombre indica, es equivalente a la anterior, salvo que se usa para la lectura de los puertos, en lugar de para la escritura. La definición es la que se muestra a continuación:

```
uchar read_hw_port (uchar Port_Address, uchar Mask);
```

De nuevo, se ha seguido un sistema lógico de nombrado de argumentos, ya que el orden se respeta con respecto a las anteriores funciones.

El primero de los parámetros es, como la función de escritura, la dirección del puerto donde queremos, en este caso, leer, mientras que el segundo nos determina la máscara de recepción. La función devuelve en su llamada el valor leído una vez enmascarado.

Al igual que lo anteriormente citado, para recoger el valor de un puerto determinado, solo necesitamos una sola instrucción en ensamblador que tarda en ejecutarse un ciclo máquina. A continuación se muestra un ejemplo de cómo se debe usar esta instrucción:

```
void main (void)
{
    uchar in_data;
    …
     set_pins();
    …
     in_data=read_hw_port(P2_ADD,0xFF);
     …
}
```

Éste, que es el peor de los casos, tarda 19 ciclos de reloj. Son pocos más que en la versión de escritura, pero, obviamente, se debe contar con que la función no tiene sentido si no se realiza una asignación de su valor, que también consume ciclos.

De nuevo, nos inclinamos por recomendar el uso de esta función siempre que esté dentro de lo posible, ya que el ahorro de ciclos y la sincronía así parecen manifestarlo.

8.2.12 read_secure_hw_port

Esta función es una ligera modificación de la anterior, en la que se recoge un tercer parámetro para la gestión de los rebotes. De esta forma, podemos asegurarnos que la lectura del puerto es correcta y que no hemos recogido ningún tipo de señal espuria.

```
uchar read_secure_hw_port (uchar Port_Address, uchar Mask,
                           uint Delay_Cycles);
```

Se necesitan tres parámetros para realizar la llamada a la rutina. Los dos primeros son, como en la versión sin rebotes, la dirección del puerto en el cual tenemos que realizar la lectura y la máscara de recepción. En el nombre de la función se devuelve el valor que se ha recogido tras pasar el proceso de la máscara.

El tercer parámetro, como hemos citado, servirá para determinar el carácter seguro de la lectura del puerto. Se interpretará este valor como el número de ciclos máquina (aproximadamente) que han de transcurrir entre dos lecturas secuenciales de la misma dirección, con lo cual podemos establecer un umbral de tiempo entre lectura y lectura, que servirá para ignorar rebotes en la lectura de sensores externos, en especial si se trata de sensores de tipo mecánico o que sean accionados por seres humanos.

La rutina hace uso de la función de espera, que bloquea el microprocesador, *lock_cycles_delay*, comentada en la parte de temporización de la librería, debido a lo cual tenemos la comodidad de disponer de las constantes simbólicas que realizan la equivalencia entre ciclos máquina del procesador y tiempo real consumido (siempre bajo un discreto y razonable margen de error). El usuario siempre dispondrá de la facilidad de interpretar este valor como una constante de tiempo o como una constante de ciclos, con lo que se puede realizar un código más versátil y legible.

Se ha de tener en cuenta que valores muy bajos de este parámetro no tienen sentido, puesto que se van a cumplir con un margen de error demasiado amplio, ya que se pierden ciclos de reloj en la llamada a la función, la conversión de parámetros (ya que la función de retardo recibe una valor de 16 bits) y el tratamiento del resultado. Esto sumaría, dependiendo de si el número de bits del ciclo es superior a 8 bits, o de si se resuelve en tiempo de compilación o de ejecución, alrededor de 10 ciclos adicionales al valor. Por tanto, estaríamos hablando de resultados razonables y cercanos a la realidad a partir de más o menos 100 ciclos de reloj.

8.3 Funciones de transmisión por UART

Para realizar una librería que fuese realmente estándar, se tuvieron que realizar ciertas restricciones, una de ellas es usar tan sólo dos de los modos de transmisión que nos ofrece la uart, los nombrados como modos 1 y 3. La estructura de las tramas en estos modos consta de ocho bits de datos más uno de parada en el primer caso, y de nueve bits de datos en el segundo. La gestión del noveno bit de datos correrá de parte

del usuario, y ninguna de estas funciones va a acceder a él, debido a que se le puede dar múltiples significados, en función de la aplicación en que se use.

Para el acceso a este noveno bit, se usa el cuarto bit de SCON como noveno bit en transmisión, y el tercer bit del mismo registro en recepción.

8.3.1 uart_init_port_irq

Esta función sobrescribe la configuración actual que tengamos de pines y deja los pines usados por la UART del micro como de salida (TxD) y de entrada (RxD), independientemente de la configuración previa que tuviéramos prediseñada.

```
void uart_init_port_irq (uchar Irq_enable, uchar Tx_irq,
                         uchar Br_irq);
```

La función también especifica que interrupciones asociadas a la UART van a ser establecidas y cuáles enmascaradas. Los tres parámetros son:

- Irq_enable: establece o enmascara las interrupciones asociadas a la UART de forma global. Si esta establecida a FALSE, ninguno de los parámetros siguientes tendrá sentido. En cambio, enviando un valor TRUE, vamos a establecer, cuanto menos, la interrupción asociada a la bandera que se activa tras la recepción de un dato. El resto de interrupciones se configura con los siguientes dos valores.
- Tx_irq: Si su valor es TRUE, se establecerá, además de la interrupción en la recepción, la interrupción asociada a la transmisión de datos. Así es como en realidad funcionan las UART convencionales de los dispositivos basados en C51.
- Br_irq: Esta bandera establece el reset del sistema mediante la llegada de una señal de *Break* por el puerto serie. Esta señal consiste en la recepción de once bits consecutivos a valor alto, activando la bandera que indica que una señal de break ha sido detectada. Activando esta interrupción, el vector asociado apunta a una rutina que, si no ha sido modificada o sobrescrita, ni el vector de reset ha cambiado de valor, recibe la nueva programación del micro mediante puerto serie, permitiendo reprogramar la memoria flash del micro (memoria de programa) directamente sobre el sistema. La rutina de carga o *Bootloader* se encuentra en el último de los 8Kbytes de memoria, y se recomienda no sobrescribirla ni cambiar el valor de la tabla de vectores de interrupción que la asocia a la interrupción.

8.3.2 uart_start

La función *uart_start* pone el modo de funcionamiento a la UART, establece su velocidad y la arranca permitiendo el intercambio de datos

```
void uart_start (uchar Uart_mode, uchar Bauds_pow);
```

Como se comentó en anteriores capítulos, sólo vamos a establecer la Uart en los modos 1 y 3, que son los que se suelen usar normalmente, ya que son los que permiten establecer comunicaciones a velocidades estándar de una forma sencilla.

Uart_mode establece el modo, y su valor sólo puede valer 1 y 3. Recordamos que el modo 1 son 8 bits de datos más uno de parada, y que el modo 3 son 9 bits. El noveno bit a transmitir (TB8) se encuentra en el cuarto bit de SCON, y el noveno bit recibido se almacena en el tercer bit del mismo registro (RB8).

Bauds_pow establecerá la frecuencia de baudios con la que vamos a establecer las comunicaciones. Su valor no puede ser mayor a cuatro, y con él se establecerá la velocidad (baudios por segundo) de la siguiente manera:

$$Bps = 2400 \cdot 2^{Bauds\_pow}$$

Así, los valores posibles son:

Bauds_pow	Bps
0	2400
1	4800
2	9600
3	19200
4	38400

Tabla 9. Baudios en función del parámetro *Bauds_pow*

Para valores superiores a 4, y para cualquier otro valor no lógico, así como para valores de modo no contemplado, se tomarán valores por defecto, es decir, puerto funcionando a 2400 bps en modo 1.

A pesar de que el modo 3 se presenta como disponible, en realidad las funciones que se van a describir a continuación no van a tener en cuenta el valor del noveno bit de transmisión o recepción. Esto es así porque no podemos establecer ninguna especificación sobre la funcionalidad de este bit, ya que puede pertenecer a un protocolo

propio, representar un bit de paridad par o impar, o significar cualquier cosa que el usuario programador decida. En cualquier caso, la recepción y la transmisión del bit va a estar permitida si se especificó en el modo, pero el tratamiento de RB8 y TB8 se ha de hacer fuera de las funciones que se incluyen en esta librería. Para facilitar esta tarea, en la cabecera están ambos definidos mediante la estructura ***sbit***, con lo que se podrá leer y escribir en él directamente con instrucciones simples, como si de una variable cualquiera se tratara.

```
sbit TB8  = SCON^3;
sbit RB8  = SCON^2;
```

8.3.3 uart_polling_tx

Llamando a *uart_polling_tx* nos aseguramos la transmisión de un carácter por la uart, ya que se espera hasta el final de la transmisión mediante espera activa (*polling*). Para que la rutina funcione de forma correcta no debe estar activa ninguna interrupción asociada a ella, ya que se chequea la bandera que la activa continuamente, y el resultado puede ser imprevisible si la interrupción la baja, o lee y borra el tampón de recepción, etc.

```
void uart_polling_txd (unsigned char Data);
```

El único parámetro que necesita que se le pase es el carácter a transmitir, y en el nombre de la función no va a devolver ningún valor.

8.3.4 uart_polling_rx

Esta función realiza una lectura del buffer de recepción sólo si la bandera de recepción se encuentra a valor alto.

```
uchar uart_polling_rxd (unsigned char *Data);
```

El carácter se ha de pasar mediante una variable por referencia, ya que el valor que se va a devolver en la función nos va a servir como bandera para saber si la lectura ha sido posible mediante espera activa o alguna interrupción de contador que la verifique cada intervalo de tiempo determinado.

El siguiente código muestra un ejemplo de cómo se ha de usar el valor devuelto:

```
void main (void)
{
    uchar data_rx;
    …
```

```
    while(!uart_polling_rx(&data_rx));
    …
}
```

Este pequeño programa esperaría en el bucle *while* hasta que la función devuelve un valor de uno lógico (en la función internamente la bandera se llama *reading_ok*). Este uno lógico es modificado por el operador de negación, y se fuerza un cero lógico que hace salir del bucle de espera. Mientras la función no lea nada, devolverá cero en la llamada, por lo que permanecerá iterando en el bucle ya que la condición se evalúa como verdadera. Este método presenta una forma sencilla de leer desde la Uart mediante una sola línea de código.

8.3.5 uart_rx_enable

Esta función sólo nos sirve para activar la recepción de datos en la Uart. Es una función extremadamente sencilla que tan sólo activa una bandera, pero, si se especifica como función por sí misma, es por la posibilidad de presentárselo a la capa de software de nivel más alto, ya que es una operación que se va a realizar con frecuencia. Su código es tan simple que sólo tiene que modificar el valor de un registro.

```
void uart_rx_enable(void);
```

8.3.6 uart_rx_disable

```
    void uart_rx_disable(void);
```

Es el mismo caso que el anterior, salvo que ahora se impide la recepción mediante la escritura en la bandera apropiada.

8.4 Funciones de comunicación con el lcd

Las funciones de intercambio de datos con el display van a estar montadas sobre una capa de adaptación, que permitirá establecer un flujo de datos de ocho bits, de forma que la interfaz no necesite ninguna programación adicional por parte del desarrollador. Las funciones que manejan datos de ocho bits van a poder comunicarse con las que tocan el nivel físico, ya que van a formatear esas comunicaciones aportándoles significación a cada carácter y troceándolo y enviándolo convenientemente en bloques de cuatro bits, que será la interfaz habitual de uso, con el único propósito de ahorrar pines y cableado, a costa de mayor lentitud en las comunicaciones.

En estas funciones vamos a usar las definiciones siguientes:

```
#define   DATA  1
#define   INSTRUCTION 0

#define TRUE    1
#define FALSE   0
```

La definición DATA e INSTRUCTION son necesarias para seleccionar el tipo de operación que vamos a efectuar sobre el display, ya que la señal de control RS, dependiendo de su valor, puede hacer referencia al registro de instrucción del display o a la memoria interna, que es la que se representa a la través de la pantalla de cristal líquido.

La siguiente definición (TRUE y FALSE) es la típica representación mediante banderas asociadas a los valores 0 y 1 de la lógica booleana.

También se definen las instrucciones que usamos comúnmente con mediante una constante que la haga sencilla de manejar.

```
#define LCD_CLEAR_DISPLAY  0x01
#define LCD_FIRST_LINE     0x00
#define LCD_SECOND_LINE  0xC0
#define LCD_BLANK 0x20
```

En lo que se refiere a las salidas del micro utilizadas para la comunicación, las funciones de librería van a funcionar siempre que se cumplan dos requisitos:

- Las cuatro líneas del bus de datos que vamos a usar han de pertenecer a un mismo puerto de los definidos en el hardware.
- Los pines han de ser correlativos y estar definidos en la parte alta de los ocho bits del puerto. La correspondencia ha de ser la lógica según el peso de los bits. Así, el bit 7 del puerto ha de estar conectado con el bit 7 del bus de datos del display.

A la hora de definir nuestra cabecera, se ha optado por usar el puerto 0, pero el usuario programador podrá cambiarlo a su antojo por cualquier otro, siempre que se cumplan las dos condiciones anteriores. En cuanto a las señales de control, por simplicidad las hemos asociado al mismo puerto, pero, en caso de que el cableado lo recomendara, se podrán usar pines de cualquier otro puerto, sin más que definirlos en la cabecera.

```
//especial function registers for lcd_com.c
//LCD_CONFIGx must be defined according to LCD_PORT Px
//(registers PxM1 and PxM2)
//Data always in more significative nibble

sfr  LCD_PORT = 0x80;       //P0
```

```
sfr  LCD_CONFIG1 = 0x84;    //P0M1
sfr  LCD_CONFIG2 = 0x85;    //P0M2

sbit LCD_D7 = LCD_PORT^7;
sbit LCD_D6 = LCD_PORT^6;
sbit LCD_D5 = LCD_PORT^5;
sbit LCD_D4 = LCD_PORT^4;
sbit LCD_EN = LCD_PORT^3;
sbit LCD_RW = LCD_PORT^2;
sbit LCD_RS = LCD_PORT^1;
```

Junto con la definición del registro del puerto, se deben definir los dos registros donde se especifican el carácter de los pines con respecto al sentido de los datos. El nombre de estos dos registros va a depender del fabricante de semiconductores, por eso se recomienda referirse al manual de usuario del micro que se esté usando.

En cualquier caso, el definirlos en las cabeceras nos va a permitir independizar ciertas funciones de la capa física, ya que, si bien acceden a ella, no es necesario que se conozcan todos sus parámetros para la que las rutinas se ejecuten perfectamente, como las rutinas de inicialización de los puertos

De esta forma, tan sólo tres funciones van a formar el paquete básico de rutinas asociadas al nivel físico, que son las asociadas única y exclusivamente a la escritura y lectura de datos en formato de nibbles y la que divide los bytes en dos caracteres, y todas las funciones de temporización y control de flujo dependerán de niveles superiores.

8.4.1 lcd_init_ports

La rutina *lcd_init_ports* asocia los valores adecuados a los registros que han sido definidos en el fichero de cabecera de la librería como los registros asociados al lcd.

```
void lcd_init_ports (void);
```

Puesto que se suponen definidos los pines de forma correcta, esta función define el nibble alto del puerto que haya sido definido como entrada salida. Además, suponiendo que, como parece lo más lógico, se habrán asociado las señales de control a los bits bajos del puerto, se les asignará un rol de salida.

Esta configuración de pines sobrescribe la que hubiera anteriormente, y de igual forma, puede ser sobrescrita. En el caso de que las señales de control se asocien a pines de otros puertos, esta función no debe ser usada. Para estos casos, es preferible utilizar *set_pins*, y detallar cada una de las funciones de todos los pines del micro de forma individual.

8.4.2 lcd_nibble_tx

Esta función escribe el dato en el nibble alto del puerto previamente definido en el archivo de cabecera. Presupone, al igual que el resto de las funciones relacionadas con el lcd, que ya se han definido todos los puertos necesarios.

```
void lcd_nibble_tx (uchar Nibble, uchar Rs);
```

Recibe dos argumentos como parámetro. El primero de ellos es un dato de ocho bits que ha de llevar la información útil en los cuatro más altos, por lo que, tal como se recibe, va a ser enmascarado y enviado al puerto. El segundo de los parámetros es el valor de la señal de control Rs, que, recordemos, puede hacer que el dato se almacene en el registro de instrucción o de datos del lcd.

8.4.3 lcd_busy

Esta función se encarga de chequear el bit de ocupado (*busy flag*), que se corresponde con el bit más alto de un carácter leído del registro de instrucción. A pesar de que sólo es necesario el nibble alto, el lcd esperará dos lecturas, ya que el nibble bajo ha de ser leído aunque será descartado.

```
uchar lcd_busy (void);
```

El valor que se devuelve es el de la bandera leída, es decir, devolverá TRUE si el lcd permanece ocupado y FALSE si no lo está y puede recibir instrucciones de nuevo.

Se puede usar para realizar polling al lcd, de la forma:

```
…
while(!lcd_busy());
…
```

Con este ejemplo esperamos el desbloqueo del controlador del lcd, que dependiendo de la instrucción que esté ejecutando, puede llegar a estar más de un milisegundo con la bandera de ocupado activada.

La lectura de la bandera ha dado lugar, en pruebas prácticas, a numerosos problemas cuando el controlador del lcd estaba ejecutando la instrucción CLEAR_DISPLAY. A pesar de que la función es lo suficientemente sencilla como para no tener dudas de su correcto funcionamiento, ya que solo lee del puerto e interpreta el resultado, con esta rutina la bandera a veces se lee a nivel bajo cuando en realidad el micro sigue ocupado. Una de las posibles explicaciones es que esa instrucción es, en realidad una instrucción compleja, e internamente es descompuesta en una secuencia de

instrucciones simples, puesto que lo que se hace es escribir el carácter espacio en todas las posiciones de la memoria del display. Quizás estas instrucciones también alteren el valor de la bandera, pero ni la hoja de datos del modelo de lcd usado ni el estándar en el que está basado advierten de ese hecho. En esos casos, es recomendable realizar una espera de, al menos, dos milisegundos antes de retomar las comunicaciones con el display.

