UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

PROYECTO FINAL DE CARRERA

CONTROL REMOTO MEDIANTE MICROCONTROLADOR DE LOS ACCESORIOS DE UN VEHICULO.

ALUMNO: JAIME CASAS RUSTARAZO

DIRECTOR: ÁNGEL RODAS JORDÁ

Fecha: 5-9-2011

INDICE DE CONTENIDOS

1.- Introducción.

Preliminares:

Se dispone de un Mitsubishi Eclipse 2.0i 210cv el cual posee ciertas modificaciones como resultado de un costoso proceso de personalización con la finalidad de servir de espectáculo en las concentraciones automovilísticas *tuning* a nivel nacional e internacional habilitadas al efecto. En el maletero de dicho vehículo se ha llevado a cabo la instalación de un equipo de audio de alta fidelidad, así como los dispositivos mostrados a continuación:

- Subida y bajada de portón automático. Posee un cilindro neumático de simple efecto capaz de elevar y bajar automáticamente el portón del maletero.
- Cerradura electrónica del portón. Debido a la automatización de la subida y bajada del portón, se ha sustituido la tradicional apertura de cerradura mecánica mediante llave por una cerradura servocontrolada.
- Subida y bajada de altavoces. En el interior del maletero se halla una plataforma horizontal en la cual se encuentran 4 altavoces, sujetada por dos pistones neumáticos mediante los cuales es posible elevar o descender dicha plataforma
- Entrada y salida de las etapas. Situada debajo de la plataforma de los altavoces se halla una base que contiene las 4 etapas de audio encargadas de alimentar los altavoces del equipo Hi-Fi. Dicha base también está controlada por pistones neumáticos de simple efecto de forma que actuando sobre ellos es posible ocultar o mostrar las etapas. Con el fin de etapas la base describe una trayectoria horizontal deslizándose hacia el interior del vehículo, quedando totalmente colocada sobre los asientos traseros, y para mostrarlas se lleva a cabo el mismo movimiento en sentido contrario, extrayéndolas y haciéndolas accesibles desde el maletero.
- Sirena acústica. El vehículo posee una sirena en el maletero cuya finalidad es la de avisar unos segundos antes y después de la apertura o cierre del portón,

evitando de esta manera en la medida de lo posible accidentes derivados con las partes móviles y los espectadores que se encuentren en las inmediaciones del vehículo.

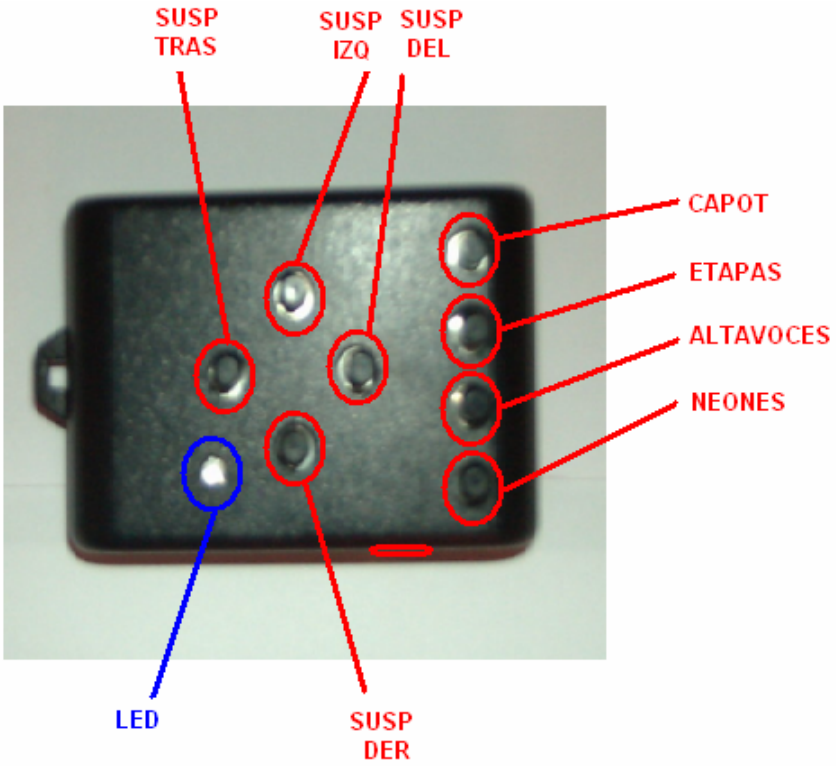
Además de los elementos descritos del maletero del automóvil, sobre este también se han instalado los siguientes elementos:

- Suspensión neumática en las 4 ruedas. Se ha sustituido la suspensión mecánica tradicional por suspensión neumática en las 4 ruedas del vehículo. Ello significa que la suspensión está provista de cilindros neumáticos que posibilitan la acción de regular en altura cada una de las ruedas en todo momento de manera independiente. Dichos pistones neumáticos se accionan, igual que los demás mecanismos descritos, mediante electroválvulas.
- El coche dispone de iluminación tuning en el maletero, interior del habitáculo y bajos exteriores, haciendo uso para ello de diodos LED, *flashes* y tubos de neón.

2.- Objetivos. Descripción del proyecto:

El proyecto desarrollado consiste en el diseño y construcción de un controlador lógico programable controlado por radiofrecuencia, con la finalidad de hacer posible el control automatizado de todos y cada uno de los elementos descritos del vehículo.

Dicho sistema se compone de dos unidades claramente diferenciadas. La primera de ellas, el emisor, consiste en un telemando de reducido tamaño provisto de 9 botones, cada uno de ellos con una finalidad definida. El emisor envía el estado de los mismos a través de ondas de radiofrecuencia bajo un protocolo previamente establecido.



La segunda unidad, el receptor, es el encargado de recibir y reconocer los comandos emitidos por el telemando y actuar en consecuencia controlando los mecanismos descritos del vehículo, siguiendo los patrones que previamente le haya sido asignado a cada pulsación. El receptor permite programar el comportamiento asociado a cada tecla del emisor pulsada, de forma que es posible, además de asignar un único movimiento por pulsación (ejemplo, subir portón o encender neones), asignar a una pulsación una secuencia de movimientos preestablecidos. Con el fin de llevar a cabo la programación de secuencias en el receptor se ha dotado al mismo de conexión USB y se ha desarrollado un programa de ordenador en el cual se muestra gráficamente el vehículo y sus actuadores de forma que hace posible la programación del receptor de una forma muy sencilla e intuitiva para el usuario.

3.- Hardware.

Se describirá a continuación el hardware de ambos elementos, primeramente del emisor y seguidamente del receptor.

3.1.- Emisor

Con el fin de facilitar la comprensión del funcionamiento del emisor previamente seguirá un esquema descriptivo ascendente. Se comenzará mostrando primeramente los componentes que componen el sistema y posteriormente se analizará el desarrollo y la construcción del emisor propiamente dicho.

3.1.1.- Componentes

Existen dos componentes fundamentales dentro del emisor que necesitan ser analizados. Estos componentes son, por una parte, el microcontrolador que gobierna el circuito. Por la otra, el módulo de radiofrecuencia encargado de emitir las ondas electromagnéticas.

3.1.1.1.- Microcontrolador PIC 16F628

El diseño del emisor se realiza en torno al microcontrolador PIC16F628, de la empresa Microchip. Dicho microcontrolador es el encargado de componer las tramas de datos en base al estado de los interruptores y llevar a cabo el envío a través del módulo de radiofrecuencia.

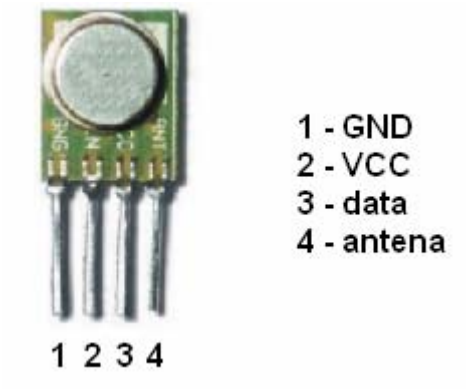
El *pinout* o distribución de pines del circuito integrado se muestra en la siguiente figura.

Las características más notables de dicho microcontrolador se enumeran a continuación.

- Juego de instrucciones RISC. El hecho de poseer instrucciones sencillas hace posible que éstas se ejecuten a una velocidad relativamente grande en comparación con los sistemas CISC, que exigen mayor tiempo por instrucción y segmentaciones del procesador complejas.
- Oscilador de hasta 20 MHz. Por una parte dispone de un oscilador interno, al mismo tiempo que permite el uso de un oscilador interno de una frecuencia de hasta 20 MHz. El oscilador interno es inadecuado para esta aplicación ya que a pesar de que es útil en muchos otros proyectos, éste hace uso de la UART del PIC y esto requiere un oscilador externo de cuarzo para lograr una estabilidad adecuada en la onda. Se ha utilizado un cristal de 4 MHz por disponibilidad, ya que es suficiente para nuestros propósitos y no se justifica la necesidad de uno de mayor velocidad.
- Memoria de programa flash de 2k-palabras. Posee una memoria flash capaz de alojar 2k instrucciones de programa, suficiente para nuestros propósitos.
- Memoria RAM de 224 bytes. Los 224 bytes de memoria volátil suplen los requisitos de esta aplicación.
- Memoria EEPROM de 128 bytes. Posee 128 bytes de memoria persistente o no volátil, sin embargo en este proyecto no se utiliza.
- Pila de 8 niveles. Posee una pila interna 8 niveles, necesaria para llevar a cabo la llamada a subrutinas (funciones).
- 16 terminales genéricos de I/O con una intensidad máxima de 25 mA.
- 3 temporizadores internos.
- Módulo UART, comparadores analógicos, módulo PWM. En este caso se ha hecho uso de la UART para establecer la conexión con el módulo de radiofrecuencia. La señal se envía a este módulo a través de una conexión serie RS232 implementada en hardware por el propio microcontrolador.
- Entrada de interrupción externa. Existe una patilla mediante la cual es posible controlar interrupciones externas por flanco de subida o de bajada (configurable). No se utiliza en este proyecto.

3.1.1.2.- Módulo RF TX433N

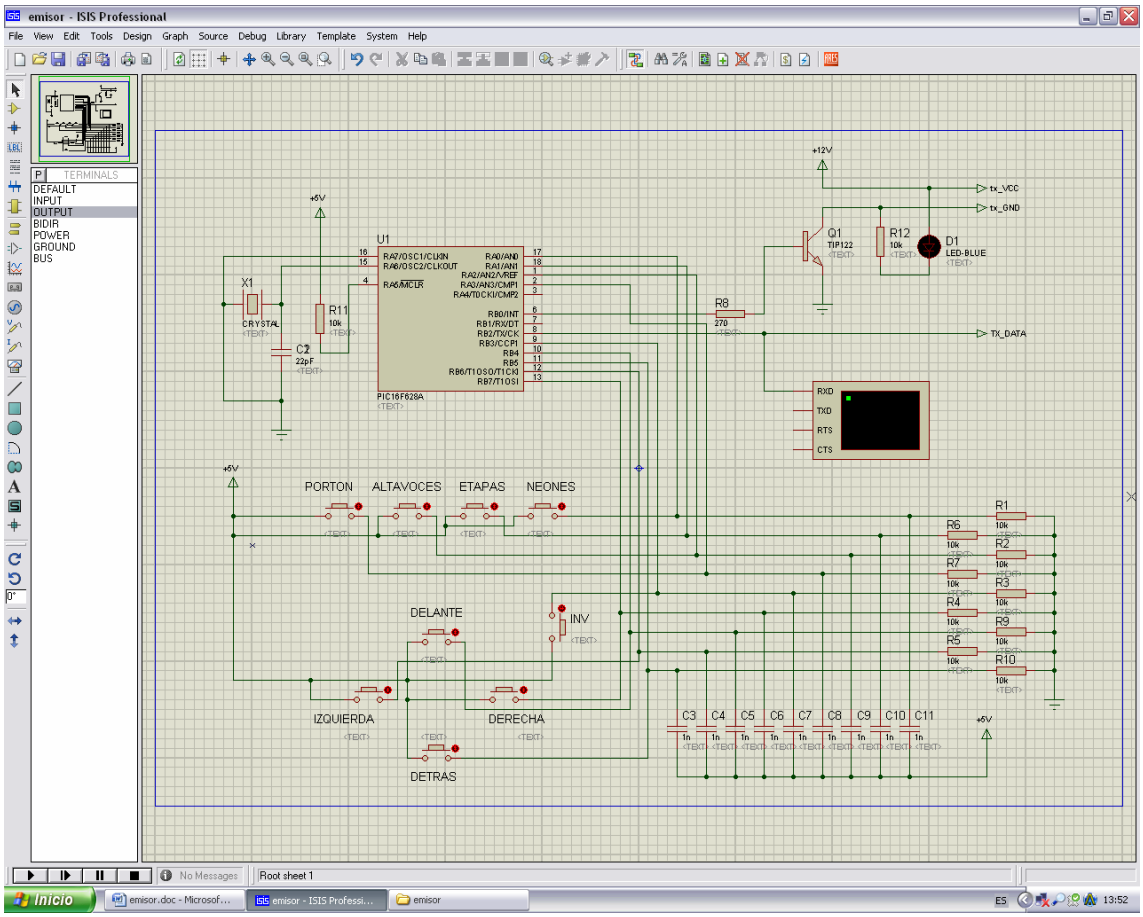
Con el fin de hacer posible la transmisión de ondas electromagnéticas por parte del receptor se hace uso de un módulo emisor de radiofrecuencia de la empresa Velleman. Dicho módulo responde a la referencia tx433n. La imagen y el *pinout* del mismo es el siguiente:



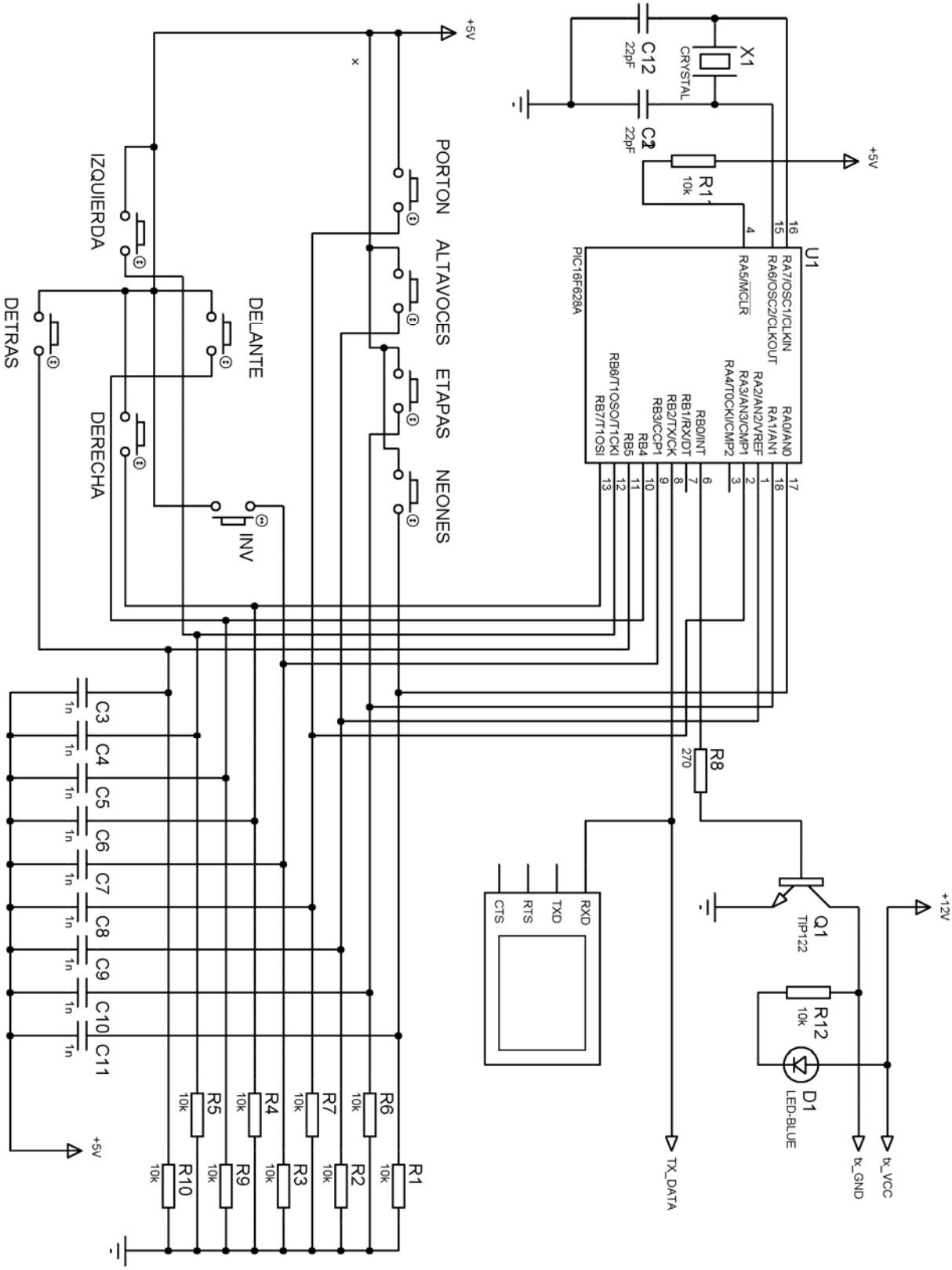
Funciona a una tensión variable dentro del rango desde 3 hasta 12V, a mayor voltaje mayor será la potencia de la onda irradiada. Utiliza el modo de modulación SAW y trabaja en la frecuencia de 433,92 MHz. Se obtienen buenos resultados hasta una velocidad de transmisión de 2.4kbps, bastante lejana de la utilizada en este proyecto. No requiere ningún ajuste y se adquiere listo para su funcionamiento, el fabricante asegura distancias de hasta 100 metros en visión directa con el módulo receptor utilizando una antena de cuarto de onda. Sin embargo y debido a las reducidas dimensiones del mando emisor se ha optado por una antena en bucle. La reducción de alcance no es de importancia ya que aun así presenta una potencia de salida adecuada para las necesidades que nos ocupan.

3.1.2.- Desarrollo y construcción

El trazado de esquema del emisor así como las simulaciones del mismo han sido llevados a cabo utilizando para ello el paquete informático de aplicaciones *Proteus*. Se incluye como anexo los esquemas y el proyecto para su testeo en dicho programa. La siguiente captura de pantalla muestra el entorno de trabajo durante la edición del esquema electrónico del mando emisor.



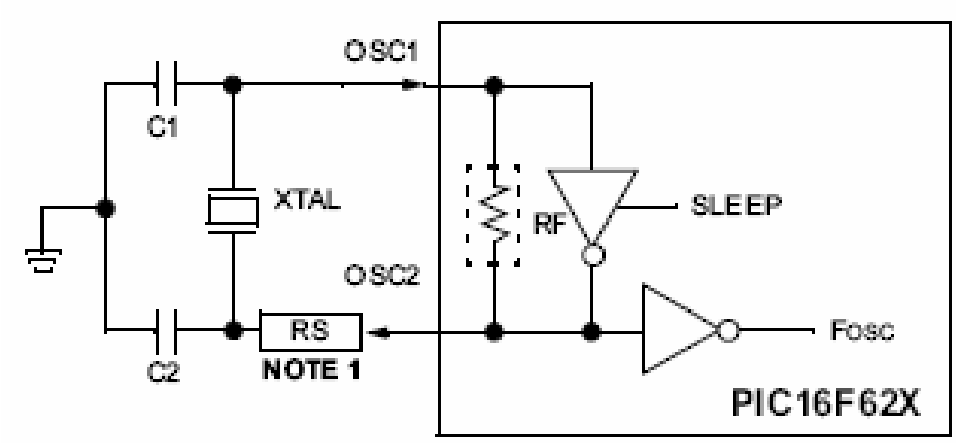
Haciendo uso de la opción “exportar” del menú “File -> Export Graphics -> Export Bitmap” es posible extraer exportar el esquema y obtener así una representación gráfica del circuito electrónico una vez concluido su desarrollo.


+5V
X1
CRYSTAL
C12
22pF
C2
22pF
R1
10k
U1
PIC16F628A
RA7/OSC1/CLKIN
RA6/OSC2/CLKOUT
RA5/MCLR
RA0/AN0
RA1/AN1
RA2/AN2/VREF
RA3/AN3/CMP1
RA4/T0CKI/CMP2
RB0/INT
RB1/RX/DT
RB2/TX/CK
RB3/CCP1
RB4
RB5
RB6/T1OSO/T1CKI
RB7/T1OSI
PORTON
ALTAVOCES
ETAPAS
NEONES
IZQUIERDA
DELANTE
DERECHA
DETRAS
INV
C3
C4
C5
C6
C7
C8
C9
C10
C11
1n
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
10k
270
RXD
TXD
RTS
CTS
Q1
TIP122
R12
D1
LED-BLUE
+12V
tx_VCC
tx_GND
TX_DATA

El desarrollo del emisor se basa en torno al microcontrolador PIC16F628, de la empresa Microchip. Se trata del sucesor del ya obsoleto PIC16F84, mejorando a dicho dispositivo en posibilidades y prestaciones.

A pesar de que según el *datasheet* del microcontrolador éste permite el uso de cristales de cuarzo desde hasta 20 MHz, el emisor no posee una carga computacional que vaya a sacar un partido de tan elevada frecuencia ya que, por disponibilidad, se utilizará un cristal de cuarzo de 4 MHz más que suficiente para este cometido.

Según el gráfico de la página 93 del *datasheet* (PIC16F628.pdf), adjuntado como anexo:



Note 1: A series resistor may be required for some crystals.
2: See Table 14-1 and Table 14-2 for recommended values of C1 and C2.

Muestra el esquema de conexión del oscilador. Tal y como dice la nota 2, y ojeando la tabla adjunta en la misma página, se puede observar lo siguiente:

Mode	Freq	OSC1(C1)	OSC2(C2)
LP	32 kHz 200 kHz	68 - 100 pF 15 - 30 pF	68 - 100 pF 15 - 30 pF
XT	100 kHz 2 MHz 4 MHz	68 - 150 pF 15 - 30 pF 15 - 30 pF	150 - 200 pF 15 - 30 pF 15 - 30 pF
HS	8 MHz 10 MHz 20 MHz	15 - 30 pF 15 - 30 pF 15 - 30 pF	15 - 30 pF 15 - 30 pF 15 - 30 pF

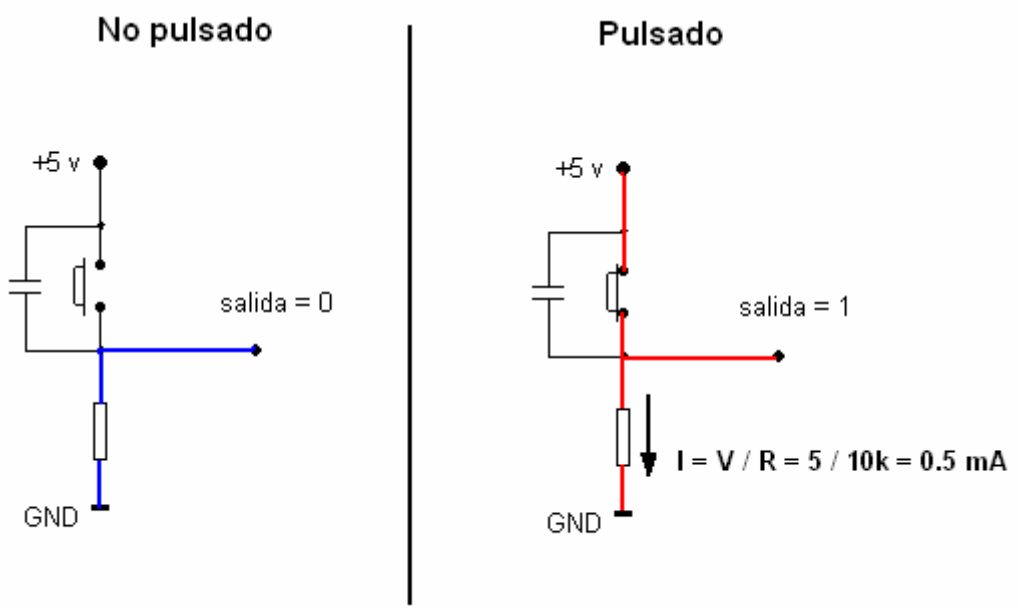
Note 1: Higher capacitance increases the stability of the oscillator but also increases the start-up time. These values are for design guidance only. Rs may be required in HS mode as well as XT mode to avoid overdriving crystals with low drive level specification. Since each crystal has its own characteristics, the user should consult the crystal manufacturer for appropriate values of external components.

Con el uso de dicha frecuencia han de colocarse un condensador en cada patilla del cuarzo dentro del rango 15-30 pF. Debido a la experiencia previa del empleo de este microcontrolador durante su uso en otros proyectos, se utilizarán condensadores de 22 pF los cuales se corresponden con C2 y C12 en el esquema, cableados por una parte a masa y por otra a cada patilla del cristal X1. El cristal de cuarzo va conectado, al mismo tiempo, a las patillas 15 y 16 del microcontrolador (OSC1 y OSC2).

La patilla 4, o MCLR permite provocar un reset por hardware en el microcontrolador de forma externa. Esta patilla es activa a nivel bajo, y se deshabilitará por medio de su conexión a positivo a través de una resistencia limitadora de corriente de 10k.

El mando emisor dispone de 9 pulsadores NA (normalmente abiertos) directamente conectados a los puertos de entrada del PIC. Mientras no se presiona un pulsador, a la entrada del puerto del microcontrolador le llega un valor lógico negativo debido a la resistencia *pull-down* conectada a él evitando de este modo dejar la línea en un estado flotante o indeterminado. Cuando el pulsador se presiona, se conecta directamente a VCC la entrada del microcontrolador entregando un nivel lógico 1.

No existe un valor concreto adecuado como resistencia de *pull-down* o *pull-up*, valores muy pequeños se comportan como un puente al cerrar el pulsador provocando el flujo directo entre tensión y masa creando un cortocircuito, y valores extremadamente altos actúan como una línea abierta no obteniendo el resultado esperado. Un valor de 10 Kohm es aceptado como estándar y permite obtener niveles lógicos adecuados de tensión, cuando el pulsador se oprime circula una corriente de:

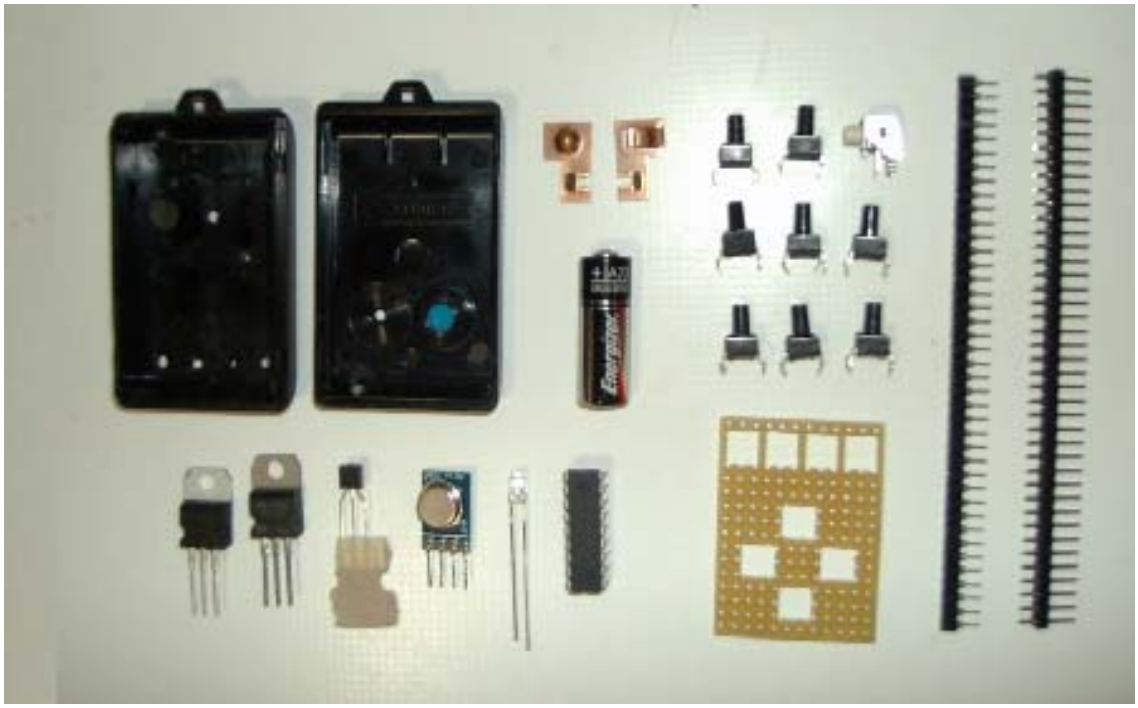


I = V / R = 5v / 10k = 0.5 mA, lo cual no supone un consumo a tener en cuenta durante la pulsación de la tecla.

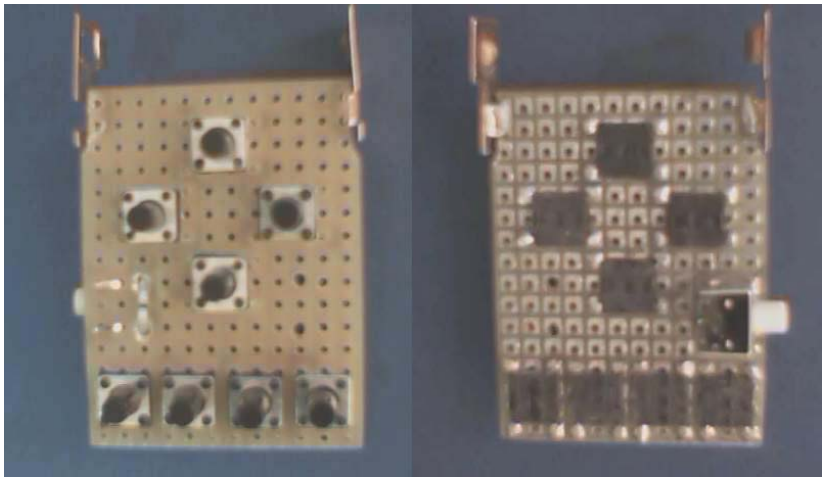
Los condensadores cerámicos de 1 nF de montaje superficial SMD incluidos entre los bornes de cada pulsador del mando tienen como función reducir en la medida de lo posible los rebotes derivados de la pulsación de los botones. Los rebotes son estados transitorios a alta velocidad durante el cambio de estado de los botones e interruptores como consecuencia de su naturaleza mecánica, haciendo uso de condensadores se reduce notablemente la aparición de transitorios no deseados en las entradas del microcontrolador.

El transistor Q1 es un transistor Darlington gobernado por la salida RB0 del PIC (patilla 6). Se trata de un transistor NPN, la colocación de la resistencia R8 de 270 ohmios permite al PIC polarizar el transistor y trabajar en corte y saturación de forma que actúa a modo de interruptor. La intención de este transistor es poder activar o desactivar la alimentación del módulo emisor de radiofrecuencia. Este módulo transmite siempre que se le proporcione la tensión de alimentación a su entrada por lo que el motivo de este transistor es capacitar al microcontrolador de la posibilidad de activar o desactivar el módulo para evitar que éste transmita “basura” (información no útil) en los periodos de inactividad, esto es, cuando no hay ninguna tecla pulsada y no se debe transmitir información. Ello permite también economizar el consumo de la batería.

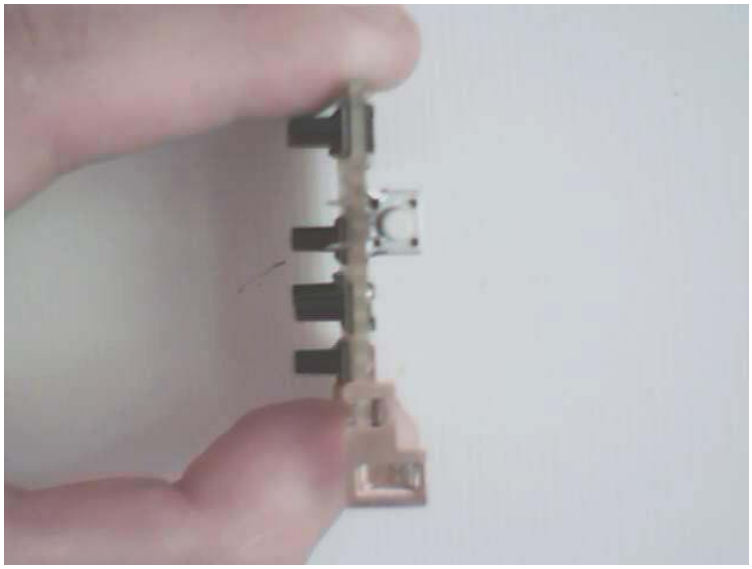
En paralelo con la alimentación del módulo se ha colocado un diodo LED de color azul con su correspondiente resistencia limitadora de intensidad. Con ello se logra mostrar una indicación visual de la actividad del mando, siempre que esté transmitiendo la indicación luminosa permanecerá encendida de forma análoga a como sucede en los mandos de los garajes. A continuación se procederá a mostrar el proceso de construcción del mando. Primeramente se muestran los componentes electrónicos utilizados en el montaje (a excepción del estaño y cables).



El primer paso consiste en soldar los nueve micro interruptores en la placa de circuito impreso, previamente cortada a la medida exacta de la caja en la caja destinada a su alojamiento.

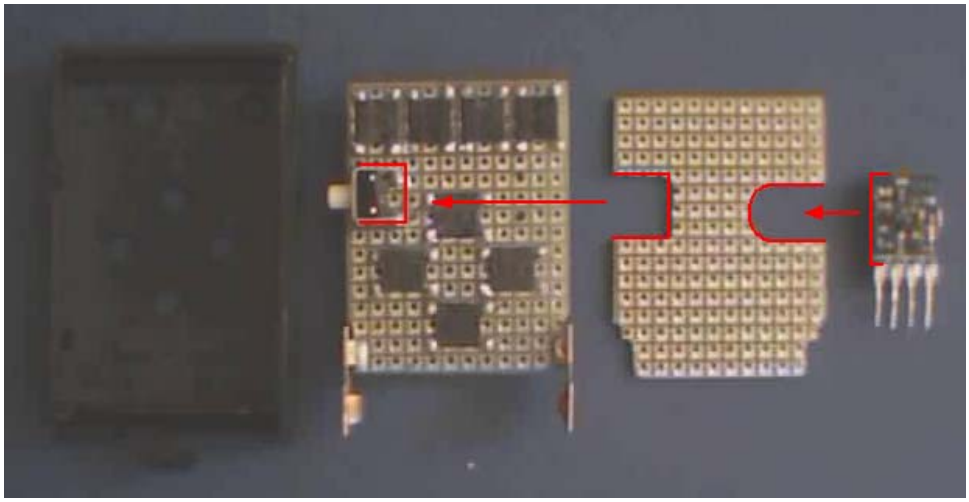


Debido al reducido espacio disponible, en lugar de colocar los componentes por la cara de la fibra como es habitual o en la cara del cobre, se ha recortado con un taladro Dremel el contorno de los interruptores en la placa de forma que estos quedan atravesados en ella. La siguiente imagen de perfil muestra este detalle. Obsérvese uno de los pulsadores en la posición de 90 grados en la placa. Se ha optado por la colocación en el lateral del mando debido a que este pulsador presenta una funcionalidad especial.



Una vez hecho esto el siguiente paso ha sido taladrar la carcasa de plástico acorde a la posición de los pulsadores. El encaje de la placa es exacto.

Con el fin de hacer posible la conexión de todos los componentes internos se ha optado por utilizar otra placa de circuito impreso auxiliar además de la que tiene los interruptores. La finalidad es colocar la segunda placa encima de la primera, siendo esta segunda placa la que alojará el microcontrolador, módulo de radiofrecuencia, transistor, regulador, etc. Debido al reducido tamaño ha sido necesario mecanizar esta segunda placa para hacer posible el encaje con el pulsador vertical de la placa de los interruptores, así como para posibilitar el encaje del módulo de radiofrecuencia. A pesar de que en la siguiente imagen no se aprecia, en su parte posterior el módulo posee un componente circular que encaja en la placa.

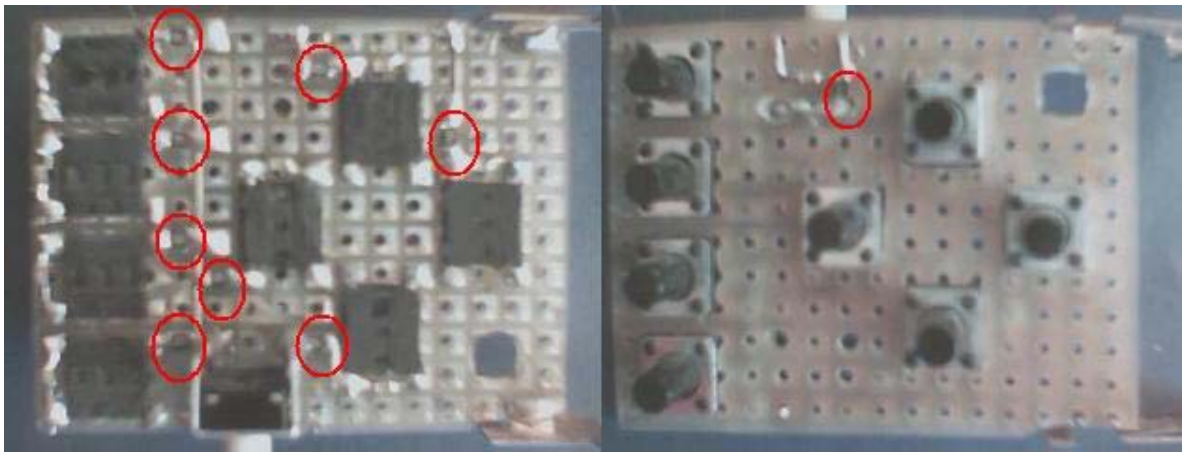


Para evitar el contacto eléctrico entre los componentes de ambas placas al superponerlas una encima de la otra, se colocará entre ambas una lámina de plástico delgada. Para ello se ha cortado una tapa de plástico transparente de las utilizadas en encuadernación con un cutex, ajustando el tamaño al contorno de las placas de circuito impreso.

Una vez soldados los microinterruptores sobre la placa se procederá a soldar sobre la misma las resistencias de *pull-down* de 10kohm y los condensadores de amortiguación de transitorios de 1nF de cada. Para ello se han soldado condensadores y resistencias en formato SMD debido al reducido espacio, ya que encima de la placa de

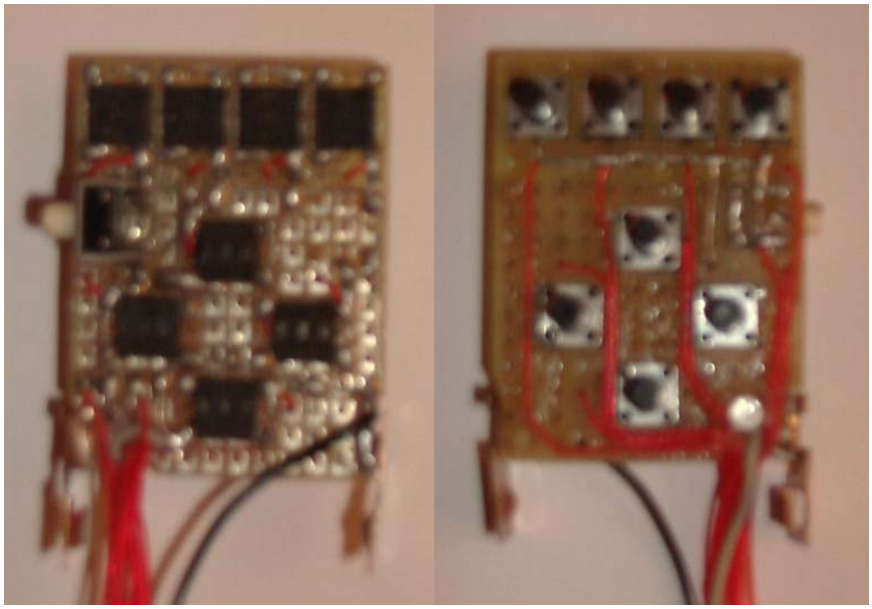
circuito impreso de los interruptores debe colocarse la del microcontrolador y el espacio entre ellas es muy reducido. La soldadura de estos componentes es una tarea delicada que exige pulso, concentración y tiempo. Antes de comenzar a se limpiarán las superficies a soldar concienzudamente y se afinará la punta del soldador si la situación así lo requiere. Puede utilizarse una lupa de soldadura para facilitar la tarea.

En las siguientes imágenes se muestran las resistencias y condensadores SMD ya soldados. En la primera imagen se muestran las resistencias en rojo, y en la segunda los condensadores en azul.



Tras ello el siguiente paso consistirá en trazar el cableado de los interruptores. Para ello se ruteará una línea de ánodo común y acto seguido se sacará un cable por cada interruptor, del lugar adecuado entre la resistencia y el condensador tal y como muestra el esquema en Proteus. Para llevar a cabo estas conexiones se han utilizado cable de 0,25 mm de sección, suficiente para nuestros propósitos y, quizá, la única opción ya que el reducido espacio en la placa impide el uso de cableado con secciones mayores. Sin embargo para esta zona concreta del circuito no es necesario más ya que la intensidad a circular es ínfima, los cables únicamente han de entregar niveles de tensión a la entrada de los puertos del microcontrolador.

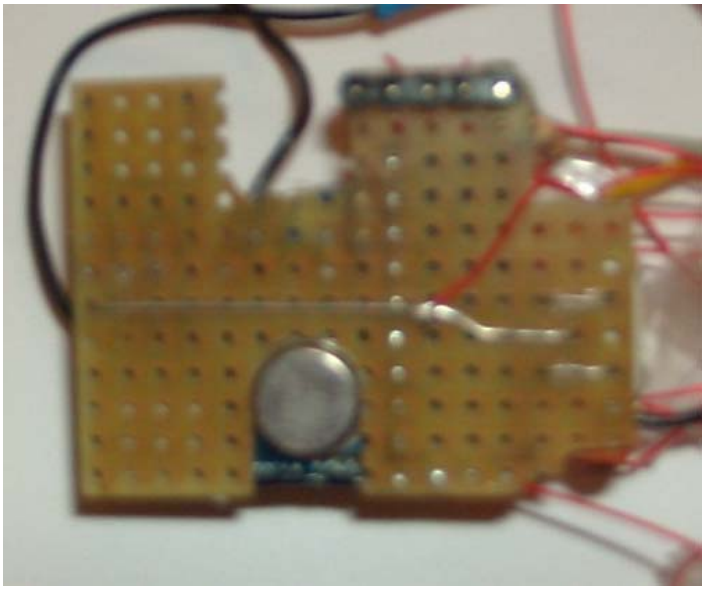
Frente y reverso:



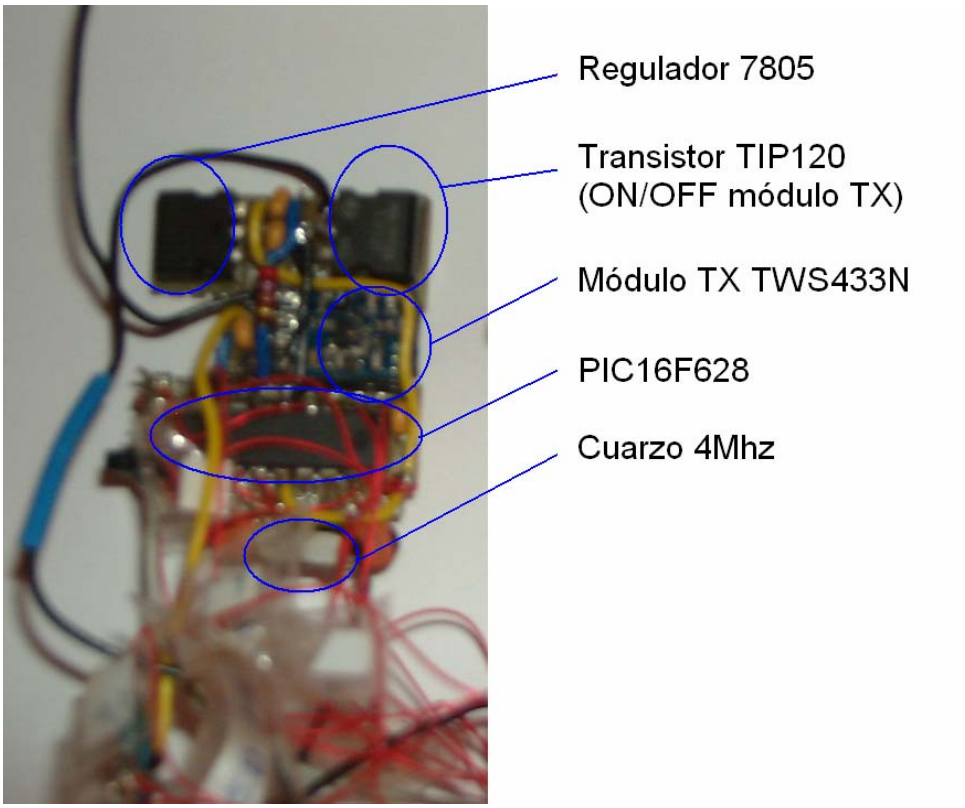
Obsérvese en la anterior imagen la colocación del diodo LED azul en la parte inferior derecha de la placa, ya soldada y cableada.

Una vez hecho ello se pasará a montar la placa de los componentes activos propiamente dichos. En ella se situará el módulo emisor de radiofrecuencia, el microcontrolador PIC, el transistor de alimentación del módulo, el regulador de tensión 7805, los condensadores de desacoplo de frecuencias de cada integrado y las resistencias de apoyo del circuito.

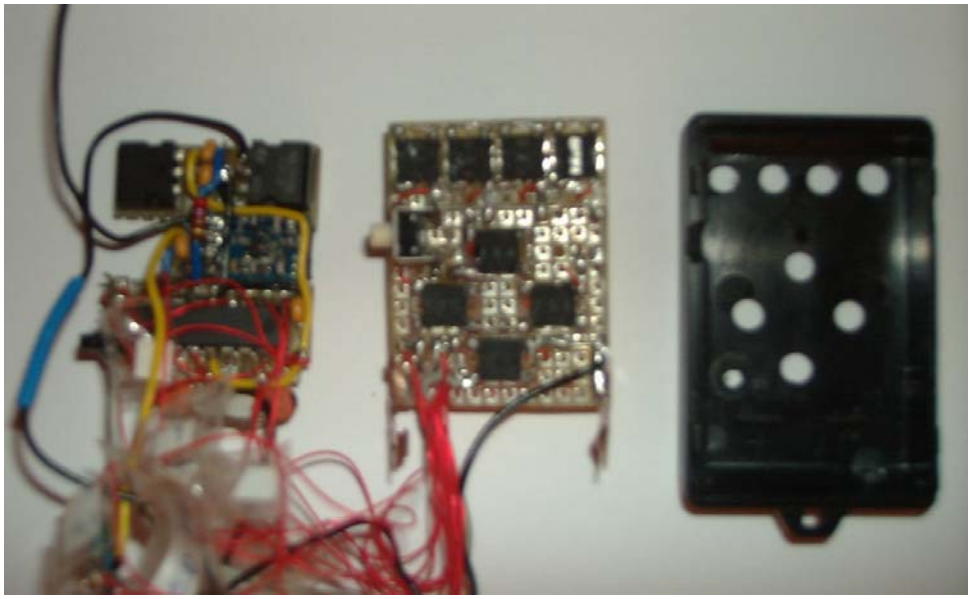
La parte posterior de dicha placa se muestra a continuación. La cara mostrada es la cara posterior de los componentes, la cara que colinda con la placa de los microinterruptores. Se puede observar la mecanización circular de la placa para el correcto acople del módulo emisor



En la siguiente imagen puede observarse esta misma placa en su parte superior, con el cableado de los interruptores conectado al PIC de forma provisional para llevar a acabo pruebas de programación de firwmare y ajuste del funcionamiento.



Préstese atención a esta imagen:



En ella queda claro el procedimiento de ensamblaje. La placa de la izquierda se coloca sobre la primera, y el conjunto formado por ambas se introduce en la caja plástica. El anclaje de estos tres componentes es exacto. En la parte inferior del interior del mando se colocará la pila de 12 voltios. En la imagen se muestra el cableado provisional durante la fase de pruebas, sin embargo una vez afinado el firmware y programada la placa de forma definitiva, se cortarán los cables (los cables rojos de la figura) lo más cortos posibles y se soldarán de nuevo una vez encajadas ambas placas, obteniendo así un conjunto compacto y reducido.

Tras el cableado definitivo entre ambas placas y su montaje dentro de la carcasa plástica, el mando presenta el siguiente aspecto:

3.2.- Receptor

Del mismo modo que se realizó durante la descripción del mando emisor, se va a proceder a describir previamente los componentes que conforman el receptor siguiendo de este modo un enfoque ascendente. Se partirá del conocimiento del funcionamiento de los componentes que forman el circuito receptor para después poder entender el funcionamiento como conjunto.

3.2.1.- Componentes

El receptor está compuesto por diversos componentes electrónicos clave. Uno de ellos, y el principal, es el microcontrolador PIC18F2550 que da vida al sistema. Tras ello, el módulo receptor de radiofrecuencia es el encargado de recibir las ondas emitidas por el emisor. Además el sistema incorpora una memoria EEPROM y un timer en tiempo real. Se procederá al análisis de cada uno de ellos.

3.2.1.1.- Microcontrolador PIC 18F2550

El microcontrolador PIC18F2550 es el que controla el receptor. En el se delegan todas las tareas de procesamiento que dan funcionalidad al sistema. Este microcontrolador se encuentra en encapsulados de 28, 40 y 44 patillas ya que debido al alto número de funciones de las que dispone algunas de ellas se encuentran multiplexadas. En el caso que nos ocupa es suficiente con el encapsulado de 28 patillas. El *pinout* del mismo se muestra a continuación:

Las características más notables del mismo son las siguientes:

- Frecuencia de reloj de hasta 48 MHz. Del mismo modo que el PIC16F628 utilizado en el receptor este microcontrolador también implementa un oscilador interno. Sin embargo y debido a la necesidad de hacer uso de la UART del mismo se ha optado por el uso de un cristal de cuarzo externo de 20 MHz, dotando al sistema de un oscilador estable y de onda simétrica.
- Soporte USB 2.0. Incorpora un módulo hardware que implementa las funciones y mecanismos necesarios para dotar al microcontrolador de conexión USB en el papel de dispositivo (no host).
- Modos *Idle* y *Sleep*. Permiten inducir el circuito integrado en modos especiales de bajo consumo. El modo *Idle* desactiva la unidad central de proceso y las partes asociadas al mismo, permaneciendo únicamente activos aquellos dispositivos de E/S mediante los cuales es posible devolver al procesador a su estado normal ante un evento predeterminado.

- Interrupciones externas. Se dispone de la posibilidad de configuración de varios pines de entrada del microcontrolador como patillas de entrada de interrupción activa por flanco ascendente o descendente (configurable por software).
- PWM, entradas analógicas, UART. En este caso concreto de aplicación se hace uso de la UART del chip para establecer la comunicación con el módulo receptor de radiofrecuencia.
- Módulo MSSP. Se utiliza para establecer la comunicación por i2c con la memoria EEPROM y el timer RTC.
- Memoria de programa de 32k-instrucciones. Está dotado con una memoria flash capaz de alojar una cantidad de 32k instrucciones, más que suficiente para nuestros propósitos.
- Memoria RAM de 2KB. Se dispone de un banco de 2KB de memoria volátil RAM y acceso aleatorio.
- Memoria EEPROM de 256 bytes. El dispositivo pone a disposición del desarrollador 256 bytes de memoria no volátil o persistente.

Las razones de elección de este microcontrolador y no otro de la familia de chips que Microchip pone a disposición es el hecho de que está dotado de módulo USB, al mismo tiempo que ofrece los recursos necesarios no siendo requerido otro de categoría superior.

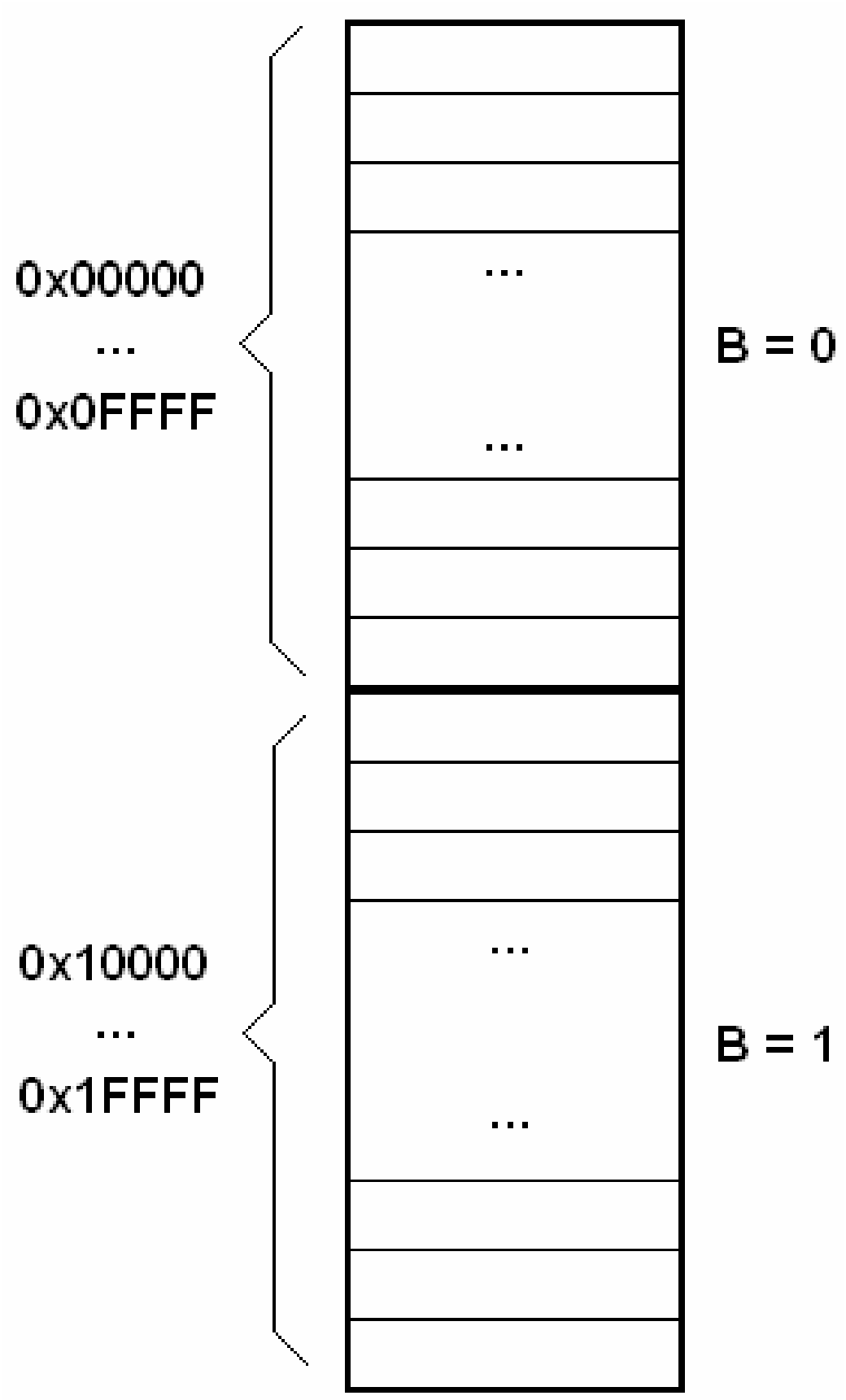
3.2.1.2.- Memoria EEPROM 24LC1025

El receptor hace uso de una memoria EEPROM 24LC1025 de microchip, cuya finalidad es posibilitar el guardado de secuencias y las acciones asignadas a los pulsadores. Dicha memoria comparte el bus i2c con el timer DS1307, y es el microcontrolador el encargado iniciar la comunicación con los dispositivos cuando así sea necesario y llevar a cabo el correcto direccionamiento de los mismos con el fin de discernir entre uno u otro.

De la misma forma que se ha implementado el driver que ofrece acceso a las funciones del timer también se ha programado una API que ofrece toda la funcionalidad de la memoria EEPROM mencionada. Para ello, y como es proceso habitual durante la programación de drivers de dispositivo, se ha comenzado estudiando el datasheet del circuito integrado con la finalidad de conocer los aspectos importantes, secuencias de transferencias de datos, modos de funcionamiento, direccionamiento, acceso, mapeado interno, rangos de direcciones, etc. Y todo lo necesario para llevar a cabo exitosamente la implementación que permita la usabilidad de éste dispositivo.

Constitución interna

La memoria EEPROM 24LC1025 posee 1Mbit de capacidad organizada en 128 K-palabras de 8 bits (1024 Kbits), siendo necesario 16 bits para poder direccionar todo el rango disponible. Debido a la arquitectura interna del dispositivo se encuentra organizada como si de dos memorias de 512 Kb se tratase, estableciendo dos bloques lógicos de 512 Kb y habilitando un bit de selección de bloque con el fin de acceder a uno u otro durante las operaciones de lectura y escritura. Dicho bit puede considerarse como si del bit de dirección número de mayor peso de la dirección se tratase.

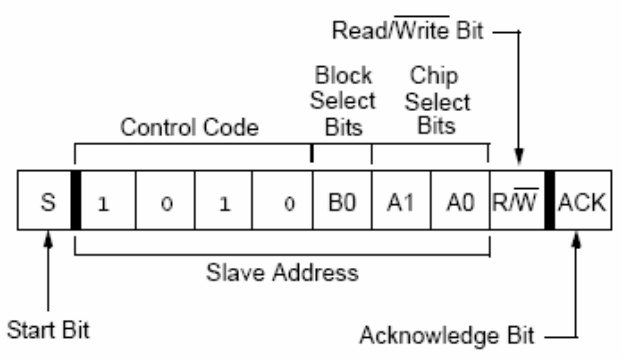

0x00000
...
0x0FFFF
B = 0
0x10000
...
0x1FFFF
B = 1

Direccionamiento del dispositivo

Name	PDIP	SOIC	Function
A0	1	1	User Configurable Chip Select
A1	2	2	User Configurable Chip Select
A2	3	3	Non-Configurable Chip Select. This pin must be hard-wired to logical 1 state (Vcc). Device will not operate with this pin left floating or held to logical 0 (Vss).
Vss	4	4	Ground
SDA	5	5	Serial Data
SCL	6	6	Serial Clock
WP	7	7	Write-Protect Input
Vcc	8	8	+1.8 to 5.5V (24AA1025) +2.5 to 5.5V (24LC1025) +2.5 to 5.5V (24FC1025)

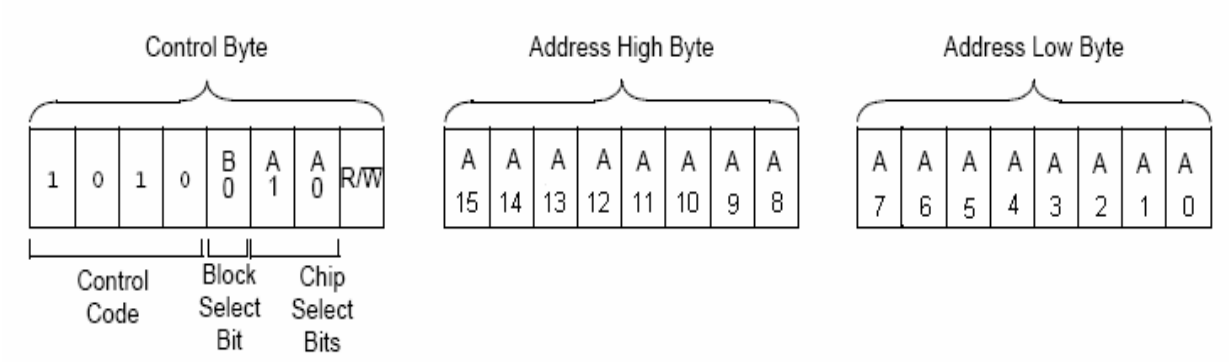
Primeramente se prestará atención al subconjunto destacad en la siguiente tabla, extraída de la página 5 del *datasheet*. Se puede observar que se dispone de dos patillas mediante las cuales puede establecerse la dirección física del dispositivo en el bus, A0 y A1. A pesar de que primeramente pudiera parecer que se dispone de tres patillas, la tabla indica que la pata A2 no se trata de una pata de configuración y que debe dejarse fija a valor lógico 1 para que el dispositivo funcione correctamente. Por lo tanto, disponiendo de dos patillas para configurar la dirección del dispositivo es posible conectar hasta 4 memorias EEPROM en el mismo bus, permitiendo una capacidad total de 4Mbit.

Formato del byte de direccionamiento:



Observando el diagrama del byte de dirección puede observarse que está compuesto por una serie de cuatro bits fijos, seguidos de 3 bits de dirección (de los cuales internamente el más significativo se corresponde al bit de selección de bloque) y un bit de lectura/escritura en la posición del bit de menos peso. El valor de A1 y A0 debe corresponderse con el valor cableado eléctricamente en las patillas 1 y del chip.

Una vez direccionado el chip, el siguiente paso consiste en enviar la dirección de trabajo sobre la cual se quiere operar. Para ello se fracciona la dirección en dos mitades, enviando primeramente el byte más significativo de la misma (bits15 a 8) y posteriormente el byte menos significativo (bits 7 a 0). Hasta aquí puede considerarse el máximo común divisor a la hora de llevar a cabo cualquiera de las operaciones posibles sobre la memoria.



Volviendo a la misma tabla, préstese ahora atención a la línea enmarcada en rojo:

Name	PDIP	SOIC	Function
A0	1	1	User Configurable Chip Select
A1	2	2	User Configurable Chip Select
A2	3	3	Non-Configurable Chip Select. This pin must be hard-wired to logical 1 state (Vcc). Device will not operate with this pin left floating or held to logical 0 (Vss).
Vss	4	4	Ground
SDA	5	5	Serial Data
SCL	6	6	Serial Clock
WP	7	7	Write-Protect Input
Vcc	8	8	+1.8 to 5.5V (24AA1025) +2.5 to 5.5V (24LC1025) +2.5 to 5.5V (24FC1025)

Como se puede observar el circuito integrado posee también de una patilla de habilitación de escritura la cual queda habilitada a nivel bajo. En caso de colocarla a nivel alto, las operaciones de escritura quedan inhibidas a pesar de que las de lectura no se ven afectadas (página 5 del *datasheet*).

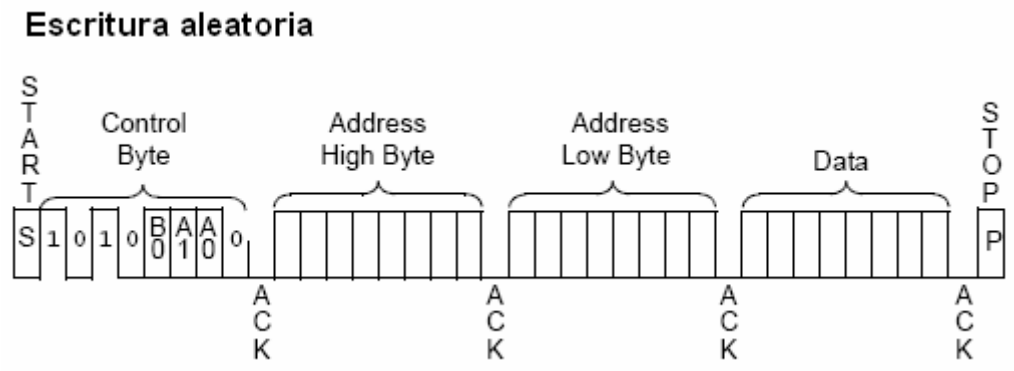
"This pin must be connected to either VSS or VCC. If tied to VSS, write operations are enabled. If tied to VCC, write operations are inhibited, but read operations are not affected"

Según la tabla de la página 3 del *datasheet*, se tiempo de *setup* mínimo de dicha patilla (TW *setup* min) es de 600 nanosegundos, y el tiempo de mantenimiento (TW *hold* min) es de 1300 ns. Se tiene por tanto, que antes de iniciar cualquier transferencia de escritura será necesario colocar la patilla 7 del chip a nivel bajo durante al menos 600 ns. antes, y será necesario mantener dicho nivel al menos 1300 ns. tras la última operación de escritura llevada a cabo.

			100	—		2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
10	TSU:STO	Stop condition setup time	4000 600 250	— — —	ns	1.8V ≤ Vcc ≤ 2.5V 2.5V ≤ Vcc ≤ 5.5V 2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
11	TSU:WP	WP setup time	4000 600 600	— — —	ns	1.8V ≤ Vcc ≤ 2.5V 2.5V ≤ Vcc ≤ 5.5V 2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
12	THD:WP	WP hold time	4700 1300 1300	— — —	ns	1.8V ≤ Vcc ≤ 2.5V 2.5V ≤ Vcc ≤ 5.5V 2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
13	TAA	Output valid from clock **(Note 2)**	— — —	3500 900 400	ns	1.8V ≤ Vcc ≤ 2.5V 2.5V ≤ Vcc ≤ 5.5V 2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
14	TBUF	Bus free time: Time the bus must be free before a new	4700 1300	— —	ns	1.8V ≤ Vcc ≤ 2.5V 2.5V ≤ Vcc ≤ 5.5V

Operaciones de escritura

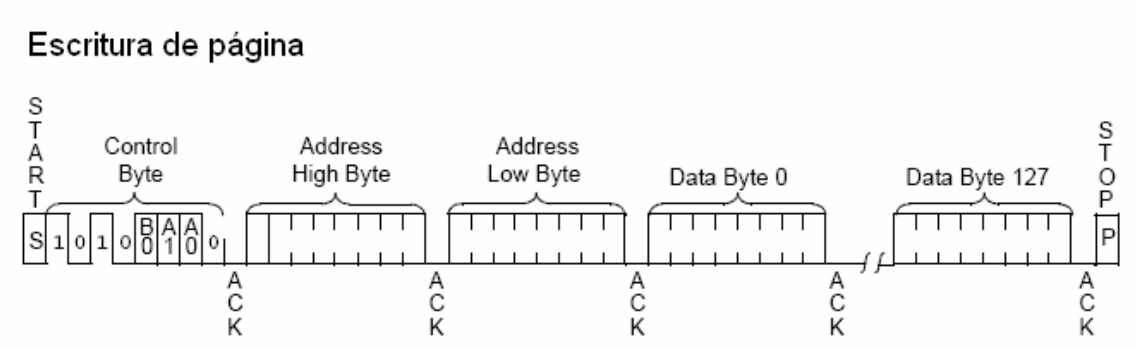
Obsérvese el siguiente diagrama, extraído de la página 7 del datasheet.



En él se refleja el fujo de datos de una operación de escritura aleatoria, esto es, a una dirección cualquiera dentro del rango direccionable por la memoria. Como se puede ver, la comunicación se inicia cuando el microcontrolador coloca el bit de START en el bus. Acto seguido, envía el byte de dirección de dispositivo constituido por una parte fija, el bit de bloque, la dirección configurada según los pines A1 y A0, y el bit de lectura/escritura, que al tratarse de un caso de escritura será 0.

A continuación se envía el byte más significativo de la dirección seguida del byte menos significativo de la misma. A cada envío, el chip responde con un bit de reconocimiento ACK. Una vez seleccionada la dirección, sucesivas escrituras escribirán posiciones consecutivas de la memoria hasta que el microcontrolador de por finalizada la comunicación escribiendo el bit de STOP en el bus.