8.4.4 lcd_uchar_tx

Esta función escribe un solo carácter en el display. No comprueba si se sale de rango ni si se produce salto de línea, ya que para eso se incluyen funciones más elaboradas en capas más altas. Su única función es esperar a que el dispositivo se encuentre libre, y posteriormente, desgranar el carácter en dos bloques de cuatro bits. Estos bloques, como ya se ha comentado en repetidas ocasiones, se van a transmitir por los cuatro bits más significativos de los dos caracteres enviados a la función encargada de transmitir los *nibbles*.

```
void lcd_uchar_tx (uchar Data, uchar Rs);
```

El uso principal de esta función, cuando se llama directamente desde el programa principal, es el envío de instrucciones al controlador CMOS del display. Para ello, la instrucción se pasa como el primer parámetro (Data) y como segundo se pasa el valor apropiado de la señal de control del display Rs que apunta al registro de instrucción. También se pueden llamar dentro de las funciones de formateo de datos, capaces de interpretar una cadena y enviarla al display carácter a carácter.

Como ejemplo, para limpiar la pantalla del lcd podríamos usar esta función de la siguiente manera:

```
lcd_uchar_tx (LCD_CLEAR_DISPLAY, INSTRUCTION);
```

ya que tenemos definidos como constantes simbólicas tanto la instrucción, como el valor necesario en Rs para activar el registro de instrucción. Si por el contrario queremos usarla para enviar una cadena de caracteres se podría codificar algo como:

```
…
uchar i;
uchar* cadena = “Hola”;
…
for (i=0; i<5;i++)
    lcd_uchar_tx (*(cadena+i), DATA);
…
```

9 FUNCIONES INDEPENDIENTES DEL MEDIO FÍSICO

9.1 Funciones de temporización

Basadas en la función comentada en el capítulo anterior, que se basa en el control de la temporización mediante un bucle *for* vacío, se definen las siguientes funciones, a las que se les puede pasar como parámetro, directamente, el valor en segundos o milisegundos que queremos bloquear el microcontrolador. No hay motivos lógicos para realizar esta función también con un argumento de microsegundos, puesto que las conversiones de datos, ya que estas funciones trabajan con argumentos de 16 bits, así como el tiempo de paso de parámetros, hacen que el márgen de error se lo suficientemente grande en determinados valores como para desaconsejar vehementemente su uso.

9.1.1 lock_ms_delay

```
void lock_ms_delay (uchar Val_ms);
```

Esta función recibe como parámetro un valor de ocho bits que indica el número de milisegundo que queremos esperar. Debido a su longitud, el rango es de 0 a 255 milisegundos en bloqueo.

La rutina realiza, a nivel interno, conversiones, comparaciones y multiplicaciones de parámetros a dieciséis bits, por lo que se pierde un tiempo que puede influir en la calidad de la aproximación que usa.

Llamada	Ciclos	Tiempo	Error relativo
lock_ms_delay(0)	51	13,84µs	no aplica
lock_ms_delay(1)	4011	1,088ms	8,8%
lock_ms_delay(10)	39662	12,01ms	21,1%
lock_ms_delay(100)	396178	107,47ms	7,47%

Tabla 10. lock_ms_delay

Como se aprecia en la tabla, el tiempo fijo de la función cuando el parámetro es nulo es poco más de un 1% de la sensibilidad de la función, por lo que los errores se aplican al tratamiento de los datos. Vemos que hay un valor que difiere del resto en cuanto al error del resultado, que parece demasiado grande. Este error se ha repetido tras

varias compilaciones, y la explicación reside en el tratamiento de los datos que hace el compilador. Si un dato ha de usar más de un registro, el valor más bajo que lo necesite va a sufrir un incremento importante en su error relativo, pero pasará desapercibido para valores más altos del parámetro.

En cualquier caso, y siempre que se necesitaran valores con más exactitud, se puede recurrir al método de prueba-error al compilar y simular y acotar los resultados.

9.1.2 lock_s_delay

```
void lock_s_delay (uchar Val_s);
```

La funcionalidad es similar a la rutina anterior, salvo que ahora el parámetro enviada estipula el número de segundos que se espera al micro.

En esta función no se trabaja con valores de dieciséis bits, pero se establecen estructuras de control para realizar todas las esperas posibles, ya que no es posible cubrir tal cantidad de tiempo con una sóla llamada a la función *lock_cycles_delay*. De nuevo, volvemos a tener que jugar con un márgen de error en las resoluciones de la función.

Llamada	Ciclos	Tiempo	Error relativo
lock_s_delay(0)	19	<10μs	no aplica
lock_s_delay(1)	3961557	1,0746s	7,5%
lock_s_delay(10)	39615218	10,7460s	7,5%

Tabla 11. lock_s_delay

Como podemos apreciar, el error cometido, al contrario que en el caso anterior, permanece más o menos constante. La función realiza un ciclo de un bucle *for* por cada 100ms, lo que incluye una comparación y un decremento, que, al trabajar con valores de 8 bits, se realizan en tres ciclos máquina, por lo que su valor es ridículo en comparación con el valor del parámetro, y de ahí se deduce que el error que hemos obtenido es, en realidad, el error relativo que produce una llamada del tipo

```
lock_cycles_delay (MS_100);
```

que son la base sobre la cual se ha montado el código de la función que estamos analizando.

9.2 Funciones de entrada/salida

El conjunto de funciones de entrada/salida que aquí comentamos se basan en la definición de los puertos del hardware que tenemos predefinidos en el micro. No se recomienda realizar escrituras de cadenas de caracteres con los puertos determinados pin a pin por el usuario, puesto que puede perderse el control de la temporización, y sólo debe usarse bajo una completa supervisión del programador, supervisión que se vuelve casi imposible si las funciones de acceso a los puertos se encuentran a un nivel diferente que las funciones que pretendemos usar. Por esta razón, sólo se han incluido en la librería las funciones que acceden a los puertos que parece lógico que se usarán para transmitir cantidades considerables de datos de forma secuencial, como cadenas de caracteres que pueden ser latcheadas por cualquier otro dispositivo externo.

9.2.1 write_hw_port_string

Con esta función podemos escribir caracteres en uno de los puertos predefinidos en el hardware del microprocesador con un intervalo de tiempo determinado entre cada uno de los caracteres.

```
void write_hw_port_string (uchar Port_address, uchar *Char_seq,
                           uint Delay_cycles);
```

El primero de los parámetros es la dirección del puerto donde queremos escribir. Para estas direcciones, se pueden usar las constantes definidas en la cabecera Px_ADD. El segundo parámetro es la secuencia de caracteres que necesitamos transmitir. No va a tener limitación de tamaño, por lo que es necesario que el último de los caracteres sea un terminador '\0'

Por último, el tercero de los parámetros hace referencia al número de ciclos de espera dentro del bucle for, ya que va a ser el argumento con el que se va a llamar, dentro de la función, a *lock_cycles_delay*. En el fichero de cabecera también están definidas las equivalencias entre el número de ciclos que se consumen en cada bucle y el tiempo total consumido por la llamada, de forma que podamos asegurar el cumplir de forma muy exacta los intervalos de tiempo necesarios entre cada carácter.

9.2.2 read_hw_port_string

Esta función realiza la tarea complementaria a la anterior, permitiendo la lectura de valores de uno de los puertos de ocho bits del hardware.

```
uchar read_hw_port_string (uchar Port_address, uchar *Readed,
                           uchar Max, uint Delay_cycles);
```

Los parámetros son los mismos y tienen el mismo significado que en el caso anterior, salvo un parámetro nuevo *uchar Max*. Puesto que la cadena que se le pase ha de tener un espacio reservado de memoria mediante, por ejemplo, *malloc*, tenemos que especificarle a la función el número máximo de caracteres que puede almacenar la cadena. Con este parámetro podemos especificar el número máximo de caracteres (incluyendo el carácter terminador '\0') que podemos leer del puerto.

En el nombre de la función se devuelve el número de caracteres que han sido tomados del puerto, también incluyendo el carácter terminador. En el caso de que sólo restara un carácter por recibir, la función automáticamente pone el terminador y termina su ejecución.

9.3 Funciones de transmisión por UART

Las funciones de transmisión que se comentan a continuación están fabricadas sobre las funciones *uart_polling_tx* y *uart_polling_rx*, por lo que se basan en la transmisión mediante esperas activas, bloqueando el microcontrolador, mientras éstas se llevan a cabo. Así nos aseguramos que la función se ha ejecutado correcta y totalmente antes de devolver el flujo de programa.

Las dos funciones, lectura y escritura, van a heredar su nombre de las clásicas *printf* y *scanf* de la librería ANSI C *stdio.h*, ya que, en realidad, su comportamiento es similar si definimos la salida estándar de un ordenador personal en la Uart. La principal diferencia reside en que no se podrá llamar con un número indeterminado de argmentos, ya que esa opción no ha sido tenida en cuenta debido a las limitaciones de memoria de los micros C51.

9.3.1 uart_printf

Tal como se ha comentado, la función envía los caracteres mediante bucles de espera activa, asegurándonos el funcionamiento.

```
uchar uart_printf (unsigned char *Char_Sequence);
```

En principio, no hay ningún tipo de limitación en cuanto al número de caracteres a escribir, salvo la memoria del microprocesador, por lo que no se recomienda el uso de esta función con cadenas de caracteres largas, es preferible dividirlas en tareas más pequeñas.

Se devuelve, para posibles usos de control de flujo, el número de caracteres que han sido enviados. El carácter terminador no se incluye puesto que éste no se envía en ningún momento.

9.3.2 uart_scanf

Esta función recoge hasta *Max_readed*-1 caracteres del puerto serie, o bien *Max_readed* si contamos el carácter terminador.

```
uchar uart_scanf (unsigned char *Char_Sequence,
                            unsigned char Max_Readed);
```

En el nombre de la función se devuelve el número de caracteres recibidos, incluyendo el terminador de cadena. Si el penúltimo carácter no es el terminador, la función lo pone automáticamente y termina su ejecución devolviendo el flujo de programa.

Otra de las características de esta función es que habilita la recepción de la UART cuando entra y la deshabilita cuando sale. Por tanto, esta función sólo ha de usarse cuando se sepa en qué intervalos va a responder el otro interlocutor (por ejemplo, habilitándolo mediante un señal del micro, estableciéndolo como esclavo dentro de un mapa de memoria, etc.). Para el resto de los casos en los que no se sepa cuándo se van a recibir los datos, es preferible el manejo de la recepción de la UART mediante interrupciones, pero estos casos son demasiado específicos como para codificar una función genérica que sirva en la mayoría de los casos.

9.4 Funciones de comunicación con el lcd

Las funciones que se detallan a continuación van a tratar los datos y parámetros como bytes completos, y no cuatro bits como en las funciones base, descritas en el capítulo anterior. Para que estas rutinas funcionen adecuadamente necesitan las definiciones previas citadas en el capítulo anterior, ya que estas funciones usan las anteriores para funcionar.

9.4.1 lcd_init

Antes de poder usar el lcd, debemos ejecutar este código, que se encarga de llamar al inicializador de puertos (*lcd_init_ports*), y posteriormente, se ocupa de ejecutar una rutina que envía al display todas las instrucciones de inicialización necesarias para que se interpreten correctamente los futuros accesos al dispositivo.

```
void lcd_init (void);
```

Si hemos inicializado o asignado los puertos de forma manual, no debemos usar esta función, puesto que pisará la configuración que tengan los puertos definidos en la sección del fichero de cabecera donde se define la dirección del lcd. En lugar de llamar a esta función, se debe usar *lcd_init_instructions*, que también es llamada desde *lcd_init*, que sólo se encarga de poner en los pines las instrucciones de inicialización, sin testear si son correctos o si han sido y están bien definidos.

9.4.2 lcd_init_instructions

```
void lcd_init_instructions (void);
```

Tal como se describe en el punto anterior, esta rutina escribe en el display las instrucciones de inicialización del mismo, sin tener en cuenta ningún aspecto relacionado con la descripción de los pines. Esta función es parte de *lcd_init*, pero se decidió separarla y hacerla independiente para el caso en el que se disponga de una configuración global de los pines usando otras funciones diferentes, como *set_pins*.

Puesto que el lcd tarda un tiempo en procesar cada instrucción, y puesto que no podemos leer el bit de ocupado entre algunas de las instrucciones de inicialización, según recomendaciones del fabricante, entre cada uno de los datos enviados y el siguiente se efectúa un bloqueo del microprocesador de cinco milisegundos, que se corresponde con aproximadamente el triple del tiempo que tarda la instrucción más lenta en ejecutarse y dejar el controlador del lcd libre para recibir más datos.

9.4.3 lcd_sprintf

Esta función escribe una cadena de caracteres en el display, tal como su propio nombre indica, haciendo una analogía con la clásica función *sprintf* incluida en la librería estándar ANSI *stdio.h*.

```
void lcd_sprintf (uchar *(Char_Sequence));
```

La primera instrucción que se ejecuta dentro de la función es dejar el display en blanco mediante la transmisión de la palabra LCD_CLEAR_DISPLAY. Por este motivo, esta función no es recomendable si se desea conservar algo de la información que permanece en la memoria interna del display, y por ende, se encuentra representado en la pantalla.

Posteriormente, la función se dedica a soltar uno a uno los caracteres por el puerto, cambiando de línea del display cuando fuera necesario (en el carácter 16 o cuando se reciba '\n'). Una vez que se cambia de línea, estas dos comprobaciones carecen de sentido, ya que se activa el flag *second_line* para evitar comportamientos indeseados.

Un ejemplo de uso podría ser el siguiente, asumiendo que se han definido correctamente los pines a los que va unido el display:

```
void main (void)
{
…
    string char_seq = "hello world!\nHELLO WORLD";
    lcd_init();
    lcd_sprintf (char_seq);
…
}
```

Con este código se genera la siguiente salida:

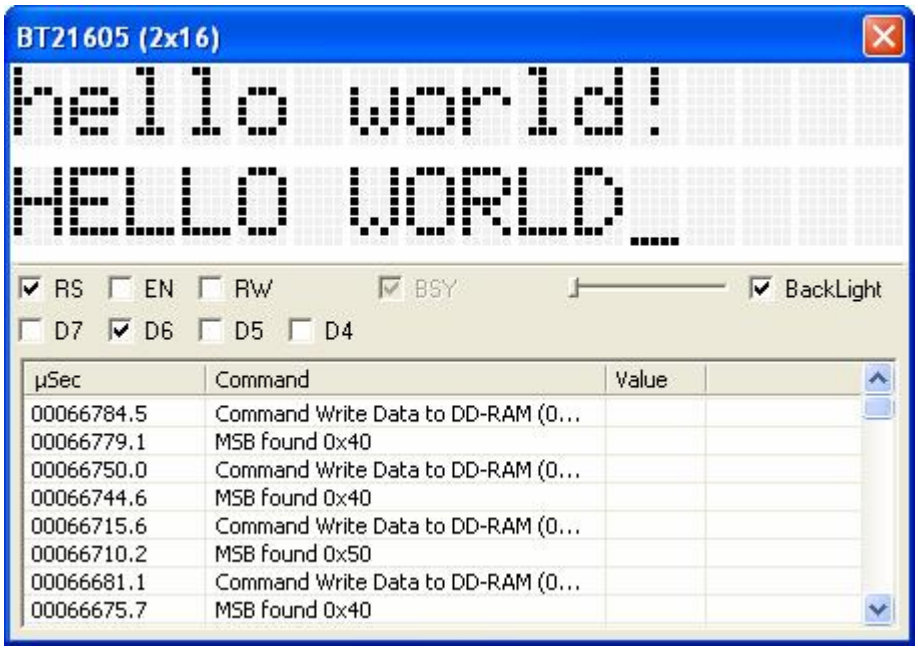

Ilustración 19. Detalle del simulador de lcd

Todas las esperas que son necesarias mientras el controlador del lcd se encuentra ocupado, se realizan mediante lcd_busy, por lo que el procesador se bloquea mientras dure la transmisión. Así, nos aseguramos, cuando la función devuelve el control a la rutina que la invocó, que ya se han completado todas las transmisiones.

Por último, cabe destacar que la ram interna del lcd queda apuntando al siguiente carácter después del último que hemos enviado, puesto que se ha inicializado así, que es

la forma habitual de hacerlo, es decir, que la posición varíe y se actualice con cada nuevo carácter escrito, apuntando siempre al siguiente espacio libre.

9.4.4 lcd_show_line1

Esta rutina borra la primera línea del lcd y escribe en ella un mensaje de máximo 16 caracteres pasado como parámetro.

```
void lcd_show_line1(uchar *(Message));
```

Para escribir en una sola línea necesitamos seguir una secuencia de acciones determinada. En primer lugar, la rutina sitúa la posición de la RAM interna del lcd al principio de la línea en la que queremos escribir, en nuestro caso, la primera.

Para borrar el contenido de la línea, simulamos una instrucción de limpieza de la pantalla, que escribe el código del espacio en blanco en todas las posiciones de la RAM, pero tan sólo en la primera línea.

Por último, volvemos a situar el cursor sobre el primero de los caracteres de la línea y escribimos el argumento que se pasa en la llamada a la función, hasta un máximo de 16 caracteres.