El dispositivo permite también el acceso a bloque de datos en modo página. El siguiente diagrama representa una instancia de dicho modo de escritura.



Como se puede observar el modo de acceso no difiere del modo aleatorio. Simplemente hay que tener en cuenta una serie de consideraciones a la hora de llevar a cabo la operación.

- El tamaño de la página es fijo e igual a 128. Deberán realizarse por lo tanto 128 accesos consecutivos a una dirección 128*n donde n es un numero entero.

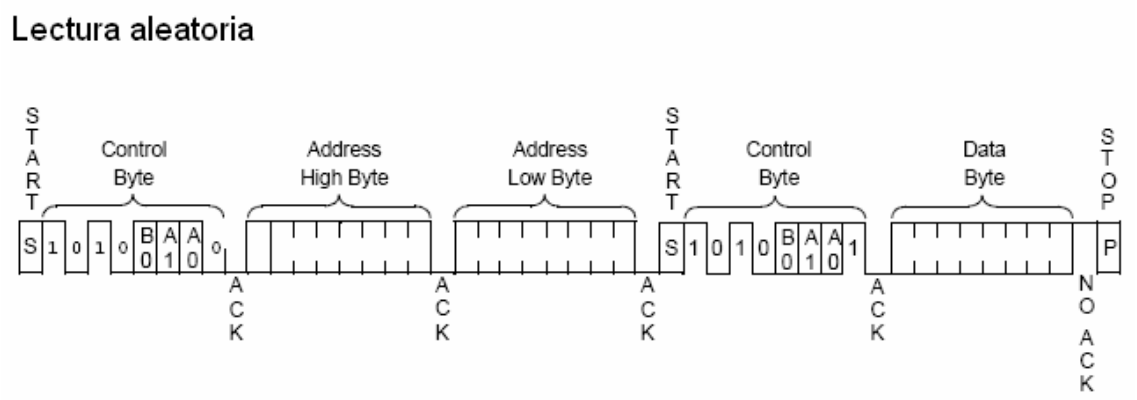
- La dirección de la página deberá ser múltiplo de 128, y al ser de tamaño 128 bits acabará en una posición inmediatamente anterior al principio de la siguiente página. Esto es, la dirección de página que comience en 12*n acabará en 128*(n+1)-1.

- Los datos se copian a un buffer intermedio y son actualizados una vez recibido el último byte. En caso de que la línea de habilitación de escritura WP se ponga a nivel alto deshabilitando la escritura durante un proceso de escritura de página, el dispositivo reconocerá las operaciones (enviará ACKs) pero no actualizará los datos al finalizar la recepción.

En caso de que no se respeten las los dos primeros puntos, al llegar al final lógico del tamaño de la página de trabajo actual se seguirá escribiendo desde el comienzo de la misma, sobrescribiendo los datos que en esas posiciones hubieran y no saltando a la página siguiente como en un principio cabría esperar (pág. 8 *datasheet*).

Operaciones de lectura

El siguiente diagrama refleja el flujo de información durante un proceso de lectura aleatoria a una dirección cualquiera dentro del rango direccionable por la memoria.



Primeramente se envía el bit de START seguido del byte de dirección de dispositivo, con el bit de lectura/escritura a 0, es decir en modo escritura. Ello es debido a que antes de leer la posición requerida, es necesario escribir en el dispositivo dicha dirección del mismo modo que sucedía con el timer DS1307. Acto seguido se envía la dirección fraccionada en dos envíos, primeramente los 8 bits más significativos y posteriormente los 8 bits menos significativos. Hasta aquí el proceso es idéntico al de escritura.

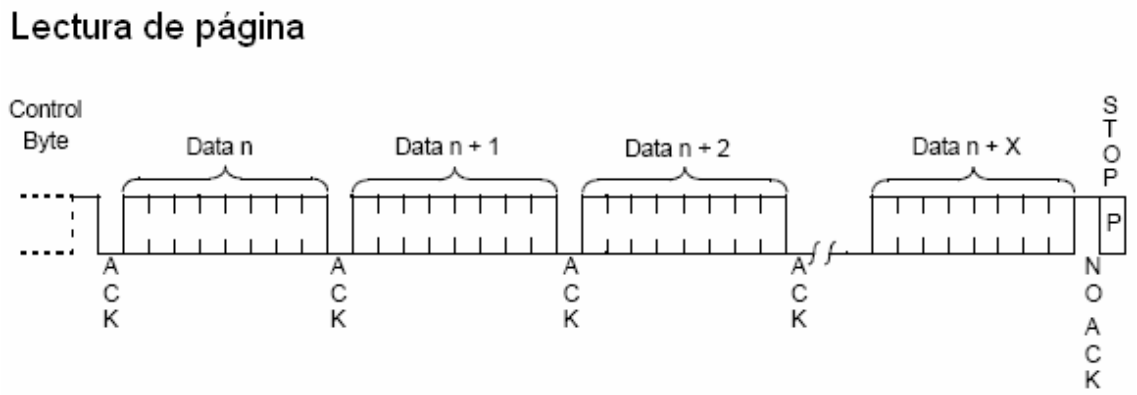
Una vez hecho esto el microcontrolador inicia una nueva transferencia i2c mediante la escritura del bit de START de nuevo sobre el bus, sin enviar el bit de STOP. Tras ello, se direcciona de nuevo el dispositivo, esta vez con el bit de lectura/escritura a 1, indicando una operación de lectura.

En este momento, sucesivas lecturas obtendrán la información contenida en sucesivas posiciones a partir de la dirección indicada anteriormente. Cada envío deberá ser reconocido con un bit de reconocimiento ACK, excepto el último, que será respondido con no-reconocimiento NACK de forma que de este modo el dispositivo interpreta que no ha de enviar más posiciones y se va a finalizar la comunicación.

Tras ello, el microcontrolador envía al bus el bit de STOP y da por concluida la lectura de datos.

Del mismo modo que se posibilita la opción de lectura en modo página, también es posible la escritura de este modo. A continuación se muestra el diagrama del flujo de datos durante una operación de escritura de página, como puede observarse no difiere

del modo de escritura aleatoria sino más bien es necesario tener una serie de consideraciones, que son las mismas que en modo de lectura.



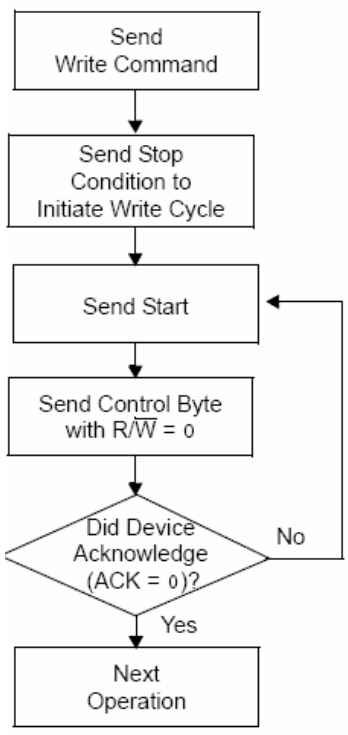
Control de verificación de escritura mediante espera activa

Recuérdese que durante el proceso de escritura el dispositivo no genera bits de reconocimiento de actualización de los registros. Los bits ACK que devuelve al microcontrolador verifican que el dato ha sido recibido correctamente, es decir operan a nivel de transferencia y no de funcionamiento interno de la memoria. Que los datos hayan sido transmitidos correctamente no quiere decir que ya hayan sido escritos. De hecho, durante la transferencia los datos se copian a un buffer intermedio y es tras el último byte de escritura enviado y cuando el microcontrolador cierra la transferencia con el bit de STOP, cuando se inicia el ciclo de escritura de la memoria que actualiza las direcciones de memoria especificadas.

Puede ser necesario, sin embargo, conocer cuando la memoria ha finalizado de actualizar los datos. Por ejemplo, durante un proceso de escritura de múltiples páginas, puede ser interesante saber cuando ha acabado de actualizar los datos de la página para iniciar una nueva transferencia i2c con los datos de la siguiente página.

Con el fin de llevar a cabo un control del proceso de escritura de datos puede implementarse un algoritmo de espera activa aprovechándose de una característica de la memoria. Cuando se direcciona el dispositivo para iniciar una operación de escritura y todavía hay una escritura en curso, el dispositivo no devuelve ACK sino NAK, mostrando la indisponibilidad en la que se encuentra.

El siguiente diagrama, extraído de la página 10 del *datasheet*, muestra las directivas a tener en cuenta a la hora de implementar este algoritmo de espera activa.



Como puede observarse, se parte del momento en el que se envía el bit de stop tras los comandos de escritura pertinentes. Tras ello, la memoria comenzará el proceso de actualización de las direcciones en cuestión volcando sobre ellas el buffer de recepción intermedio.

La espera activa comienza inmediatamente después de enviar el bit de STOP. El microcontrolador escribe en el bus el bit de START para acto seguido direccionar el dispositivo en modo escritura. Si el dispositivo responde con ACK, ha terminado de refrescar los registros internos con los datos previamente enviados y se encuentra disponible para una nueva operación de escritura. En caso de no devolver el bit ACK, todavía se encuentra bajo este proceso. En este caso, se vuelve a intentar iniciar una transferencia de escritura hasta que la memoria devuelva ACK. Cuando devuelva ACK, se cierra la comunicación y se da por concluida la escritura con espera.

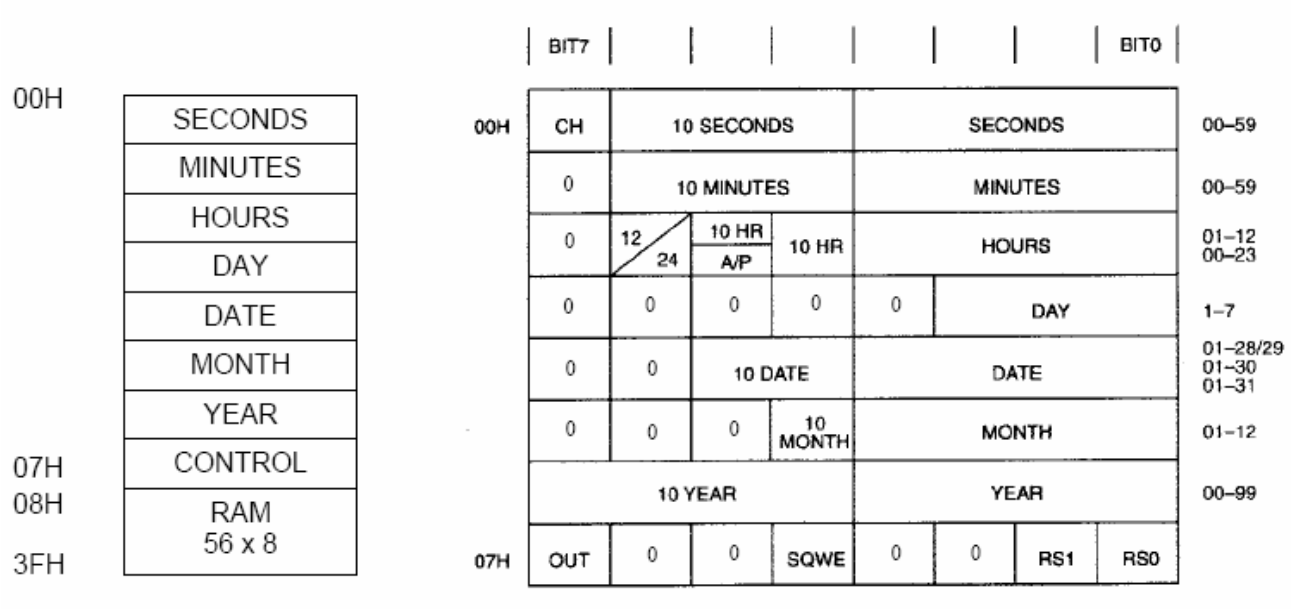
3.2.1.3.- Timer RTC DS1307

Con la finalidad de hacer posible el acceso a las funciones del *timer* RTC DS1307 se ha llevado a cabo el desarrollo de un driver que pone a disposición del programador una API de acceso y manejo al mismo.

Para ello como primer paso ha sido necesario el estudio del *datasheet* ya que éste documento refleja las consideraciones a tener en cuenta, el *timing*, mapeo de registros, bits de configuración, las velocidades, protocolos y formato de acceso al bus y todo lo relacionado con la comunicación.

Como se puede observar, el *timer* posee un mapeo de registros internos que hay que se debe tener en cuenta a la hora de la implementación del driver. Comenzando por la dirección 0x00 de forma ascendente, se encuentran localizados los registros correspondientes a los segundos, minutos, horas, día de la semana, día del mes, mes, año y registro de control. Mediante la escritura y lectura de estos registros es posible acceder a toda la funcionalidad del chip.

Extraído de las págs.. 4 y 5 del *datasheet*:



Es importante hacer notar que el valor de los registros de hora y fecha (0x00 a 0x06) no se encuentran en formato binario natural sino en formato BCD, pasar este detalle por alto imposibilita por completo la programación de las funciones de acceso ya que daría lugar a escritura y lectura de valores erróneamente interpretados (página 4 del *datasheet*). El formato de representación BCD difiere del binario natural. El formato de representación BCD representa cada byte descomponiendo el mismo en dos mitades, el *nibble* (grupo de 4 bits) superior y el *nibble* inferior para posteriormente interpretarlos por separado como dígitos independientes. Por ejemplo, para el valor en decimal 148:

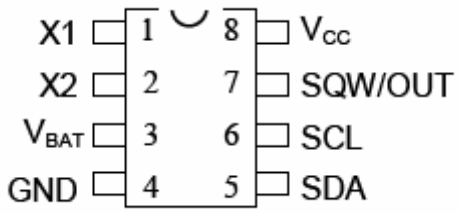
El bit 7 del registro de los segundos (registro dirección 0x00) controla el estado del reloj. Poniendo el bit a 0 se habilita el dispositivo, si se pone a 1 se detiene inhabilita el paso del tiempo pausando la cuenta del tiempo.

El *timer* ofrece dos modos de funcionamiento, funcionamiento en modo 12 horas y modo 24 horas. En caso de encontrarse funcionando bajo el formato horario de 12 horas, el bit 5 del registro horario (0x02) indica AM (bit = 0) o PM (bit = 1) (página 4 del *datasheet*).

La diferencia entre el registro 0x03 y el 0x04 es que el registro 0x04 contiene el día del mes mientras que el 0x03 contiene el día de la semana representado por un número entero en el rango [0-7], estableciendo una correspondencia lineal con los días de la semana en forma ascendente:

0x01 → Lunes
0x02 → Martes
0x03 → Miércoles
0x04 → Jueves
0x05 → Viernes
0x06 → Sábado
0x07 → Domingo

El *timer* pone también a disposición del desarrollador acceso a la onda cuadrada generada por el cristal de cuarzo soldado a sus patillas 1 y 2. Dicha onda es accesible a través de la patilla 7 del chip:



El control de la salida sobre esta patilla se delega en el registro de control mapeado en la dirección 0x07 del dispositivo. Escribiendo sobre él es posible habilitar o deshabilitar la salida de onda, así como seleccionar entre cuatro frecuencias de salida disponibles en base a divisores de frecuencia internos, o fijar el nivel de salida lógico cuando la salida de onda se encuentra deshabilitada.

Registro de control (0x07)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OUT	0	0	SQWE	0	0	RS1	RS0

OUT → Controla el nivel lógico de salida de la patilla cuando la salida de onda se encuentra deshabilitada. Si OUT=0 la salida
queda fija a nivel alto, si OUT=1 la salida queda fija a nivel bajo.

SQWE → Habilita o deshabilita la salida de onda a través de la patilla 7. Si SQWE=1 la salida se encuentra habilitada, si SQWE=0 se encuentra deshabilitada.

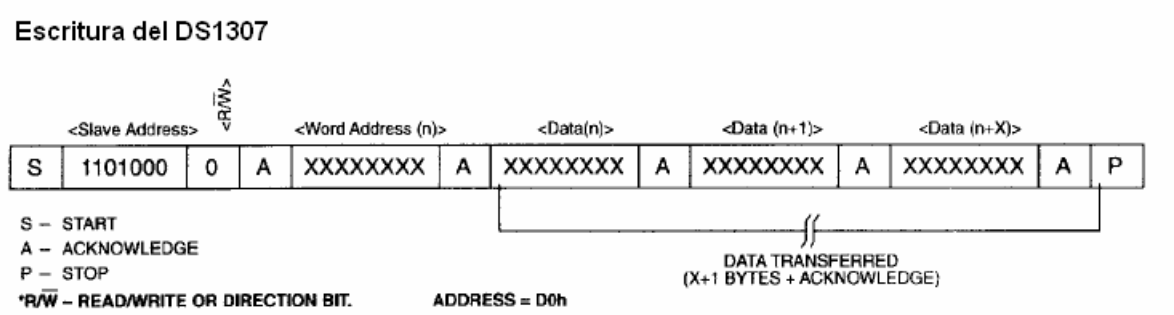
RS1, RS0 → Controlan la frecuencia de salida de la onda a través de la patilla 7 según la siguiente tabla (página 5 del *datasheet*):

RS1	RS0	SQW OUTPUT FREQUENCY
0	0	1Hz
0	1	4.096kHz
1	0	8.192kHz
1	1	32.768kHz

Si la frecuencia de salida se configura a 1 Hz, el flanco de bajada de la onda corresponde con el instante de actualización de los registros internos del *timer*.

Es importante recalcar a la hora de realizar la programación del microcontrolador que a pesar de que existen diversas velocidades posibles a la hora de abrir la conexión i2c, este dispositivo únicamente permite operar bajo la velocidad de trabajo de 100 KHz, limitando de este modo la velocidad de trabajo del bus a pesar de que los demás dispositivos anclados a él permitan trabajar con velocidades mayores.

Escritura de los registros del *timer*.



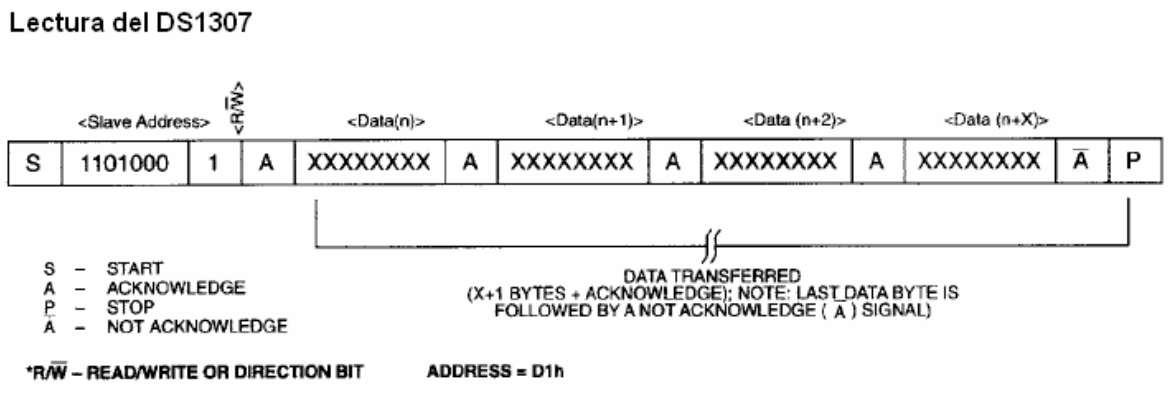
La figura muestra el orden de los datos durante un flujo de datos de un proceso de escritura de los registros del *timer*. Primeramente el microcontrolador debe generar el bit de START.

Acto seguido debe direccionar el dispositivo escribiendo la dirección del mismo en el bus. La dirección del dispositivo está formada por 8 bits, de los cuales los 7 bits de mayor peso conforman la dirección física fija del mismo. El bit de menos peso indica si se está solicitando una operación de escritura o de lectura. Tanto si este bit vale 0 como 1, el dispositivo queda direccionado si los 7 bits de mayor peso de la dirección se corresponden con el valor en binario "1101000". Es decir, el dispositivo se puede direccionar a través de la dirección 0xD0 para una operación de escritura y con la dirección 0xD1 para una operación de lectura.

El siguiente byte hace referencia a la dirección del registro sobre el cual se va a realizar la operación de escritura. Una vez especificado el registro, una posterior escritura actualizará el valor del registro apuntado. El registro apuntador es autoincremental, esto significa que posteriores valores escritos en el bus actualizarán cada vez el registro inmediatamente posterior. Esta característica, presente en muchos dispositivos, permite agilizar el intercambio de información permitiendo llevar a cabo escrituras sobre registros consecutivos sin necesidad de iniciar una nueva transferencia y direccionamiento por separado.

Tras cada byte enviado, el dispositivo receptor del mismo ha de responder con un bit de reconocimiento ACK (*Acknowledgment*). Tras escribir el último valor del último registro deseado, el microcontrolador debe generar el bit de STOP dando por concluido el intercambio de información y liberando el bus.

Lectura de los registros del *timer*.



De manera análoga al proceso de escritura en el dispositivo, el intercambio de información deberá comenzar colocando el bit de START en el bus i2c para, acto seguido, escribir en el mismo la dirección del *timer* en modo lectura, esto es, con el bit menos significativo a 1. El archivo será direccionado para una operación de lectura, por tanto, escribiendo el valor 0xD1 en el bus i2c. A continuación una lectura del bus i2c

por parte del microcontrolador obtendrá el valor del registro seleccionado en ese momento. Sucesivas lecturas obtendrán valores de los registros inmediatamente consecutivos, de igual forma que sucedía durante el proceso de escritura.

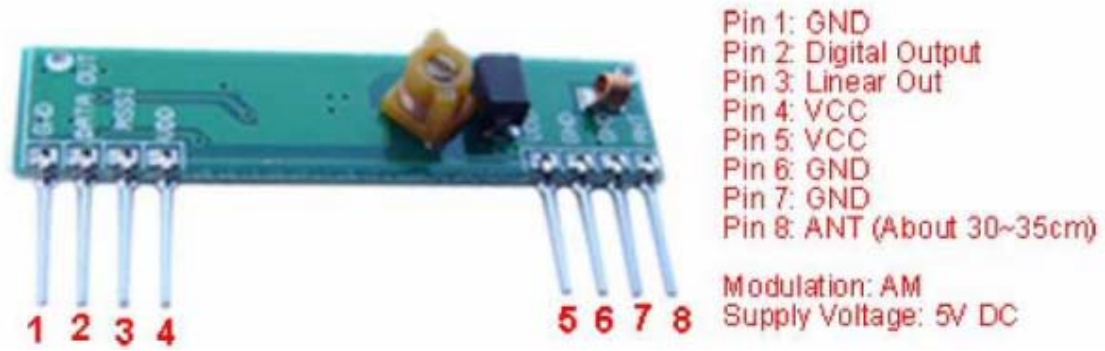
Cada vez que el microcontrolador recibe un byte debe indicarlo mediante el bit de reconocimiento ACK, excepto cuando se dispone a leer el último valor en cuyo caso debe responderse con No-*Acknowledgement*, o bit NACK, haciendo saber de este modo al *timer* que la operación de lectura ha finalizado y no ha de volcar más datos al bus. Acto seguido, el microcontrolador finalizará el intercambio de datos mediante la escritura en el bus del bit de STOP. Viendo el esquema del flujo de datos.

Viendo la figura que indica el orden de los datos durante el proceso de lectura se puede observar que en ningún momento durante la orden de lectura se indica el valor del registro sobre el cual se desea llevar a cabo la operación. De ello se deduce, que a la hora de realizar una operación de lectura deberá anteponerse una operación de escritura, en la cual el microcontrolador escribe el registro apuntador del *timer*, para posteriormente iniciar la secuencia de lectura sobre el registro previamente indicado.

El *datasheet* del dispositivo pone a disposición del desarrollador más información sobre aspectos de *timing* de bus, electricidad, dimensiones, disipación de calor, características eléctricas, etc. Sin embargo con la información comentada hasta ahora se dispone del suficiente conocimiento como para llevar a cabo la implementación del driver del chip.

3.2.1.4.- Módulo RF RX433N

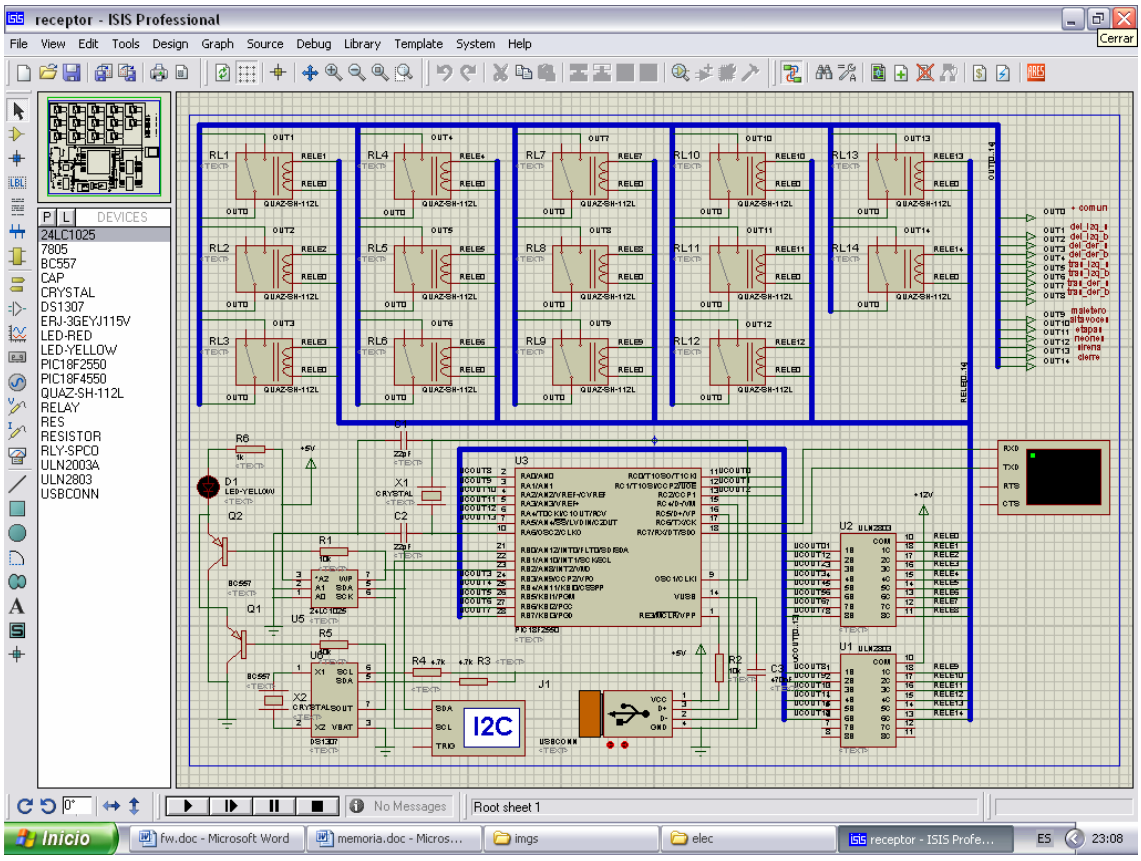
De igual modo que la transmisión por parte del mando emisor se delega en el módulo de radiofrecuencia tx433n, para la recepción se hace uso del módulo complementario, el cual responde a la referencia rx433n, también de la empresa Velleman. La asignación de pines de dicho módulo es la siguiente:



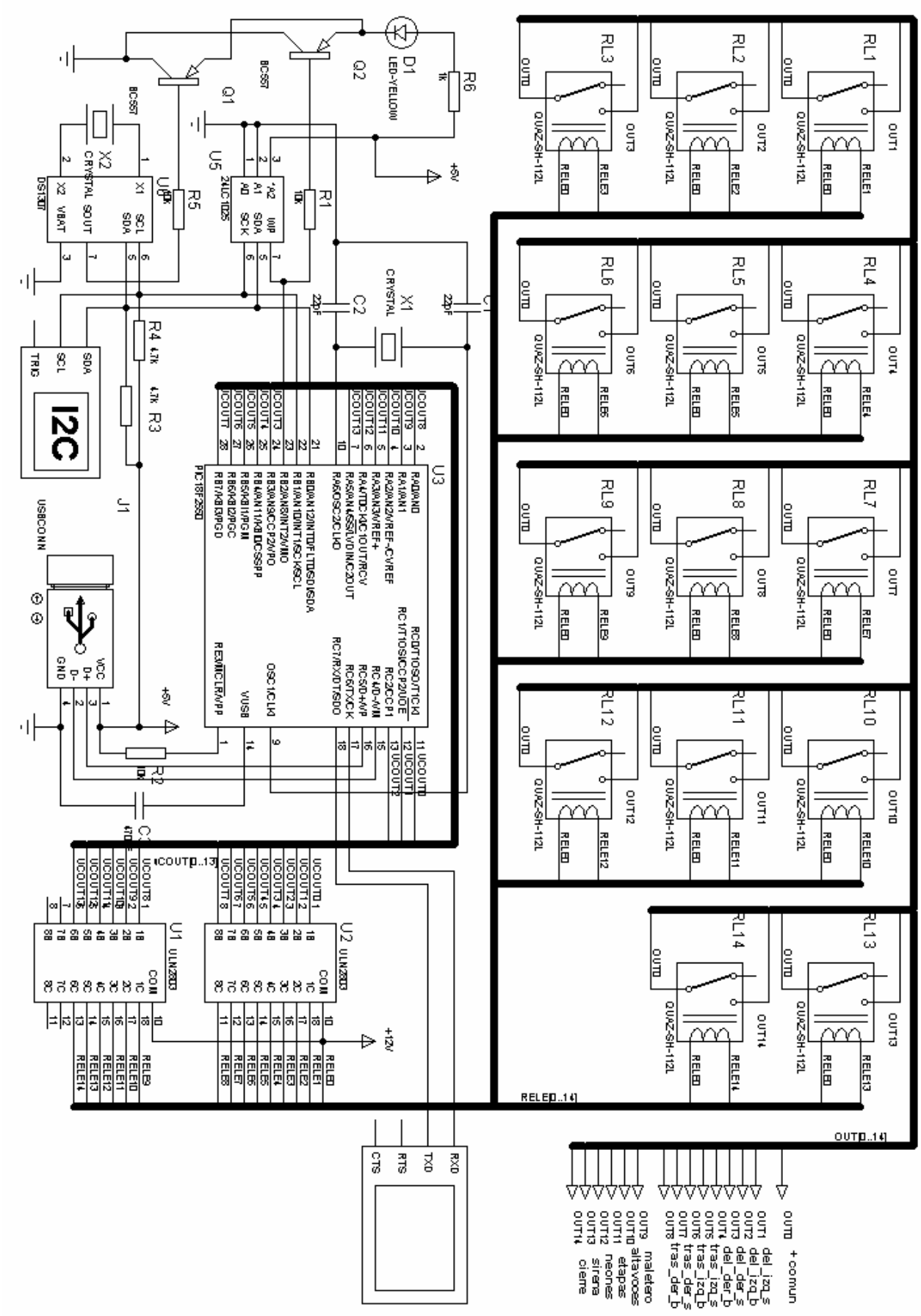
De igual modo que el emisor, trabaja en la frecuencia de 433,92 MHz en modulación ASK. Posee una sensibilidad de -108 dBm y es efectivo en cuanto a recepción de datos sin corrupción hasta una tasa de transferencia de 3 KB/s. Para su correcto funcionamiento ha de ser alimentado con una tensión continua de 5 voltios.

3.2.2.- Desarrollo y construcción

Siguiendo con las herramientas utilizadas para el desarrollo de emisor, se ha llevado a cabo del mismo modo el diseño y simulación del receptor. Se incluye como anexo todo el proyecto en Proteus, con los esquemas y archivos necesarios para las simulaciones pertinentes. La siguiente captura de pantalla muestra un pantallaza de la suite de desarrollo electrónico durante el proceso de diseño del esquema del receptor.

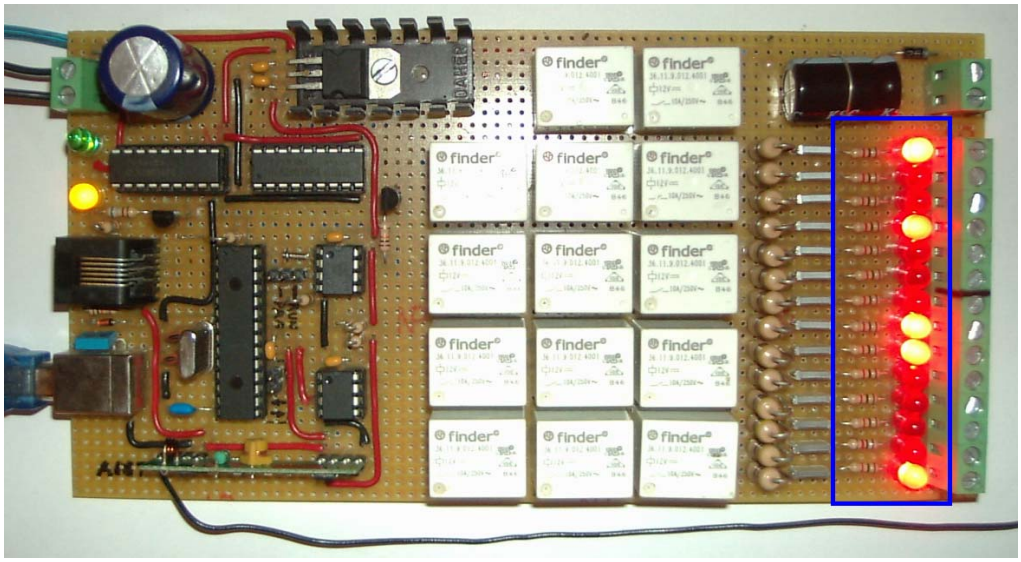


Existen sin embargo algunos componentes que no se han incluido dentro del proyecto en Proteus. Posteriormente se hablará de ellos y los motivos. Haremos uso de la opción “exportar” del menú “File -> Export Graphics -> Export Bitmap” con el fin de exportar el esquema y obtener una imagen del mismo en formato habitual. En la siguiente imagen puede verse el esquema del receptor una vez exportado:

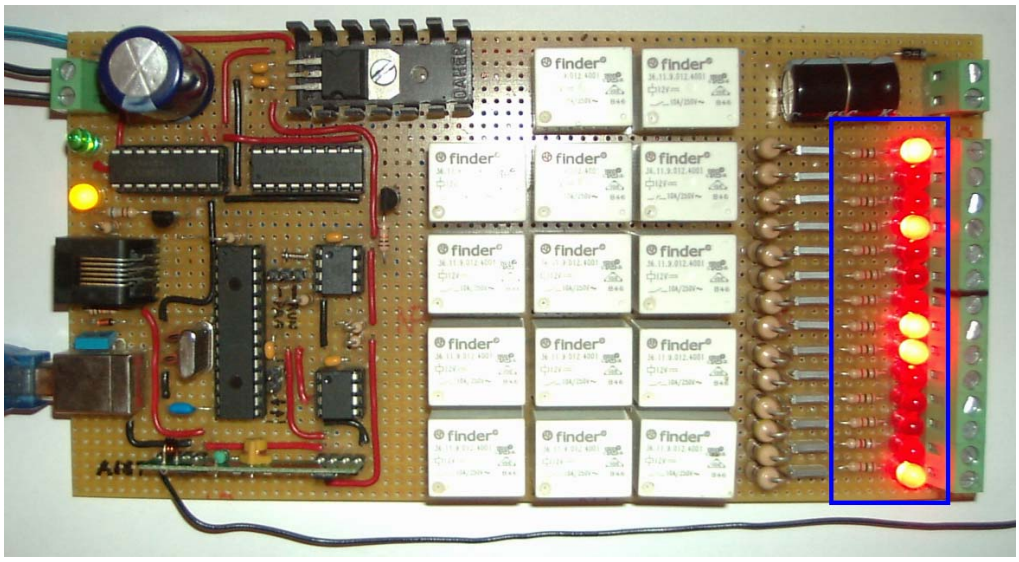

RL1
RL2
RL3
RL4
RL5
RL6
RL7
RL8
RL9
RL10
RL11
RL12
RL13
RL14
QUAZ-SH-112L
OUTD
RELED
OUT1
OUT2
OUT3
OUT4
OUT5
OUT6
OUT7
OUT8
OUT9
OUT10
OUT11
OUT12
OUT13
OUT14
RELE1
RELE2
RELE3
RELE4
RELE5
RELE6
RELE7
RELE8
RELE9
RELE10
RELE11
RELE12
RELE13
RELE14
RELEp..14
OUTp..14
+comun
del_izq_s
del_izq_b
del_der_s
del_der_b
tras_izq_s
tras_izq_b
tras_der_s
tras_der_b
maletero
altavoces
etapas
neones
sirena
cierre
U1 ULN2803
U2 ULN2803
COM
+12V
U3
PIC18F2550
RXD
TXD
RTS
CTS
USBCONN
I2C
SDA
SCL
TRIG
D1
LED-YELLOW
Q1
Q2
BC557
U5
24LC1025
U6
DS1307
X1
CRYSTAL
X2
R1
R2
R3
R4
R5
R6
C1
C2
C3
22pF
4.7k
1k
10k
+5V

Como se ha mencionado existen diversos componentes que no aparecen en el esquema. El motivo de ello es la falta de espacio en la superficie de trabajo de Proteus, y el hecho de que su ausencia no tiene consecuencias en la simulación práctica de los circuitos por ordenador. Los elementos que no aparecen en los esquemas son los siguientes:

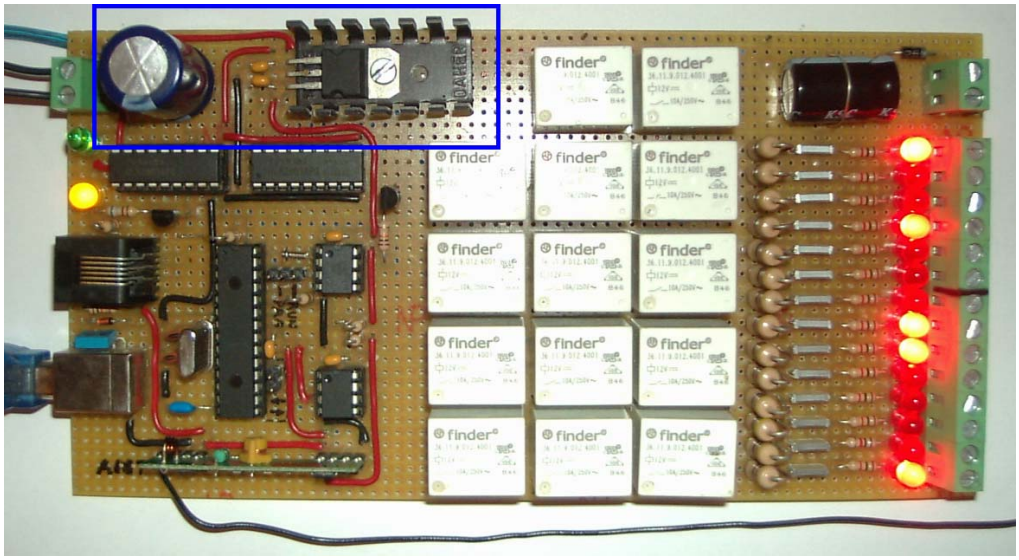
- Leds rojos y resistencias limitadoras de intensidad asociadas. Un conjunto LED-resistencia por cada relé. Cada vez que un LED está activo se enciende su resistencia indicadora.



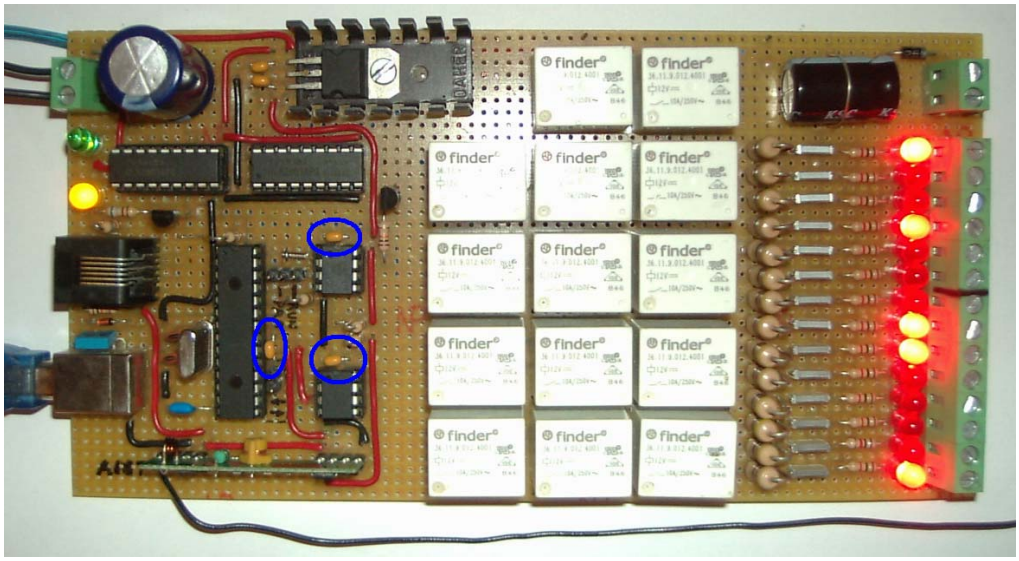
- Conjunto Resistencia-Condensador. Una resistencia de 10Khz en serie con un condensador de 100nF, y cada conjunto resistencia-condensador descrito en paralelo con los contactos del relé. Esto evita los chispazos en los contactos mecánicos de los relés y evita su deterioro con el tiempo, al mismo tiempo que reduce considerablemente las emisiones electromagnéticas producidas como consecuencia del cierre/apertura de los contactos.



- Fuente de alimentación. Reduce la tensión a 5 voltios estabilizados para alimentar los circuitos integrados. La fuente de alimentación está constituida por un regulador integrado 7805 de 3 patillas, así como dos condensadores cerámicos de 100 nF y uno electrolítico de 2200 uF.

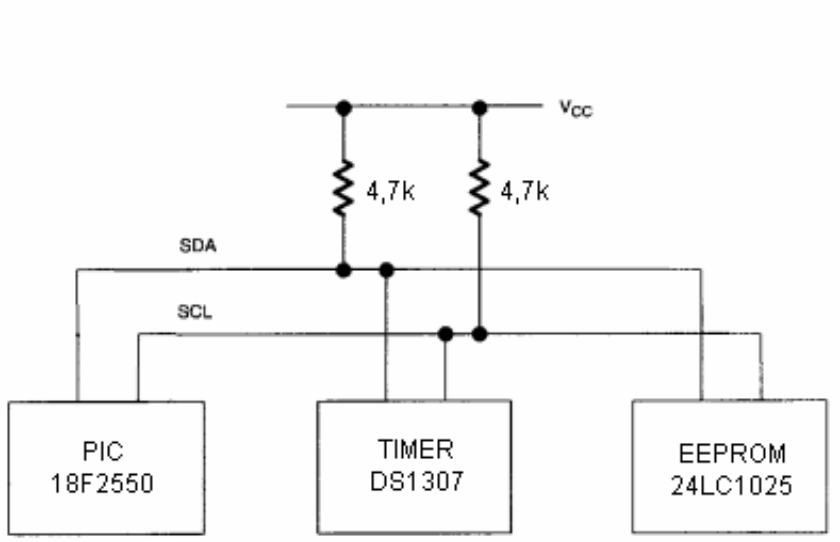


- Un condensador de 100 nF en paralelo con la alimentación lo más cerca posible de cada circuito integrado. Ayuda a reducir transitorios de alta frecuencia en las líneas de alimentación que podrían ocasionar un funcionamiento erróneo de los componentes o el cuelgue del microcontrolador.



Toda la circuitería tiene como epicentro el microcontrolador PIC18F2550. Como fuente de oscilación se ha utilizado un cristal de cuarzo de 20 MHz, acompañado de dos condensadores cerámicos de 22pF.

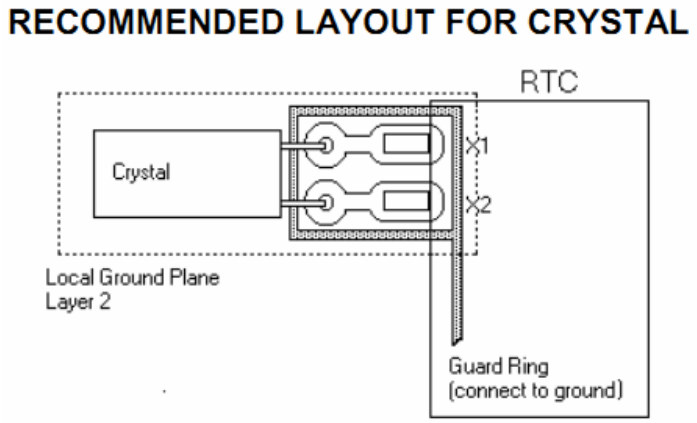
Para llevar a cabo el trazado de la conexión del bus i2c, que une el microcontrolador con la memoria EEPROM y el Timer RTC, se han colocado estos tres elementos lo más cerca posible entre sí de forma que la longitud de las líneas sea lo más corta posible. Tal y como marca el estándar se han colocado dos resistencias de *pull-up* de 4,7 Kohm a positivo, ello es debido a que los dispositivos que hacen uso de este bus implementan la salida en configuración de colector abierto para permitir el acceso compartido a las líneas de datos por lo que únicamente pueden colocar la línea a nivel bajo, siendo necesario fijar las líneas a nivel alto de forma externa mediante resistencias cuando ninguno de los dispositivos anclados al bus está manteniendo la línea a nivel bajo. Esta es la función de las resistencias de pull-up.



El módulo DS1307 funciona a dúo con un cristal de 32.768 KHz, siendo éste la base de tiempos de onda cuadrada que el circuito integrado utiliza de forma interna. Es necesario soldar el cristal de cuarzo lo más cerca posible al circuito integrado, asimismo el fabricante recomienda el uso de planos de masa para el cristal. Sin embargo a nivel práctico esto no es necesario. Se ha soldado un cable proveniente de masa la carcasa metálica del cristal de cuarzo haciendo el efecto de Jaula de Faraday (obsérvese el reducido tamaño del cristal en las imágenes del receptor).

La finalidad de la incorporación de un chip cuya capacidad es la de ofrecer la hora en todo momento cobra sentido a la hora de la reproducción de secuencias preprogramadas. Cada secuencia preprogramada lleva asociado un tiempo, codificado en la propia instrucción, que indica la duración mediante la cual tiene validez dicha instrucción. Es decir, si una instrucción bajo ejecución de la secuencia indica que sendas ruedas delanteras deben elevarse, también se indica el tiempo durante el cual debe llevarse a cabo. Es decir, el tiempo que se estarán elevando antes de dar por concluida la instrucción y pasar a leer la siguiente orden programada en la memoria. Con el fin de controlar el tiempo transcurrido desde que se inicia la ejecución de la instrucción se incorpora en el circuito del receptor este timer. Mediante diversas consultas sucesivas (espera activa) es posible conocer el momento exacto en el cual ha

transcurrido el tiempo deseado y se debe dar por concluida la ejecución de la instrucción, pasando a la siguiente.



La conexión de la memoria EEPROM tampoco presenta mayores complicaciones, únicamente es necesario tener en cuenta la conexión de las patillas del bus i2c y el cableado de la patilla de protección de escritura WP a la patilla correspondiente a RB2 del microcontrolador.

El motivo de la incorporación de una memoria EEPROM en el circuito es dotar al mismo de la capacidad de almacenar una cantidad relativamente elevada de datos. El PIC también posee una zona de memoria regrabable por el firmware y accesible al programador, sin embargo está limitada a 256 bytes, una capacidad muy reducida para este caso. Por ello se opta por el uso de una memoria externa, la finalidad de la misma es servir de almacén regrabable de las secuencias de instrucciones programadas por el usuario.

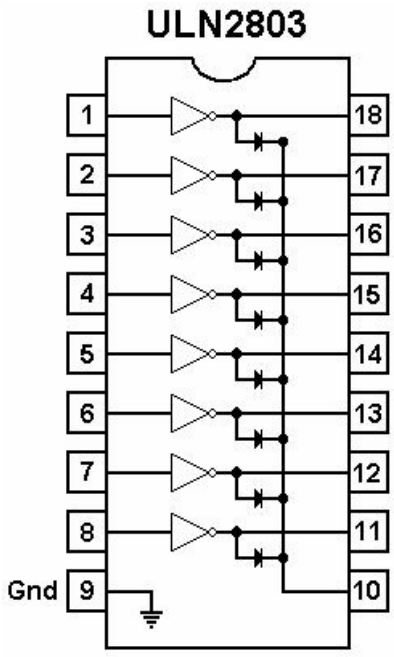
El diodo LED amarillo D1 está controlado por dos transistores PNP BC557 en paralelo. Ello permite ser activado al mismo tiempo por el microcontrolador cuando se habilita la escritura en la EEPOM a través de la patilla WP de la misma, y por el timer DS1307 en una de sus patillas que ofrece una señal cuadrada. La finalidad de ello es hacer posible que el led esté apagado, fijo o parpadeando a una frecuencia de 1 Hz (en este último caso se aprovecha de la patilla de salida que habilita el timer). Ello ofrece la posibilidad de ser usado para mostrar visualmente la actividad interna del receptor, utilizando este LED como elemento indicador según el caso.

Por otra parte se dota al sistema de un conector USB de tipo B, cableado directamente al microcontrolador a través de las patillas habilitadas al efecto en él. El condensador cerámico de 470 nF es necesario para asegurar el correcto funcionamiento del bus.

La resistencia de 10k conectada a positivo entrega un nivel alto a la entrada MCLR del PIC. Esta entrada resetea el microcontrolador cuando se encuentra a nivel bajo, mediante esta resistencia se asegura el correcto funcionamiento del mismo evitando situaciones de inestabilidad, si se deja esta patilla al aire sin conexión el

funcionamiento del microcontrolador no será el esperado. En algunos casos puede no llegar a funcionar directamente.

Los dos integrados ULN2803 conforman la etapa de potencia que separa la parte lógica del receptor de la parte destinada al control de los dispositivos, motivo de la existencia del circuito.



Cada uno de estos circuitos incorpora en su interior 8 drivers de corriente capaces de servir para el control de cargas de un consumo relativamente alto. Cada uno de estos drivers está constituido en base a un transistor Darlington en corte/saturación. En el modo de saturación es capaz de soportar una intensidad máxima de 500 mA, suficiente para el caso que nos ocupa, teniendo en cuenta que la intensidad consumida por cada relé ronda los 30 mA. Hay que tener en cuenta que el ULN2803 no entrega corriente, sino que cierra a masa la carga conectada a él. Pasar por alto esta consideración durante un diseño electrónico que haga uso de estos integrados puede derivar en fallas difíciles de detectar.

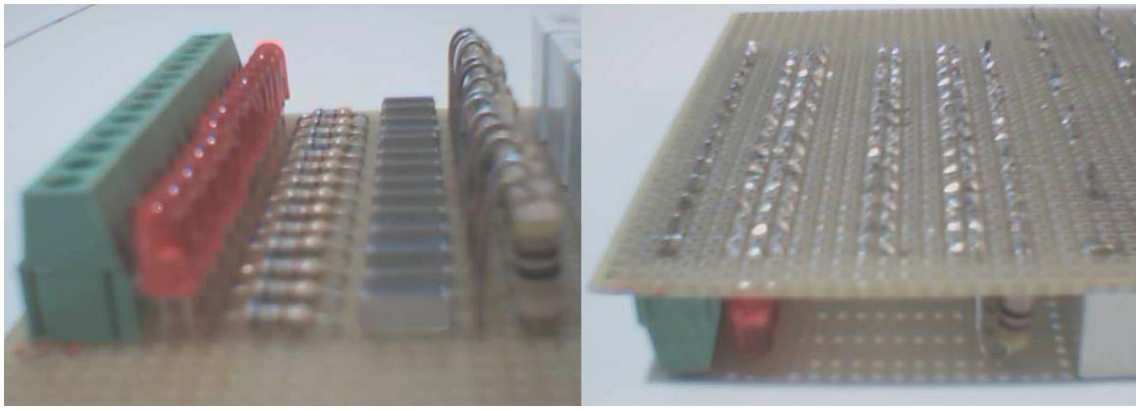
La salida de cada uno de los drivers de corriente internos del chip incorpora un diodo en paralelo a una línea de masa común. En la figura superior puede observarse la conexión. Se trata de un diodo con una tensión inversa de ruptura bastante alta, idóneo para el empleo de relés, motores, solenoides y otras cargas inductivas que puedan producir corrientes inversas durante su funcionamiento. De este modo nos ahorramos también el diodo que de otro modo habría que colocar por cada relé de forma externa.

Una vez descrita la visión general del circuito electrónico del receptor se pasará a mostrar los pasos de montaje llevados a cabo.

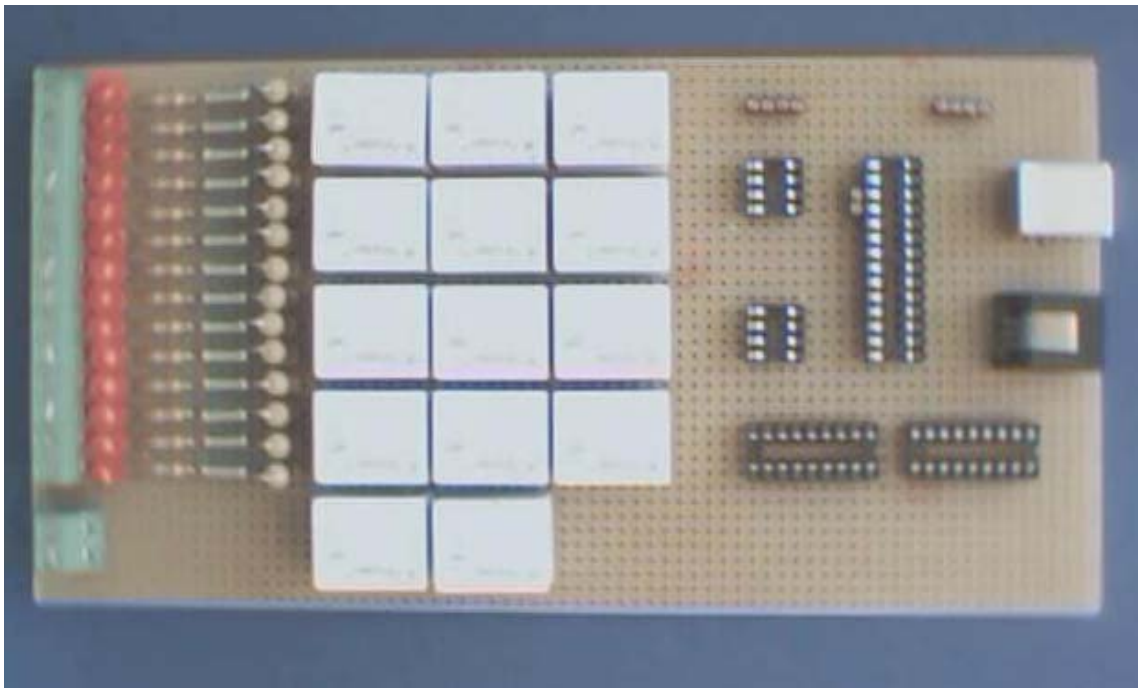
Primeramente se ha cortado la placa de circuito impreso al tamaño adecuado y se han soldado los relés. Posteriormente se ha procedido del mismo modo con las regletas de conexión de los actuadotes y los LEDs señalizadores de estado, uno por cada LED. Cada LED va emparejado de una resistencia de 1k, la cual debe ser colocada en serie.

Del mismo modo, se han soldado también el conjunto condensador-resistencia en paralelo entre los contactos de salida de cada uno de los relés.

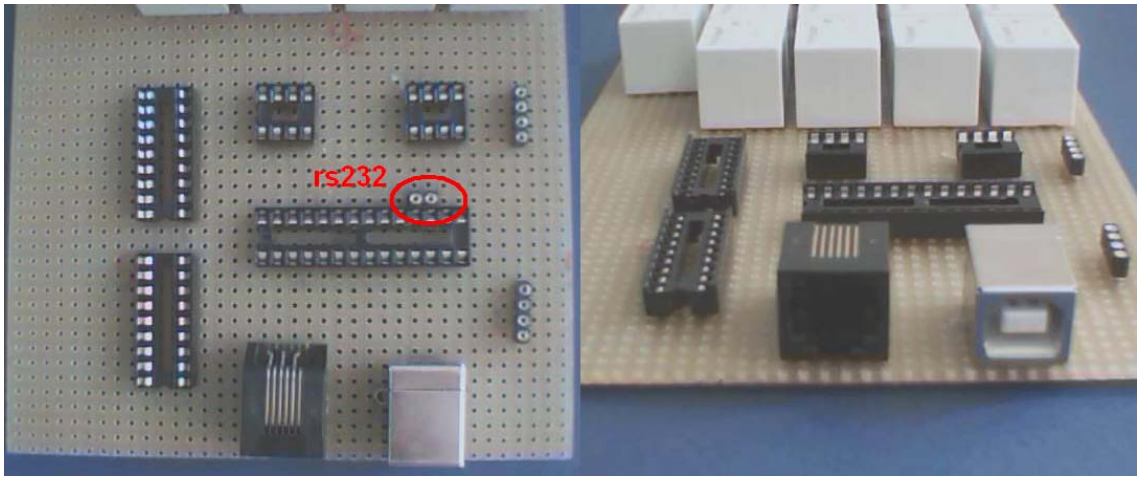
Aquí se muestra otro ángulo de la placa:



Como siguiente paso se colocarán los zócalos de todos los integrados, así como el conector RJ-12 para el programador ICD2 y el conector USB de tipo B en su localización correspondiente.

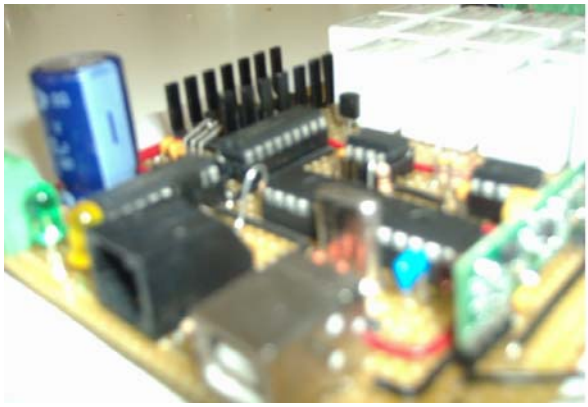


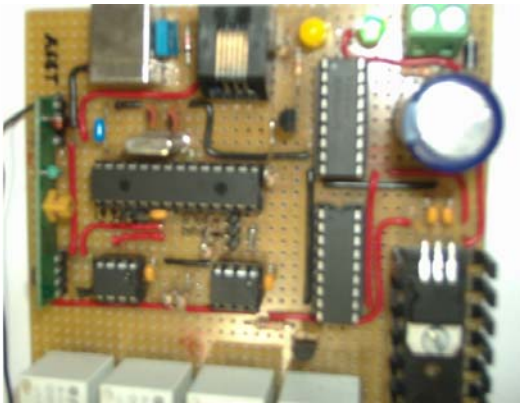
La conexión de dos patillas marcada en rojo permite acceder directamente a las patillas TX y RX de la USART del microcontrolador. Ello permite comunicarse directamente con el PIC desde el Hyperterminal de Windows, siendo de gran utilidad en tareas de depuración durante el periodo de desarrollo del firmware.



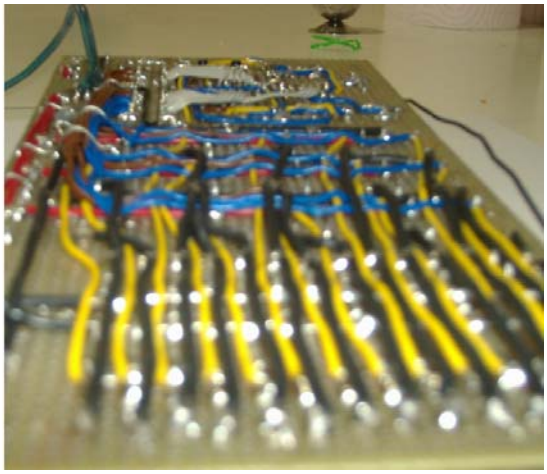
No es preciso detallar el proceso de soldado de cada uno de los componentes del sistema, se muestran a continuación imágenes de la placa ya finalizada una vez soldados todos sus componentes y llevado a cabo el conexionado de todos ellos por la parte posterior de la misma.

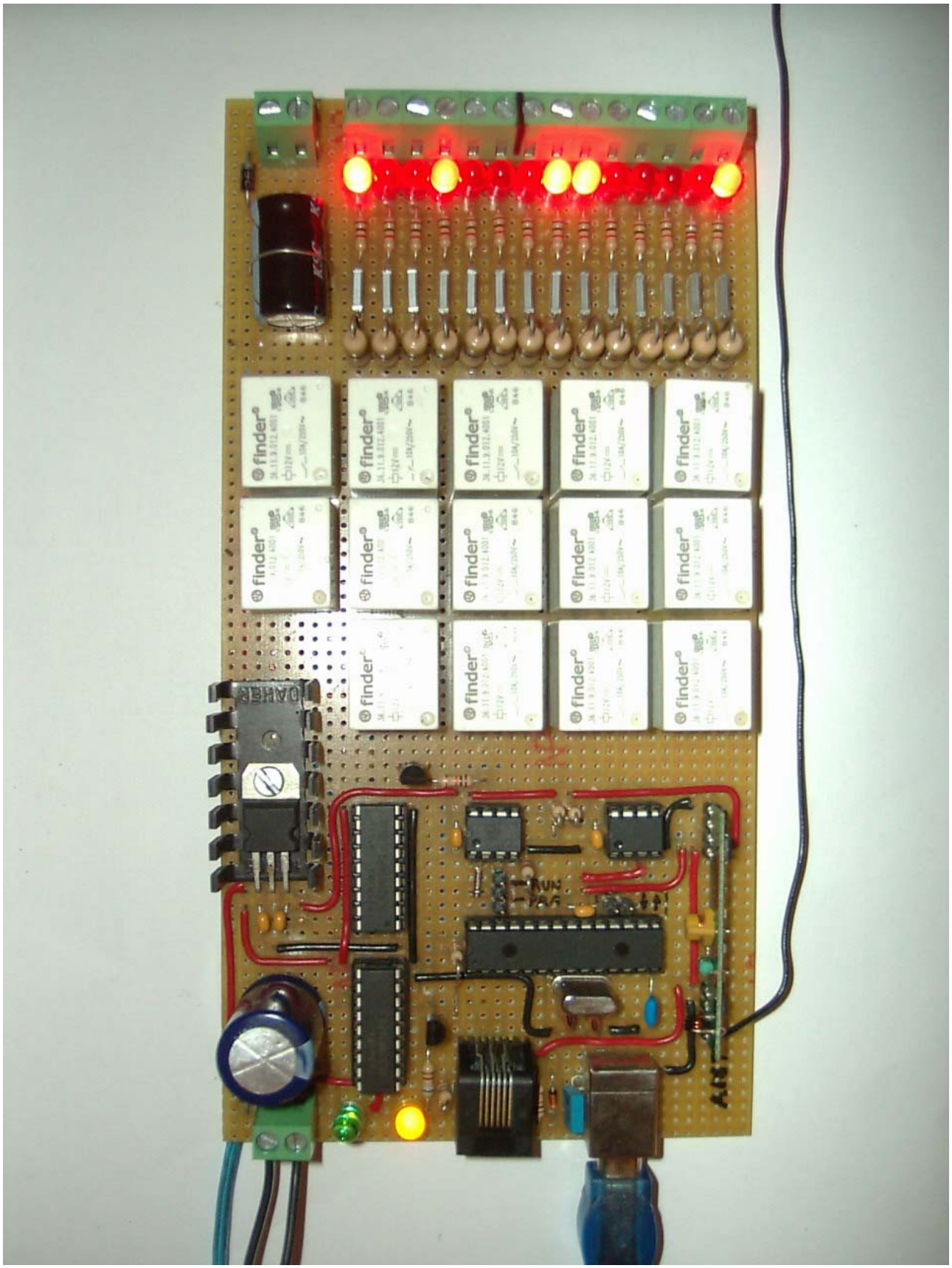
Vista superior:

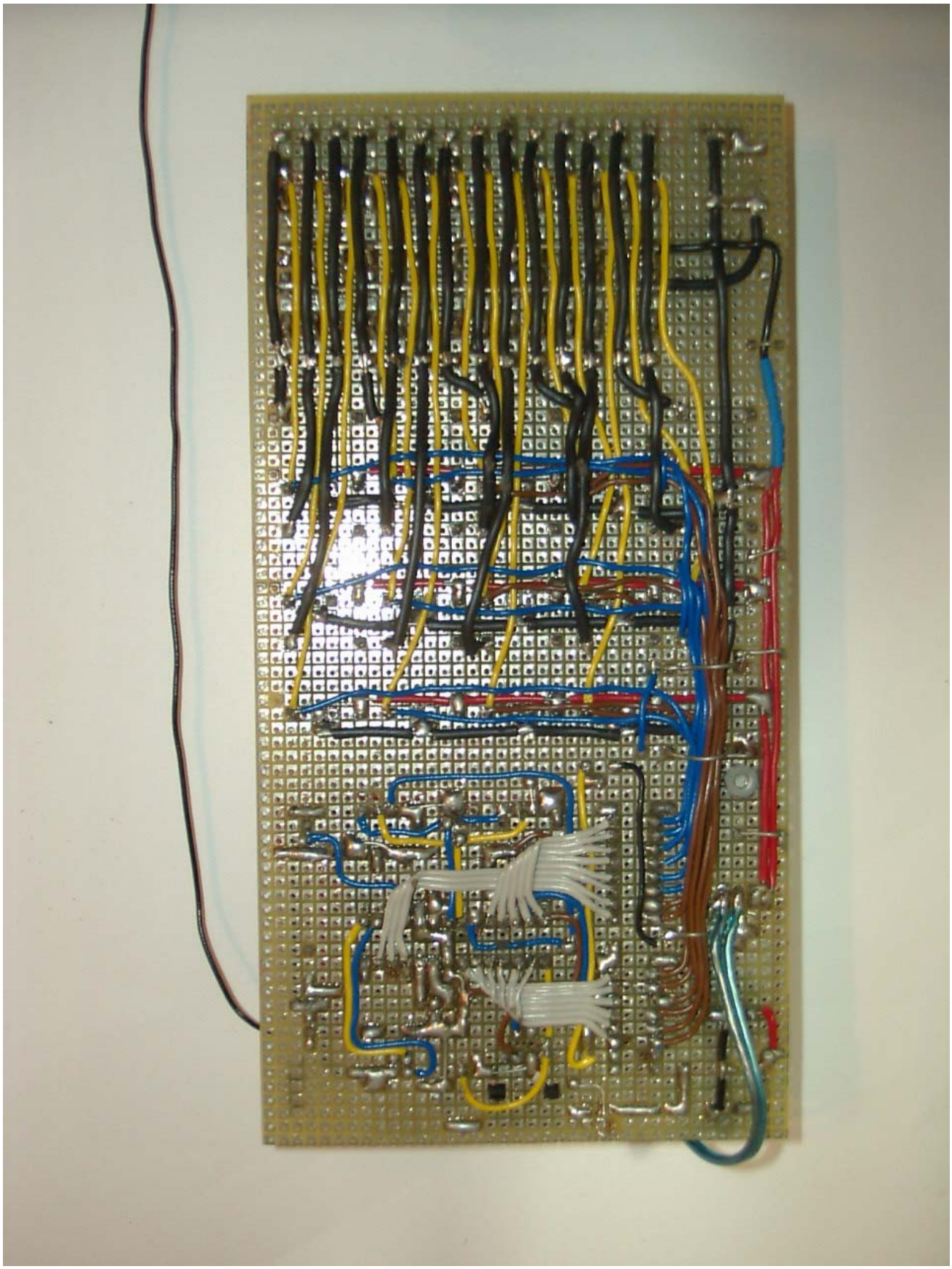




Vista anterior:




finder



4.- Software

A continuación se tratará los componentes software del sistema. Este concepto engloba tanto el firmware de los microcontroladores como el programa de usuario para la programación de secuencias en el receptor.