La diferencia entre esta función y la anterior puede verse en el siguiente ejemplo:

```
void main (void)
{
    …
    lcd_init();
    lcd_sprintf ("This is line 1!\nThis is line 2");
    lcd_show_line1("Line1 Rewrited");
    …
}
```

La instrucción lcd_sprintf deja el lcd con el siguiente aspecto:

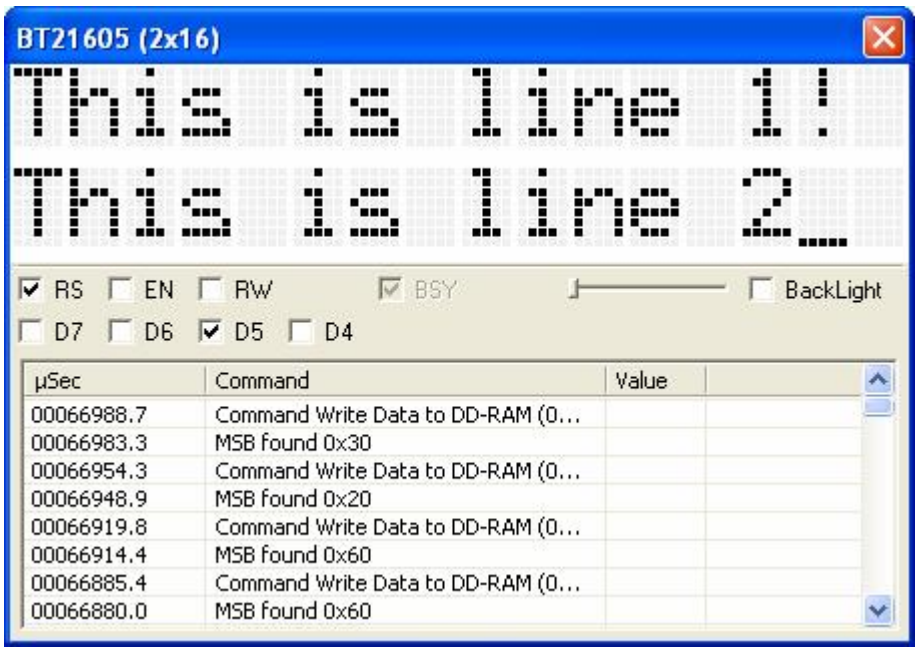

Ilustración 20. Simulación lcd

Una vez que devuelve el flujo al módulo principal, se llama a lcd_show_line1, cuya ejecución nos permite obtener el siguiente resultado:

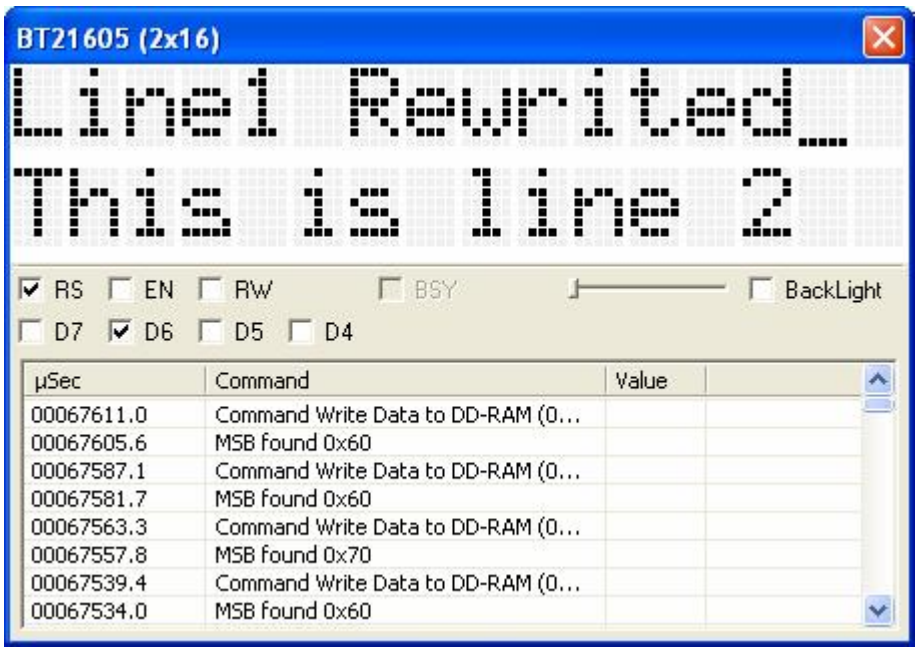

Ilustración 21. Simulación lcd

Vemos cómo el cursor permanece al final de la primera línea, por lo que habrá que recolocarlo si se quiere escribir en un sitio diferente.

9.4.5 lcd_show_line2

También será útil una función que escriba dos argumentos en la segunda línea del display. De esta forma podremos tener la primera línea del display funcionando a modo de “encabezado” y manejar una selección de opciones en la segunda.

```
void lcd_show_line2(uchar *(Message1), uchar *(Message2));
```

El funcionamiento es similar al de la función anterior, ya que primero necesitamos colocar el cursor al principio de la segunda línea, y posteriormente “limpiarla” a base de escribir espacios en blanco.

El proceso de escritura comienza dejando un espacio en blanco en el primero de los caracteres, y posteriormente escribe hasta un máximo de siete caracteres en el lcd pertenecientes al primer argumento de la función. A partir del espacio número nueve y hasta el dieciséis, se trata del espacio reservado para el segundo de los parámetros. El procedimiento es el mismo: el primer carácter que se escribe es un espacio en blanco, y a continuación, se escriben hasta siete caracteres, si los hubiere, correspondientes al segundo argumento.

La razón de dejar un espacio en blanco delante de cada cadena escrita es para tener la posibilidad de escribir un carácter de selección, ya que esta función está pensada para, escribiendo un encabezado en la primera línea, dejar la segunda para representación de dos opciones a elegir por un usuario.

Si se pretende usar el lcd para cualquier otro fin, se recomienda el uso de la función *lcd_sprintf*, que, recordemos, usa las dos líneas, pero limpia el dispositivo antes de cualquier lectura.

El siguiente código muestra el resultado final de usar los ejemplos anteriores y añadirles dos parámetros en la segunda línea.

```
void main (void)
{
    …
    lcd_init();
    lcd_sprintf ("This is line 1!\nThis is line 2");
    lcd_show_line1("Line1 Rewrited");
    lcd_show_line2("op1", "op2");
    …
}
```

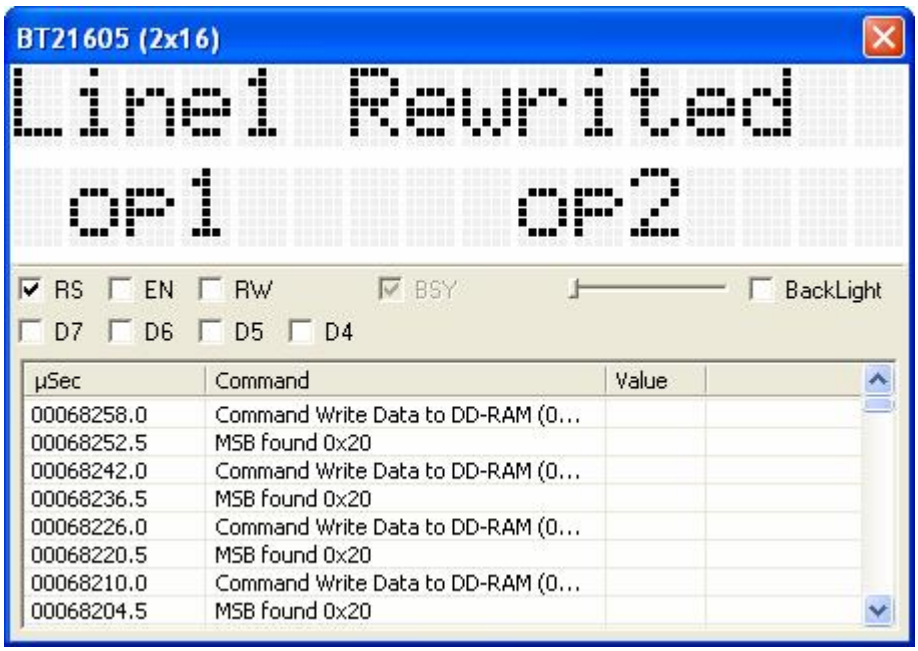

Ilustración 22. Simulación lcd

10 CASO DE ESTUDIO. IMPLEMENTACIÓN DE UN SISTEMA INTERACTIVO DE MENÚS

10.1 Introducción

Usando algunas de las funciones de la librería que acabábamos de crear, se debe poder codificar un programa basado en los siguientes preceptos:

- Se obtienen datos a través de señales que llegan a uno de los puerto paralelo del microcontrolador.
- La interacción con el usuario se va a realizar a través de la representación de datos mediante un display lcd estándar como el que es objeto de las pruebas de este proyecto.
- Los datos van a llegar de y van poder escribirse en un puerto serie que se comunicará con cualquier dispositivo que admita este protocolo de representación de la información, tanto física como sintácticamente.

Con estos requisitos básicos, se propuso un programa de pruebas, cuyas características se describen a continuación:

- El sistema microcontrolador va a estar conectado a un dispositivo accesible mediante el puerto serie. Este dispositivo va a disponer de una serie de registros internos direccionables y accesibles tanto para lectura como para escritura. Para poder acceder a ellos, se desarrolla un protocolo mediante el cual la UART es responsable de poner las direcciones mediante la transmisión de dos octetos, antes del octeto que contiene los datos. Se van a usar 15 bits para indicar la dirección del registro, guardando el bit más significativo para decidir si se accede en lectura (‘0’) o en escritura (‘1’). El siguiente octeto en la línea será un octeto de datos, bien transmitido desde el microcontrolador al dispositivo externo, o viceversa.
- Existirá un display lcd mediante el cual se mostrará información a un usuario. Con este display podremos armar un sistema de menús con el que podremos interactuar con el usuario.
- La interacción con el usuario se realizará mediante pulsadores conectados a entradas de uno de los puertos paralelo del microprocesador.

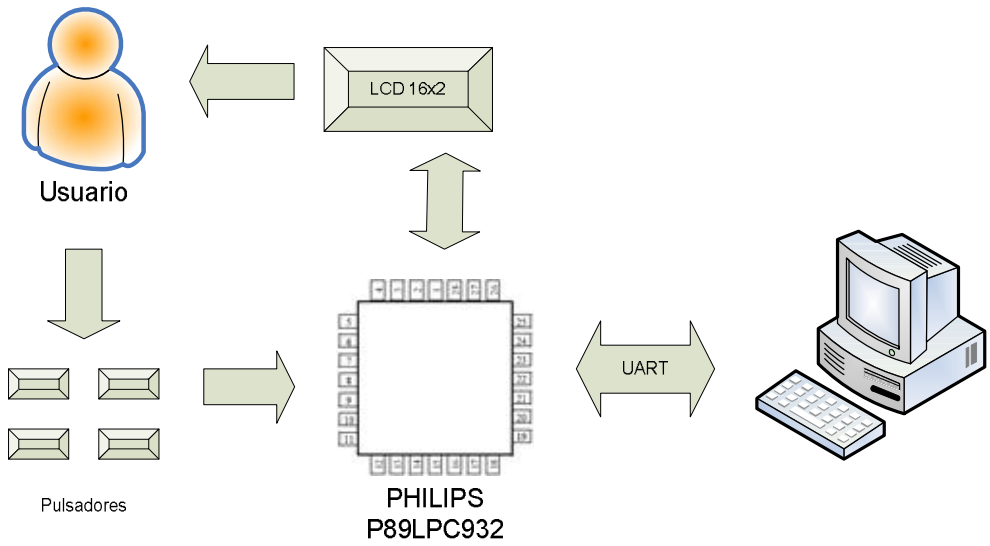

Ilustración 23. Arquitectura hardware del programa

Todas estas características se han de conjugar con las restricciones y las capacidades propias tanto del compilador, como de las características de este micro.
Por ejemplo, todas las cadenas y estructuras constantes serán almacenadas en memoria, y puesto que la memoria de código también es limitada, sólo se compilarán aquellas funciones de la biblioteca que necesitemos.

10.1.1 Formato de la información legible

Para representar cada una de las componentes de que van a constar los nodos de nuestro árbol de menú, primero se diseño cómo van a ser las selecciones de opciones y cómo se tenía que mostrar la información en el lcd, de forma que al usuario le resulta fácil interactuar con el sistema.

Puesto que la matriz lcd cuenta con dos filas, se usó la primera de ellas para indicar el nodo padre, que es el que se visita en ese momento. La segunda línea se reservó para ir mostrando los diferentes nodos hijos a los que se puede acceder desde el mismo. En el caso de los nodos terminales asociados a un valor, en adelante, nodos numérico, se decidió que la primera línea mostrase el parámetro que se estuviese modificando en ese momento, y en la segunda el valor que el usuario puede modificar y asignar.

Los nodos se incrustan en la memoria de código al principio de la ejecución del programa usando las siguientes estructuras:

```
struct tree_node{       string description;
                        uchar num_opt;
```

```
                        string opt1;
                        string opt2;
                        string opt3;
                        string opt4;};

struct number_node{     string description;};
```

La estructura del nodo se escogió de tal forma que se pudieran tener hasta cuatro nodos accesibles, puesto que ese iba a ser el máximo valor en la aplicación que se pensaba codificar. En cuanto a los nodos numéricos, sólo íbamos a necesitar guardar su descripción. Los valores a los que iban relacionados, puesto que tenían que ser usados por otras partes del programa, y puesto que el paso de parámetros a las funciones es un recurso caro, se determinó, aún a costa de dificultar la legibilidad del programa, que se guardaran en variables globales. Su significado se comentará más adelante, en cuanto se comenten los módulos que iban a requerir conocer su valor.

```
//global
uchar uart_pow = 2;
uint global_address = 0x0000;
uchar global_tx_value = 0x00;
uchar global_rx_value = 0x00;
```

10.1.2 Interacción

Los comandos que el usuario debía mandar al microcontrolador, una vez decidida su acción tras la lectura del lcd, iban a venir impuestos por pulsadores que actuaban a modo de pad. Se definieron cuatro botones o pulsadores necesarios, asociados a una funcionalidad concreta.

```
sbit button_left    = P2^4;
sbit button_right   = P2^5;
sbit button_ok      = P2^6;
sbit button_back    = P2^7;
```

- button_left: mediante este comando se puede desplazar la selección de entre las opciones que nos muestra el menú. De igual forma, servirá para disminuir el valor que estemos modificando en el caso de que nos encontremos en un nodo numérico.
- button_right: el funcionamiento de esta orden es similar. Se va a permitir desplazar el menú en la dirección inversa a la que desplaza el comando anterior, y en los nodos numéricos se aumentará el valor de la variable que se estuviera editando.

- button_ok: este botón nos sirve para entrar y activar la opción seleccionada, y en los nodos terminales, aceptar y guardar las modificaciones realizadas a los parámetros.
- button_back: con este comando podremos volver un paso atrás en el árbol de menú, volviendo al nodo previo. También servirá para rechazar las modificaciones que hayan tenido lugar sobre una variable modificable en un nodo Terminal.

10.1.3 Conexión con dispositivos externos

Tal como hemos apuntado con anterioridad, se va a suponer un dispositivo de memoria de acceso secuencial conectado a nuestro sistema microcontrolador mediante el puerto serie. Este dispositivo externo va a ser accesible tanto para lectura como para escritura, y se basa en un protocolo de comunicaciones muy simple:

Este artilugio va a poder ser direccionado mediante 15 líneas de direcciones, es decir, hasta 32 K-posiciones disponibles. Esta dirección va a estar encapsulada en dos bytes que se van a enviar por la uart. Puesto que tenemos un bit libre, éste determinará el carácter de la instrucción, bien lectura, bien escritura, y dependiendo de él, el tercero de los bytes de los que consta cada comunicación simple va a estar puesto en la línea por un dispositivo u otro. En realidad, este proceso simulado podría ser implementado mediante el latcheo de los datos recibidos por la uart, que ponen una dirección de quince líneas en un bus, y el bit que queda puede ser la entrada al pin de '*output enable*' del trozo de memoria.

En cuanto a las características físicas de esta transmisión, se enviará primero el más significativo de los bytes, con el bit más significativo de este byte conteniendo el bit de lectura/escritura. El segundo byte enviado completa la dirección total a la que se está accediendo. Las cadenas estarán compuestas por el bit de arranque o inicio de transmisión más nueve bits. Ocho de ellos contienen los datos y el último indica el final de la transmisión.

10.2 Codificación, ejecución y simulación

10.2.1 Preprocesado y cabeceras

En la cabecera de nuestro programa se incluye, por supuesto, la librería que hemos creado durante el desarrollo de este documento, y se incrustan todos los nodos que vamos a usar en el programa., como:

```
code struct tree_node tx_node = {"Tx: select",
                                 4,
                                 "bps",
                                 "Address",
                                 "Value",
                                 "Send"};
```

Esta estructura tiene seis parámetros:

- El primero de ellos es el encabezado, el nombre del nodo, lo que se mostrará en la primera línea del display cuando accedamos a él desde un nodo de un nivel superior.
- El segundo nos indica cuántas de las cuatro cadenas de caracteres siguientes van a tener un significado lógico, que en este nodo concreto son todas las posibles. En el caso de que alguna de ellas no tuviera utilidad, se debería dejar vacía pero especificada, es decir, el parámetro sería (“”).
- Los cuatro parámetros siguientes indican el número de ramificaciones (hijos), que va a tener el nodo, y la cadena muestra la forma en la que se va a mostrar la opción al usuario.

Para los nodos terminales sin ramificaciones, en los que no se van a mostrar opciones, se especifica otro tipo de estructura:

```
code struct number_node address_node = {"Set address"};
```

Tan sólo incluirán en el código su descripción, puesto que es el único punto estático de los mismos.

10.2.2 Programa principal. Inicialización

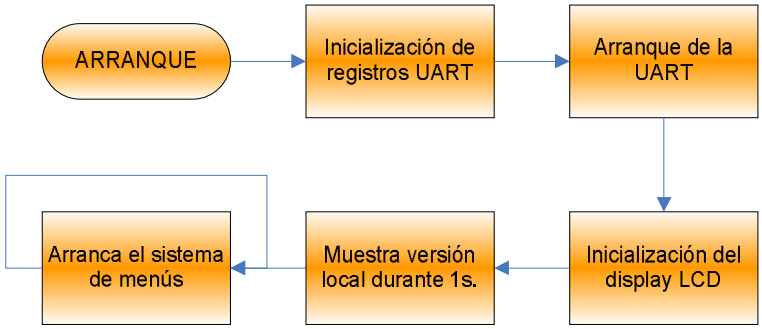

Ilustración 24. Diagrama del programa principal

Una vez que comienza a ejecutar se el código, lo primero que se hace es inicializar los componentes que vamos a necesitar.

- Los pines del puerto 2, a los que van conectados los pulsadores, se va a dejar con la inicialización por defecto, puesto que los define como bidireccionales.
- La uart se inicializa y arranca con las funciones de librería. Los parámetros de la definición del arranque son: no hay interrupciones en transmisión ni recepción y sí permitimos el *Break Detect*, para que el dispositivo pueda ser reprogramado en el mismo sistema. En cuanto a los ajustes del arranque, se usará en modo 1 (8 bits de datos más un bit de pausa), y se hará correr el generador de baudios a 9600 bps. Está será la configuración por defecto, la que se aplicará en la transmisiones salvo que el usuario las modifique expresamente.
- Por último, se inicializa el lcd, enviando las instrucciones de arranque, y se muestra en pantalla, durante un segundo, el identificador local de código con la versión actual.

10.2.3 Arranque del menú

Una vez llegado a este punto, el microcontrolador arranca el sistema de menús llamando a la función oportuna, que ejecuta el código asociado al nodo raíz. Si se examina el código con cierta profundidad, se vislumbra que el resto de las funciones asociadas a otros nodos es muy similar y, a trozos, idéntica, por lo que parece que hubiera sido más óptimo el realizar una función genérica que pudiera ser personalizable mediante el envío de algunos parámetros. Sin embargo, como hemos comentado en repetidas ocasiones, el paso de parámetros en un recurso caro, tanto por la memoria de datos que ocupan los registros y variables, como por el carácter de ciertos tipos de

datos, como los registros, los bits, los punteros a función, etc., a los cuales el compilador les impide ser pasados a una función, o ni tan siquiera indireccionados.

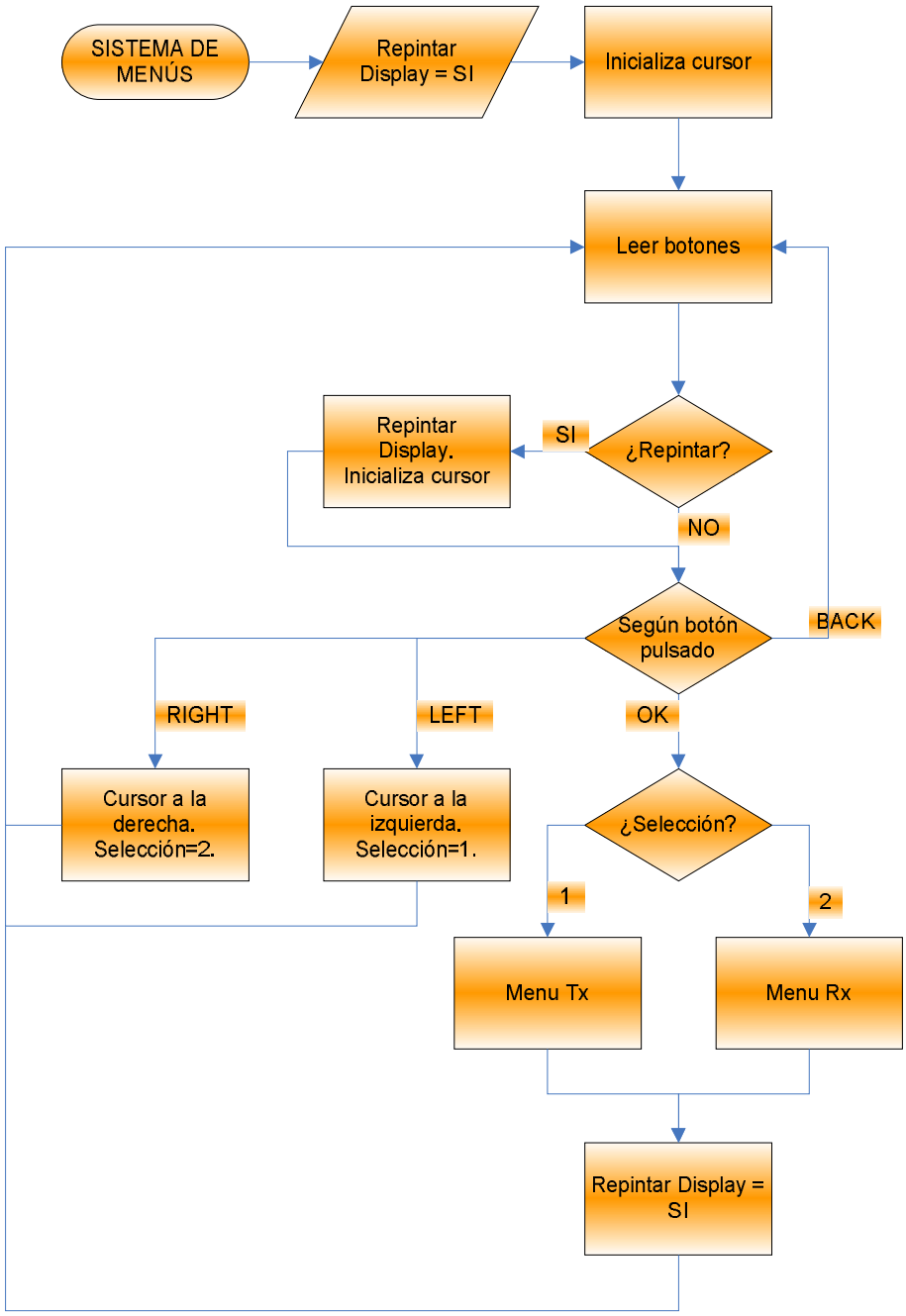

Ilustración 25. Diagrama del sistema de menús

Los demás menús de los nodos, puesto que tienen que devolver el flujo de la función a la rutina que los ha llamado, cuando se pulsa *BACK* o se termina la función que se llama tras pulsar *OK*, terminan en lugar de continuar iterando.

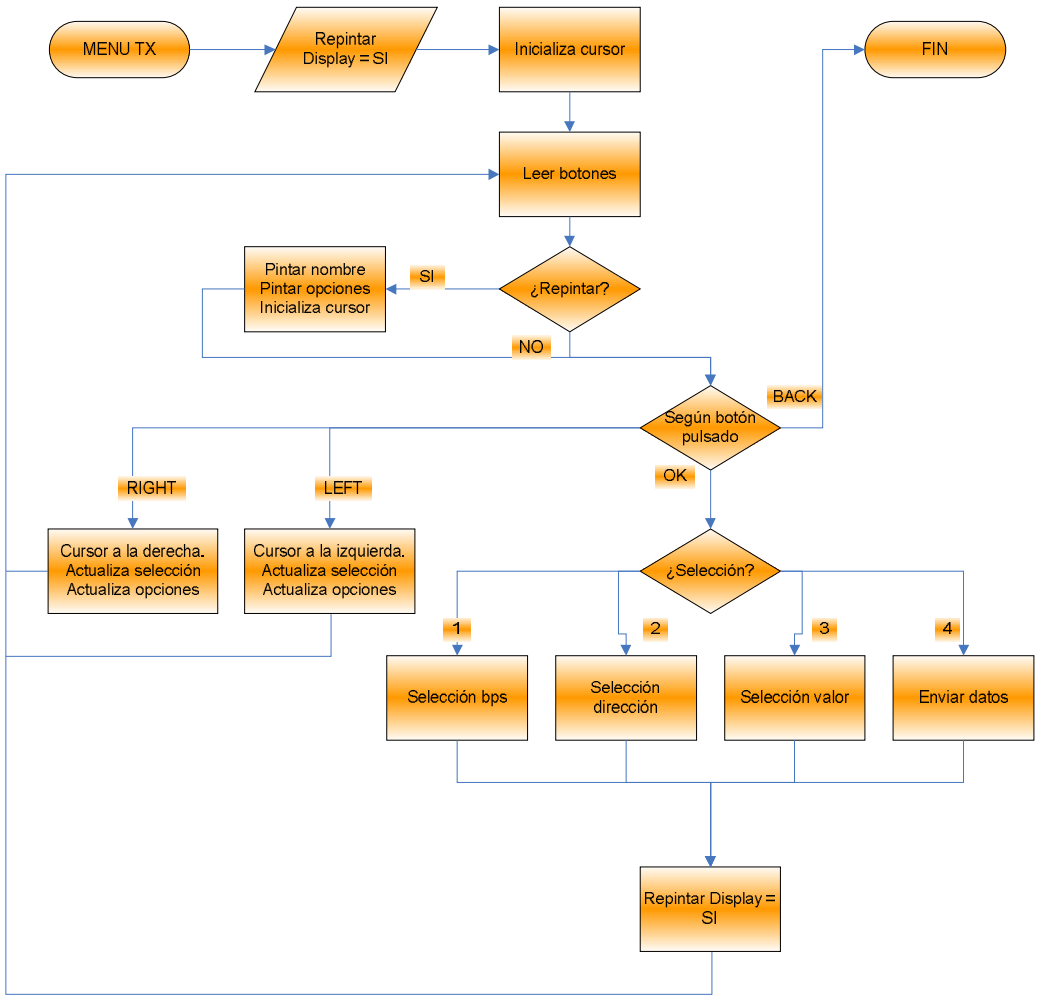

Ilustración 26. Menú de transmisión

Una vez arrancado, el aspecto que muestra el lcd es similar al que sigue:

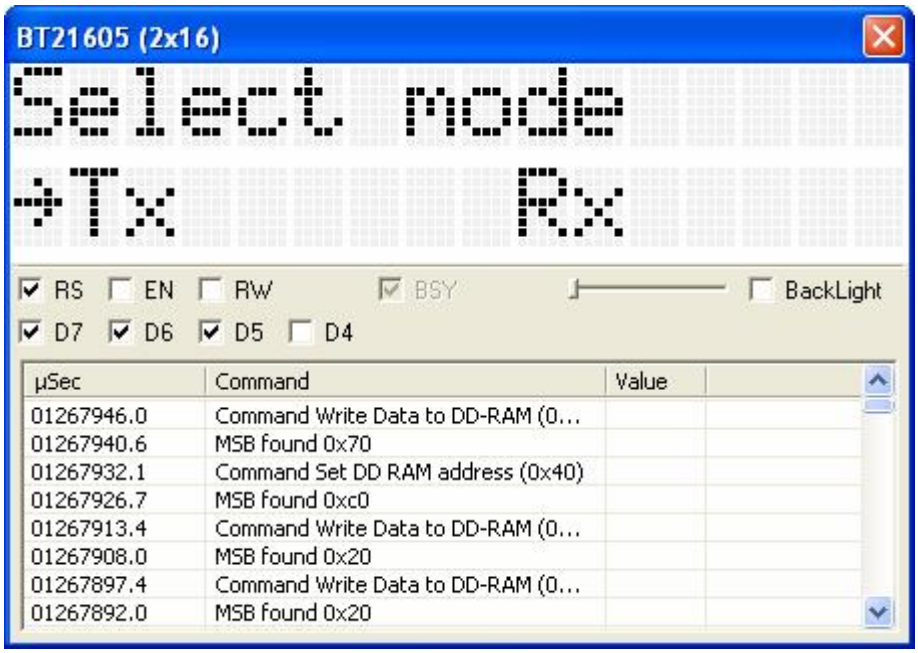

Ilustración 27. Simulación lcd

La flecha que indica el nodo que está siendo seleccionado se representa mediante una función codificada para ese fin, no es parte de la librería. Como la librería limpia las líneas que escribe, esta función debe ejecutarse después de cualquier actualización de la información de la pantalla.

Las funciones se han codificado de tal forma que el punto de regreso tras la llamada a un nodo hijo es exactamente el mismo punto por donde se quedó la ejecución del, de modo que el flujo de programa puede quedar perfectamente definido a primera vista, facilitando la legibilidad del código.

En estos menús, los cuatro botones que se conectan al puerto dos van a ser los responsables de la navegación a través del sistema.

10.2.4 Nodos numéricos y terminales

Por último, solo nos queda por hablar de los nodos terminales. Este tipo de nodos guarda, o bien el valor de una variable, o bien una acción determinada que sólo se ejecuta cuando se activa desde ellos.

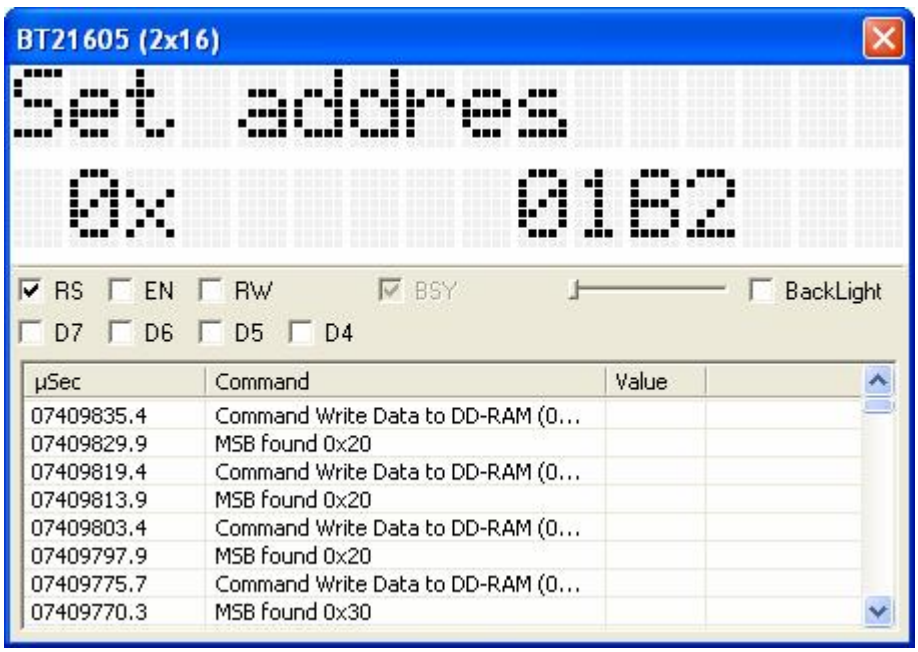

Ilustración 28. Simulación lcd

En el caso de los nodos numéricos, como el que muestra la ilustración, el valor de la variable que se está modificando se incrementa o decrementa de uno en uno, mientras que se pulse los botones direccionales mediante pulsaciones normales. En el caso de que una pulsación dure más de dos décimas de segundo, se considerará que se

trata de una pulsación larga, y el valor se modificará en intervalos de dieciséis cada décima de segundo que permanezca pulsado desde ese momento. Para los nodos en los que sólo se modifiquen dos cifras, el valor de la modificación del valor, para el caso de una pulsación larga, es de ocho.

Los nodos terminales muestran el resultado de una acción cuando han sido invocados. Por ejemplo, una vez que en el menú de transmisión se han especificado los baudios, la dirección del envío y el dato que queremos enviar, disponemos de una opción *'Send'*. Al pulsar el botón de aceptación comenzará la transmisión con los parámetros especificados, y cuando su tarea haya finalizado, mostrará por pantalla el siguiente mensaje:

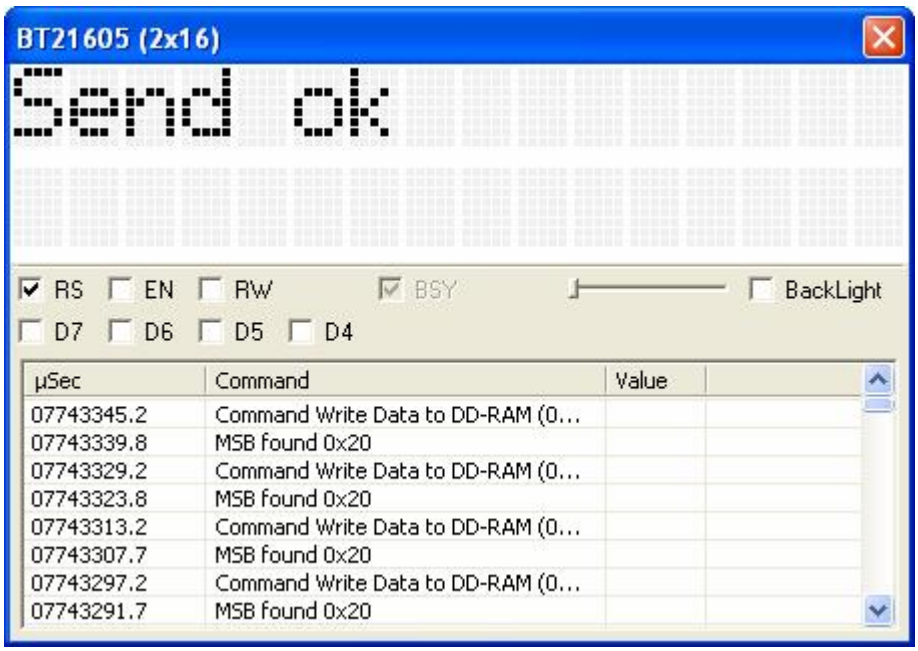

Ilustración 29. Simulación lcd

Este mensaje desaparecerá, para volver posteriormente al menú de opciones del nodo padre, en cuanto se pulse un botón.