4.1.- Librerías de componentes desarrolladas.

Con el fin de permitir al microcontrolador PIC18F2550 alojado en el receptor hacer uso de las funciones ofrecidas por la memoria EEPROM y el timer RTC se han desarrollado sendas librerías en CCS. El código llevado a cabo ofrece una API (Application Programming Interface), esto es, una serie de funciones bien definidas que dan acceso a los chips de los cuales son motivo.

Dicho de otro modo, se han desarrollado drivers para el manejo de la memoria 24LC1025 y el timer DS1307. El código generado se ha extraido en archivos aparte a modo de librería estática, ofreciendo por cada chip un archivo de cabecera *.h* y un archivo de implementación *.c* de forma que puede ser reutilizado en futuros proyectos. Tan solo basta con incluir el archivo de cabecera con la directiva del preprocesador "include" al principio del código include.

4.1.1.- Memoria 24LC1025

Siguiendo con la metodología de trabajo del lenguaje C, se desarrollará el archivo de cabecera 24LC1025.h el cual contendrá las definiciones de datos y de funciones de la API del dispositivo que posteriormente se implementarán en el archivo de código de mismo nombre y extensión .c, 24LC1025.c.

Se comenzará analizando el archivo 24LC1025.h:

```
26  #ifndef EEPROM_24LC1025_H
27  #define EEPROM_24LC1025_H
28
29
30  #ifndef EEPROM_24LC1025_PINOUT
31      #error EEPROM_24LC1025: Pinout no definido.
32  #endif
33
34
35  #define EEPROM_24LC1025_write_enable()      output_low(EEPROM_24LC1025_WP);
36  #define EEPROM_24LC1025_write_disable()     output_high(EEPROM_24LC1025_WP);
37
38  #define EEPROM_24LC1025_ADDR_BANK0_R    0xA1
39  #define EEPROM_24LC1025_ADDR_BANK0_W    0xA0
40  #define EEPROM_24LC1025_ADDR_BANK1_R    0xA9
41  #define EEPROM_24LC1025_ADDR_BANK1_W    0xA8
42
43  #define EEPROM_24LC1025_PAGE_SIZE       128
44  #define WRITE_WAIT                      0xFF
45  #define WRITE_NOWAIT                    0x00
46
47
48  void EEPROM_24LC1025_init();
49  void EEPROM_24LC1025_write(int32 addr, byte data, byte wait);
50  byte EEPROM_24LC1025_read(int32 addr);
51  int EEPROM_24LC1025_page_write(int32 addr, byte *buffer, byte wait);
52  int EEPROM_24LC1025_page_read(int32 addr, byte *buffer);
53
54  #include "24LC1025.c"
55
56  #endif
```

El código se encierra en un bloque de instrucción condicional al preprocesador del compilador #ifndef-#endif con el fin de evitar duplicidad de definiciones en caso de incluir el archivo de la librería mas de una vez.

Se controla que se ha definido el *pinout* del dispositivo comprobando si está definida la macro EEPROM_24LC1025_PINOUT, en caso contrario el compilador interrumpirá la compilación y mostrará un error. Los pines necesarios a definir son, por una parte, los relativos a la conexión i2c y por la otra el pin EEPROM_24LC1025_WP, que está directamente cableado a la patilla 7 del integrado y controla la protección de escritura.

Las líneas 35 y 36 implementan la habilitación o inhabilitación de escritura poniendo a 0 o 1 respectivamente dicha patilla, debido a que únicamente se componen de una instrucción se ha optado por crear una macro en lugar de una función para ello.

Acto seguido, se define la dirección del dispositivo para ambos bancos (banco 0 y banco 1) y ambos modos de acceso, lectura y escritura. Es importante hacer notar que esta definición tiene en cuenta que las patillas A1 y A0 de la memoria están cableadas a 0, en caso de no ser así la definición de la dirección divergirá de la especificada (líneas 38 a 41).

Posteriormente se define el tamaño de página a pesar de que siempre va a ser fijo, por comodidad, facilidad de lectura de código y de mantenimiento del mismo. Del

mismo modo se definen dos constantes cuya función es ser especificadas como parámetro en algunas funciones de la API (líneas 43 a 45).

void EEPROM_24LC1025_init(): función de inicialización del dispositivo. Debe ser llamada antes que cualquier otra función de la API.

void EEPROM_24LC1025_write(int32 addr, byte data, byte wait): permite la escritura aleatoria del dispositivo. “addr” contiene la dirección en una variable de 32 bits, “data” contiene el dato en un valor de 8 bits a escribir y “wait” hace referencia a si la función debe bloquearse hasta que esté actualizado el dato o debe retornar de inmediato (valor EEPROM24_LC1205_WAIT o EEPROM24LC1025_NOWAIT).

byte EEPROM_24LC1025_read(int32 addr): lleva a cabo una operación de lectura aleatoria en memoria. Lee el valor apuntado por la dirección de 32 bits contenida en “addr”.

int 24LC1025_page_write(int32 addr, byte *buffer, byte wait): Escribe la página de 128 bits de tamaño de datos de tipo byte en la dirección especificada por “addr”. El parámetro “wait” permite controlar, del mismo modo que su análoga para acceso aleatorio, si debe esperar a que el proceso de escritura haya concluido o si el retorno de la función es inmediato tras finalizar la transferencia i2c con el dispositivo.

En caso de que la dirección especificada no sea múltiplo de 128 y no sea por tanto una dirección válida no se lleva a cabo ninguna operación y devuelve “1”. En caso de que la dirección sea correcta, se lleva a cabo la operación y se devuelve “0”. Se transferirán exactamente 128 bits, en caso de que el buffer sea mayor solo se transferirán los 128 primeros bits y en caso de que sea menor se transferirán los datos de las posiciones de memoria siguientes fuera del rango del tamaño del buffer.

int 24LC1025_page_read(int32 addr, byte *buffer): Lee una página de 128 bits de tamaño de datos de tipo byte en la dirección especificada por “addr” y los coloca en “buffer”. En caso de que la dirección especificada no sea múltiplo de 128 no se lleva a cabo ninguna operación y se devuelve “1”, en caso de que sea válida se lleva a cabo la lectura de página y devuelve “0”. En caso de que el buffer sea de mayor tamaño solo se rellenarán con datos los 128 primeros bytes, y en caso de que sea menor el resultado es impredecible corrompiendo las direcciones de memoria adyacentes al buffer.

Finalmente, en la línea 54 se incluye el archivo de código que implementa las funciones definidas y se cierra el bloque condicional del preprocesador dando por concluido de este modo el archivo de definiciones. A continuación se estudiará el archivo 24LC1025.c por partes:

EEPROM_24LC1025_init():

```
37  void EEPROM_24LC1025_init()
38      {
39      EEPROM_24LC1025_write_disable();
40      }
```

La función de inicialización únicamente controla la patilla de habilitación / deshabilitación de escritura, inhabilitándola.

EEPROM_24LC1025_write()

```
53  void EEPROM_24LC1025_write(int32 addr, byte data, byte wait)
54      {
55      int status;
56      byte command;
57
58      if(addr > 65535) command = EEPROM_24LC1025_ADDR_BANK1_W;
59      else command = EEPROM_24LC1025_ADDR_BANK0_W;
60
61      EEPROM_24LC1025_WRITE_ENABLE();
62      delay_ms(1);
63
64      #ifdef USE_INTERRUPTS
65          disable_interrupts(GLOBAL);
66      #endif
67
68      i2c_start();
69      i2c_write(command);
70      i2c_write(addr >> 8);
71      i2c_write(addr);
72      i2c_write(data);
73      i2c_stop();
74
75      if(wait == WRITE_WAIT)
76          {
77          do
78              {
79              i2c_start();
80              status = i2c_write(command);
81              } while(status == 1);
82          i2c_stop();
83          }
84
85      #ifdef USE_INTERRUPTS
86          enable_interrupts(GLOBAL);
87      #endif
88
89      delay_ms(2);
90      EEPROM_24LC1025_WRITE_DISABLE();
91      }
```

El primer paso consiste en determinar si el acceso se va a producir en el banco 0 o el banco 1. En caso de que la dirección sea menor que los primeros 65535 bits, se encontrará en el banco 1. En caso de que sea mayor, el banco seleccionado deberá ser el 1 (líneas 58 y 59).

Posteriormente se habilita la escritura del dispositivo y se espera 1 ms con el fin de cumplir el tiempo de set-up. Volvamos a echar un vistazo al *datasheet*:

			250	—		2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
11	TSU:WP	WP setup time	4000	—	ns	1.8V ≤ Vcc ≤ 2.5V
			600	—		2.5V ≤ Vcc ≤ 5.5V
			600	—		2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
12	THD:WP	WP hold time	4700	—	ns	1.8V ≤ Vcc ≤ 2.5V

El tiempo mínimo de set-up de la señal de protección es de 600 ns. Con 1 ms superamos con creces dicho tiempo. A pesar de que puede reducirse bastante, no es conveniente apurar el *timing* de las señales, con 1 ms es suficiente para nuestro cometido y no supone una pérdida de tiempo considerable.

Acto seguido se deshabilitan las interrupciones en caso de estar habilitadas, como viene siendo habitual en las operaciones con el bus i2c, y después se inicia la transferencia . Primeramente se lanza el bit de START, acto seguido la dirección del dispositivo, la parte alta de la dirección, la parte baja, se escribe el dato de 8 bits y se envía al bus el bit de STOP (líneas 68 a 73).

En caso de que haya sido seleccionada la opción de llevar a cabo una espera activa con el fin de no retornar hasta que haya sido finalizado el proceso de escritura interno de la memoria, el algoritmo entra en un bucle, contenido entre las líneas 75 y 83. Dicho bucle comienza tratando de iniciar una nueva operación de escritura, escribiendo el bit de START y posteriormente la dirección del dispositivo en modo escritura. Si no devuelve el bit de reconocimiento ACK, sigue iterando hasta que éste sea devuelto, considerando que el dispositivo se encuentra listo para una nueva operación de escritura y que por tanto la operación anterior ha sido finalizada.

Finalmente, se deshabilita la escritura del dispositivo y se lleva a cabo una espera de 2 ms. Con el fin de cumplir con el *timing* de las especificaciones. Volviendo al *datasheet*:

			600	—		2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
12	THD:WP	WP hold time	4700	—	ns	1.8V ≤ Vcc ≤ 2.5V
			1300	—		2.5V ≤ Vcc ≤ 5.5V
			1300	—		2.5V ≤ Vcc ≤ 5.5V (24FC1025 only)
13	TAA	Output valid from clock	—	3500	ns	1.8V ≤ Vcc ≤ 2.5V

Del mismo modo y aunque un tiempo de *hold* de 1.3 ms es suficiente para garantizar el correcto funcionamiento, se le dará un margen adecuado y se pondrá una espera de 2 ms.

EEPROM_24LC1025_read()

```
102  byte EEPROM_24LC1025_read(int32 addr)
103      {
104      byte data;
105      byte command;
106
107      if(addr > 65535) command = EEPROM_24LC1025_ADDR_BANK1_W;
108      else command = EEPROM_24LC1025_ADDR_BANK0_W;
109
110      #ifdef USE_INTERRUPTS
111          disable_interrupts(GLOBAL);
112      #endif
113
114      i2c_start();
115      i2c_write(command);
116      i2c_write(addr >> 8);
117      i2c_write(addr);
118      i2c_start();
119      i2c_write(command+1);
120      data = i2c_read(0);
121      i2c_stop();
122
123      #ifdef USE_INTERRUPTS
124          enable_interrupts(GLOBAL);
125      #endif
126
127      return data;
128      }
```

Esta función realiza una operación de lectura aleatoria sobre la dirección indicada. Para ello y de igual modo que su homóloga de escritura, primeramente comprueba el valor de la dirección y establece un banco de trabajo u otro (líneas 107 y 108). Acto seguido, deshabilita las interrupciones y comienza la lectura de la dirección.

Para ello, primeramente inicia una operación de escritura sobre el dispositivo, envía la dirección fraccionada en dos partes, y posteriormente inicia una nueva transferencia sin cerrar la anterior (sin poner el bit de STOP) en la cual direcciona de nuevo el dispositivo en modo lectura (con el bit menos significativo a 1) para posteriormente leer un byte del bus sin enviar el bit de ACK, indicando al dispositivo que únicamente se va a leer una dirección. Tras ello, se cierra la conexión con el bit de STOP (líneas 114 a 121).

EEPROM_24LC1025_page_write()

```
143  int EEPROM_24LC1025_page_write(int32 addr, byte *buffer, byte wait)
144      {
145      byte command;
146      int status;
147      int i;
148
149      if(addr%EEPROM_24LC1025_PAGE_SIZE != 0) return 1;
150
151      if(addr > 65535) command = EEPROM_24LC1025_ADDR_BANK1_W;
152      else command = EEPROM_24LC1025_ADDR_BANK0_W;
153
154      EEPROM_24LC1025_WRITE_ENABLE();
155      delay_ms(1);
156
157      #ifdef USE_INTERRUPTS
158          disable_interrupts(GLOBAL);
159      #endif
160
161      i2c_start();
162      i2c_write(command);
163      i2c_write(addr>>8);
164      i2c_write(addr);
165      for(i=0; i<EEPROM_24LC1025_PAGE_SIZE; i++) i2c_write(*(buffer++));
166      i2c_stop();
167
168      if(wait == WRITE_WAIT)
169          {
170          do
171              {
172              i2c_start();
173              status = i2c_write(command);
174              } while(status == 1);
175
176          i2c_stop();
177          }
178
179      #ifdef USE_INTERRUPTS
180          enable_interrupts(GLOBAL);
181      #endif
182
183      delay_ms(2);
184      EEPROM_24LC1025_WRITE_DISABLE();
185
186      return 0;
187      }
```

Este algoritmo realiza una operación de escritura de página. Para ello primeramente comprueba si la dirección especificada es múltiplo de 128, en caso contrario devuelve “1” y finaliza (línea 149).

En caso contrario, del mismo modo que el resto de operaciones primeramente comprueba el banco de acceso (líneas 151 y 152), tras ello habilita la patilla de habilitación de escritura y comienza la transferencia.

Se escribe el bit de START, se pone en el bus la dirección del dispositivo direccionado como escritura, se pasa la dirección de inicio de página en dos mitades, y se llevan a cabo 128 operaciones de escritura en el bus tomando para ello los datos del buffer de manera consecutiva (línea 165).

Tras ello, y si el parámetro “wait” así lo indica, se lleva a cabo una espera activa hasta que el ciclo de escritura finalice tal y como se ha explicado anteriormente en la función de escritura aleatoria. Acto seguido, se habilitan de nuevo las interrupciones y finaliza la función.

EEPROM_24LC1025_page_read()

```
200  int EEPROM_24LC1025_page_read(int32 addr, byte *buffer)
201      {
202      byte command;
203      int i;
204
205      if(addr%EEPROM_24LC1025_PAGE_SIZE != 0) return 1;
206
207      if(addr > 65525) command = EEPROM_24LC1025_ADDR_BANK1_W;
208      else command = EEPROM_24LC1025_ADDR_BANK0_W;
209
210      #ifdef USE_INTERRUPTS
211          disable_interrupts(GLOBAL);
212      #endif
213
214      i2c_start();
215      i2c_write(command);
216      i2c_write(addr>>8);
217      i2c_write(addr);
218      i2c_start();
219      i2c_write(command+1);
220      for(i=0; i<EEPROM_24LC1025_PAGE_SIZE-1; i++) *(buffer++) = i2c_read();
221      *buffer = i2c_read(0);
222      i2c_stop();
223
224      #ifdef USE_INTERRUPTS
225          enable_interrupts(GLOBAL);
226      #endif
227
228      return 0;
229      }
```

Función complementaria a EEPROM_24LC1025_page_write(), lee una página completa de 128 bits de la dirección especificada en memoria y la almacena en la dirección del buffer especificado como parámetro.

Para ello primeramente comprueba si la dirección de página es válida, esto es, múltiplo de 128. En caso contrario devuelve “1” y el algoritmo finaliza. En caso de tratarse de una dirección válida se inicia la operación.

Para ello primeramente, y al igual que en el resto de operaciones, se comprueba a que banco hace referencia dicha dirección (líneas 207 y 208). Acto seguido se deshabilitan

las interrupciones y se comienza una operación de escritura para especificar la dirección sobre la que se leerá. Para ello se envía el bit de START, la dirección del dispositivo en modo escritura, y tras ello la dirección de la página fraccionada en dos mitades como ya se ha visto anteriormente.

Después se inicia una nueva transferencia sin cerrar la anterior, direccionando ahora el dispositivo en modo lectura colocando para ello el bit menos significativo a 1 (línea 219) y se entra en un bucle "for" que lleva a cabo 128 lecturas consecutivas (línea 220). La última lectura se efectúa fuera del bucle ya que ésta deberá enviar el bit de no reconocimiento NAK, esto se consigue pasando como parámetro el valor "0" a la función "i2c_read()".

Una vez hecho esto, se escribe el bit de STOP y se da por concluida la operación de escritura de página.

4.1.2.- Timer RTC DS1307.

El driver está compuesto, siguiendo con la metodología de desarrollo del lenguaje C, de un archivo cabecera .h que contiene las definiciones y prototipos de las funciones y el archivo del driver propiamente dicho de extensión .c, que implementa la API previamente definida en el archivo de cabecera.

El archivo DS1307.h

Se procederá a analizar primeramente el archivo DS1307.h por partes:

```
26  #ifndef DS1307_H
27  #define DS1307_H
28
29  #ifndef DS1307_PINOUT
30      #error DS1307: Pinout no definido
31  #endif
32
```

Primeramente se hace uso del juego de directivas de preprocesador del compilador #ifndef-#endif para llevar a cabo el control de inclusión de la librería. Esto permite evitar errores cuando se incluye una librería mas de una vez en un mismo proyecto, debido a inclusiones simultaneas desde diversos archivos fuente diferentes lo que daría lugar a errores debido a duplicidad de definiciones.

Cuando se hace uso de este driver en un proyecto es necesario haber definido previamente el *pinout* del dispositivo, el control de ello se lleva a cabo obligando al desarrollador a definir la macro "DS1307_pinout" cuando define el *pinout* del dispositivo. En caso de que no esté definida, se da por hecho que no se ha definido el *pinout* del dispositivo y el compilador muestra el error "DS1307: *Pinout* no definido.".

```
34      // API RTC DS1307
35      void DS1307_init(byte config);
36      void DS1307_set_date_time(byte year, byte mth, byte day, byte weekday, byte hour, byte min, byte sec);
37      void DS1307_get_date_time(byte *year, byte *mth, byte *day, byte *weekday, byte *hour, byte *min, byte *sec);
38      void DS1307_set_date(byte year, byte mth, byte day, byte weekday);
39      void DS1307_get_date(byte *year, byte *mth, byte *day, byte *weekday);
40      void DS1307_set_time(byte hour, byte min, byte sec);
41      void DS1307_get_time(byte *hour, byte *min, byte *sec);
42      void DS1307_get_day_of_week(char *str);
43      void DS1307_out_osc_conf(byte conf);
44      void DS1307_pause();
45      void DS1307_resume();
46      void DS1307_out_osc_on();
47      void DS1307_out_osc_off();
```

El siguiente bloque de sentencias define la API del dispositivo, que serán implementadas en el archivo de código de extensión .c, se incluyen las siguientes funciones:

DS1307_init(): Función que debe ser llamada primeramente con el fin de inicializar los registros internos y llevar al dispositivo a un estado conocido. Cualquier función que se llame de la *API* debe situarse posteriormente a esta función. Opera sobre el registro de configuración.

DS1307_set_date_time(): establece la fecha y la hora del dispositivo.

DS1307_get_date_time(): obtiene la fecha y la hora del dispositivo.

DS1307_set_date(): establece solo la fecha del dispositivo.

DS1307_get_date(): obtiene solo la fecha del dispositivo.

DS1307_set_time(): establece solo la hora del dispositivo.

DS1307_get_time(): obtiene solo la hora del dispositivo.

DS1307_get_day_of_week(): obtiene un string con el nombre del dia de la semana.

DS1307_out_osc_conf(): cambia la configuración de la patilla 7 de salida de onda cuadrada. Permite reconfigurar el dispositivo operando sobre el registro de configuración.

DS1307_out_pause(): pausa el *timer*. Inhabilita el paso del tiempo.

DS1307_out_resume(): reanuda la ejecución del *timer*.

DS1307_out_out_osc_on(): habilita la salida de onda cuadrada por la patilla 7, según la configuración previamente establecida mediante DS1307_out_osc_conf() o DS1307_init().

DS1307_out_osc_off(): deshabilita la salida de onda cuadrada por la patilla 7.

Asimismo, también se definen dos funciones internas que no forman parte de la API del chip.

```
49      // Funciones internas
50      byte DS1307_bcd2bin(byte bcd);
51      byte DS1307_bin2bcd(byte bin);
```

DS1307_bcd2bin(): Convierte un valor BCD en su representación en binario natural.

DS1307_bcd2bin(): Convierte un valor en binario natural en su representación en BCD.

A continuación se lleva a cabo la definición del mapeado de registros internos del dispositivo, con el fin de asignar un alias a cada dirección de registro y facilitar la legibilidad y mantenimiento del código.

```
54      // Definición de los registros de memoria internos.
55      #define DS1307_ADDR_R               0xD1
56      #define DS1307_ADDR_W               0xD0
57      #define DS1307_SEC_ADDR             0x00
58      #define DS1307_MIN_ADDR             0x01
59      #define DS1307_HOUR_ADDR            0x02
60      #define DS1307_WEEKDAY_ADDR         0x03
61      #define DS1307_DAY_ADDR             0x04
62      #define DS1307_MONTH_ADDR           0x05
63      #define DS1307_YEAR_ADDR            0x06
64      #define DS1307_CONF_ADDR            0x07
```

El siguiente bloque de código del archivo define los parámetros de configuración que serán utilizados durante la llamada a las funciones DS1307_init() y DS1307_osc_conf(). Pueden ser empleados conjuntamente mediante el operador “or”, representado en C por el símbolo “|”.

```
66      // Parámetros de configuración de la onda cuadrada de salida durante
67      // La función "init()". Onda accesible desde la patilla 7 del chip.
68      #define DS1307_OUT_ON               0x10
69      #define DS1307_OUT_OFF              0x00
70      #define DS1307_OUT_HIGH_WHEN_OFF    0x80
71      #define DS1307_OUT_LOW_WHEN_OFF     0x00
72      #define DS1307_OUT_13768_KHZ        0x03
73      #define DS1307_OUT_8192_KHZ         0x02
74      #define DS1307_OUT_4096_KHZ         0x01
75      #define DS1307_OUT_1_HZ             0x00
76
```

Finalmente, se define un *array* de *strings* conteniendo los días de la semana para hacer posible la implementación de la función DS1307_get_day_of_week(). Tras ello, se incluye el archivo de código y se cierra el bloque condicional del preprocesador mediante la sentencia "#endif".

```
78      char days_of_week[7][11] =
79          {
80          "Lunes\0",
81          "Martes\0",
82          "Miércoles\0",
83          "Jueves\0",
84          "Viernes\0",
85          "Sábado\0",
86          "Domingo\0"
87          };
88
89
90      #include "DS1307.c"
91
92      #endif
```

El archivo DS1307.c

El archivo DS1307.c contiene la implementación de las funciones del driver definidas en el archivo de cabecera.

Función DS1307_init():

```
38      void DS1307_init(byte config)
39          {
40          #ifdef USE_INTERRUPTS
41              disable_interrupts(GLOBAL);
42          #endif
43
44          i2c_start();
45          i2c_write(DS1307_ADDR_W);
46          i2c_write(DS1307_CONF_ADDR);
47          i2c_write(config);
48          i2c_stop();
49
50          #ifdef USE_INTERRUPTS
51              enable_interrupts(GLOBAL);
52          #endif
53          }
```

Inicializa el dispositivo escribiendo el registro de configuración con los valores pasados como parámetro. Se utiliza en conjunto con las definiciones comprendidas entre las líneas 66 y 76 del archivo de cabecera, por ejemplo DS1307_init(DS1307_INIT_ON | DS1307_INIT_1_HZ) inicializa el *timer* activando la salida de la onda cuadrada por la patilla 7 a una frecuencia establecida a 1 Hz. Realiza una operación de escritura, para ello sigue la secuencia detallada anteriormente sobre el bus i2c.

Primeramente se escribe el bit de START, para continuar direccionando el dispositivo en modo escritura y tras ello se indica la dirección a modificar. Después se vuelca en el bus el valor del dato, que será tomado por el dispositivo para actualizar el registro indicado, y acto seguido el microprocesador pone fin a la transferencia escribiendo en el bus el bit de STOP.

El bloque #ifdef-#endif agrega al programa las instrucciones comprendidas en él siempre y cuando se cumpla la condición de que la macro USE_INTERRUPTS esté definida. Ello se realiza porque las transferencias i2c no son prorrogables ni interrumpibles, es necesario deshabilitar las interrupciones de forma GLOBAL durante cada transferencia del bus para evitar que una interrupción detenga el flujo normal del programa y corrompa la transferencia. Una vez finalizado el intercambio de información i2c, se rehabilitan de nuevo las interrupciones.

Función DS1307_set_date_time()

```
62  void DS1307_set_date_time(byte year, byte mth, byte day, byte weekday, byte hour, byte min, byte sec)
63      {
64      byte day_bcd;
65      byte mth_bcd;
66      byte year_bcd;
67      byte hour_bcd;
68      byte min_bcd;
69      byte sec_bcd;
70
71      day_bcd = DS1307_bin2bcd(day);
72      mth_bcd = DS1307_bin2bcd(mth);
73      year_bcd = DS1307_bin2bcd(year);
74      hour_bcd = DS1307_bin2bcd(hour);
75      min_bcd = DS1307_bin2bcd(min);
76      sec_bcd = DS1307_bin2bcd(sec);
77
78      #ifdef USE_INTERRUPTS
79          disable_interrupts(GLOBAL);
80      #endif
81
82      i2c_start();
83      i2c_write(DS1307_ADDR_W);
84      i2c_write(DS1307_SEC_ADDR);
85      i2c_write(sec_bcd);
86      i2c_write(min_bcd);
87      i2c_write(hour_bcd);
88      i2c_write(weekday);
89      i2c_write(day_bcd);
90      i2c_write(mth_bcd);
91      i2c_write(year_bcd);
92      i2c_stop();
93
94      #ifdef USE_INTERRUPTS
95          enable_interrupts(GLOBAL);
96      #endif
97      }
```

Lleva a cabo la operación de escritura sobre la totalidad de registros que componen la fecha y la hora. Primeramente se convierten los valores pasados como parámetros, los cuales están en formato binario natural, a BCD que es el formato bajo el que opera el *timer*. Para ello se hace uso de la función interna "DS1307_bin2bcd()" (líneas 71 a 76). Una vez convertidos los valores se inicia la transferencia a través del bus. Se coloca el bit de START, se direcciona el dispositivo en modo escritura, se coloca la dirección del registro de los segundos y acto seguido se escribe en el bus cada uno de los valores consecutivamente, ello es posible a que como ya se ha indicado previamente, sucesivas operaciones de escritura operan sobre sucesivos registros internos debido a la propiedad de autoincremento del registro apuntado tras cada lectura o escritura. Ello facilita enormemente la programación del dispositivo y agiliza el uso del bus. A la hora de llevar a cabo escrituras múltiples como en este caso hay que tener muy en cuenta el mapeado de los registros internos, que ha de corresponder con el orden de escritura de los valores en orden ascendente.

Una vez llevadas a cabo las escrituras pertinentes se finaliza la transferencia mediante la escritura por parte del PIC del bit de STOP en el bus. Si se ha definido la variable USE_INTERRUPTS, se deshabilitarán las interrupciones durante el acceso al bus i2c.

Función DS1307_get_date_time()

```
106  void DS1307_get_date_time(byte *year, byte *mth, byte *day, byte *weekday, byte *hour, byte *min, byte *sec)
107      {
108      byte day_bcd;
109      byte mth_bcd;
110      byte year_bcd;
111      byte hour_bcd;
112      byte min_bcd;
113      byte sec_bcd;
114
115      #ifdef USE_INTERRUPTS
116          disable_interrupts(GLOBAL);
117      #endif
118
119      i2c_start();
120      i2c_write(DS1307_ADDR_W);
121      i2c_write(DS1307_SEC_ADDR);
122      i2c_start();
123      i2c_write(DS1307_ADDR_R);
124      sec_bcd = i2c_read();
125      min_bcd = i2c_read();
126      hour_bcd = i2c_read();
127      *weekday = i2c_read();
128      day_bcd = i2c_read();
129      mth_bcd = i2c_read();
130      year_bcd = i2c_read(0);
131      i2c_stop();
132
133      #ifdef USE_INTERRUPTS
134          enable_interrupts(GLOBAL);
135      #endif
136
137      *year = DS1307_bcd2bin(year_bcd);
138      *mth = DS1307_bcd2bin(mth_bcd);
139      *day = DS1307_bcd2bin(day_bcd);
140      *hour = DS1307_bcd2bin(hour_bcd & 0x3F);
141      *min = DS1307_bcd2bin(min_bcd & 0x7F);
142      *sec  = DS1307_bcd2bin(sec_bcd & 0x7F);
143      }
```

Función complementaria a la función anterior, obtiene la fecha y la hora actuales del dispositivo. Obsérvese la implementación de la operación de lectura, como se explicó anteriormente van precedidas de una operación de escritura en la cual se escribe el registro interno apuntador de dirección, con el fin de que las lecturas llevadas a cabo a continuación operen sobre los registros adecuados.

Primeramente se inicia la transferencia mediante el bit de START, acto seguido se direcciona el dispositivo en modo escritura, se escribe la dirección que se quiere leer y se inicia una nueva transferencia sin cerrar la anterior (sin enviar el bit de STOP). Esta vez se direcciona el dispositivo en modo lectura y se procede a realizar sucesivas llamadas a la función i2c_read(), que leen un byte del bus en cada invocación. Una vez más se hace uso de la propiedad autoincremental del apuntador de registro interno del dispositivo. Nótese que tras la última lectura se invoca a i2c_read() con el parámetro "0", esto indica que no debe llevarse a acabo escritura del bit de reconocimiento ACK, sino del bit de no-reconocimiento NACK, que indica al dispositivo que se trata de la última lectura. Tras ello, el PIC cierra la conexión con la escritura del bit de STOP, como es habitual.

Una vez finalizada la lectura de los datos y debido a que los valores se encuentran guardados en formato BCD, se realiza la conversión de formato BCD a binario natural para posteriormente ser devueltos a las direcciones especificadas como punteros durante la llamada a la función (líneas 137 a 142).

Antes de cada conversión se lleva a cabo el aislamiento de los bits de datos de cada valor leído a través de la función AND ("&") con el fin de no tener en cuenta los bits de configuración que se sitúan en algunos bits específicos junto con el valor propio de los registros registro. Por ejemplo, en el registro de los segundos no se tendrá en cuenta el bit más significativo ya que no forma parte del dato propiamente dicho sino que se trata de un bit de configuración que habilita o deshabilita el dispositivo, para ello se realizará la operación "& 0x7F" con el fin de ponerlo a cero previamente antes de la conversión a binario natural, si ello no se llevara a cabo daría lugar a resultados erróneos ya que se tomarían este bit de configuración como un bit más del registro de los segundos.

Función DS1307_get_date()

```
152  void DS1307_set_date(byte year, byte mth, byte day, byte weekday)
153      {
154      byte year_bcd;
155      byte mth_bcd;
156      byte day_bcd;
157
158      year_bcd = DS1307_bin2bcd(year);
159      mth_bcd = DS1307_bin2bcd(mth);
160      day_bcd = DS1307_bin2bcd(day);
161
162      #ifdef USE_INTERRUPTS
163          disable_interrupts(GLOBAL);
164      #endif
165
166      i2c_start();
167      i2c_write(DS1307_ADDR_W);
168      i2c_write(DS1307_WEEKDAY_ADDR);
169      i2c_write(weekday);
170      i2c_write(day_bcd);
171      i2c_write(mth_bcd);
172      i2c_write(year_bcd);
173      i2c_stop();
174
175      #ifdef USE_INTERRUPTS
176          enable_interrupts(GLOBAL);
177      #endif
178      }
```

De manera análoga a la función DS1307_get_date_time(), realiza la lectura de los registros pertinentes para obtener la fecha y la hora del dispositivo. Implementa un subconjunto de las funciones que implementa DS1307_get_date_time(), por lo tanto puede obviarse su explicación y evitar de este modo caer en la redundancia.

Función DS1307_get_date()

```
187  void DS1307_get_date(byte *year, byte *mth, byte *day, byte *weekday)
188      {
189      byte year_bcd;
190      byte mth_bcd;
191      byte day_bcd;
192
193      #ifdef USE_INTERRUPTS
194          disable_interrupts(GLOBAL);
195      #endif
196
197      i2c_start();
198      i2c_write(DS1307_ADDR_W);
199      i2c_write(DS1307_WEEKDAY_ADDR);
200      i2c_start();
201      i2c_write(DS1307_ADDR_R);
202      *weekday = i2c_read();
203      day_bcd = i2c_read();
204      mth_bcd = i2c_read();
205      year_bcd = i2c_read(0);
206      i2c_stop();
207
208      #ifdef USE_INTERRUPTS
209          enable_interrupts(GLOBAL);
210      #endif
211
212      *year = DS1307_bcd2bin(year_bcd);
213      *mth = DS1307_bcd2bin(mth_bcd & 0x1F);
214      *day = DS1307_bcd2bin(day_bcd & 0x3F);
215      }
```

Función complementaria a DS1307_get_date(), obtiene la fecha del sistema. Del mismo modo que en el caso anterior, se hace innecesaria su explicación puesto que se trata de un subconjunto de las funciones de DS1307_get_date_time().

Función DS1307_set_time()

```
224  void DS1307_set_time(byte hour, byte min, byte sec)
225      {
226      byte hour_bcd;
227      byte min_bcd;
228      byte sec_bcd;
229
230      hour_bcd = DS1307_bin2bcd(hour);
231      min_bcd = DS1307_bin2bcd(min);
232      sec_bcd = DS1307_bin2bcd(sec);
233
234      #ifdef USE_INTERRUPTS
235          disable_interrupts(GLOBAL);
236      #endif
237
238      i2c_start();
239      i2c_write(DS1307_ADDR_W);
240      i2c_write(DS1307_SEC_ADDR);
241      i2c_write(sec_bcd);
242      i2c_write(min_bcd);
243      i2c_write(hour_bcd);
244      i2c_stop();
245
246      #ifdef USE_INTERRUPTS
247          enable_interrupts(GLOBAL);
248      #endif
```

Establece la hora del sistema. Implementa un subconjunto de las funciones implementadas por DS1307_set_date_time().

```
258  void DS1307_get_time(byte *hour, byte *min, byte *sec)
259      {
260      byte hour_bcd;
261      byte min_bcd;
262      byte sec_bcd;
263
264      #ifdef USE_INTERRUPTS
265          disable_interrupts(GLOBAL);
266      #endif
267
268      i2c_start();
269      i2c_write(DS1307_ADDR_W);
270      i2c_write(DS1307_SEC_ADDR);
271      i2c_start(),
272      i2c_write(DS1307_ADDR_R);
273      sec_bcd = i2c_read();
274      min_bcd = i2c_read();
275      hour_bcd = i2c_read(0);
276      i2c_stop();
277
278      #ifdef USE_INTERRUPTS
279          enable_interrupts(GLOBAL);
280      #endif
281
282      *hour = DS1307_bcd2bin(hour_bcd & 0x3F);
283      *min = DS1307_bcd2bin(min_bcd & 0x7F);
284      *sec = DS1307_bcd2bin(sec_bcd & 0x7F);
285      }
```

Obtiene la hora del sistema, ignorando el resto de información. Implementa un subconjunto del código perteneciente a la función DS1307_get_date_time().

Función DS1307_get_day_of_week()

```
295  void DS1307_get_day_of_week(char *str)
296      {
297      byte day;
298      byte mth;
299      byte year;
300      byte weekday;
301
302      DS1307_get_date(&year, &mth, &day, &weekday);
303      sprintf(str, "%s", days_of_week[weekday]);
304      }
```

Coloca en la variable cadena de caracteres pasa como parámetro el nombre del día de la semana. Para ello basta con leer el registro 0x04, el cual contiene un entero comprendido entre 0 y 6 que representa el día de la semana. La conversión entre entero-string se lleva a cabo haciendo uso de la estructura “days_of_week[]” definida en el archivo DS1307.h.

Función DS1307_out_osc_conf()

```
313  void DS1307_out_osc_conf(byte conf)
314      {
315
316      #ifdef USE_INTERRUPTS
317          disable_interrupts(GLOBAL);
318      #endif
319
320      i2c_start();
321      i2c_write(DS1307_ADDR_W);
322      i2c_write(DS1307_CONF_ADDR);
323      i2c_write(conf);
324      i2c_stop();
325
326      #ifdef USE_INTERRUPTS
327          enable_interrupts(GLOBAL);
328      #endif
329      }
```

Opera sobre el registro de configuración, de manera análoga al funcionamiento de la función DS1307_init() por lo que no se comentará su funcionamiento.

DS1307_pause()

```
338  void DS1307_pause()
339      {
340      byte secs;
341
342      #ifdef USE_INTERRUPTS
343          disable_interrupts(GLOBAL);
344      #endif
345
346      i2c_start();
347      i2c_write(DS1307_ADDR_W);
348      i2c_write(DS1307_SEC_ADDR);
349      i2c_start();
350      i2c_write(DS1307_ADDR_R);
351      secs = i2c_read(0);
352      i2c_start();
353      i2c_write(DS1307_ADDR_W);
354      i2c_write(DS1307_SEC_ADDR);
355      i2c_write(secs | 0x80);
356      i2c_stop();
357
358      #ifdef USE_INTERRUPTS
359          enable_interrupts(GLOBAL);
360      #endif
361      }
```

La función D1307_pause() detiene el funcionamiento del *timer*. Tras la llamada a esta función el paso del tiempo se deshabilitará y el *timer* quedará pausado hasta la llamada de la función de reanudación DS1307_resume().

Con el fin de llevar a cabo el pausado del dispositivo, la función modifica el bit de mayor peso del registro del segundero. Si se pone dicho bit a 0, se habilita el funcionamiento del *timer*. Si se pone a 1, se deshabilita. Hay que tener en cuenta que los demás bits del registro deben permanecer inalterados, para ello se deberá leer previamente su valor (líneas 347 a 351) para posteriormente modificar el bit sobre el valor leído mediante una operación OR (“|”) y la máscara 0x80, con lo cual se consigue poner a uno dicho bit manteniendo inalterados el resto de ellos. Una vez puesto a uno el bit de mayor peso se procederá a actualizar el registro de los segundos con dicho valor modificado, siguiendo para ello el procedimiento habitual.

DS1307_resume()

```
370  void DS1307_resume()
371      {
372      byte secs;
373
374      #ifdef USE_INTERRUPTS
375          disable_interrupts(GLOBAL);
376      #endif
377
378      i2c_start();
379      i2c_write(DS1307_ADDR_W);
380      i2c_write(DS1307_SEC_ADDR);
381      i2c_start();
382      i2c_write(DS1307_ADDR_R);
383      secs = i2c_read(0);
384      i2c_start();
385      i2c_write(DS1307_ADDR_W);
386      i2c_write(DS1307_SEC_ADDR);
387      i2c_write(secs & 0x7F);
388      i2c_stop();
389
390      #ifdef USE_INTERRUPTS
391          enable_interrupts(GLOBAL);
392      #endif
393      }
```

Función complementaria a la función DS1307_pause(), reanuda el paso del tiempo del *timer*. Para ello, siguiendo el proceso análogo a la función pause(), obtiene el valor del registro de los segundos (líneas 378 a 383), pone a cero el bit de mayor peso mediante la operación AND (“&”) y la máscara 0x7F, y actualiza el valor del mismo (línea 387).

Función DS1307_out_osc_off()

```
435  void DS1307_out_osc_off()
436      {
437      byte conf;
438
439      #ifdef USE_INTERRUPTS
440          disable_interrupts(GLOBAL);
441      #endif
442
443      i2c_start();
444      i2c_write(DS1307_ADDR_W);
445      i2c_write(DS1307_CONF_ADDR);
446      i2c_start();
447      i2c_write(DS1307_ADDR_R);
448      conf = i2c_read(0);
449      i2c_start();
450      i2c_write(DS1307_ADDR_W);
451      i2c_write(DS1307_CONF_ADDR);
452      i2c_write(conf & 0xEF);
453      i2c_stop();
454
455      #ifdef USE_INTERRUPTS
456          enable_interrupts(GLOBAL);
457      #endif
```

Deshabilita la salida de onda cuadrada de la patilla 7 del chip. Como se vio anteriormente, ello se lleva a cabo mediante la escritura del bit 4 del registro de control.

Registro de control (0x07)

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OUT	0	0	SQWE	0	0	RS1	RS0

SQWE = 1 → Salida habilitada.
SQWE = 0 → Salida deshabilitada.

A la hora de modificar un bit del registro hay que tener en cuenta que no es posible la modificación individual a nivel de bit, la modificación de un registro supone la escritura de nuevo del registro completo de 8 bits. Para conservar el valor de los demás bits durante la actualización, se lleva previamente a cabo una operación de lectura sobre el registro y se guarda su valor (lineas 443 a 448). Acto seguido, y sobre el valor leido, se pone a 0 el bit correspondiente mediante la operación AND (&) junto con una máscara de bits, que resetea el bit 4 permaneciendo inalterados el resto de los bits del registro, para posteriormente escribir de nuevo el valor del registro de configuración (linea 452).

Función DS1307_out_osc_on()

```
402  void DS1307_out_osc_on()
403      {
404      byte conf;
405
406      #ifdef USE_INTERRUPTS
407          disable_interrupts(GLOBAL);
408      #endif
409
410      i2c_start();
411      i2c_write(DS1307_ADDR_W);
412      i2c_write(DS1307_CONF_ADDR);
413      i2c_start();
414      i2c_write(DS1307_ADDR_R);
415      conf = i2c_read(0);
416      i2c_start();
417      i2c_write(DS1307_ADDR_W);
418      i2c_write(DS1307_CONF_ADDR);
419      i2c_write(conf | 0x10);
420      i2c_stop();
421
422      #ifdef USE_INTERRUPTS
423          enable_interrupts(GLOBAL);
424      #endif
425      }
```

Opera de la misma manera que la función DS1307_out_osc_off(), solo que en lugar de poner a_0 el bit SQWE lo pone a 1 con el fin de hacer accesible la onda cuadrada a través d la patilla 7 del dispositivo.

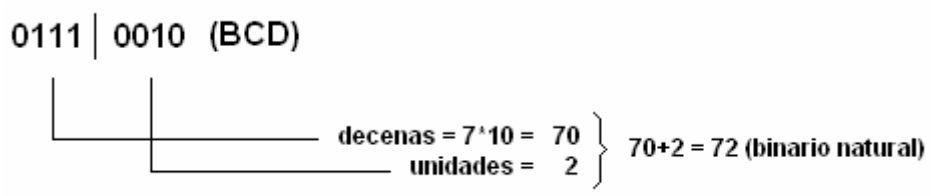
Para ello primeramente obtiene el valor del registro de configuración (lineas 410 a 415), posteriormente pone a 1 el bit 4 del mismo mediante la operación lógica a nivel de bit OR ("|"), para finalmente iniciar una operación de escritura y actualizar el valor del registro el el timer (lineas 417 a 419). Una vez finalizado el proceso, finaliza la transferencia de la forma habitual escribiendo en el bus el bit de STOP.

Función DS1307_bcd2bin()

```
466
467     byte DS1307_bcd2bin(byte bcd)
468         {
469         byte bin;
470         byte dec;
471         byte uni;
472
473         dec = (bcd>>4)*10;
474         uni = bcd & 0x0F;
475         bin = dec+uni;
476         return bin;
477         }
```

Realiza la conversión del número en formato BCD especificado como parámetro en su correspondiente valor en representación en binario natural.

En un número representado en BCD, el nibble (conjunto de 4 bits) superior del byte representa el valor de las decenas y el nibble inferior el valor de las unidades. Teniendo esto en cuenta, el algoritmo se basa en el aislamiento de ambos nibbles para su tratamiento por separado.



Primeramente se obtiene el valor de las decenas. Ello se consigue desplazando el valor 4 bits hacia le derecha, debido a que la instrucción de desplazamiento rellena por la izquierda con ceros. Acto seguido se multiplica por 10 para obtener el valor de dicho número (linea 473).

Se procede de forma similar con las unidades, para ello se realiza la operación AND con la máscara 0x0F de forma que se pone a cero el nibble de mayor peso. El valor obtenido es el valor de las unidades (linea 474). Finalmente, se suman ambos valores y se obtiene el valor esperado en binario natural (linea 475).

Función DS1307_bin2bcd()

```
486     byte DS1307_bin2bcd(byte bin)
487         {
488         byte bcd;
489         byte nibble_H;
490         byte nibble_L;
491
492         nibble_H = bin/10;
493         nibble_L = bin - nibble_H*10;
494         bcd = nibble_H<<4 | nibble_L;
495         return bcd;
496         }
```

Esta opera de manera inversa que la función presentada anteriormente, realizando la conversión de un valor en binario natural pasado como parámetro a su representación en BCD, con el fin de ser utilizado durante las funciones de escritura del timer.

Para ello primeramente obtiene el valor de las decenas, haciendo uso de la divisón entera y despreciando de este modo el resto de la operación (linea 497). Acto seguido, se obtiene el valor de las unidades. Para ello, al valor inicial se le resta el valor ya obtenido de las decenas (previa multipliación por 10) y el resultado es el valor esperado (linea 493). Otra forma de hacerlo sería realizando la operación módulo (“%”) la cual obtiene el resto de una división: realizando la operación módulo de la forma “bin % 10” obtenemos el valor de las unidades del número en representación binaria.

4.2.- Pruebas de componentes.

Una vez implementados los dirvers de los componentes necesarios se han llevado a cabo ciertas pruebas sobre ellos con el fin de verificar su correcto funcionamiento y la correcta implementación del código de dichos drivers. Del mismo modo el hecho de llevar a cabo inicialmente pruebas sencillas sobre cada uno de los componentes permite familiarizarse con su uso y aprender el manejo concreto y los pormenores de cada uno de ellos. Se pasará posteriormente, por tanto, a explicar las pruebas realizadas sobre cada uno de los componentes que así se ha considerado conveniente llevar a cabo.

4.2.1.- USART

En el presente proyecto es de vital importancia el correcto funcionamiento de la USART de ambos microcontroladores, el PIC16F628 y el PIC18F4550, del emisor y receptor respectivamente, debido a que a través de su conexión directa con los módulos de radiofrecuencia de emisor y receptor, suponen la apertura al canal de comunicación inalámbrica.

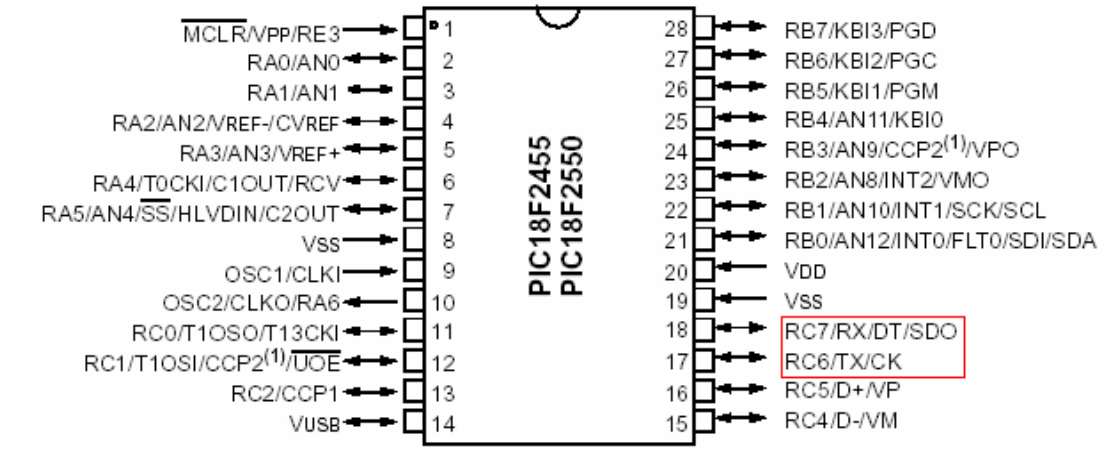
Con el fin de llevar a cabo un primer contacto con la USART de ambos *pics* se ha implementado un sencillo programa cuya única función es la de enviar el mismo carácter que recibe, dicho de otro modo, implementa la función "echo". Hay que tener en cuenta que los PIC poseen lógica TTL y funcionan a niveles de tensión comprendidos entre 0 y 5 voltios, sin embargo los niveles de tensión de funcionamiento del PC operan en el rango entre -12 y 12 voltios. Para llevar a cabo la conversión de niveles de tensión y permitir la comunicación bidireccional entre el PIC y el ordenador, se hará uso del conversor de niveles desarrollado.

El código de esta prueba no supone una alta complejidad y el código se encuentra autodocumentado, sin embargo se procederá a hacer una explicación del mismo:

```
21
22      #include <18F2550.h>
23
24      #fuses HS,NOWDT,NOPROTECT,NOLVP,BROWNOUT,NOMCLR
25      #use delay(clock=20000000)
26
27      // Pinout
28      #define RS232_TX    PIN_C6
29      #define RS232_RX    PIN_C7
30
31
32
33      // Apertura de la USART
34
35      #use rs232(baud=2400,xmit=RS232_TX,rcv=RS232_RX,bits=8,parity=N)
36
37
38      #int_rda
39
40      // Interrupción de recepción de carácter
41
42      void rs232_int()
43          {
44          char input;
45          input = getc();
46          printf("Echo: %c\r\n", input);
47          }
48
49
50      void main()
51          {
52          enable_interrupts(INT_RDA);
53          enable_interrupts(GLOBAL);
54          while(true);
55          }
56
```

Las líneas 28 y 29 se definen los pines utilizados como entrada y salida rs232. La creación de variables definición para llevar a cabo la configuración el *pinout* de los componentes supone una buena práctica de programación que mejora la legibilidad y mantenimiento futuro del código.

Acto seguido se procede a inicializar la USART del PIC mediante la sentencia "use", propia del compilador CCS. Esta sentencia inicializa la USART del microcontrolador en base a las opciones pasadas como parámetros, en este caso concreto se crea una conexión serie a una velocidad de 2400 baudios, usando como patilla de transmisión la patilla correspondiente al bit 6 del puerto C (patilla 17) y la correspondiente al bit 7 del mismo puerto como recepción (patilla 18) ya que son las que se corresponden con la USART. La unidad de envío de datos estará compuesta por 8 bits (1 byte) y no se utilizará control de errores mediante bit de paridad (opción parity=N).



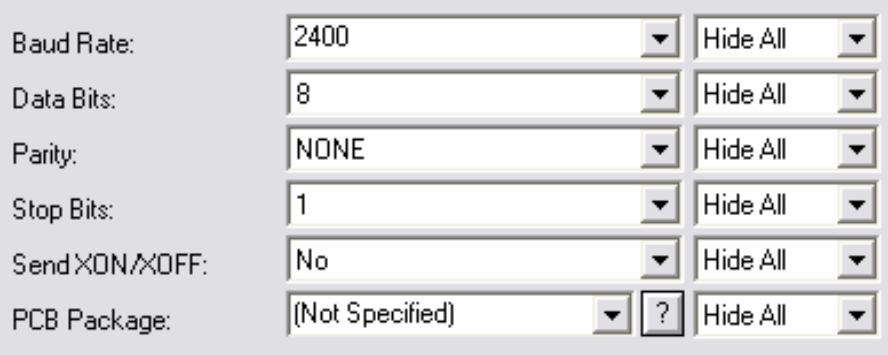
Es importante mencionar que el compilador CCS inicializará la USART hardware siempre y cuando la definición de pines de envío y recepción indicados se corresponda con los establecidos como entrada y salida de la misma, en caso contrario se creará una conexión RS232 por software y no se utilizará la USART del dispositivo. También es importante dejar constancia de que la definición del puerto serie deberá llevarse a cabo siempre posteriormente de la especificación de la frecuencia del microcontrolador (línea 25) ya que internamente el compilador CCS necesita conocer la base de tiempos para establecer la comunicación y cumplir con el *timing* de la señal de la transmisión serie.

La sentencia "#int_rda" indica que la función descrita a continuación hace referencia al manejador de interrupción de la USART. Dicha función debe tener tipo de retorno nulo (*void*) y ser lo mas escueta posible, denominador común que debe cumplir toda aquella función software encargada de llevar a cabo el control de interrupciones de cualquier índole. Cada vez que se encuentre disponible un carácter en el buffer de lectura de la USART, se producirá una interrupción que delegará el control en la función indicada tras esta sentencia.

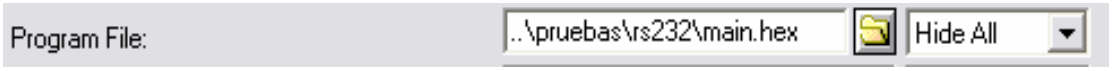
Acto seguido se pasa a la implementación del manejador de interrupción propiamente dicho en la línea 42, el cual únicamente lee un carácter a través de la función “getc()” y lo devuelve por el mismo canal de comunicación mediante la instrucción “printf()” previamente formateado con la inclusión de retorno de línea para mejorar la presentación de la aplicación.

Seguidamente se procede a la implementación del “main” o función principal del programa, la cual únicamente activa las interrupciones del dispositivo con el fin de habilitar el manejador asociado previamente.

Una vez escrito el código se procederá a su verificación. Para ello primeramente se simulará en *Proteus*, como circuito electrónico se utilizará el mismo del receptor. Haciendo doble *click* sobre el símbolo del terminal virtual se accede a su configuración, en la cual se especificarán las siguientes opciones:



Haciendo doble *click* en el símbolo del microcontrolador se abrirá la ventana de propiedades del mismo mediante la cual se podrá seleccionar el firmware a cargar en el PIC, directamente de la carpeta del proyecto de *MPLab* previamente compilado.



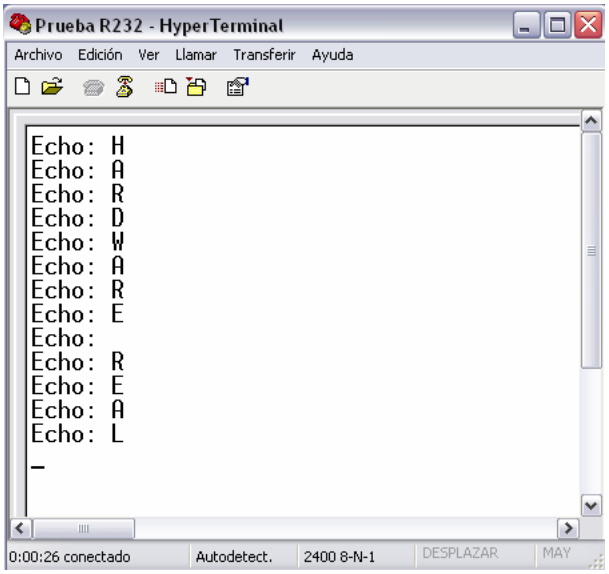
Una vez cargado el archivo rs232.hex se simulará el proyecto y se comprobará que a cada carácter pulsado en el terminal virtual, el microcontrolador devuelve el mismo carácter pulsado.

Tras llevar a cabo esta comprobación básica se procederá a llevar a cabo la comprobación sobre el hardware real. Para ello se abrirá *MPLab* y se seleccionará el programador IC2 en el menú "Programmer -> Select programmer". Se conectará la alimentación del mismo y se conectará a la placa del circuito electrónico a través de la opción "Programmer -> Connect". Acto seguido se programará el PIC con el firmware previamente compilado. Una vez hecho esto se procederá a alimentar el circuito electrónico se conectará la salida de la USART del microcontrolador al conversor de niveles de tensión mediante los pines habilitados al efecto en el circuito, y el conversor de niveles se conectará al ordenador directamente. Después de ello se definirá una nueva conexión serie mediante el *Hyperterminal* de Windows que deberá responder a la siguiente configuración:

Baudios: 2400
Bits de datos: 8
Paridad: N
Bits de parada: 1
Control de flujo: Ninguno

Se ha guardado la conexión de *Hyperterminal* bajo el nombre "Prueba RS232.ht", dicho archivo se encuentra en la carpeta "receptor\pruebas\rs232" y haciendo doble *click* sobre él se crea una nueva conexión automáticamente ya configurada. Es posible, por tanto, ejecutar este archivo para abrir automáticamente un *Hyperterminal* ya configurado para la prueba con la única salvedad de la necesidad de especificar el puerto que se utilizará para la comunicación a través de la opción "Archivo -> Propiedades". En el caso que nos ocupa se ha utilizado un conversor serie-usb de forma que es posible disponer de conexión serie en ordenadores que carecen de este puerto de forma nativa. Dicho conversor crea una conexión virtual RS232 que ofrece todas las funciones de un puerto serie real.

Una vez programado el microcontrolador y encendido el circuito electrónico es posible verificar el correcto funcionamiento del hardware real y comprobar que la USART funciona correctamente, permitiendo interactuar con ella desde el PC y devolviendo el eco del carácter enviado.



4.2.2.- Memoria 24LC1025

El siguiente paso será la verificación del correcto funcionamiento de la memoria 24LC1025 de Microchip, así como del driver desarrollado para su manejo. Para llevar a cabo esta verificación se ha implementado un software que permite la prueba realizar operaciones de lectura y escritura de las direcciones de la memoria en modo acceso aleatorio, así como en modo página, el cual permite el acceso a bloques completos de memoria de tamaño de página de 128 bits.

Siguiendo la misma metodología que las comprobaciones del *timer* RTC DS1307, se lleva a cabo a través de la conexión RS232 haciendo uso para ello del *Hyperterminal* de Windows, suponiendo un sistema cómodo y sencillo de interactuación con el microcontrolador durante las operaciones de verificación del dispositivo bajo análisis.

```
25  #include <18F2550.h>
26
27  #fuses HS,NOWDT,NOPROTECT,NOLVP,BROWNOUT
28  #use delay(clock=20000000)
29
30
31  #define USE_INTERRUPTS
32
33  // Definición del pinout del puerto serie
34  #define RS232_TX            PIN_C6
35  #define RS232_RX            PIN_C7
36
37  // Definición del pinout de la EEPROM
38  #define EEPROM_24LC1025_PINOUT
39  #define I2C_SDA             PIN_B0
40  #define I2C_SCL             PIN_B1
41  #define EEPROM_24LC1025_WP  PIN_B2
42
43  #use rs232(baud=2400,xmit=RS232_TX,rcv=RS232_RX,bits=8,parity=N)
44  #use i2c(sda=I2C_SDA,scl=I2C_SCL,slow)
45
46
47
48  #include "24LC1025.h"
```

De forma análoga a los casos ya estudiados, se indicará la velocidad de reloj como primer paso después de la inclusión del archivo de cabecera del dispositivo y la palabra de configuración del mismo (fuses). Acto seguido, se lleva a cabo la definición de los pines de interconexión de la conexión serie directa de la USART del microcontrolador así como del patillaje de la memoria EEPROM, que debido a que se accede a ella mediante el bus i2c al igual que al *timer* DS1307, el *pinout* permanece inalterado. Únicamente será necesario añadir una definición, la correspondiente a la patilla de habilitación y deshabilitación de escritura de la memoria (línea 41).

Una vez definidas las conexiones pertinentes se creará el puerto serie y la conexión i2c. Según la página 1 del *datasheet* de la memoria (24LC1025.pdf), incluido como anexo, el dispositivo puede operar como máximo a una velocidad de bus de 400 KHz. Sin embargo, y debido a que comparte conexión con el *timer* y como consecuencia de su limitación de frecuencia de 100 KHz, deberá utilizarse la velocidad del dispositivo mas lento por lo que será necesario fijar la velocidad del bus a 100 KHz.

Una vez llevadas a cabo las definiciones de pines y la apertura del puerto i2c se incluirá el archivo cabecera del driver de la memoria mediante la pertinente directiva del preprocesador "#include", línea 48.

```
50      void escribe_memoria();
51      void lee_memoria();
52      void comprueba_acceso_pagina();
53      int32 obten_direccion();
54      int32 obten_direccion_pagina();
55      byte obten_dato();
56      int32 ascii2int32(char c);
57      byte ascii2byte(char c);
58      int compara_buffers(byte *buffer1, byte *buffer2, int n);
59
60
61      void main()
62          {
63          char orden;
64
65          EEPROM_24LC1025_init();
66          printf("Programa de test de la memoria 24LC1025.\r\n");
67
68          while(true)
69              {
70              do
71                  {
72                  printf("Escritura aleatoria = w\r\n");
73                  printf("Lectura aleatoria   = r\r\n");
74                  printf("Acceso a pagina     = p\r\n");
75                  printf("Opcion: ");
76                  orden = getc();
77                  printf("%c\r\n", orden);
78                  } while(orden != 'r' && orden != 'w' && orden != 'p');
79
80              if(orden == 'r') lee_memoria();
81              else if (orden == 'w') escribe_memoria();
82              else comprueba_acceso_pagina();
83              }
84          }
85
```

En el código de la prueba se han definido diversas funciones. Por una parte, “escribe_memoria()” lleva a cabo la operación de escritura aleatoria cuando el usuario selecciona dicha opción de prueba. Análogamente a ella, existe una función complementaria “lee_memoria()” sobre la cual se delega la operación de lectura aleatoria. Como función complementaria a ellas, se implementa la función “comprueba_acceso_pagina()” que realiza la comprobación de lectura y escritura de páginas fijas de 128 bits de tamaño.

Se dispone también de otras funciones internas, tales como “obtén_direccion()” y “obtén_direccion_pagina()” que obtienen una dirección válida para acceso aleatorio y para acceso de página respectivamente, así como la función “obten_dato()” que lee un byte del terminal para las operaciones de prueba de escritura aleatoria. La función “ascii2byte()” realiza la conversión de un valor ASCII en un valor en binario puro de tamaño 1 byte, dicha función ya ha sido analizada en el caso de estudio del timer DS1307 y no será analizada en este apartado para evitar redundancia. La función “ascii2int32()” opera del mismo modo que “ascii2byte()” solo que devuelve el resultado en un entero de 32 bits.

La función “compara_buffers()” permite comparar dos buffers de 128 bits. Es utilizada para comparar un buffer de recepción tras una operación de lectura de página con el buffer de envío y verificar de este modo que todo el bloque de datos escritos y posteriormente leídos se corresponde con los originales.

El método main() del programa de prueba muestra un menú por el *Hyperterminal*, de forma que permite al usuario seleccionar entre un abanico de opciones disponible. En caso de seleccionar la operación de escritura aleatoria se deberá pulsar “w” (*write*), en caso de querer llevar a cabo una operación de lectura se indicará al microcontrolador mediante la pulsación del carácter “r” (*read*), y en caso de que se desee iniciar una comprobación lectura y escritura por página a memoria se deberá indicar mediante la opción “p” (page). El bucle infinito comprueba que la opción seleccionada se halle dentro de los valores esperados, y acto seguido bifurca el flujo del programa delegando el control de la prueba a las funciones adecuadas habilitadas para ello.

Función de escritura aleatoria de memoria. Permite al usuario especificar una dirección cualquiera de escritura siempre y cuando se encuentre dentro del rango direccionable por la memoria.

```
87    void escribe_memoria()
88        {
89        int32 dir;
90        byte dato;
91        dir = obten_direccion();
92        dato = obten_dato();
93        EEPROM_24LC1025_write(dir, dato, WRITE_NOWAIT);
94        }
```

Primeramente obtiene una dirección válida de memoria (línea 91). Acto seguido, obtiene un dato por teclado (línea 92) y finalmente llama a la función de la API definida en el archivo 24LC1025.h encargada de realizar una operación de escritura. El parámetro WRITE_NOWAIT tiene como función indicar que tras el envío del comando de escritura, el programa debe continuar su flujo de instrucciones sin esperar a que se haya llevado a cabo la operación. En caso de especificar WRITE_WAIT, la función de escritura se detendrá hasta que el valor haya sido actualizado en la memoria.

La función complementaria de "escribe_memoria()" es "lee_memoria()". Como su nombre indica, su función consiste en la lectura de una dirección de memoria específica. Para ello se obtendrá una dirección válida contenida dentro del rango (línea 101) y se llamará a la función "EEPROM_24LC1025_read()" que permite el acceso de lectura aleatorio. Tras ello, se imprime el valor obtenido por la USART (línea 103). Los caracteres de formateado "%03u" indican que la representación se llevará a cabo siempre con 3 dígitos, rellenando por la izquierda con ceros cuando la cifra así lo requiera, y la "u" hace referencia a que el valor a representar es del tipo *unsigned* (positivo, sin signo).

```
97   void lee_memoria()
98       {
99       int32 dir;
100      byte dato;
101      dir = obten_direccion();
102      dato = EEPROM_24LC1025_read(dir);
103      printf("Dato: %03u\r\n", dato);
104      }
```

La función de prueba "comprueba_acceso_pagina()" permite llevar a cabo la comprobación de las operaciones de escritura y lectura de forma conjunta. Debido a que trabaja haciendo uso como unidad de operación la página de 128 bits, resulta absurdo e innecesario preguntar al usuario por 128 valores. En lugar de ello y con la finalidad de simplificar la operación, crea un buffer automáticamente de tamaño 128 bytes y lo rellena con valores conocidos. Acto seguido, inicia el acceso a página del dispositivo para después proceder a leer la misma página y comprueba el buffer leído con la página inicial. En caso de que los 128 valores del buffer se correspondan, el acceso habrá resultado exitoso y en caso de que difiera algún valor se deducirá que se ha producido un fallo. Si que se deja al usuario, sin embargo, la especificación a conveniencia de la dirección de página sobre la que se realizará la comprobación.

```
106  void comprueba_acceso_pagina()
107      {
108      int i;
109      int32 dir;
110      byte buffer_escritura[128];
111      byte buffer_lectura[128];
112
113      for(i=0; i<128; i++)
114          buffer_escritura[i] = i;
115
116      dir = obten_direccion_pagina();
117      printf("Escribiendo pagina... ");
118      EEPROM_24LC1025_page_write(dir, buffer_escritura, WRITE_WAIT);
119      printf("OK\r\n");
120      printf("Leyendo pagina... ");
121      EEPROM_24LC1025_page_read(dir, buffer_lectura);
122      printf("OK\r\n");
123
124      i = compara_buffers(buffer_escritura, buffer_lectura, 128);
125      if(i == 0) printf("EXITO!\r\n");
126      else printf("ERROR!\r\n");
127      }
```

Primeramente se declaran dos buffers consistentes en dos *arrays* de bytes de tamaño fijo e igual a 128 elementos (líneas 110 y 111). Acto seguido a través de un bucle "for()" se procede a llenar el buffer de escritura con valores consecutivos y conocidos, de forma que (línea 113):

buffer_escritura[0] = 0
buffer_escritura[1] = 1
buffer_escritura[2] = 2
buffer_escritura[3] = 3
buffer_escritura[4] = 4
buffer_escritura[5] = 5
…
buffer_escritura[127] = 127

Una vez rellenado el buffer completo, el programa pregunta al usuario por la dirección de página sobre la cual se realizarán las operaciones de prueba (línea 116). Sabiendo la dirección de página y teniendo el buffer de escritura, se efectúa una llamada a la función "EEPROM_24LC1025_page_write()" de la API la cual lleva a cabo la operación de escritura de página. Tras ello se procede a leer una página completa de la misma dirección de escritura mediante la instrucción complementaria "EEPROM_24LC1025_page_read()", en la línea 121.