10.2.5 Código fuente de la aplicación

10.2.5.1 Preprocesado

```
/*********************************************************/
/*                                                       */
/*  Case study using io_lib. Menu system with options    */
/*                                                       */
/*  Author: Juan A. Quintero                             */
/*                                                       */
```

```
/*  Creation: 2005/06/26                                    */
/*  Last modified: 2005/06/28                               */
/************************************************************/


//header files
#include "../src/io_lib.h"


//definitions and code constants
struct tree_node{       string description;
                        uchar num_opt;
                        string opt1;
                        string opt2;
                        string opt3;
                        string opt4;};

struct number_node{     string description;};

//defined code nodes

code struct tree_node main_node = {"Select mode",
                                   2,
                                   "Tx",
                                   "Rx",
                                   "",
                                   ""};

code struct tree_node tx_node = {"Tx: select",
                                 4,
                                 "bps",
                                 "Address",
                                 "Value",
                                 "Send"};

code struct tree_node rx_node = {"Rx: select",
                                 4,
                                 "bps",
                                 "Address",
                                 "Receive",
                                 "Value"};

code struct tree_node bps_node = {"Set bps:",
                                  4,
                                  "2400",
                                  "4800",
                                  "9600",
                                  "19200"};

code struct number_node address_node = {"Set address"};
code struct number_node value_node = {"Value:"};
code struct number_node send_node = {"Send ok"};
code struct number_node receive_node = { "Receive ok"};


//defined buttons

sbit button_left    = P2^4;
sbit button_right   = P2^5;
sbit button_ok      = P2^6;
sbit button_back    = P2^7;
```

```
//version id

code string local_version = "Local version:\n V0.1b";


//function prototypes

void lcd_set_cursor (uchar pos);
void lcd_set_blank(uchar pos);

void start_menu (void);
void start_tx_menu(void);
void start_rx_menu(void);

void set_bps(void);
void set_address (void);
void send_data(void);
void receive_data(void);
void show_value (void);
void set_value(void);

uchar read_push_buttons(void);
void show_uinttohex(uint Value);
void show_uchartohex(uchar Value);

//global var
uchar uart_pow = 2; //default 9600bps

//shared global vars
uint global_address = 0x0000;
uchar global_tx_value = 0x00;
uchar global_rx_value = 0x00;
```

10.2.5.2 Función principal

```
void main (void)
{

    uchar i;

//Disable tx and rx irq
//Enable reset using break detect

    uart_init_port_irq(FALSE,FALSE,TRUE);

//default uart config
    uart_start(1,uart_pow);  //mode 1, 2400*(2^2)=9600bauds

//init lcd and show local id
    lcd_init();
    lcd_sprintf(local_version);

//wait for a second
    for (i=0; i<10; i++)
        lock_cycles_delay(MS_100);
```

```
//Start menú system
    start_menu();
}
```

10.2.5.3 Funciones representadoras de nodos

10.2.5.3.1 start_menu

```
void start_menu(void)
{
    uchar i=1;
    uchar repaint = TRUE;

    while(1)
    {
        read_push_buttons();

        if (repaint==TRUE)
        {

            lcd_show_line1(main_node.description);
            lcd_show_line2(main_node.opt1,main_node.opt2);
            lcd_set_cursor(1);
            repaint=FALSE;
        }

        if (button_right==TRUE)
        {
            button_right=FALSE;
            lcd_set_blank(1);
            lcd_set_cursor(2);
            i=2;
        }
        else if (button_left==TRUE)
        {
            button_left=FALSE;
            lcd_set_blank(2);
            lcd_set_cursor(1);
            i=1;
        }
        else if (button_back==TRUE)
        {
            button_back=FALSE;
        }
        else if(button_ok==TRUE)
        {
            button_ok=FALSE;
            if(i==1)
                start_tx_menu();
            else
                start_rx_menu();
            repaint=TRUE;

        }
    }
}
```

10.2.5.3.2 start_tx_menu

```
void start_tx_menu(void)
{
    uchar loop=TRUE;
    uchar right_opt;
    uchar i=1;
    uchar selected;
    uchar repaint = TRUE;
    uchar moved_opt = FALSE;

    while(loop)
    {
        if (repaint==TRUE)
        {  lcd_show_line1(tx_node.description);
            lcd_show_line2(tx_node.opt1,tx_node.opt2);
            lcd_set_cursor(1);
            right_opt=2;
            repaint=FALSE;
            moved_opt = FALSE;
        }

        read_push_buttons();

        if (button_right==TRUE)
        {
            button_right=FALSE;
            if(i==1)
            {
                lcd_set_blank(1);
                lcd_set_cursor(2);
                i=2;
            }
            else if (moved_opt==FALSE)
            {

                lcd_show_line2(tx_node.opt3, tx_node.opt4);
                right_opt = 4;
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
                moved_opt=TRUE;
            }
        }
        else if (button_left==TRUE)
        {
            button_left=FALSE;
            if(i==2)
            {
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
            }
            else if (moved_opt == TRUE)
            {
                lcd_show_line2(tx_node.opt1, tx_node.opt2);
                right_opt = 2;
                lcd_set_blank(1);
```

```
                lcd_set_cursor(2);
                i=2;
                moved_opt = FALSE;
            }
        }
        else if (button_back==TRUE)
        {
            button_back=FALSE;
            loop = FALSE;
        }
        else if(button_ok==TRUE)
        {
            button_ok = FALSE;
            selected = (right_opt-(i%2));
            switch (selected)
            {
                case 1:
                    set_bps();
                    break;
                case 2:
                    set_address();
                    break;
                case 3:
                    set_value();
                    break;
                case 4:
                    send_data();
                    break;
            }
            repaint = TRUE;
        }
    }
}
```

10.2.5.3.3 start_rx_menu

```
void start_rx_menu(void)
{
    uchar loop=TRUE;
    uchar right_opt;
    uchar i=1;
    uchar selected;
    uchar repaint = TRUE;
    uchar moved_opt = FALSE;

    while(loop)
    {
        if (repaint==TRUE)
        {
            lcd_show_line1(rx_node.description);
            lcd_show_line2(rx_node.opt1,rx_node.opt2);
            lcd_set_cursor(1);
            right_opt=2;
            repaint=FALSE;
            moved_opt = FALSE;
        }
```

```
        read_push_buttons();

        if (button_right==TRUE)
        {
            button_right=FALSE;
            if(i==1)
            {
                lcd_set_blank(1);
                lcd_set_cursor(2);
                i=2;
            }
            else if (moved_opt==FALSE)
            {

                lcd_show_line2(rx_node.opt3, rx_node.opt4);
                right_opt = 4;
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
                moved_opt=TRUE;
            }
        }
        else if (button_left==TRUE)
        {
            button_left=FALSE;
            if(i==2)
            {
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
            }
            else if (moved_opt == TRUE)
            {
                lcd_show_line2(rx_node.opt1, rx_node.opt2);
                right_opt = 2;
                lcd_set_blank(1);
                lcd_set_cursor(2);
                i=2;
                moved_opt = FALSE;
            }
        }
        else if (button_back==TRUE)
        {
            button_back=FALSE;
            loop = FALSE;
        }
        else if(button_ok==TRUE)
        {
            button_ok = FALSE;
            selected = (right_opt-(i%2));
            switch (selected)
            {
                case 1:
                    set_bps();
                    break;
                case 2:
                    set_address();
                    break;
                case 3:
                    receive_data();
                    break;
```

```
                case 4:
                    show_value();
                    break;
            }
            repaint = TRUE;
        }
    }
}
```

10.2.5.4 Funciones de nodos terminales

10.2.5.4.1 set_bps

```
void set_bps(void)
{
    uchar loop=TRUE;
    uchar right_opt;
    uchar i=1;
    uchar selected;
    uchar repaint = TRUE;
    uchar moved_opt = FALSE;

    while(loop)
    {
        if (repaint==TRUE)
        {
            lcd_show_line1(bps_node.description);
            lcd_show_line2(bps_node.opt1,bps_node.opt2);
            lcd_set_cursor(1);
            right_opt=2;
            repaint=FALSE;
            moved_opt = FALSE;
        }

        read_push_buttons();

        if (button_right==TRUE)
        {
            button_right=FALSE;
            if(i==1)
            {
                lcd_set_blank(1);
                lcd_set_cursor(2);
                i=2;
            }
            else if (moved_opt==FALSE)
            {

                lcd_show_line2(bps_node.opt3, bps_node.opt4);
                right_opt = 4;
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
                moved_opt=TRUE;
            }
        }
```

```
        else if (button_left==TRUE)
        {
            button_left=FALSE;
            if(i==2)
            {
                lcd_set_blank(2);
                lcd_set_cursor(1);
                i=1;
            }
            else if (moved_opt == TRUE)
            {
                lcd_show_line2(bps_node.opt1, bps_node.opt2);
                right_opt = 2;
                lcd_set_blank(1);
                lcd_set_cursor(2);
                i=2;
                moved_opt = FALSE;
            }
        }
        else if (button_back==TRUE)
        {
            button_back=FALSE;
            loop = FALSE;
        }
        else if(button_ok==TRUE)
        {
            button_ok = FALSE;
            selected = (right_opt-(i%2)-1);
            uart_start(1,selected);
            loop = FALSE;
        }
    }
}
```

10.2.5.4.2 set_address

```
oid set_address (void)
{
    uchar loop=TRUE;
    uint address=0x0000;

    lcd_show_line1 (address_node.description);
    show_uinttohex(address);

    while (loop)
    {
        read_push_buttons(); //0,1s entre lecturas
        if(button_right == TRUE)
        {
            button_right=FALSE;
            read_push_buttons();
            if(button_right==FALSE)
            {
                address++;
                show_uinttohex(address);
            }
            while(button_right==TRUE) //pulsación larga
```

```
            {
                if(address<32768-16)
                    address+=16;
                show_uinttohex(address);
                read_push_buttons();
            }
        }
        else if(button_left == TRUE)
        {
            button_left=FALSE;
            read_push_buttons();
            if(button_left==FALSE)
            {
                address--;
                show_uinttohex(address);
            }
            while(button_left==TRUE)
            {
                if(address>16)
                    address-=16;
                show_uinttohex(address);
                read_push_buttons();
            }
        }
        else if(button_back == TRUE)
        {
            button_back =FALSE;
            loop = FALSE;
        }
        else if(button_ok == TRUE)
        {
            button_ok=FALSE;
            loop=FALSE;
            global_address = address;
        }

    }
}
```

10.2.5.4.3 set_value

```
void set_value(void)
{
    uchar loop=TRUE;
    uchar value=0x00;

    lcd_show_line1 (value_node.description);
    show_uchartohex(value);

    while (loop)
    {
        read_push_buttons(); //0,1s entre lecturas
        if(button_right == TRUE)
        {
            button_right=FALSE;
            read_push_buttons();
            if(button_right==FALSE)
```

```
            {
                value++;
                show_uchartohex(value);
            }
            while(button_right==TRUE) //pulsación larga
            {
                if(value<256-8)
                    value+=8;
                show_uchartohex(value);
                read_push_buttons();
            }
        }
        else if(button_left == TRUE)
        {
            button_left=FALSE;
            read_push_buttons();
            if(button_left==FALSE)
            {
                value--;
                show_uchartohex(value);
            }
            while(button_left==TRUE)
            {
                if(value>8)
                    value-=8;
                show_uchartohex(value);
                read_push_buttons();
            }
        }
        else if(button_back == TRUE)
        {
            button_back =FALSE;
            loop = FALSE;
        }
        else if(button_ok == TRUE)
        {
            button_ok=FALSE;
            loop=FALSE;
            global_tx_value = value;
        }

    }
}
```

10.2.5.4.4 show_value

```
void show_value (void)
{
    lcd_show_line1(value_node.description);
    show_uchartohex(global_rx_value);
    while(!read_push_buttons());
}
```

10.2.5.4.5 send_data

```
void send_data(void)
{
    uchar add;

    add = (global_address>>8);
//set MSB=1->tx
//sending address
    add |= 0x80;
    uart_polling_txd (add);

    add = global_address;
    uart_polling_txd (add);

//sending data
    uart_polling_txd (global_tx_value);

    lcd_show_line1(send_node.description);
    lcd_show_line2 ("","");

    while(!read_push_buttons);
}
```

10.2.5.4.6 receive_data

```
void receive_data(void)
{
    uchar add;
    uchar *buffer;

    add = (global_address>>8);
//set MSB=0->rx
//sending address
    add &= 0x7F;

    uart_polling_txd (add);

    add = global_address;
    uart_polling_txd (add);

//receiving data
    while(!uart_polling_rxd (buffer));
    global_rx_value = *buffer;

    lcd_show_line1(receive_node.description);
    lcd_show_line2 ("","");

    while(!read_push_buttons);
}
```

10.2.5.5 Formateo de datos y otros

10.2.5.5.1 show_uinttohex

```
void show_uinttohex(uint Value)
{
    uchar i;
    char sequence[5];

    uchar nibble;

    for (i=0;i<4;i++)
    {
        nibble = ((Value)>>((3-i)*4)&(0x0F));
        if(nibble<10)
            sequence[i]=nibble+48;
        else
            sequence[i]=nibble+55;
    }
    sequence[i]='\0';

    lcd_show_line2("0x",sequence);
}
```

10.2.5.5.2 show_uchartohex

```
void show_uchartohex(uchar Value)
{
    uchar i;
    char sequence[3];
    uchar nibble;

    for (i=0;i<2;i++)
    {
        nibble = ((Value)>>((1-i)*4))&(0x0F);
        if(nibble<10)
            sequence[i]=nibble+48;
        else
            sequence[i]=nibble+65;
    }
    sequence[i]='\0';

    lcd_show_line2("0x",sequence);
}
```

10.2.5.5.3 lcd_set_cursor

```
void lcd_set_cursor(uchar pos)
{
    if (pos==1)
        lcd_uchar_tx(LCD_SECOND_LINE, INSTRUCTION);
    else
        lcd_uchar_tx(LCD_SECOND_LINE+8, INSTRUCTION);

    lcd_uchar_tx(0x7E, DATA);
}
```

10.2.5.5.4 lcd_set_blank

```
void lcd_set_blank(uchar pos)
{
    if (pos==1)
        lcd_uchar_tx(LCD_SECOND_LINE, INSTRUCTION);
    else
        lcd_uchar_tx(LCD_SECOND_LINE+8, INSTRUCTION);

    lcd_uchar_tx(0x20, DATA);
}
```

10.2.5.5.5 read_push_buttons

```
uchar read_push_buttons(void)
{
    uchar readed;
    readed = read_secure_hw_port(P2_ADD,0xF0,MS_100);

    button_left = (readed>>4)&(0x01);
    button_right = (readed>>5)&(0x01);
    button_ok = (readed>>6)&(0x01);
    button_back = (readed>>7)&(0x01);

    return ((readed)&(0xF0));
}
```

11 ANEXO I – CÓDIGO FUENTE DE LA LIBRERÍA

11.1 io_lib.h

```
/*********************************************************/
/*                                                       */
/*  File:   io_lib.h                                     */
/*                                                       */
/*  Author: Juan A. Quintero                             */
/*                                                       */
/*  Description: Header file including type definitions */
/*      and constant values.                             */
/*                                                       */
/*  Creation: 2005/06/18                                 */
/*  Last modified: 2005/06/23                            */
/*********************************************************/




//type definitions

typedef unsigned char uchar;
typedef unsigned int uint;
typedef unsigned long ulong;
typedef char * string;


/*********************************************************/
/*********************************************************/
/* USER EDITABLE PARAMETERS                              */
/*********************************************************/
/*********************************************************/


//especial function registers for lcd_com.c
//LCD_CONFIGx must be defined according to LCD_PORT Px
//(registers PxM1 and PxM2)
//Data always in more significative nibble

sfr  LCD_PORT = 0x80;       //P0
sfr  LCD_CONFIG1 = 0x84;    //P0M1
sfr  LCD_CONFIG2 = 0x85;    //P0M2

sbit LCD_D7 = LCD_PORT^7;
sbit LCD_D6 = LCD_PORT^6;
sbit LCD_D5 = LCD_PORT^5;
sbit LCD_D4 = LCD_PORT^4;
sbit LCD_EN = LCD_PORT^3;
sbit LCD_RW = LCD_PORT^2;
sbit LCD_RS = LCD_PORT^1;


/*********************************************************/
/*********************************************************/
/* Static and hardware mandatory definitions for         */
/*  Philips P89LPC932                                    */
/*  DO NOT EDIT!! unless you change the micro model      */
/*  and read the register location the user manual       */
```

```
/*********************************************************/
/*********************************************************/


//port adress definitions
//common in all 8051 based uc

#define P0_ADD  0x80
#define P1_ADD  0x90
#define P2_ADD  0xA0
#define P3_ADD   0xB0

sfr P0   = P0_ADD;  //0x80;
sfr P1   = P1_ADD;  //0x90;
sfr P2   = P2_ADD;  //0xA0;
sfr P3   = P3_ADD;  //0xB0;

sfr P0M1   = 0x84;
sfr P0M2   = 0x85;
sfr P1M1   = 0x91;
sfr P1M2   = 0x92;
sfr P2M1   = 0xA4;
sfr P2M2   = 0xA5;
sfr P3M1   = 0xB1;
sfr P3M2   = 0xB2;


//uart registers
sfr SCON = 0x98;
sfr SBUF = 0x99;


//SCON bits
sbit SM0  = SCON^7; // alternatively "FE"
sbit FE   = SCON^7;
sbit SM1  = SCON^6;
sbit SM2  = SCON^5;
sbit REN  = SCON^4;
sbit TB8  = SCON^3;
sbit RB8  = SCON^2;
sbit TI   = SCON^1;
sbit RI   = SCON^0;


sfr AUXR1  = 0xA2;
sfr SSTAT  = 0xBA;

sfr BRGR0  = 0xBE;
sfr BRGR1  = 0xBF;
sfr BRGCON = 0xBD;


//IEN0
sfr IEN0 = 0xA8;

sbit EA   = IEN0^7;
sbit EWDRT = IEN0^6;
sbit EBO   = IEN0^5;
sbit ES   = IEN0^4;
sbit ESR  = IEN0^4;
sbit ET1  = IEN0^3;
```

```
sbit EX1  = IEN0^2;
sbit ET0  = IEN0^1;
sbit EX0  = IEN0^0;



//TCON
sfr TCON = 0x88;

sbit TF1  = TCON^7;
sbit TR1  = TCON^6;
sbit TF0  = TCON^5;
sbit TR0  = TCON^4;
sbit IE1  = TCON^3;
sbit IT1  = TCON^2;
sbit IE0  = TCON^1;
sbit IT0  = TCON^0;


//timer

sfr TAMOD = 0x8F;
sfr TL1 = 0x8B;
sfr TH1 = 0x8D;


//symbolic constant

#define TRUE   1
#define FALSE  0


//defines for RS control signal
#define  DATA  1
#define  INSTRUCTION 0


//used lcd instructions definitions

#define LCD_CLEAR_DISPLAY  0x01
#define LCD_FIRST_LINE     0x80
#define LCD_SECOND_LINE  0xC0
#define LCD_BLANK 0x20


//Time-Machine cycles equivalence constant

#define MS_100  33500
#define MS_20 6701
#define MS_5   1550
#define US_4100 1375
#define US_100  30


//PROTOTYPES

/*******************************************************/
/*                                                     */
/*  Prototypes for io_functions.c                      */
/*                                                     */
/*******************************************************/
```

```
void set_pins (void);
void write_out_value (ulong Out_Val, ulong Mask);
void write_my_int_port_value (uint Out_Val, uint Mask);
void write_my_byte_port1_value (uchar Out_Val, uchar Mask);
void write_my_byte_port2_value (uchar Out_Val, uchar Mask);
ulong read_in_value (ulong Mask);
uint read_my_int_port_value (uint Mask);
uchar read_my_byte1_port_value (uchar Mask);
uchar read_my_byte1_port_value (uchar Mask);
void write_hw_port (uchar Port_Address, uchar Out_Val, uchar
Mask);
uchar read_hw_port (uchar Port_Address, uchar Mask);
uchar read_secure_hw_port (uchar Port_Address, uchar Mask, uint
Delay_Cycles);
void write_hw_port_string(uchar Port_address, uchar
*Char_sequence, uint Delay_cycles);
uchar read_hw_port_string (uchar Port_address, uchar *Readed,
uchar Max, uint Delay_cycles);



/*******************************************************/
/*                                                     */
/*  Prototypes for lcd_com.c                           */
/*                                                     */
/*******************************************************/


void lcd_sprintf (uchar *(Char_Sequence));
void lcd_show_line1(uchar *(Message));
void lcd_show_line2(uchar *(Message1), uchar *(Message2));
void lcd_uchar_tx (uchar Data, uchar Rs);
void lcd_nibble_tx (uchar Nibble, uchar Rs);
void lcd_init (void);
void lcd_init_ports (void);
void lcd_init_instructions (void);
uchar lcd_busy (void);



/*******************************************************/
/*                                                     */
/*  Prototypes serial_com.c                            */
/*                                                     */
/*******************************************************/

void uart_init_port_irq (uchar Irq_enable, uchar Tx_irq, uchar
Br_irq);
void uart_start (uchar Uart_mode, uchar Bauds_pow);

void uart_polling_txd (uchar Data);
uchar uart_polling_rxd (uchar *Data);

void uart_rx_enable(void);
void uart_rx_disable(void);

uchar uart_printf (uchar *Char_Sequence);
uchar uart_scanf (uchar *Char_Sequence, uchar Max_Readed);
```

```
/*******************************************************/
/*                                                     */
/*  Prototypes for timing_functions.c                  */
/*                                                     */
/*******************************************************/

void lock_cycles_delay (uint N_Cycles);
void lock_ms_delay (uchar Val_ms);
void lock_s_delay (uchar Val_s);
void timer1_init (uint value, uchar enable_irq);
uchar timer1_expired (void);
```