Tras haber efectuado estas operaciones, en "buffer_escritura[]" se encontrará el valor original que ha sido escrito en la página cuya dirección ha sido especificada por el usuario, y en "buffer_escritura[]" los datos de la misma página tras la operación de lectura. En el caso de que las funciones de escritura y lectura funcionen correctamente y el intercambio de datos se haya llevado a cabo de forma satisfactoria, el contenido de ambos buffers debería ser el mismo.
Con la finalidad de realizar esta comprobación se llama a la función "compara_buffers()" (línea 124) que comprueba que los datos contenidos en ambos

arrays coinciden uno a uno. Tras ello, se muestra un mensaje informando del resultado de la operación (líneas 125 y 126).

Para obtener una dirección válida de dispositivo sobre la que operar, se pone a disposición la implementación de la función “obten_direccion()”. Dicha función pregunta al usuario a través del *Hyperterminal* por una dirección de memoria dentro del rango [000000,131071] consecutivamente hasta que no se introduzca una dirección válida, a través del bucle “do while()” que tiene comienzo en la línea 136.

```
129  int32 obten_direccion()
130      {
131      int32 dir;
132
133      char a,b,c,d,e,f;
134      int32 ai,bi,ci,di,ei,fi;
135
136      do
137          {
138          printf("Direccion (000000-131071): ");
139
140          a = getc();
141          putc(a);
142          b = getc();
143          putc(b);
144          c = getc();
145          putc(c);
146          d = getc();
147          putc(d);
148          e = getc();
149          putc(e);
150          f = getc();
151          putc(f);
152
153          printf("\r\n");
154
155          ai = ascii2int32(a);
156          bi = ascii2int32(b);
157          ci = ascii2int32(c);
158          di = ascii2int32(d);
159          ei = ascii2int32(e);
160          fi = ascii2int32(f);
161
162          dir = fi+ei*10+di*100+ci*1000+bi*10000+ai*100000;
163          } while(dir<0 || dir>131071);
164
165      return dir;
166      }
```

Se obtienen 6 caracteres del puerto serie que conforman la dirección requerida, haciendo eco tras cada recepción para mostrar por pantalla en todo momento lo que teclea el usuario. Como siguiente paso, se convierte cada carácter leído a su correspondiente entero natural en una variable de 32 bits líneas 155 a 160) para finalmente sumar todos ellos atendiendo a la posición que ocupan, obteniendo de este modo la dirección de memoria (línea 162).

A la hora de llevar a cabo una operación de comprobación de acceso a página, es necesario tener en cuenta que la dirección de la misma debe ser múltiplo de 128 según indica la pág. 8 del *datasheet* del dispositivo (24LC1025.pdf). En caso contrario, una vez llegado al límite de página la escritura continuará sobrescribiendo los valores contenidos a partir del inicio de la misma.

*"**Note:** Page write operations are limited to writing bytes within a single physical page, regardless of the number of bytes actually being written. Physical page boundariesstart at addresses that are integer*
multiples of the page buffer size (or 'page size') and end at addresses that are integer multiples of [page size – 1]. If a Page Write command attempts to write across a physical page boundary, the result is that the data wraps around to the beginning of the current page (overwriting data previously stored there), instead of being written to the next page as might be expected. It is therefore, necessary for the application software to prevent page write operations that would attempt to cross a page boundary."

```
169  int32 obten_direccion_pagina()
170       {
171       int32 dir;
172       printf("Debe ser multiplo de 128,\r\n");
173
174       do
175            {
176            dir = obten_direccion();
177            } while(dir % 128 != 0);
178
179       return dir;
180       }
```

Con el fin de obtener una dirección válida, un bucle "do while()" comprueba que la dirección es múltiplo de 128 y pregunta de nuevo al usuario repetidamente hasta que éste introduzca una dirección válida. Para comprobar que la dirección es múltiplo de 128 se lleva a cabo la división por 128 y se comprueba el resto, en caso de que el resto sea igual a cero es que la dirección es múltiplo (línea 177).

Tras efectuar las operaciones de lectura y escritura de la memoria se comparan ambos buffers, de la manera anteriormente descrita. Para ello se hace uso de la función "compara_buffers()" la cual lleva a cabo la ´comparación de ambos buffers verificando de este modo la validez de las operaciones de escritura/lectura de página. La

comprobación se lleva a cabo elemento a elemento a través de un bucle “for()” (línea 188), de forma que en el momento en que se encuentra una incoherencia entre ambos en alguno de sus elementos, se detiene la comprobación mediante la comprobación de la variable “error” en cada pasada del bucle.

```
182
183     int compara_buffers(byte *buffer1, byte *buffer2, int n)
184         {
185         int1 error = 0;
186         int i;
187
188         for(i=0; i<n && !error; i++)
189             if(*(buffer1++) != *(buffer2++)) error=1;
190
191         return error;
192         }
```

Una vez finalizado el bucle de comprobación y ya sea tanto en caso de éxito tanto como de error, la variable “error” contendrá el valor de la comprobación. En caso de haber resultado exitosa, el bucle habrá terminado porque la variable “i” habrá llegado al valor límite de tamaño de página pasado como parámetro (y fijado en 128 durante la llamada a la función) y la variable “error” valdrá 0 (no error). En caso de haber finalizado el bucle porque se haya producido un error en la comprobación, la variable “error” valdrá 1 y éste será el valor devuelto por la orden “*return*” (línea 191).

Durante la comprobación de escritura en acceso aleatorio en memoria, se pide al usuario por un valor de tamaño 8 bits (tamaño de palabra de la memoria). La obtención de dicho valor queda delegada a la función “obtén_dato()”.

4.2.3.- Timer RTC DS1307.

Con el fin de verificar el correcto funcionamiento del *timer* RTC DS1307 y su conexión con el microcontrolador a través de i2c así como el correcto funcionamiento del driver de dispositivo desarrollado, se ha programado una aplicación de prueba que permite operar sobre él y llevar a cabo la prueba de las operaciones básicas de puesta en hora y lectura de la misma.

Como método de establecer una comunicación directa entre microcontrolador y usuario y poder en todo momento operar directamente con las funciones de lectura y escritura del *timer*, se ha hecho uso de la conexión RS232 de la USART del PIC de forma que una vez en ejecución es posible poner en hora el *timer* así como efectuar operaciones de lectura a conveniencia desde el *Hyperterminal* de Windows.

```
26      #use delay(clock=20000000)
27
28
29      // Pinout DS1307
30      #define DS1307_PINOUT
31      #define I2C_SDA      PIN_B0
32      #define I2C_SCL      PIN_B1
33
34      // Pinout RS232
35      #define RS232_TX     PIN_C6
36      #define RS232_RX     PIN_C7
37
38
39      #use i2c(master,sda=I2C_SDA,scl=I2C_SCL,slow)
40      #use rs232(baud=2400,xmit=RS232_TX,rcv=RS232_RX,bits=8,parity=N)
```

Primeramente se define la frecuencia de funcionamiento del cuarzo del PIC, 20 MHz. Esto es necesario especificarlo antes de la definición de la conexión serie e i2c ya que, como se ha especificado en la prueba anterior, estas funciones operan a nivel de señal directamente sobre las patillas de los buses y se hace necesario para la generación de ondas y *timing* de la comunicación. Acto seguido se define el *pinout* de la conexión I2C sobre la que se basa el DS1307 así como de la conexión serie que permitirá operar a través de *Hyperterminal*. Se define la macro "DS1307_PINOUT" con la finalidad de hacer saber a la librería desarrollada para el *timer* que se ha definido el *pinout*, en caso contrario mostrará un error haciendo notar que es necesaria la definición de las patillas del dispositivo.

Tras ello, se definirán las conexiones i2c y rs232. La conexión rs232 será la misma que la definida en la prueba anterior, y la conexión i2c se especificará que el dispositivo es el máster, que la patilla de datos es la definida por I2C_SDA y la de reloj por I2C_SCL, y que se va a utilizar una velocidad de transmisión lenta, 100 KHz, ya que es la única velocidad soportada por el dispositivo (*datasheet* pág. 6, DS1307.pdf).

"Within the 2-wire bus specifications a regular mode (100kHz clock rate) and a fast mode (400kHz clock rate) are defined. The DS1307 operates in the regular mode (100kHz) only. "

```
42
43      #include "DS1307.h"
44
45
46      void escribe_rtc();
47      void lee_rtc();
48      byte ascii2byte(char c);
49
50
51      void main()
52          {
53          char orden;
54          // Fijamos RB2 como salida a 1 para evitar dejar encendido
55          // indefinido el led amarillo compartido por la eeprom, y
56          // las salidas a los reles todas a 0.
57          set_tris_b(0x00);
58          output_b(0x04);
59
60          // Parpadeo del led a frecuencia de un herzio
61          DS1307_init(DS1307_OUT_ON | DS1307_OUT_1_HZ);
62          while(1)
63              {
64              do
65                  {
66                  printf("Leer o escribir RTC? (r/w): ");
67                  orden = getc();
68                  printf("\r\n");
69                  } while(orden != 'r' && orden != 'w');
70
71              if(orden == 'r') lee_rtc();
72              else escribe_rtc();
73              }
74          }
```

Tras definir los pines de conexión y abrir las conexiones rs232 e i2c, el siguiente paso es la inclusión del archivo de cabecera del driver del *timer* (línea 43). El orden es importante ya que este archivo se basa en las definiciones previas.

Se han definido 3 funciones, escribe_rtc() (línea 46), lee_rtc() (línea 47) y ascii2byte() (línea 48). Sobre la primera de ellas se delega el control de las operaciones de escritura, análogamente a ello el *main* pasa el control pasa a la segunda función cuando se trata de una operación de lectura, y la tercera operación tiene como finalidad la conversión de caracteres numéricos ASCII a sus correspondientes en binario natural.

La función *main* de la prueba no supone una complejidad excesiva, primeramente fija a nivel alto la salida RB2 del microcontrolador y a nivel bajo el resto de salidas, ello tiene la finalidad de de apagar inicialmente el LED amarillo por una parte y de desactivar todos los relés de la placa. Es importante recordar que el diodo LED está gobernado por un transistor PNP BC557 operando en corte/saturación y que un nivel alto en su salida hace entrar al mismo en corte desactivando el LED. Contrariamente a lo que ocurre con los relés, las salidas del microcontrolador están

directamente cableadas a las etapas de potencia ULN2803 con lo que para desactivarlos es necesario establecer las salidas del microcontrolador asociadas a nivel bajo.

Posteriormente se procede a inicializar el *timer*, para ello se llama a la función DS1307_init() en la línea 61 con los parámetros DS1307_OUT_ON y DS1307_OUT_1_HZ, que habilitan la salida 7 del *timer* haciendo accesible la onda cuadrada generada a una frecuencia de 1 Hz. Esta salida está directamente cableada a la base del transistor mencionado BC557 de forma que será posible verificar la oscilación del cuarzo de 16.768 KHz del *timer* viendo parpadear el LED a la frecuencia establecida de 1 Hz.

Como se puede observar en la línea de código número 68, la parte principal del programa la conforma un bucle infinito el cual pregunta al usuario por la operación a llevar a cabo. Es posible leer la hora (opción “r”) y escribirla en la configuración del chip (opción “w), el programa se cerciora que la orden introducida sea la correcta mediante el while() condicional de la línea 69, el cual sigue iterando si la opción es errónea. Si la opción introducida hace referencia a una operación de lectura, se delega el control en la función “lee_rtc()” (línea 71), y en caso de tratarse de una operación de escritura se llama a la función “escribe_rtc()” (línea 72).

La función de escritura del *timer* se muestra a continuación:

```
77  // Pregunta al usuario la hora y actualiza el timer
78  void escribe_rtc()
79      {
80      byte hora;
81      byte min;
82      byte seg;
83      int unidades;
84      int decenas;
85      char c;
86
87      printf("Hora (00-23): ");
88      c=getc();
89      putc(c);
90      decenas = ascii2byte(c);
91      c=getc();
92      putc(c);
93      unidades = ascii2byte(c);
94      hora = decenas*10+unidades;
95      printf("\r\n");
96
97      printf("Min (00-59): ");
98      c=getc();
99      putc(c);
100     decenas = ascii2byte(c);
101     c=getc();
102     putc(c);
103     unidades = ascii2byte(c);
104     min = decenas*10+unidades;
105     printf("\r\n");
106
107     printf("Seg (00-59): ");
108     c=getc();
109     putc(c);
110     decenas = ascii2byte(c);
111     c=getc();
112     putc(c);
113     unidades = ascii2byte(c);
114     seg = decenas*10+unidades;
115     printf("\r\n");
116
117     DS1307_set_time(hora, min, seg);
118     printf("Hora establecida en: %02i:%02i:%02i\r\n", hora, min, seg);
119     }
```

Primeramente la función pregunta al usuario la hora a introducir (línea 88). En todo momento se realiza un eco al *Hyperterminal* (línea 89) con la finalidad de mostrar el carácter que se está tecleando, en caso de no realizar esta operación el usuario no vería gráficamente la pulsación de las teclas sobre el terminal.

Acto seguido se realiza la conversión del carácter recibido a su correspondiente en binario natural ya que el *Hyperterminal* opera sobre caracteres ASCII. En caso de enviar un número sobre el terminal, éste envía su correspondiente en representación ASCII y no el número en binario real, que es lo deseado.

DEC	HEX	OCT	CHAR	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH	DEC	HEX	OCT	CH
0	0	000	NUL	32	20	040		64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	80	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Por ejemplo, en caso de introducir en el terminal el valor numérico “7”, el carácter recibido por el PIC no será el 0x07 como sería lo esperado sino su correspondiente en representación ASCII, en este caso el valor enviado por el *Hyperterminal* sería 0x37. Se hace necesario por tanto para la correcta interacción PC-PIC cuando se va a llevar a cabo el intercambio de valores numéricos de la función de conversión de ASCII a binario durante la comunicación PC a PIC, y binario a ASCII durante la comunicación PIC a PC. Sin embargo en este último caso la operación se simplifica bastante ya que puede realizarse implícitamente mediante la función “printf()” con el parámetro %i, seguido del número a convertir. Por ejemplo, para imprimir por el *Hyperterminal* el número 9, contenido en la variable dato, basta con utilizar la siguiente orden de la biblioteca estándar de C:

printf(“%c”, dato);

Una vez obtenido el valor de las unidades y las decenas, se construye el número completo (línea 94) multiplicando el valor de las decenas por 10 y sumándole las unidades._Es importante comentar que en caso de introducir valores erróneos, por ejemplo introducir como hora un número fuera del rango [0-23] dará lugar al erróneo funcionamiento del dispositivo. Es función del programador cerciorarse que los valores introducidos se encuentran dentro de los rangos establecidos, sería posible introducir en el código instrucciones de control de los valores fuera de rango sin embargo debido a que se trata de un programa de prueba, no supone un problema serio ya que el usuario es el encargado de introducir los valores adecuados y se evita introducir complejidad innecesaria al código.

Del mismo modo, puede ser interesante como ejercicio adicional hacer pruebas con valores fuera de rango con el fin de observar el comportamiento del *timer*.

El fragmento de código comprendido entre las líneas 97 a 105 repite el proceso para obtener los minutos, y el código contenido entre 107 y 115 realiza la misma operación nuevamente con el fin de obtener el valor de los segundos.

En la línea 117 se llama a la función DS1307_set_time(), la cual actualiza el *timer* con la hora especificada. Tras ello, en la línea 118 se muestra un mensaje de retorno al ordenador que será mostrado por el *Hyperterminal* y confirma que se ha llevado a cabo la operación.

```
118       printf("Hora establecida en: %02i:%02i:%02i\r\n", hora, min, seg);
```

El parámetro “%02i” de la función *printf* indica que el valor pasado es un entero, y que se mostrará siempre con dos cifras rellenando por la izquierda con ceros. La finalidad de ellos es mejorar la presentación y mostrar la hora de la forma habitual.

La función lee_rtc() es mas sencilla que la anterior, y únicamente realiza una llamada a la función DS1307_get_time() con el fin de obtener la hora del dispositivo y acto seguido envía por la USART un mensaje con el resultado:

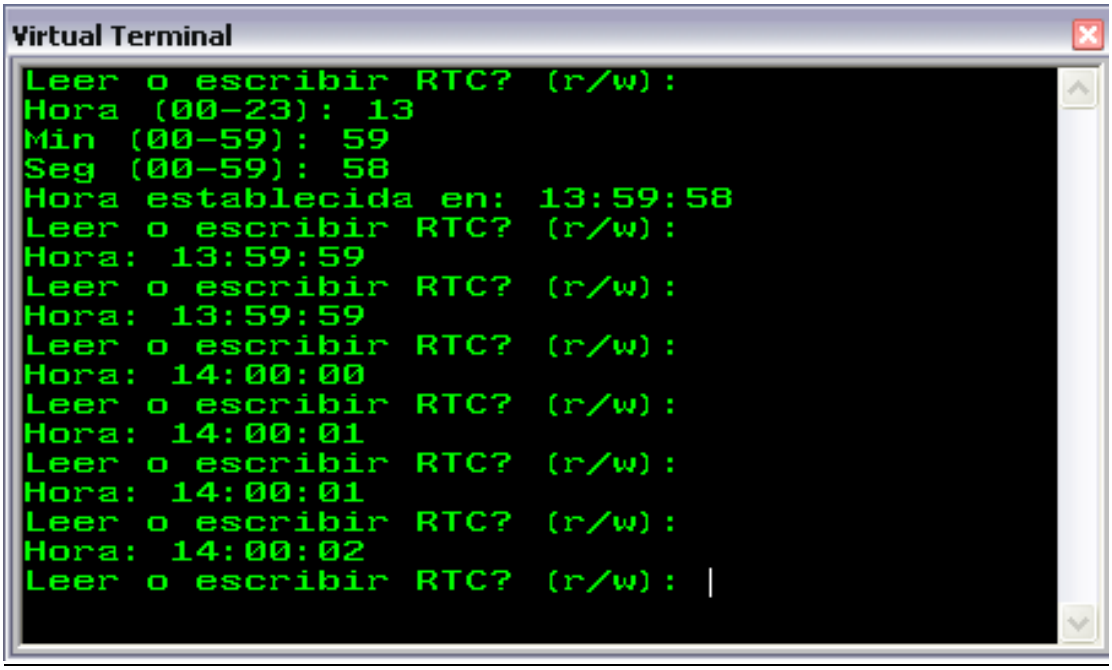
```
122   // Lee la hora del timer y la muestra por el terminal
123   void lee_rtc()
124       {
125       byte hora;
126       byte min;
127       byte seg;
128
129       DS1307_get_time(&hora, &min, &seg);
130       printf("Hora: %02i:%02i:%02i\r\n", hora, min, seg);
131       }
```

El último fragmento de código de el programa de prueba hace referencia a la función anteriormente nombrada ascii2byte() la cual se encarga de convertir el carácter ASCII pasado como parámetro en un valor de retorno entero correspondiente al valor en binario natural. Para ello la función la compone un bloque “swtich()” con un “case” por valor, de forma que dependiendo del valor obtenido la conversión es inmediata. En caso de un valor erróneo se ejecuta la cláusula por defecto “default”, la cual devuelve 0:

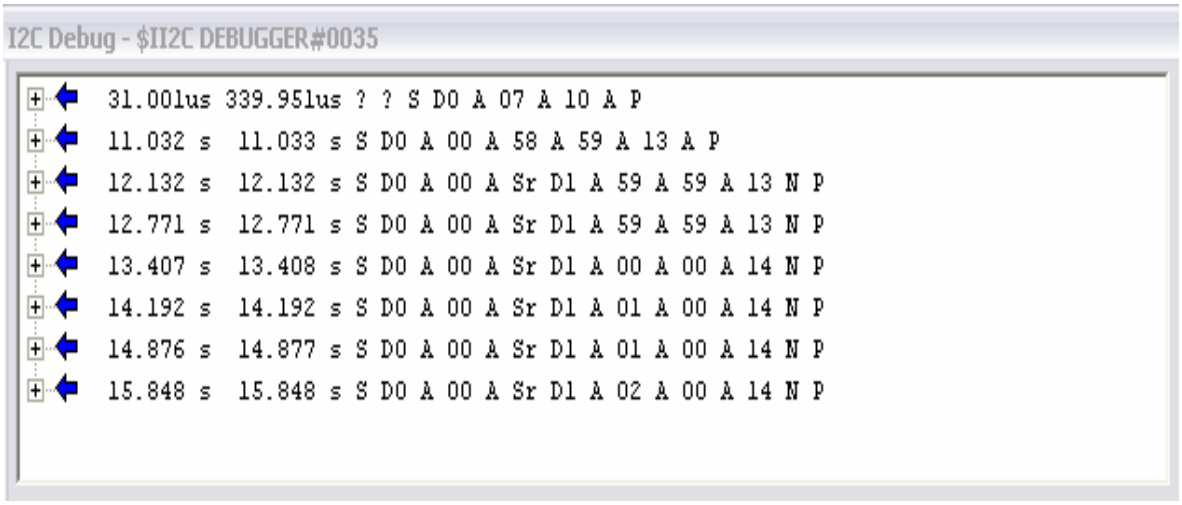
```
134  // Convierte un número en binario puro a ASCII para su correcta representación
135  // por el terminal del puerto serie.
136  byte ascii2byte(char c)
137      {
138      byte result;
139
140      switch(c)
141          {
142          case '1':
143              result = 1;
144              break;
145          case '2':
146              result = 2;
147              break;
148          case '3':
149              result = 3;
150              break;
151          case '4':
152              result = 4;
153              break;
154          case '5':
155              result = 5;
156              break;
157          case '6':
158              result = 6;
159              break;
160          case '7':
161              result = 7;
162              break;
163          case '8':
164              result = 8;
165              break;
166          case '9':
167              result = 9;
168              break;
169          default:
170              result = 0;
171          }
172      return result;
173      }
174
```

Una vez tratado el tema del código se simulará con el fin de verificar el correcto funcionamiento del mismo, así como del *timer* y la conexión i2c. Para ello se cargará en *Proteus* de forma análoga al procedimiento llevado a cabo durante la prueba de la USART del PIC, en este caso se hará doble *click* sobre el microcontrolador en el esquema y se seleccionará el firmware ya compilado bajo el nombre DS1307.hex. En este caso se hará uso también del monitor de actividad I2C del simulador, el cual se conectará directamente en paralelo a las líneas SDA y SCL como si se tratase de un dispositivo más y se configurará a velocidad de 100 KHz en el menú de propiedades desplegado al hacer doble *click* sobre él.

Se iniciará la simulación y e inmediatamente se abrirá el terminal virtual de *Proteus* que simula una conexión RS232, así como el monitor de actividad del bus i2c. A continuación puede observarse el diálogo de prueba entre el microcontrolador y el usuario durante la prueba de verificación del *timer*:



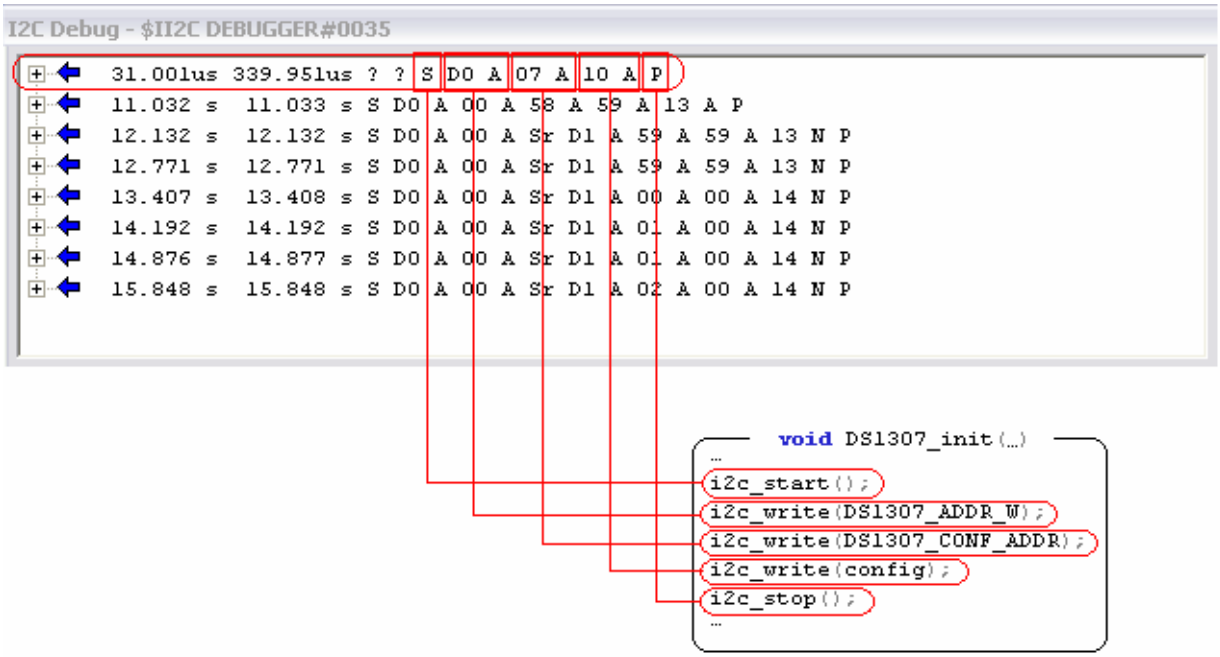
Dicho diálogo ha dado lugar a un intercambio de información directo PIC-DS1307 a través del bus I2C, información que puede ser analizada mediante el monitor i2c que *Proteus* pone a disposición del usuario para dicho cometido. Los datos capturados por el monitor son los siguientes.



Como puede observarse existe una correspondencia directa entre las órdenes enviadas al PIC a través del *Hyperterminal* y la actividad del puerto I2C llevada a cabo por el PIC. La primera línea (seg. 11.032) se corresponde con la inicialización del *timer*. La segunda con la puesta en hora. De aquí en adelante, cada una de las tramas de datos hace referencia a una operación de lectura del chip.

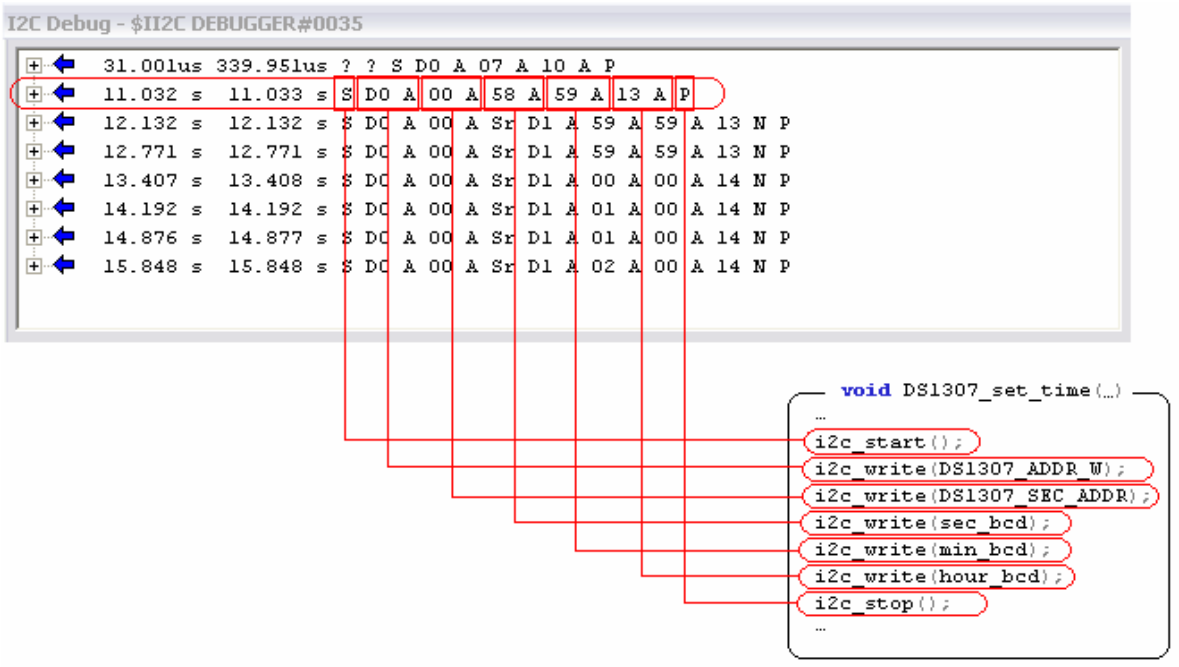
Es posible comprobar como las instrucciones de código de acceso al bus I2C del driver desarrollado llevan a cabo su cometido, controlando la información enviada y recibida a través del bus con los datos adecuados y en el orden especificado tal y como se puede ver a continuación.

Primeramente se analizará la función de inicialización.



Como se puede observar, y de acorde a las especificaciones del bus i2c, primeramente se envía el bit de inicio (representado por el carácter “S” en el visor). Acto seguido se direcciona el timer volcando al bus la dirección del mismo en modo escritura, lo cual se lleva a cabo colocando el bit menos significativo (LSB) de la dirección a 0. La dirección del dispositivo en modo escritura será, por tanto, 0xD0. Una vez hecho esto se procede a enviar el apuntador de registro sobre el cual se quiere trabajar. El registro de configuración es el 0x07 (pág. 5, datasheet DS1307.pdf), definida por la macro DS1307_CONF_ADDR. Tras ello, se envía por el bus el valor a guardar en el registro especificado, en este caso la palabra de configuración. Como último paso y siguiendo con el cumplimiento del protoclo del bus i2c, se envía el bit de STOP concluyendo el envío de información y cerrando la comunicación PIC-dispositivo. Como se puede observar en la figura, cada vez que el microcontrolador envía un byte al dispositvo a través del bus este responde con un bit de confirmación o ACK, representado en la traza por el carácter “A” (Acknowledgment).

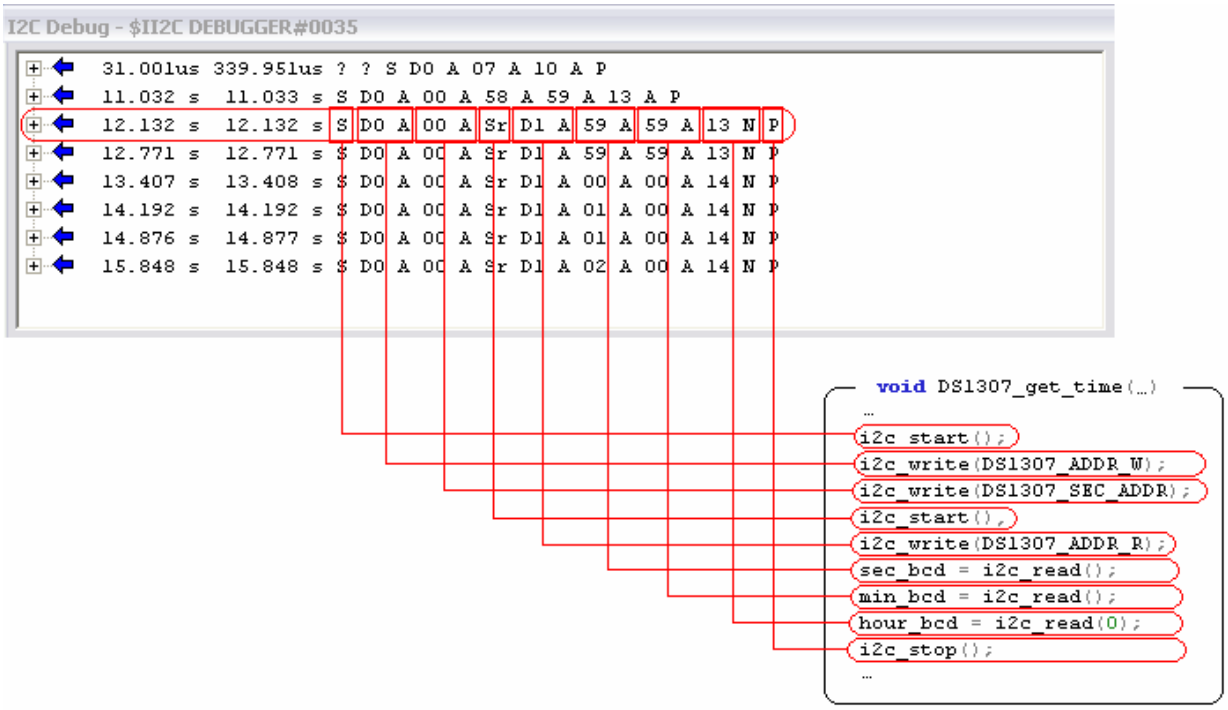
Se procederá al análisis del intercambio de información a través del bus durante una operación de escritura, a través de la cual el microcontrolador establece la hora del dispositivo escribiendo los valores correspondientes en los registros al efecto. El diagrama que refleja el flujo de datos desde el microcontrolador hacia el timer queda ilustrado en la siguiente figura:



Primeramente el microcontrolador manda el bit de start indicando de este modo que va a iniciarse una transferencia a través del bus. Acto seguido, vuelca en el mismo la dirección del dispositivo sobre el que operar. De manera análoga a la función de inicialización, tras ello se envía la dirección del registro objeto de escritura, en este caso es el correspondiente al valor de los segundos. Sucesivas operaciones de escritura actualizan los registros inmediatamente posteriores (contador interno autincremental). La dirección del registro segundero está definida en el archivo de cabecera del driver DS1307.h bajo el nombre DS1307_SEC_ADDR. Una vez indicado el registro, se envía

la información a escribir en él. Un segundo envío actualizará el valor del registro del minutero y una tercera escritura actualizará el valor de las horas. Del mismo modo que en el caso anterior y tal como especifica el estándar, con el fin de finalizar el intercambio de información el microcontrolador enviará el bit de STOP dando por concluida la puesta en hora del chip.