11.2 io_functions.c

```
/*******************************************************/
/*                                                     */
/*  File: io_functions.c                               */
/*                                                     */
/*  Author: J. Quintero                                */
/*                                                     */
/*  Description: Functions for read from and write in  */
/*      c51's parallel ports or in user defined ports  */
/*      with any number of bits. Also functions for    */
/*      preparing pins and ser its io-direction.       */
/*                                                     */
/*  Creation: 2005/06/16                               */
/*  Last modified: 2005/06/22                          */
/*******************************************************/

#include "io_lib.h"



//Philips LPC932 port pin locations
//Read the user manual before editing it.
//DO NOT EDIT unless you change the uc model

code uchar hw_port0_pins[] = {3, 26, 25, 24, 23, 22, 20, 19};
code uchar hw_port1_pins[] = {18, 17, 12, 11, 10, 6, 5, 4};
code uchar hw_port2_pins[] = {1, 2, 13, 14, 15, 16, 27, 28};
code uchar hw_port3_pins[] = {9,8};


/*******************************************************/
/*  User mandatory parameter, if functions of this     */
/*  module will be used                                */
/*******************************************************/



//user defined byte and int ports
//fill the arrays in spite of you didn't need it

code uchar my_byte1_port_pins[8] = {3, 26, 25, 24, 23, 22, 20,
19};
```

```
code uchar my_byte2_port_pins[8] = {18, 17, 12, 11, 10, 6, 5, 4};

code uchar my_int_port_pins[16] = {3, 26, 25, 24, 23, 22, 20, 19,
                                   18, 17, 12, 11, 10, 6, 5, 4};


//for defined ports for undetermined number of pins
//my_num_outs must be the number of elements of my_out_pins
//my_num_ins must be the number of elements of my_in_pins
//fill this data in spite of you didn't need it

code uchar my_num_outs = 4;
code uchar my_out_pins [] = {3,18,1,9};

code uchar my_num_ins = 2;
code uchar my_in_pins [] = {3,4};


//define the i/o pins
//fill this array with you desired defaults is mandatory!
//Beware of the funcionalities asociated to the pins


//values  0->i/o
//              1->input
//              2->output
//index+1 represents pin number

code uchar my_pin_io[]={2, //pin1      =     P2^0;
                        2,    //pin2      =     P2^1;
                        2,    //pin3      =     P0^0;
                        0,    //pin4      =     P1^7;
                        0,    //pin5      =     P1^6;
                        1,    //pin6      =     P1^5; EXT RESET
                        1,    //pin7 Vss -ignored value-
                        0,    //pin8      =     P3^1; XTAL1
                        0,    //pin9      =     P3^0; XTAL2//CLK OUT
                        0,    //pin10     =     P1^4; EXT IRQ
                        0,    //pin11     =     P1^3; I2C DATA
                        0,    //pin12     =     P1^2; I2C CLOCK
                        2,    //pin13     =     P2^2; SPI MOSI
                        1,    //pin14     =     P2^3; SPI MISO
                        0,    //pin15     =     P2^4; SPI ^SS
                        0,    //pin16     =     P2^5; SPI CLK
                        1,    //pin17     =     P1^1; UART RXD
                        2,    //pin18     =     P1^0; UART TXD
                        0,    //pin19     =     P0^7;
                        0,    //pin20     =     P0^6;
                        1,    //pin21     Vdd -ignored value-
                        0,    //pin22     =     P0^5;
                        0,    //pin23     =     P0^4;
                        2,    //pin24     =     P0^3;
                        2,    //pin25     =     P0^2;
                        2,    //pin26     =     P0^1;
                        0,    //pin27     =     P2^6;
                        0};   //pin28     =     P2^7;



void set_pins (void)
{
```

```
//Set all the hardware pin i/o
//Functions read from my_pin_io, array that sets the
//pins with the following values:
//  1->input pin
//  2->output pin
//  0->i/o pin (default)


    uchar i;                //index var
    uchar pin_index;  //my_pin_io[] index var

//set port0 pins i/o
    P0 &= 0x00;
    P0M1 = 0x00;            //i/o default value
    P0M2 = 0x00;

    for (i=0; i<8; i++)
    {
          pin_index = (hw_port0_pins[i]-1);

          switch (my_pin_io[pin_index])
          {
                case 1:           //input pin
                      P0M1 |= (1<<i);
                      break;
                case 2:           //output pin
                      P0M2 |= (1<<i);
                      break;
          }
    }

//set port1 pins i/o
    P1 &= 0x00;
    P1M1 = 0x00;            //i/o default value
    P1M2 = 0x00;

    for (i=0; i<8; i++)
    {
          pin_index = (hw_port1_pins[i]-1);

          switch (my_pin_io [pin_index])
          {
                case 1:           //input pin
                      P1M1 |= (1<<i);
                      break;
                case 2:           //output pin
                      P1M2 |= (1<<i);
                      break;
          }
    }

//set port2 pins i/o
    P2 &= 0x00;
    P2M1 = 0x00;            //i/o default value
    P2M2 = 0x00;

    for (i=0; i<8; i++)
    {
          pin_index = (hw_port2_pins[i]-1);
```

```
          switch (my_pin_io [pin_index])
          {
                case 1:           //input pin
                      P2M1 |= (1<<i);
                      break;
                case 2:           //output pin
                      P2M2 |= (1<<i);
                      break;
          }
    }

//set port3 pins i/o. Only two bits least significative
    P3 &= 0x00;
    P3M1 &= 0xFC;           //i/o default value
    P3M2 &= 0xFC;

    for (i=0; i<2; i++)
    {
          pin_index = (hw_port3_pins[i]-1);

          switch (my_pin_io [pin_index])
          {
                case 1:           //input pin
                      P3M1 |= (1<<i);
                      break;
                case 2:           //output pin
                      P3M2 |= (1<<i);
                      break;
          }
    }
}



void write_out_value (ulong Out_Val, ulong Mask)
{

//Set the output value for the user defined array my_out_pins
//it could have up to 26 legal values.
//Thif functions need my_num_outs to be defined.

    uchar i=0;
    uchar loop = TRUE;
    uchar port_pos;
    uchar pin_index;
    uchar out_byte[4];  //to separate long values
    uchar mask_byte[4];
    uchar num_byte = 0;
    uchar offset;
    uchar aux;

//divide 32 bits in 4 bytes, and save it in an array

    out_byte[0] = (uchar) Out_Val;
    out_byte[1] = (uchar) (Out_Val>>8);
    out_byte[2] = (uchar) (Out_Val>>16);
    out_byte[3] = (uchar) (Out_Val>>24);

    mask_byte[0] = (uchar) Mask;
    mask_byte[1] = (uchar) (Mask>>8);
    mask_byte[2] = (uchar) (Mask>>16);
```

```
    mask_byte[3] = (uchar) (Mask>>24);


    while (i<my_num_outs)
    {

//set num_byte and offset (position in the actual byte)

          num_byte = (i/8);
          offset = (i%8);
          aux = (out_byte[num_byte]>>offset)&(0x01);
          if (((mask_byte[num_byte]>>offset)&(0x01))>0)
          {
                pin_index = my_out_pins [i];

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index == hw_port0_pins[port_pos])
                      {
                            P0 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port1_pins[port_pos])
                      {
                            P1 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port2_pins[port_pos])
                      {
                            P2 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if (pin_index == 9)
                      {
                            P3 |= aux;
                            loop = FALSE;
                      }
                      else if (pin_index == 8)
                      {
                            P3 |= (aux<<1);
                            loop = FALSE;
                      }
                }
                loop = TRUE;
          } //end if
          i++;
    } //end while
}



void write_my_int_port_value (uint Out_Val, uint Mask)
{

//Set  the  output  value  for  the  16bits  user  defined  array
//my_int_port_pins*

    uchar i=0;
    uchar loop = TRUE;
    uchar port_pos;
    uchar pin_index;
```

```
    uchar out_byte[2];  //to separate int values
    uchar mask_byte[2];
    uchar num_byte = 0;
    uchar offset;
    uchar aux;


//divide 16 bits in 4 bytes, and save it in an array

    out_byte[0] = (uchar) Out_Val;
    out_byte[1] = (uchar) (Out_Val>>8);

    mask_byte[0] = (uchar) Mask;
    mask_byte[1] = (uchar) (Mask>>8);


    while (i<16)
    {

//set num_byte and offset (position in the actual byte)

          num_byte = (i/8);
          offset = (i%8);

          aux = (out_byte[num_byte]>>offset)&(0x01);

          if (((mask_byte[num_byte]>>offset)&(0x01))>0)
          {
                pin_index = my_int_port_pins [i];

//search for the pins in the hardware defined ports 0,1,2,3

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index == hw_port0_pins[port_pos])
                      {
                            P0 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port1_pins[port_pos])
                      {
                            P1 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port2_pins[port_pos])
                      {
                            P2 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if (pin_index == 9)
                      {
                            P3 |= aux;
                            loop = FALSE;
                      }
                      else if (pin_index == 8)
                      {
                            P3 |= (aux<<1);
                            loop = FALSE;
                      }
                }
                loop = TRUE;
```

```
          } //end if
          i++;
    } //end while
}



void write_my_byte1_port_value (uchar Out_Val, uchar Mask)
{

//Set  the  output  value  for  the  8bits  user  defined  array
my_byte_port1_pins

    uchar i=0;
    uchar loop = TRUE;
    uchar port_pos;
    uchar pin_index;
    uchar aux;


    while (i<8)
    {
          aux = (Out_Val>>i)&(0x01);
          if (((Mask>>i)&(0x01))>0)
          {
                pin_index = my_byte1_port_pins [i];


//search for the pins in the hardware defined ports 0,1,2,3

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index==hw_port0_pins[port_pos])
                      {
                            P0 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port1_pins[port_pos])
                      {
                            P1 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port2_pins[port_pos])
                      {
                            P2 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if (pin_index == 9)
                      {
                            P3 |= aux;
                            loop = FALSE;
                      }
                      else if (pin_index == 8)
                      {
                            P3 |= (aux<<1);
                            loop = FALSE;
                      }
                }
                loop = TRUE;
          } //end if
          i++;
```

```
    } //end while
}


void write_my_byte2_port_value (uchar Out_Val, uchar Mask)
{

//Set  the  output  value  for  the  8bits  user  defined  array
my_byte_port1_pins

    uint i=0;
    uchar loop = TRUE;
    uchar port_pos;
    uchar pin_index;
    uchar aux;


    while (i<8)
    {
          aux = (Out_Val>>i)&(0x01);
          if (((Mask>>i)&(0x01))>0)
          {
                pin_index = my_byte2_port_pins [i];


//search for the pins in the hardware defined ports 0,1,2,3

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if(pin_index==hw_port0_pins[port_pos])
                      {
                            P0 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port1_pins[port_pos])
                      {
                            P1 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port2_pins[port_pos])
                      {
                            P2 |= (aux<<port_pos);
                            loop = FALSE;
                      }
                      else if (pin_index == 9)
                      {
                            P3 |= aux;
                            loop = FALSE;
                      }
                      else if (pin_index == 8)
                      {
                            P3 |= (aux<<1);
                            loop = FALSE;
                      }
                }
                loop = TRUE;
          } //end if
          i++;
    } //end while
}
```

```
ulong read_in_value (ulong Mask)
{

//Get the input value for the user defined pin array my_in_pins
//it could have up to 26 legal pin values.
//Returns readed input bits.
//This functions needs my_num_ins to be defined.

    uchar i=0;
    ulong in_val = 0;
    uchar loop = TRUE;
    uchar port_pos;
    uchar pin_index;
    uchar in_byte[4];  //to separate long values
    uchar mask_byte[4];
    uchar num_byte = 0;
    uchar offset;
    uchar aux;

//divide 32 bits in 4 bytes, and save it in an array

    in_byte[0] = 0;
    in_byte[1] = 0;
    in_byte[2] = 0;
    in_byte[3] = 0;


    mask_byte[0] = (uchar) Mask;
    mask_byte[1] = (uchar) (Mask>>8);
    mask_byte[2] = (uchar) (Mask>>16);
    mask_byte[3] = (uchar) (Mask>>24);

    while (i<my_num_ins)
    {

//set num_byte and offset (position in the actual byte)

          num_byte = (i/8);
          offset = (i%8);

          if (((mask_byte[num_byte]>>offset)&(0x01))>0)
          {
                pin_index = my_in_pins[i];

//search for the pins in the hardware defined ports 0,1,2,3

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index == hw_port0_pins[port_pos])
                      {
                            aux = ((P0>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port1_pins[port_pos])
                      {
                            aux = ((P1>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if(pin_index==hw_port2_pins[port_pos])
                      {
```

```
                              aux = ((P2>>port_pos)&(0x01)) ;
                              loop = FALSE;
                        }
                        else if (pin_index == 9)
                        {
                              aux = (P3&0x01);
                              loop = FALSE;
                        }
                        else if (pin_index == 8)
                        {
                              aux = ((P3>>1)&0x01);
                              loop = FALSE;
                        }
                  }

                  in_byte[num_byte] |= (aux<<offset);
                  loop = TRUE;

            } //end if
            i++;
      } //end while

      in_val = in_byte[0];
      in_val |= (in_byte[1]<<8);
      in_val |= (in_byte[2]<<16);
      in_val |= (in_byte[3]<<24);

      return in_val; //returns readed input value
}


uint read_my_int_port_value (uint Mask)
{

//Get the input value for the 16 bits user defined pin array
my_int_port_pins
//Returns readed input bits.

      uchar i=0;
      uint in_val = 0;
      uchar loop = TRUE;
      uchar port_pos;
      uchar pin_index;
      uchar in_byte[2];   //to separate int values
      uchar mask_byte[2];
      uchar num_byte = 0;
      uchar offset;
      uchar aux;

//divide 16 bits in 2 bytes, and save it in an array

      in_byte[0] = 0;
      in_byte[1] = 0;

      mask_byte[0] = (uchar) Mask;
      mask_byte[1] = (uchar) (Mask>>8);

      while (i<16)
      {

//set num_byte and offset (position in the actual byte)
```

```
          num_byte = (i/8);
          offset = (i%8);

          if (((mask_byte[num_byte]>>offset)&(0x01))>0)
          {
                pin_index = my_int_port_pins[i];

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index == hw_port0_pins[port_pos])
                      {
                            aux = ((P0>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port1_pins[port_pos])
                      {
                            aux = ((P1>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port2_pins[port_pos])
                      {
                            aux = ((P2>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index == 9)
                      {
                            aux = (P3&0x01);
                            loop = FALSE;
                      }
                      else if (pin_index == 8)
                      {
                            aux = ((P3>>1)&0x01);
                            loop = FALSE;
                      }
                }

                in_byte[num_byte] |= (aux<<offset);
                loop = TRUE;
          } //end if
          i++;
    } //end while

    in_val = in_byte[0];
    in_val |= (in_byte[1]<<8);

    return in_val;  //returns readed input value
}



uchar read_my_byte1_port_value (uchar Mask)
{

//Get the input value for the 8 bits user defined pin array
my_byte_port1_pins
//Returns readed input byte

    uchar i=0;
    uchar in_val = 0;
    uchar loop = TRUE;
```

```
    uchar port_pos;
    uchar pin_index;
    uchar aux;

    while (i<8)
    {

          if (((Mask>>i)&(0x01))>0)
          {
                pin_index = my_byte1_port_pins[i];

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index==hw_port0_pins[port_pos])
                      {
                            aux = ((P0>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port1_pins[port_pos])
                      {
                            aux = ((P1>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port2_pins[port_pos])
                      {
                            aux = ((P2>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==9)
                      {
                            aux = (P3&0x01);
                            loop = FALSE;
                      }
                      else if (pin_index==8)
                      {
                            aux = ((P3>>1)&0x01);
                            loop = FALSE;
                      }
                }

                in_val |= (aux<<i);
                loop = TRUE;
          } //end if
          i++;
    } //end while

    return in_val;
}


uchar read_my_byte2_port_value (uchar Mask)
{

//Get the input value for the 8 bits user defined pin array
//my_byte_port2_pins
//Returns readed input byte

    uchar i=0;
    uchar in_val = 0;
    uchar loop = TRUE;
    uchar port_pos;
```

```
    uchar pin_index;
    uchar aux;

    while (i<8)
    {

          if (((Mask>>i)&(0x01))>0)
          {
                pin_index = my_byte2_port_pins[i];

                for(port_pos=0;(port_pos<8)&&(loop);port_pos ++)
                {
                      if (pin_index==hw_port0_pins[port_pos])
                      {
                            aux = ((P0>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port1_pins[port_pos])
                      {
                            aux = ((P1>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==hw_port2_pins[port_pos])
                      {
                            aux = ((P2>>port_pos)&(0x01)) ;
                            loop = FALSE;
                      }
                      else if (pin_index==9)
                      {
                            aux = (P3&0x01);
                            loop = FALSE;
                      }
                      else if (pin_index==8)
                      {
                            aux = ((P3>>1)&0x01);
                            loop = FALSE;
                      }
                }

                in_val |= (aux<<i);
                loop = TRUE;
          } //end if
          i++;
    } //end while

    return in_val;
}



void write_hw_port (uchar Port_Address, uchar Out_Val, uchar Mask)
{

//Write in a predefined hardware port
//This function needs PX_ADD to be defined

    if (Port_Address == P0_ADD)
        P0 |= (Out_Val & Mask);

    else if (Port_Address == P1_ADD)
        P1 |= (Out_Val & Mask);
```

```
    else if (Port_Address == P2_ADD)
        P2 |= (Out_Val & Mask);

}



uchar read_hw_port (uchar Port_Address, uchar Mask)
{

//Read a predefined hardware port
//This function needs PX_ADD to be defined

    uchar buffer_rx;

    if(Port_Address == P0_ADD)
          buffer_rx = P0;
    else if(Port_Address == P1_ADD)
          buffer_rx = P1;
    else if(Port_Address == P2_ADD)
          buffer_rx = P2;

    return (buffer_rx & Mask);

}


uchar read_secure_hw_port (uchar Port_Address, uchar Mask, uint
Delay_Cycles)
{

//Read a predefined hardware port, and waits Delay_Cycles
//to read again and compares their values.
//Return readed value or cero if error ocurred

    uchar buffer_rx;
    uchar buffer_aux;

    buffer_rx = read_hw_port (Port_Address, Mask);

    lock_cycles_delay (Delay_Cycles);

    buffer_aux = read_hw_port (Port_Address, Mask);

    if (buffer_rx != buffer_aux)
          buffer_rx = 0;

    return (buffer_rx & Mask);

}


void      write_hw_port_string(uchar      Port_address,      uchar
*Char_sequence, uint Delay_cycles)
{
    uchar i=0;

    while(*(Char_sequence+i)!='\0')
    {
        write_hw_port (Port_address, *(Char_sequence+i), 0xFF);
```

```
        lock_cycles_delay (Delay_cycles);
    }
}


uchar  read_hw_port_string  (uchar  Port_address,  uchar  *Readed,
uchar Max, uint Delay_cycles)
{
    uchar i = 0;
    uchar loop = TRUE;

    if (Max>1)
    {
        while(loop)
        {
            *(Readed+i) = read_hw_port(Port_address, 0xFF);
            if(i==Max-2)
            {
                *(Readed+i+1) = '\0';
                loop = FALSE;
                i++;
            }
            else if (*(Readed+i) =='\0')
            {
                loop = FALSE;
            }
            else
            {
                lock_cycles_delay (Delay_cycles);
                i++;
            }
        }
    }
    return (i+1);
}
```

11.3 lcd_com.c

```
/*******************************************************/
/*                                                     */
/*  File: lcd_com.c                                    */
/*                                                     */
/*  Author: Juan A. Quintero                           */
/*                                                     */
/*  Description: Functions for the management of an    */
/*      standart lcd ks0066. You need to ensure a right */
/*      port definition, see it in header file.        */
/*                                                     */
/*  Creation: 2005/06/18                               */
/*  Last modified: 2005/06/26                          */
/*******************************************************/



/*******************************************************/
/*  Some pins of the microcontroller have been         */
/*  redefined in order to have suitable names and code  */
```

```
/*  simplicity.                                                 */
/*                                                              */
/*  You can redefine the pins in header file, but data  */
/*  pins must be placed in the high nibble of the lcd   */
/*  port.                                                       */
/*                                                              */
/*  sfr  LCD_PORT = 0x80; //P0                                  */
/*                                                              */
/*  sfr  LCD_CONFIG1 = 0x84;    //P0M1                          */
/*  sfr  LCD_CONFIG2 = 0x85;    //P0M2                          */
/*                                                              */
/*  sbit LCD_D7 = LCD_PORT^7;                                    */
/*  sbit LCD_D6 = LCD_PORT^6;                                    */
/*  sbit LCD_D5 = LCD_PORT^5;                                    */
/*  sbit LCD_D4 = LCD_PORT^4;                                    */
/*  sbit LCD_EN = LCD_PORT^3;                                    */
/*  sbit LCD_RW = LCD_PORT^2;                                    */
/*  sbit LCD_RS = LCD_PORT^1;                                    */
/*                                                              */
/****************************************************************/


#include "io_lib.h"


void lcd_sprintf (uchar *(Char_Sequence) )
{

//Clear the display and print a string

    uchar i = 0;            //index var
    uchar second_line_flag = 0;


    lcd_uchar_tx (LCD_CLEAR_DISPLAY, INSTRUCTION);
    lock_cycles_delay(MS_5);      //waits instruction ends

//Move to the second line when '\n' received or
//the 16th char is printed

    while (( *(Char_Sequence + i) != '\0')&&(i<32))
    {
          if (*(Char_Sequence +i) == '\n')
          {
                if (second_line_flag == 0)
                {
                      while(lcd_busy());
                      lcd_uchar_tx (LCD_SECOND_LINE, INSTRUCTION);
                 second_line_flag = 1;
                }
          }
          else if (((i & 0x0F) == 0)&&(i!=0))
          {
                if (second_line_flag == 0)
                {
                      while (lcd_busy());
                      lcd_uchar_tx (LCD_SECOND_LINE, INSTRUCTION);
                      second_line_flag = 1;
                      while (lcd_busy());
                      lcd_uchar_tx (*(Char_Sequence + i), DATA);
                 second_line_flag = 1;
```

```
                }
                else
                {
                      while (lcd_busy());
                      lcd_uchar_tx (*(Char_Sequence + i), DATA);
                }

          }

//default. No special treatment: print the char

          else
          {
                while(lcd_busy());
                lcd_uchar_tx (*(Char_Sequence + i), DATA);

          }

          i++;
    }//end while
}


void lcd_show_line1(uchar *(Message))
{

    uchar i;

//set the cursor in the begining of the line

    while (lcd_busy());
    lcd_uchar_tx(LCD_FIRST_LINE, INSTRUCTION);

//write 0x20 -space code- in the first line

    for(i=0; i<16;i++)
    {
          while(lcd_busy());
          lcd_uchar_tx(LCD_BLANK, DATA);
    }

//set the cursor in the begining of the line

    while(lcd_busy());
    lcd_uchar_tx(LCD_FIRST_LINE, INSTRUCTION);

//write the message
    for(i=0; (i<16) && (*(Message+i)!= '\0'); i++)
    {
          while(lcd_busy());
          lcd_uchar_tx(*(Message+i), DATA);
    }
}

void lcd_show_line2(uchar *(Message1), uchar *(Message2))
{
//each argument is preceded of a blank space to
//specify a cursor character if needed

    uchar i;
     uchar finished = FALSE;
```

```
//set the cursor in the first position
    while (lcd_busy());
    lcd_uchar_tx(LCD_SECOND_LINE, INSTRUCTION);

//write 0x20 -space code- in the first line
    for(i=0; i<16;i++)
    {
          while(lcd_busy());
          lcd_uchar_tx(LCD_BLANK, DATA);
    }

//set the cursor in the begining
    while(lcd_busy());
    lcd_uchar_tx(LCD_SECOND_LINE, INSTRUCTION);

//write the first argument

    lcd_uchar_tx(LCD_BLANK, DATA);
    for(i=0;(i<7)&&(!finished);i++)
    {

          if( *(Message1+i) != '\0')
         {
                while(lcd_busy());
                lcd_uchar_tx(*(Message1+i), DATA);

         }
          else
             finished=TRUE;
    }

    for(;i<8;i++)
          lcd_uchar_tx(LCD_BLANK, DATA);

//and the second argument

    lcd_uchar_tx(LCD_BLANK, DATA);
    finished=FALSE;

    for(i=0;(i<7)&&(!finished);i++)
    {

          if( *(Message2+i) != '\0')
         {
                while(lcd_busy());
                lcd_uchar_tx(*(Message2+i), DATA);

         }
          else
             finished=TRUE;
    }

    for(;i<8;i++)
          lcd_uchar_tx(LCD_BLANK, DATA);

}

void lcd_uchar_tx (uchar Data, uchar Rs)
{
```

```
//Send an uchar to the display
//The info is sent in the more significative nibble

    uchar msb_nibble = 0xF0 & (Data);
    uchar lsb_nibble = 0xF0 & (Data << 4);

    while(lcd_busy());      //wait FALSE busy flag


    lcd_nibble_tx (msb_nibble, Rs);
    lcd_nibble_tx (lsb_nibble, Rs);
}



void lcd_nibble_tx (uchar Nibble, uchar Rs)
{

//sent a nibble to the display placed at port defined in LCD_PORT

    LCD_PORT &= 0x0F;             //clear port data
    LCD_RW = 0;             //lcd write enable
    LCD_RS = Rs;

    LCD_EN = 1;             //lcd command/data enabled

    LCD_PORT |= (0xF0 & Nibble);        //write to port 0

    LCD_EN = 0;             //lcd command/data disabled
}


void lcd_init (void)
{
    lcd_init_ports();
    lcd_init_instructions();
}

void lcd_init_ports (void)
{
//Init the port 0 writing the appropiate registers.
//Asumes LCD_PORT

    LCD_PORT = 0x00;        //Initialize the port to zero to avoid
                            //undesiderable effects

    LCD_CONFIG1 = 0x00;     //4  more  significative  bits  defined
                            //in/out
    LCD_CONFIG2 = 0x0F;     //for data interchanging and 4 less
                            //significative defined output for
                            //control
                            //signals
}

void lcd_init_instructions (void)
{
//Send the standart init instruction to the display
//Wait 5ms between them to ensure the correct operation

    lcd_nibble_tx (0x30,INSTRUCTION);
```

```
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x30, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x30, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x20, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x20, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x80, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x00, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x60, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x00, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x10, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0x00, INSTRUCTION);
    lock_cycles_delay (MS_5);

    lcd_nibble_tx(0xC0, INSTRUCTION);
    lock_cycles_delay (MS_5);

}

uchar lcd_uchar_rx(void)
{

     uchar received_data = (LCD_PORT & 0xF0);
     return received_data;
}

uchar lcd_busy (void)
{

//read the busy flag D7

    uchar received_data;
    uchar busy = 0;

    LCD_RS = 0;
    LCD_RW = 1;
    LCD_EN = 1;

    lock_cycles_delay (1);        // respect timing characteristics
    received_data = (LCD_PORT & 0xF0);
    lock_cycles_delay (1);        // respect timing characteristics

    LCD_EN = 0;
```

```
    lock_cycles_delay (1);        //second nibble read cycle
                                        //this nibble is unused
    LCD_EN = 1;
    lock_cycles_delay (1);
    LCD_EN = 0;

    if (received_data >= 128)     // busy if D.7 == 1
          busy = 1;

    return busy;
}
```

11.4 serial_com.c

```
/*********************************************************/
/*                                                       */
/*  File: serial_com.c                                   */
/*                                                       */
/*  Author: J. Quintero                                  */
/*                                                       */
/*  Description: Inicialization, management and data     */
/*      tx/rx uart functions                             */
/*                                                       */
/*  Creation: 2005/06/18                                 */
/*  Last modified: 2005/06/22                            */
/*********************************************************/


#include "io_lib.h"

void uart_init_port_irq (uchar Irq_enable, uchar Tx_irq, uchar
Br_irq)
{
//Init uart pins of P89LPC932. Disable interrupts
//during this function and enable it when
//return the programm flow

    EA = 0;

//set uart ports. Overwrite the pin configuration

    P1M1 &= 0xFC;
    P1M2 &= 0xFD;

//default configuration

    ES = 0;     //uart irq disabled
    SSTAT = 0x00;   //single buffer, tx irq disable
    AUXR1 &= 0xDF;  //break detect disabled


//set uart irq

    if(Irq_enable == TRUE)
          ES = 1;
```

```
    if(Tx_irq==TRUE)
        SSTAT = 0x20;   //Tx interrupt enable
    if(Br_irq == TRUE)
        AUXR1 |= 0x40;  //set break detect irq
}

void uart_start (uchar Uart_mode, uchar Bauds_pow)
{
    uint rate_generator;

    EA = 0;                //disable interrupts


    SCON = 0x50;      //Uart mode 1. RxD Enabled. Set BRGCON
                     //as Baud Rate Generator
    if(Uart_mode == 3)
        SCON |= 0x80; //Uart mode ·

//default: 2400 Bauds
    BRGR0 = 0xC0;     //9600 bauds using local
    BRGR1 = 0x0B;     //oscilator (7.373MHz)

    rate_generator = BRGR0;
    rate_generator += (BRGR1<<8);

    if(Bauds_pow<=4)
        rate_generator = rate_generator>>Bauds_pow;

    BRGR0 = rate_generator;
    BRGR1 = (rate_generator>>8);

    BRGCON = 0x03;    //establish BRG

    EA = 1;           //enable interrupts
}


void uart_polling_txd (unsigned char Data)
{
    SCON &= 0xEF;     //REN = 0
    SBUF = Data;
    while (!TI);      //Wait transmision ends
    TI = 0;
    SCON |= 0x10;     //REN = 1
}

uchar uart_polling_rxd (unsigned char *Data)
{
    uchar reading_ok = FALSE;
    if (RI != 0)
    {
          *Data = SBUF;
          RI = 0;
          reading_ok = TRUE;
    }
    return reading_ok;
}

uchar uart_printf (unsigned char *Char_Sequence)
{
    uchar i;
```

```
    for(i=0; *(Char_Sequence +i) != '\0';i++)
          uart_polling_txd(*(Char_Sequence+i));

    return i; //number of writen chars
}

uchar uart_scanf (unsigned char *Char_Sequence, unsigned char
Max_Readed)
{
    uchar i=0;
    uchar loop = TRUE;

//Max_Readed including '\0'

    if( Max_Readed >1)
    {

          uart_rx_enable();

          while (loop)
          {
                while (!uart_polling_rxd(Char_Sequence +i));

                if (*(Char_Sequence+i)=='\0')
                      loop=FALSE;
                else if (i==(Max_Readed-2))
                {
                      i++;
                      *(Char_Sequence+i) = '\0';
                      loop=FALSE;
                }
                else
                      i++;
          }
          uart_rx_disable();
    }

    return (i+1);
}

void uart_rx_enable(void)
{
    REN = 1;
}

void uart_rx_disable(void)
{
    REN = 0;
}
```

11.5 timing_functions.c

```
/*****************************************************/
/*                                                   */
/*  File: timming_functions.c                        */
/*                                                   */
```

```
/*  Author: Juan A. Quintero                                */
/*                                                          */
/*  Description: Timming control functions                  */
/*                                                          */
/*  Creation: 2005/06/18                                    */
/*  Last modified: 2005/06/22                               */
/************************************************************/

#include "io_lib.h"

void lock_cycles_delay (uint N_Cycles)
{
    uint i;
    for (i = 0; i<N_Cycles; i++);
}


void lock_ms_delay (uchar Val_ms)
{
    uint n_cycles = 330*Val_ms;
    lock_cycles_delay (n_cycles);
}


void lock_s_delay (uchar Val_s)
{
    uchar i =Val_s*10;
    for (;i>0;i--)
    {
        lock_cycles_delay (MS_100);
    }
}


void timer1_init (uint value, uchar enable_irq)
{
    EA = 0;
    TAMOD |= 0x10;  //counter 16bits mode
    TAMOD |= 0x00;

    TL1 = value;
    TH1 = (value>>8);

    TCON |= 0x40;

    if (enable_irq>0)
        IEN0 |= 0x08;
    else
        IEN0 &= 0xF7;

    EA = 1;
}


uchar timer1_expired (void)
{
    uchar expired;
    expired = ((TCON>>7)&(0x01));
    return expired;
}
```

12 ANEXO II – USO DE LA HERRAMIENTA *FLASHMAGIC* PARA REPROGRAMACIÓN EN PLACA

Para la reprogramación del microcontrolador estudiado, *Philips P89LPC932*, tenemos tres posibilidades:

- ISP (In System Programming): Ocurre cuando algún dispositivo externo causa que el microcontrolador, en lugar de aplicar su código normal de programa, ejecute una porción de código que contiene rutinas de borrado y reprogramación. Este tipo de programación tiene lugar con el micro en su medio hardware normal, es decir, soldado.
- IAP (In Application Programming): Es similar al procedimiento comentado anteriormente, salvo que el proceso de reprogramación no lo inicia ningún dispositivo externo, sino que es el mismo código de programa que, bajo ciertas condiciones, hace saltar la ejecución normal del programa hasta el lugar de la memoria donde se ubica este código de reprogramado. Al igual que antes, el proceso tiene lugar en su propia placa.
- Programado paralelo: Se requiere un programador externo, y que la unidad microcontroladora esté separada de su medio habitual.

En la mayoría de casos prácticos, nos vamos a encontrar con la necesidad de utilizar cualquiera de los dos primeros métodos comentados. Por su generalidad, y, a fin de cuentas, por la relación funcionalidad/sencillez tan elevada, que es lo que busca todo programador, se optará por el primero de ellos.

12.1 Interfaz

Para la transmisión de datos durante el proceso de reprogramación del microcontrolador, usaremos la interfaz serie del dispositivo. Para ello, se cableará a un Max232 o similar para adecuarlo al medio físico de la línea de transmisión serie de un ordenador personal, como muestra la siguiente figura.

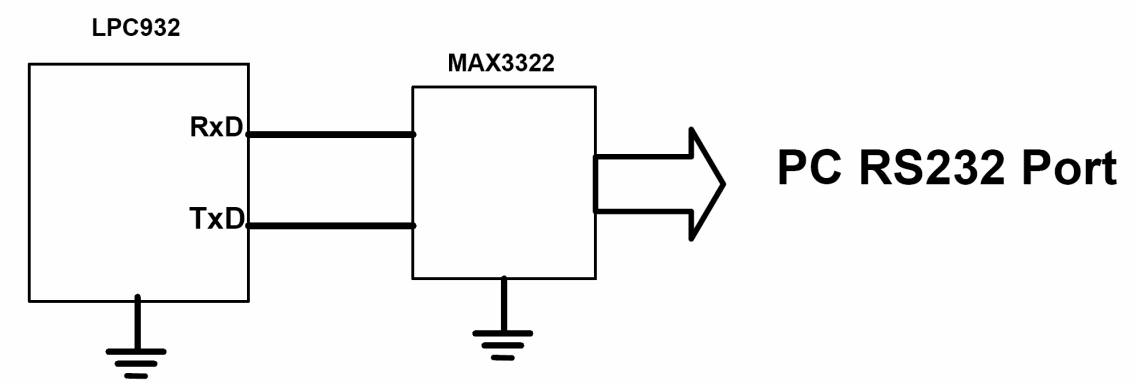

Ilustración 30. Cableado para comunicación serie con un PC

12.2 Acceso al ISP

Existen tres formas de acceder a la programación en placa.

- Mediante el bit de estado y el vector de arranque. El bit de estado se pone a '1' cuando la flash está vacía, i.e., cuando el micro acaba de salir de fábrica o cuando se ha borrado por software para forzar la programación en el siguiente arranque.
- Mediante la activación hardware, enviando por el pin de reset una secuencia de señales determinada, que varía en función del fabricante. Para el microcontrolador seleccionado, se puede entrar siguiendo

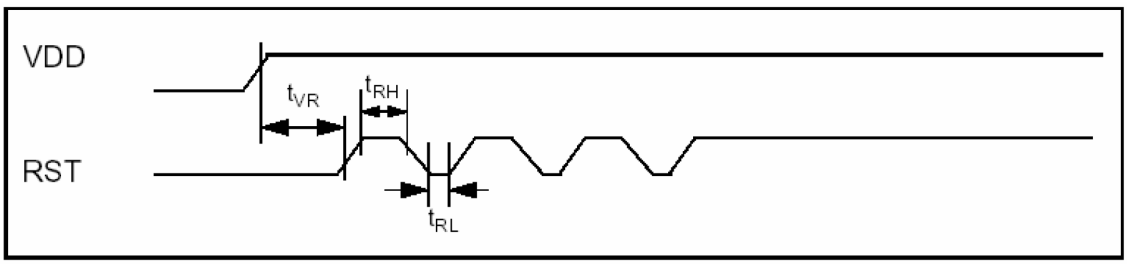

Ilustración 31. Activación Hardware

Símbolo	Parámetro	Mínimo	Máximo
t_{VR}	espera desde VDD activa	50 µs	-
t_{RH}	tiempo a uno del reset	1 µs	32 µs
t_{RL}	tiempo a cero del reset	1 µs	-

Tabla 12. Características dinámicas de la activación hardware

- Mediante la detección de una señal de *Break* por el puerto serie. Esta señal se define como un nivel bajo en recepción por un tiempo de trama completo, es decir, diez u once bits seguidos a nivel bajo, que dependerá de la configuración de la Uart para 8+1 ó 9+1 bits. Veremos a continuación lo sencillo que resulta el acceder al micro mediante este tipo de señal, y cómo podemos valernos de un software dedicado capaz de facilitar enormemente la reprogramación del dispositivo.

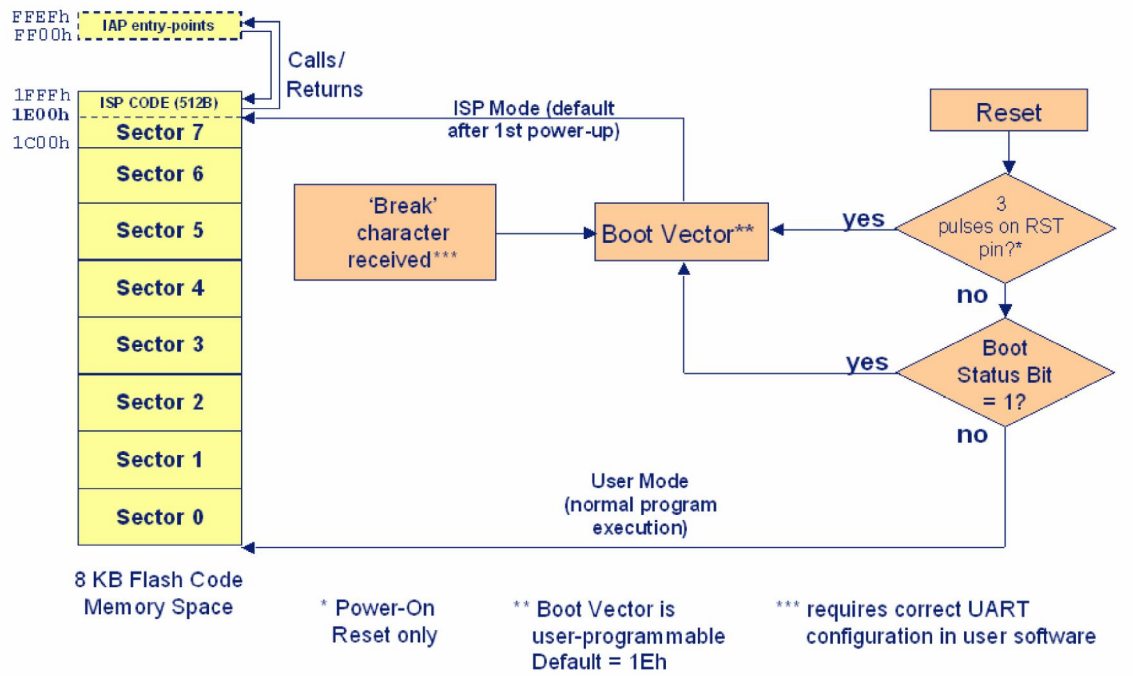

Ilustración 32. Diagrama de flujo del proceso ISP[1]

12.3 Programación del microcontrolador en modo Break Detect

Para que al recibir la señal de *break* por la Uart el micro funcione correctamente, se deben cumplir una serie de requisitos.

Una vez que se detecta la señal de ruptura, a pesar de que suele mantenerse durante muchas tramas a nivel bajo, con la primera se genera el informe de la detección. El código de la aplicación debe dar soporte a la interrupción por detección de señal de ruptura si queremos que esto sirva para activar la reprogramación. Es vital que durante

[1] Imagen extraida de *Philips Application Note 10221*[13]

la ejecución del programa no se reubique la tabla de vectores ni se modifiquen los valores de los saltos, puesto que el sitio donde se encuentra el programa de carga no se puede modificar (a no ser que pretendamos codificar nuestro propio programa de carga). Para que el código la recoja, debemos incluir en nuestro programa:

```
void enable_ISP (void)
{
    SCON = 0x50; //select the BRG as UART baud rate source
    …
    BRGR1 = 0xXX; //select the suitable rate
    BRGR0 = 0xXX; //
    BRGCON = 0x03; //enable BRG
    …
    AUXR1 |= 0x40; //enable reset on break detect by setting EBRR
}
```

Si el nivel bajo contínuo está provocado por algún tipo de dispositivo mecánico, el programa debe manejar la interrupción y limpiar su bandera. De otra forma, los resultados serían imprevisibles:

```
void isr_uart (void) interrupt 11
{
    RI = 0;
}
```

12.4 Software Flashmagic

Flashmagic es un programa gratuito capaz de reprogramar muchos microcontroladores de Philips sin más que especificar algunos parámetros simples. Al arrancar el programa, intentará iniciar una sesión de reprogramado en el dispositivo que tengamos conectado a uno de los puertos serie, y dará un error casi con toda probabilidad porque aún no tendremos nuestro dispositivo en ese modo.

Los primeros pasos que debemos seguir son, en la pantalla principal, dar los valores adecuados al puerto, a la velocidad de transmisión/recepción (se recomiendan 9600bps) y al modelo de dispositivo. Una vez determinado, en la pestaña *Advanced Options*, como se muestra en la figura a continuación, debemos desmarcar la casilla *'Use DTR and RTS to enter ISP mode'* puesto que esta señal sólo sirve para generar el modo dentro de algunas placas de prueba y testeo.

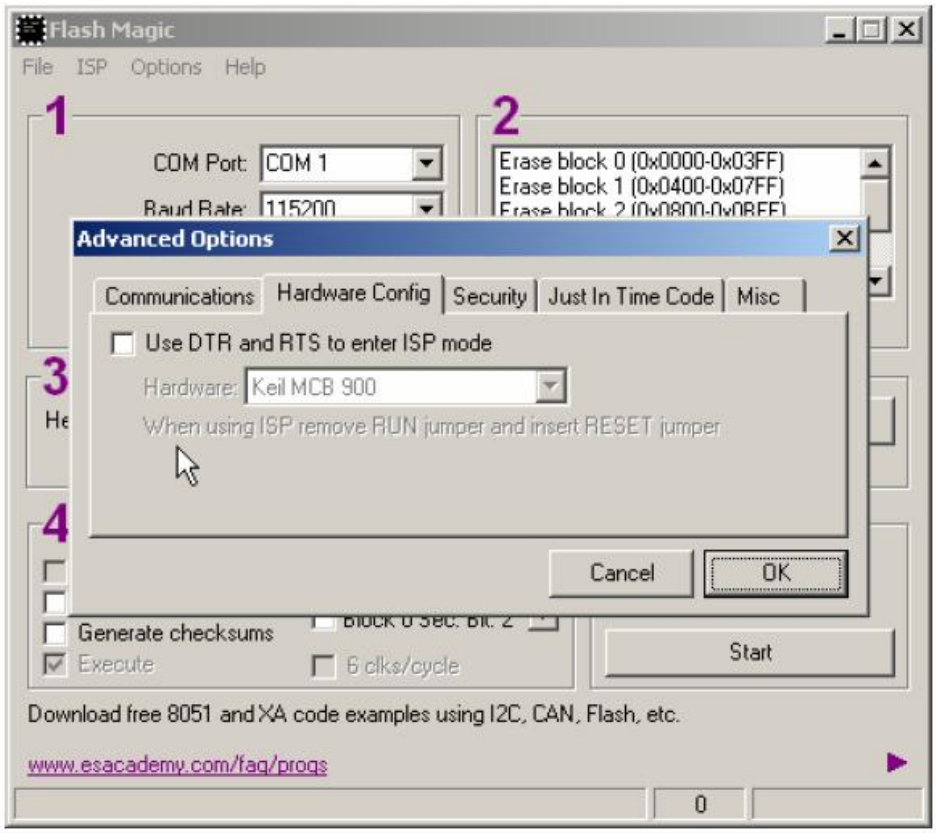

Ilustración 33. Opciones *Flashmagic*

A continuación en *'Start BootROM'* seleccionamos la casilla *'Send Break Condition'* y pulsamos el botón de comienzo para enviar la señal de *break.*

A partir de este momento, todas las opciones del modo ISP están accesibles desde los menús de la aplicación. Se recomienda leer los bits de seguridad, para determinar que zonas del código están protegidas, y proteger el último Kbyte, donde se encuentra el programa de carga preinstalado por *Philips.* Desde ese momento, y siguiendo los sencillo pasos que se muestran en la pantalla principal, la reprogramación es realmente sencilla.

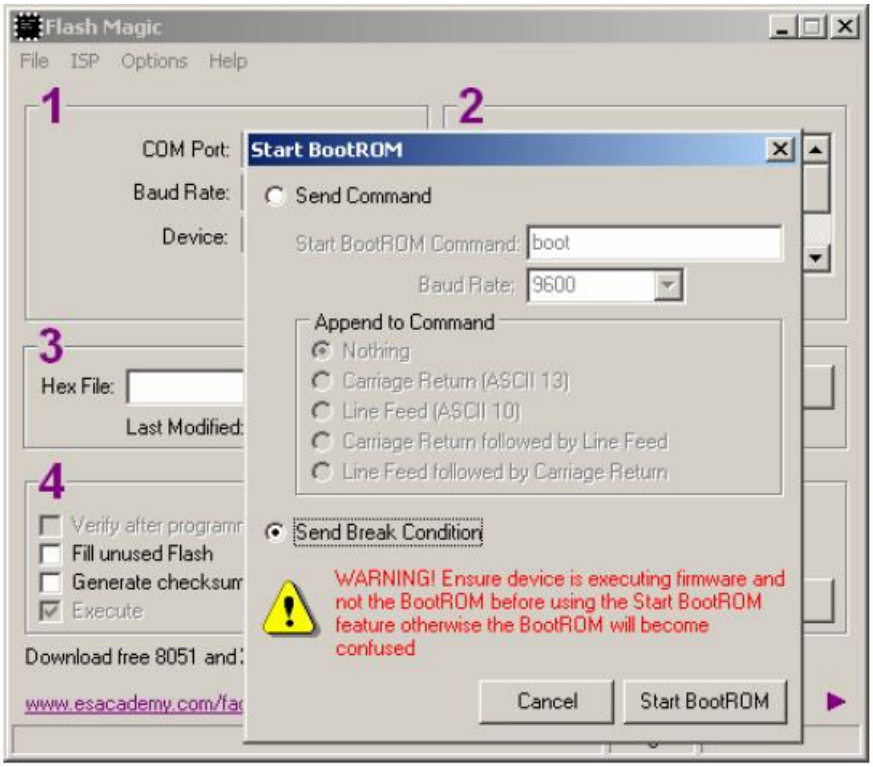

Ilustración 34. Inicio BootROM

13 ANEXO III – CONTENIDO CD

El cd adjuntado incluye el siguiente contenido:

- **/biblio**: de este directorio cuelgan los archivos que han servido de bibliografía, que son gratuitos y se pueden encontrar en Internet, disponibles para cualquier usuario.
- **/case_study**: incluye los archivos fuentes de la aplicación de ejemplo que se ha desarrollado. Se recuerda que para su compilación se han de modificar los archivos de código de la librería y eliminar las funciones que no van a ser usadas. En caso contrario, quizás se genere un código más grande que la memoria disponible para almacenarlo.
- /**doc**: Se incluye este mismo documento.
- **/lib**: incluye los todos los archivos fuente de la librería que se ha desarrollado, es decir, tanto los archivos de código como los de cabecera.
- **/software**: incluye dentro del mismo directorio, los programas necesarios para la elaboración de un proyecto sobre microcontroladores basados en C51:
  1. µVision2
  2. Flashmagic

14 BIBLIOGRAFÍA

Todos los documentos citados a continuación se encuentran en el cd anexo en el directorio '/biblio':

[1]*C51 Primer. An Introduction To The Use Of The Keil C51 Compiler On The 8051 Family.* Hitex (UK) Ltd. University of Warwick Science Park.

[2]*Getting Started With μVision2 And The C51 Microcontroller Development Tools. User's Guide.* Keil Software, Rev. Feb 2001

[3]*MCS@51 Microcontroller Family User's Manual.* Intel Corporation, Rev. Feb 1994

[4]*User Manual P89LPC932.* Philips Semiconductors. Rev. Nov 2002.

[5]*P89LPC932 Datasheet.* Philips Semiconductors. Rev. Jan 2004.

[6]*Cx51 Compiler. Optimizing C Compiler And Library Reference For Classic And Extended 8051 Microcontrollers.* Keil Software, Rev. Sep 2001.

[7]*80C51 family architecture*. Philips Semiconductors, Rev. Mar 2005.

[8]*Errata Sheet P89LPC932 Ver. 1.2.* Philips Semiconductors. Rev. Nov 2003.

[9]*Micro LCD Character Modules Data Sheet.* RS Components, Rev. Mar 2002.

[10]*AN10221. In-system programming (ISP) with the Philips P89LPC932 microcontroller.* Philips Semiconductors, Rev. Jun 2003.

[11]*Macro Assembler and Utilities. Macro Assembler, Linker/Locator, Library Manager, and Object-HEX Converter for 8051, Extended 8051, and 251 Microcontrollers.* Keil Software, Rev. Feb 2001.

[12]*Flashmagic Application Note 3. Using Flashmagic with the Philips P89LPC932*. Embedded systems academy, 2003

[13]*Application Note AN10221, In system programming with the Philips LPC932 microcontroller*. Philips semiconductors, Rev, 2

[14]*http://www.esacademy.com/software/flashmagic*