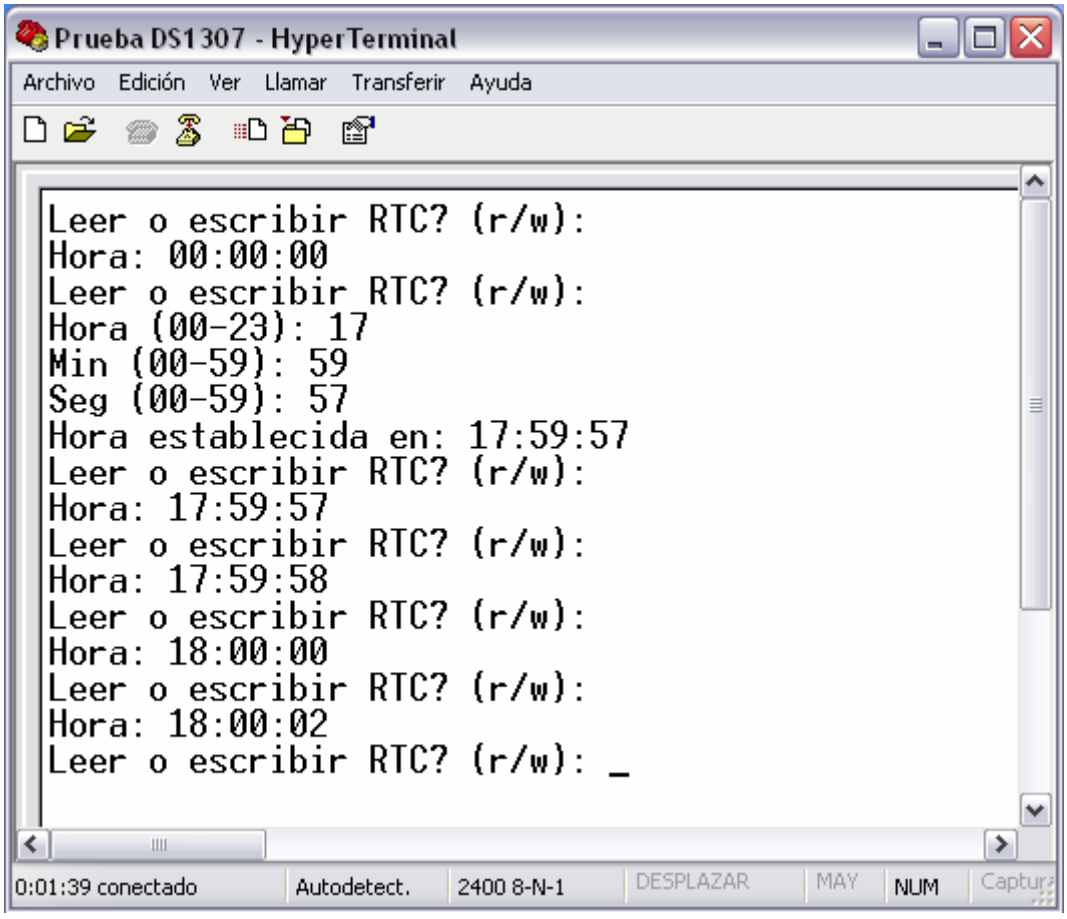
Como último caso de análisis se llevará a cabo el análisis del flujo de información como consecuencia de una operación de lectura. En este caso el procedimiento difiere del visto hasta ahora, puesto que una operación de lectura requiere llevar a cabo previamente una de escritura ya que antes de obtener el valor de un registro es necesario actualizar el apuntador interno con la dirección del mismo. La siguiente figura refleja la actividad del bus durante una operación de lectura de hora:



Primeramente el microcontrolador envía el bit de *start* inicializando de este modo la comunicación con el dispositivo. Acto seguido se vuelca al bus la dirección del *timer* en modo escritura, haciendo uso de la macro DS1307_ADDR_W. Una vez hecho esto, se especifica la dirección motivo de lectura escribiendo el registro apuntador del *timer*. Para ello, se enviará la dirección del segundero DS1307_SEC_ADDR. En estos momentos el registro apuntador conocerá la dirección sobre la cual se quiere obtener el valor, es necesario entonces iniciar una nueva transferencia sin soltar el control del bus. Para ello el microcontrolador envía de nuevo el bit de *start* sin cerrar la comunicación anterior, esto es, sin enviar el bit de parada. Ello queda gráficamente representado en el monitor del bus por el valor “Sr”. Acto seguido el PIC direcciona de nuevo el dispositivo, esta vez en modo lectura, esto se consigue colocando el bit menos significativo de la dirección de dispositivo (LSB) a 1, quedando la dirección por tanto definida por el valor 0xD1.

Del mismo modo que ocurre durante la escritura de un registro, la lectura también incrementa automáticamente en 1 el valor del registro apuntador de forma que sucesivas lecturas obtienen valores de registros contiguos en orden ascendente. De este modo, se llevarán a cabo tres lecturas consecutivas, una para los segundos, otra para los minutos y una tercera y última para las horas, respectivamente. Como se puede observar, tras cada operación de lectura el microcontrolador valida la misma mediante el envío al *timer* del bit de reconocimiento (ACK) representado en la figura por el carácter "A", excepto tras la lectura del último valor que envía un carácter de no-reconocimiento o NACK con el fin de indicar al dispositivo el fin de envío de datos. Finalmente, el microcontrolador cierra al comunicación mediante el envío del bit de stop, representado por el carácter "P".

Una vez llevada a cabo la comprobación sobre el simulador *Proteus* se realizará la misma operación sobre hardware real. Para ello se grabará el microcontrolador de la forma habitual con el mismo firmware utilizado para la simulación, DS1307.hex.



Para configurar el *Hyperterminal* hay que especificar las siguientes opciones:

Baudios: 2400
Bits de datos: 8
Paridad: Ninguno
Bits de parada: 1
Control de flujo: Ninguno

No obstante, se suministra junto con el código fuente en *MPLab* de la prueba el archivo “Prueba DS1307.ht”. Haciendo doble *click* en este archivo se abre una sesión de *Hyperterminal* totalmente funcional con los parámetros ya configurados, siendo únicamente necesario especificar el puerto de conexión.

Una vez en funcionamiento el código del microcontrolador sobre el hardware real, se comprueba que las operaciones de lectura y escritura funcionan correctamente así como del mismo modo también se observa el parpadeo del diodo LED amarillo a la frecuencia de 1 Hz. Queda de este modo comprobado el correcto funcionamiento del driver desarrollado para controlar el *timer* RTC DS1307, así como el correcto funcionamiento del mismo y del cristal de cuarzo. También queda demostrado el correcto cableado y funcionamiento del bus i2c sobre la placa.

Como observación, en el *datasheet* del *timer* (*pag*. 4 *datasheet* DS1307.pdf) indica que tras la inicialización del dispositivo el valor de los registros es indeterminado:

“Please note that the initial power-on state of all registers is not defined. Therefore, it is important to enable the oscillator (CH bit = 0) during initial configuration.”

Sin embargo tras sucesivas comprobaciones se ha observado que inicialmente la hora siempre es 00:00:00 y el bit de habilitación de siempre se encuentra desactivado, esto significa que hasta la primera puesta en hora del dispositivo éste permanecerá desactivado y fijo en 00:00:00.

4.2.3.- Conexión USB CDC

Como primer ejemplo del uso del puerto USB, y más concretamente en su clase de aplicación CDC (Class Device Comunicaction), se ha implementado una sencilla aplicación de eco que devuelve el carácter enviado.

Se crea una conexión RS232 virtual a través de USB, de forma que es posible conectar a través del Hyperterminal de Windows buscando el puerto serie COMx creado. Dicho puerto será accesible como si un puerto serie real se tratase.

Para ello es necesario la creación de un archivo de especificación de driver BLA que será proporcionado al sistema la primera vez que se conecte al ordenador, el cual será detallado posteriormente.

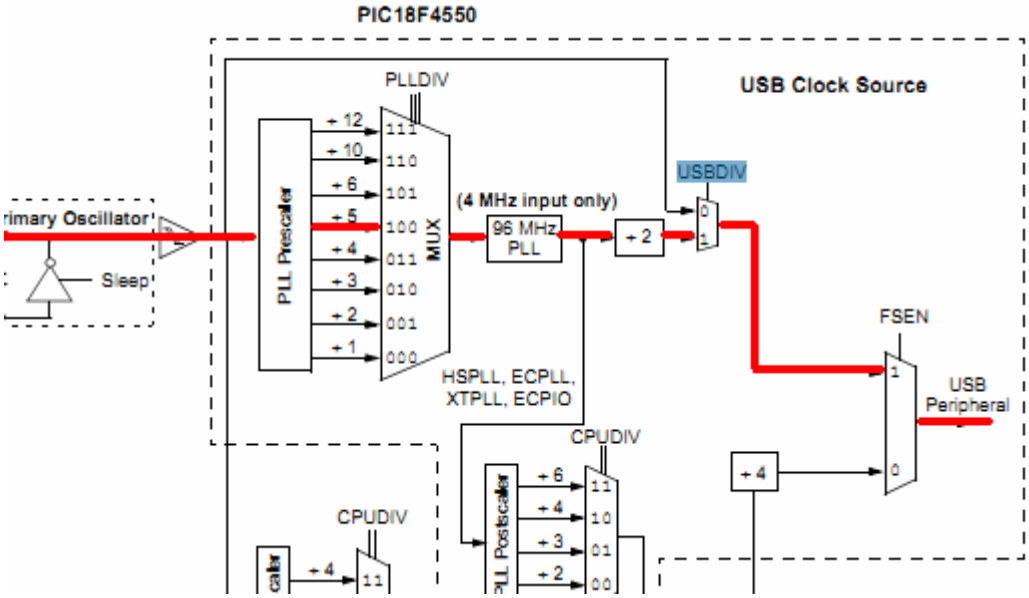
Se procederá a continuación a explicar el código del microcontrolador detalladamente:

```
2   #include <18F2550.h>
3
4   #fuses HSPLL,NOWDT,NOPROTECT,NOLVP,USBDIV,PLL5,CPUDIV1,VREGEN
5   #use delay (clock=20000000)
6
7
8
9   #include "usb_cdc.h"
10  #include "usb_desc_cdc.h"
11
```

En la línea 2 se hace la inclusión del archivo de cabecera de dispositivo, como viene siendo habitual en todos los proyectos en lenguaje C. Dicho archivo incorpora las definiciones y recursos del compilador necesarios para poder trabajar con el microcontrolador concreto en cuestión.

En las líneas 4 y 5 se indica la configuración del microcontrolador. NOWDT hace referencia al *watchdog* o perro guardián, indicando que no va a ser utilizado. NOLVP desactiva la programación a bajo voltaje.

Los fusibles HSPLL, USBDIV y PLL5 configuran el microcontrolador para obtener los 48 MHz necesarios para hacer funcionar el módulo USB a través de los 20 MHz entrantes por las patillas de entrada del oscilador. Los 20 MHz se dividen entre 5 en el preescaler del PLL para obtener los 4 MHz necesarios para el PLL propiamente dicho, ya que esta es la frecuencia necesaria para obtener los 96 MHz que posteriormente serán divididos por 2 para finalmente obtener los 48 MHz. En la siguiente figura puede verse un esquema del módulo de configuración de frecuencias, en el cual se ha resaltado la ruta configurada por los fusibles para obtener la frecuencia necesaria:



Las líneas 9 y 10 incluyen las librerías de trabajo que el compilador CCS proporciona para el uso del bus USB bajo la clase CDC. Todos los proyectos que vayan a hacer uso de esta clase de trabajo necesitarán incluir estos dos archivos.

A continuación se analiza la única función que compone el programa, la función *main*:

```
13  void main()
14      {
15      byte caracter;
16
17      set_tris_a(0x00);
18      set_tris_b(0x00);
19      set_tris_c(0x00);
20      output_a(0x00);
21      output_b(0x00);
22      output_c(0x00);
23
24      usb_cdc_init();
25      usb_init();
26
27      while(true)
28          {
29          if(usb_cdc_connected())
30              {
31              usb_task();
32              if(usb_enumerated())
33                  {
34                  caracter = usb_cdc_getc();
35                  usb_cdc_putc(caracter);
36                  }
37              }
38          }
39      }
```

Primeramente se configuran los puertos como salida, y se establece su valor en 0 (nivel bajo). Esto se hace para mantener desactivados los relés cableados a los puertos del microcontrolador.

Posteriormente en las líneas 24 y 25 se llama a las funciones *usb_cdc_init()* y *usb_init()* que inicializan las estructuras de datos y recursos que implementan la clase de uso CDC, y el módulo USB, respectivamente. Antes de poder hacer uso de las funciones del USB será necesario llamar a estas funciones de inicialización.

Tras ello se entra en un bucle infinito el cual lleva a cabo la misma función. El bus USB se trata de un bus de naturaleza impredecible, esto es, el hecho de que sea un BUS *hot-plug* hace posible la desconexión del mismo mientras se está trabajando con él. En un momento dado podría conectarse y volver a desconectarse el cable. El dispositivo podría quedarse sin alimentación. De comprobar si el cable se encuentra conectado se encarga la función *usb_cdc_connected()*.

Del mismo modo, podría ser que el cable se encontrara conectado pero todavía no hubiera sido enumerado por el PC. Esto es, que todavía no hubiera sido reconocido y su driver cargado y asociado, con lo cual seguiría estando inutilizable. De comprobar si el dispositivo se encuentra enumerado por el host, se encarga la función *usb_enumerated()*.

Se hace necesario llevar a cabo comprobaciones continuas sobre el estado del bus con el fin de llevar a cabo las gestiones de funcionamiento internas del mismo. De ello se encarga la función *usb_task()*. Esta función debe ser llamada continuamente. Llamadas sucesivas a la misma es un acto benigno. Debe ser invocada en cada pasada del bucle.

Las funciones *usb_cdc_getc()* y *usb_cdc_putc()* leen y escriben (respectivamente) un carácter del puerto serie virtual creado a través de la conexión USB.

Una vez estudiadas las funciones que componen el programa, se explicará su funcionamiento. La función *main* ejecuta constantemente las mismas instrucciones. Básicamente, comprueba si hay algún carácter a la entrada y posteriormente lo devuelve. Sin embargo, y debido a la naturaleza impredecible y cambiante del bus USB, como se ha explicado anteriormente, se añaden algunas funciones extra de apoyo. En cada pasada del bucle, y antes de llevar a cabo ninguna otra operación, primeramente se comprueba que el cable se encuentra conectado. En ese caso se llama a la función *usb-task()* y posteriormente se comprueba si se encuentra enumerado. En caso negativo no se hace nada más y se vuelve a hacer lo mismo constantemente hasta que esté enumerado (reconocido por el driver). Una vez se encuentra enumerado, cada pasada del bucle llega hasta lo más profundo de la serie de llamadas de funciones y lee un carácter para posteriormente devolverlo (función “eco”).

El *firmware* aquí mostrado es bastante sencillo ya que hace uso de las librerías que ocultan toda la complejidad del bus USB y la implementación de la clase CDC. Para que el correcto funcionamiento y comunicación entre el host (PC) y el dispositivo sea posible es necesario que exista un entendimiento mutuo entre ambos. Por ello es necesario personalizar algunos puntos del código concreto de las librerías del USB, así como también se hace necesario proporcionar un driver al PC para que reconozca el firmware del dispositivo bajo desarrollo. Para este último no será necesario desarrollarlo completamente, existe un *template* incluido en el propio compilador CCS que servirá para tener a punto un driver rápidamente en base a ciertas modificaciones ligeras de un archivo modelo. Primermante se mostrarán las modificaciones necesarias por parte de los archivos de código del firmware y finalmente se mencionarán los aspectos a personalizar en el driver proporcionado por CCS para el host.

En lo relativo al firmware, en el archivo *usb_cdc.h* se modificará la función de inicio del CDC (obsérvese que es la función que se llama en el *main* del programa) para que configure las características del puerto RS232 virtual creado con los parámetros acordes a nuestras necesidades. Como se puede ver, esta función inicializa una estructura que contiene los parámetros de una conexión serie estándar. Se adecuarán por tanto a nuestras necesidades concretas en lo relativo a los baudios, paridad, etc.

```
374  void usb_cdc_init(void) {
375     usb_cdc_line_coding.dwDTERrate=1200;
376     usb_cdc_line_coding.bCharFormat=0;
377     usb_cdc_line_coding.bParityType=0;
378     usb_cdc_line_coding.bDataBits=8;
379     (int8)usb_cdc_carrier=0;
380     usb_cdc_got_set_line_coding=FALSE;
381     usb_cdc_break=0;
382     usb_cdc_put_buffer_nextin=0;
383     usb_cdc_get_buffer_status.got=0;
384     usb_cdc_put_buffer_free=TRUE;
385  }
```

Por otra parte, se hace necesario también la modificación del archivo *usb_desc_cdc.h* para que el dispositivo sea reconocido por el driver. Las líneas concretas a modificar son las que se muestran a continuación. Existe una tabla que contiene la descripción del dispositivo con la finalidad de mostrar información en el momento en el que éste se conecta al PC. Modificando a conveniencia los datos de dicha tabla se hace posible la personalización de las cadenas de identificación que servirán para dar nombre al dispositivo en el ordenador, visible en el administrador de dispositivos.

La tabla USB_STRING_DESC contiene tres cadenas. Cada una de las cadenas comienza con un valor entero que contiene el tamaño de la misma (de la cadena en cuestión, incluyéndose el mismo valor como un elemento dentro del conteo). El segundo carácter de la cadena hace referencia al significado de la cadena, y se especifica haciendo uso de las macros definidas en las librerías. A continuación, el resto de caracteres de la cadena contiene la información propia de la misma.

La primera cadena hace referencia al tipo de codificación utilizado. No se tocará, dejándose el valor por defecto. La segunda y tercera cadenas contienen ambas la descripción del hardware. Nótese que tras cada carácter se deberá colocar otro carácter “0” siempre, por lo que cada carácter supondrá un espacio de dos unidades durante el conteo de caracteres de la cadena para especificarlo en el primer elemento de la misma.

Además de ello, deberá modificarse también la cadena USB_STRING_DESC_OFFSET que contendrá el *offset*, esto es, la posición de incio, de cada una de las cadenas de la tabla previamente nombrada.

```
205  //the offset of the starting location of each string.  offset[0
206  char USB_STRING_DESC_OFFSET[]={0,4,12};
207
208  char const USB_STRING_DESC[]={
209     //string 0
210           4, //length of string index
211           USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
212           0x09,0x04,   //Microsoft Defined for US-English
213     //string 1
214           8, //length of string index
215           USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
216           'P',0,
217           'F',0,
218           'C',0,
219     //string 2
220           40, //length of string index
221           USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
222           'E',0,
223           'C',0,
224           'L',0,
225           'I',0,
226           'P',0,
227           'S',0,
228           'E',0,
229           ' ',0,
230           'T',0,
231           'E',0,
232           'L',0,
233           'E',0,
234           'C',0,
235           'O',0,
236           'N',0,
237           'T',0,
238           'R',0,
239           'O',0,
240           'L',0
241  };
242
```

Obsérvese las líneas 185 y 186 del mismo archivo. Contienen el VID y el PID (Vendor ID y Product ID), que identifican el vendedor y el producto. Cuando se conecta el dispositivo al ordenador se produce un intercambio de información entre ambos. El host "pregunta" al dispositivo sobre su configuración, el cual devuelve entre otros datos estos dos valores los cuales son usados para localizar el driver asociado al mismo y cargarlo en el sistema. En caso de no existir el driver en el sistema, lo pide al usuario de forma que lo guardará para posteriores usos del dispositivo (solo es necesario proporcionar el driver la primera vez que se conecta el dispositivo al ordenador). El VID es un número idealmente único entre empresas que desarrollan hardware bajo USB de forma que cada una de ellas posee un identificador único. El PID identifica al identificador de producto, y cada empresa es libre de utilizar los valores que considere oportunos para ello. En este caso concreto se dejará el VID y PID proporcionados por Microchip. El VID 0x04D8 se corresponde con el de Microchip (obsérvese que se indica primero el byte de menor peso y posteriormente el de mayor peso).

```
169
170     ////////////////////////////////////////////////////////////////
171     ///
172     ///   start device descriptors
173     ///
174     ////////////////////////////////////////////////////////////////
175
176        const char USB_DEVICE_DESC[USB_DESC_DEVICE_LEN] ={
177           //starts of with device configuration. only one possibl
178              USB_DESC_DEVICE_LEN, //the length of this report   =
179              0x01, //the constant DEVICE (DEVICE 0x01)  ==1
180              0x10,0x01, //usb version in bcd  ==2,3
181              0x02, //class code. 0x02=Communication Device Class
182              0x00, //subclass code ==5
183              0x00, //protocol code ==6
184              USB_MAX_EP0_PACKET_LENGTH, //max packet size for end
185              0xD8,0x04, //vendor id (0x04D8 is Microchip)
186              0x00,0x00, //product id
187              0x00,0x01, //device release number  ==12,13
188              0x01, //index of string description of manufacturer.
189              0x02, //index of string descriptor of the product  =
190              0x00, //index of string descriptor of serial number
191              USB_NUM_CONFIGURATIONS  //number of possible configu
192        };
193
```

Una vez modificados los archivos *usb_cdc.h* y *usb_desc_cdc.h* pasará a escribirse el archivo del driver de la parte del PC. Dicho archivo posee la extensión *.inf*, y será necesario proporcionarlo al sistema la primera vez que se conecte el dispositivo bajo desarrollo. Se modificarán las siguientes líneas:

```
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%JCR%
LayoutFile=layout.inf

[Manufacturer]
%JCR%=PFC

[PFC]
%Telecontrol%=Reader, USB\VID_04D8&PID_0000

[Reader_Install.NTx86]
;Windows2000

[DestinationDirs]
DefaultDestDir=12
Reader.NT.Copy=12

[Reader.NT]
Include=mdmcpq.inf
CopyFiles=Reader.NT.Copy
```

El primer recuadro rojo indica el fabricante. El segundo recuadro es el más importante ya que en él se indica de nuevo el VID y el PID. El VID y el PID deberán ser idénticos tanto en los archivos ya nombrados en la parte del firmware, como en el archivo de extensión *.inf* de la parte del host.

```
ServiceType = 1 ; SERVICE_KERNEL_DRIVER
StartType = 3 ; SERVICE_DEMAND_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\usbser.sys
LoadOrderGroup = Base

[Strings]
JCR = "JCR"
Telecontrol = "Eclipse Telecontrol"
Serial.SvcDesc = "Programador de secuencias (Eclipse Telecontrol)"
```

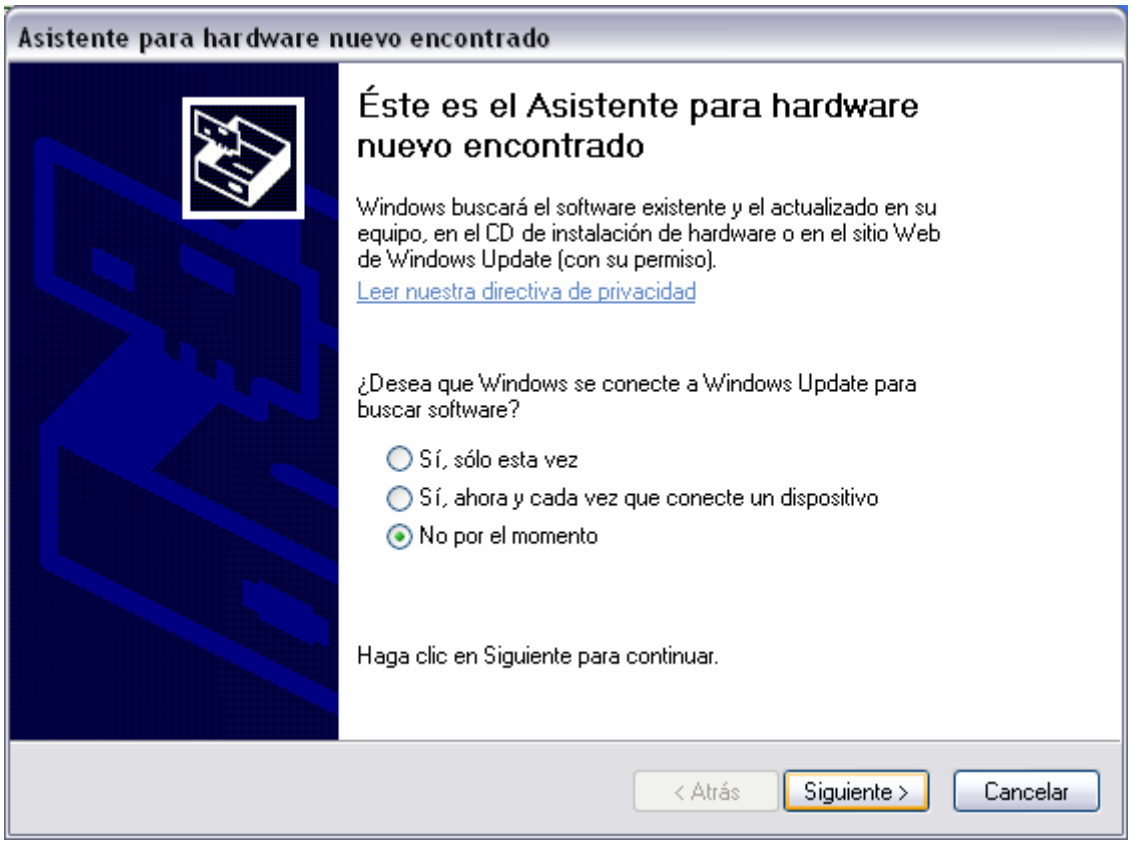
Al final del archivo se pueden modificar las cadenas de texto con el fin de personalizar los mensajes de identificación del dispositivo en el ordenador.

A continuación se mostrará el funcionamiento del dispositivo desarrollado. Se trata de un dispositivo muy sencillo pero funcional, que muestra la funcionalidad básica del bus USB en su modo CDC, y crea un puerto serie virtual a través de USB mediante el cual es posible llevar a cabo la conexión mediante *Hyperterminal* y probar el correcto funcionamiento haciendo uso de la función de "eco" implementada. Para ello conectaremos la alimentación de la placa de los relés (el receptor) con el firmware en este ejemplo descrito grabado en ella. Posteriormente se conectará al puerto USB haciendo uso del puerto habilitado para ello en la placa.

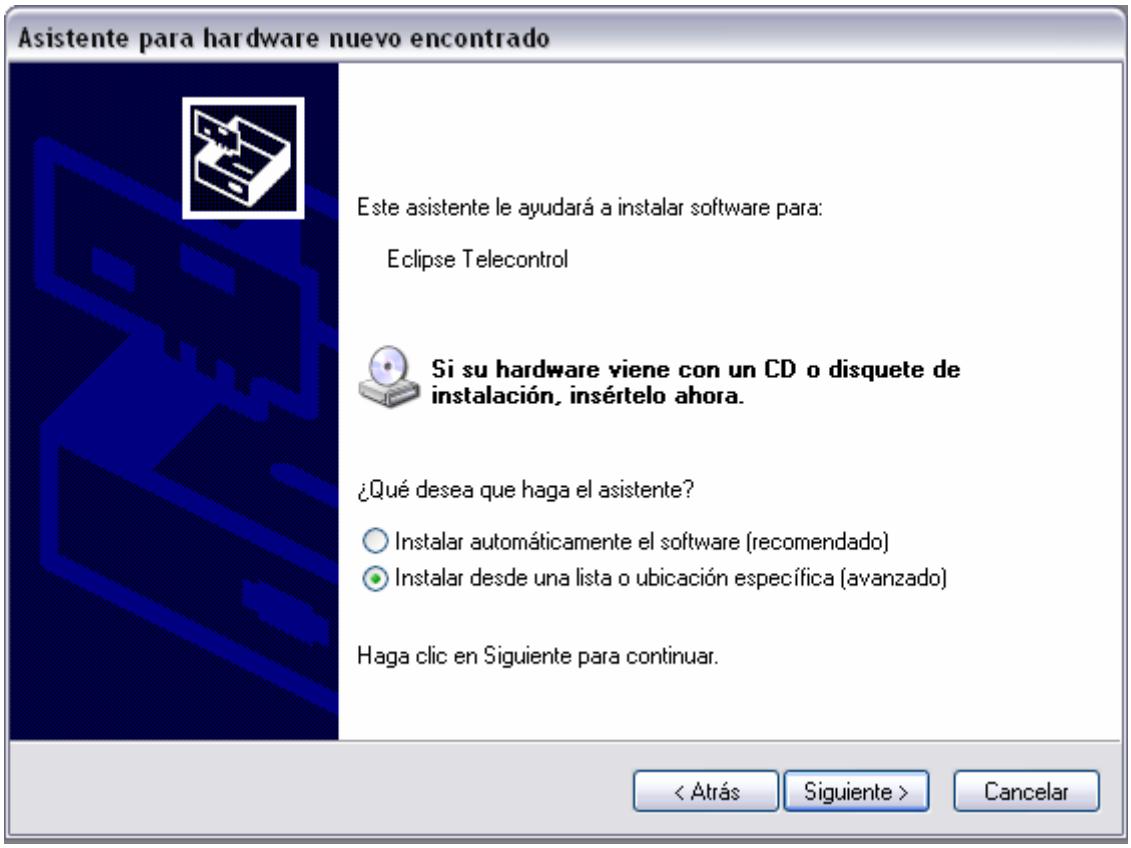
Se mostrará el siguiente mensaje en la parte inferior derecha de la pantalla:

Obsérvese que el nombre de identificación de dispositivo se corresponde con las cadenas especificadas en el archivo modificado *usb_desc_cdc.h*. En este momento, el host ha solicitado información al dispositivo y éste le ha respondido, entre otras cosas, con estas cadenas de identificación. Sin embargo a nivel práctico no tienen ninguna utilidad, para la identificación del dispositivo se utiliza, como se dijo anteriormente, la combinación VID-PID. Del mismo modo que se ha pasado la cadena "ECLIPSE TELECONTROL" entre muchos otros parámetros, también se ha recibido el VID-PID configurado en el firmware.

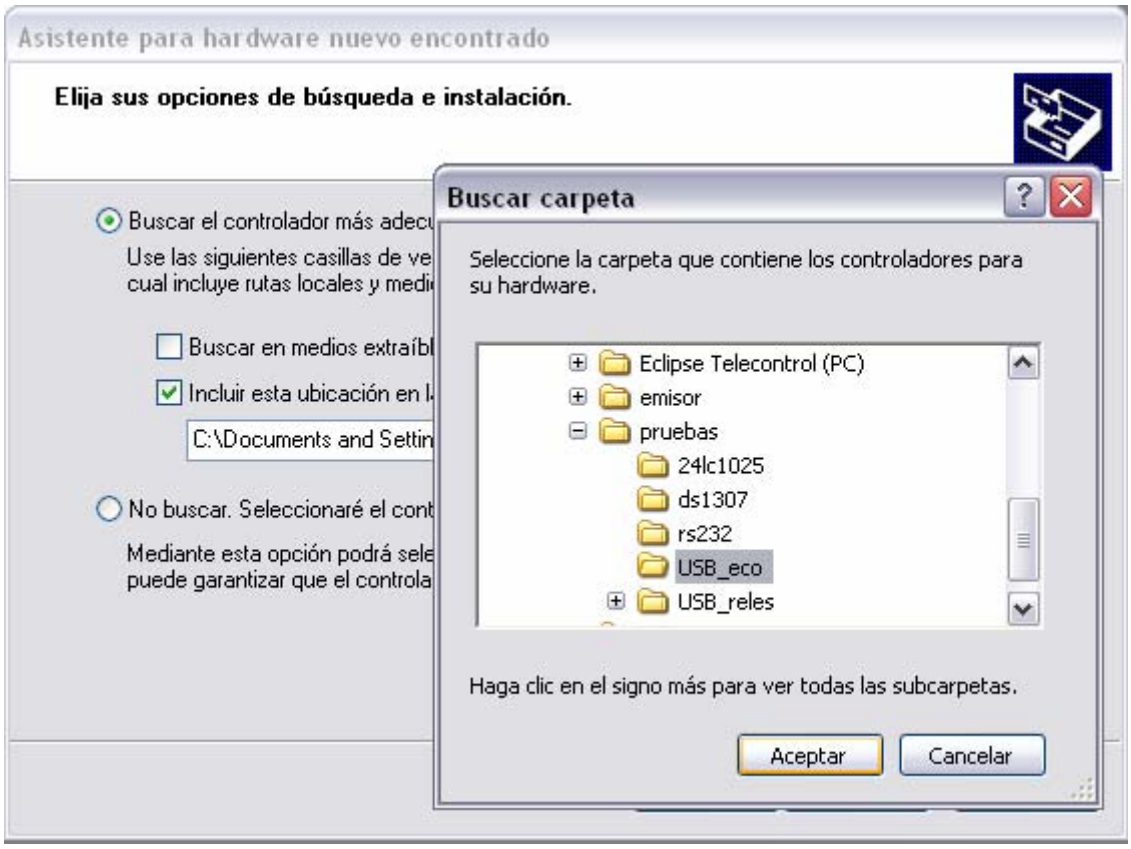
En este momento el PC buscará en sus archivos si posee los drivers asociados a dicho binomio VID-PID. Al ser la primera vez que se conecta el dispositivo no se dispondrá del driver con lo cual se mostrará el siguiente mensaje:



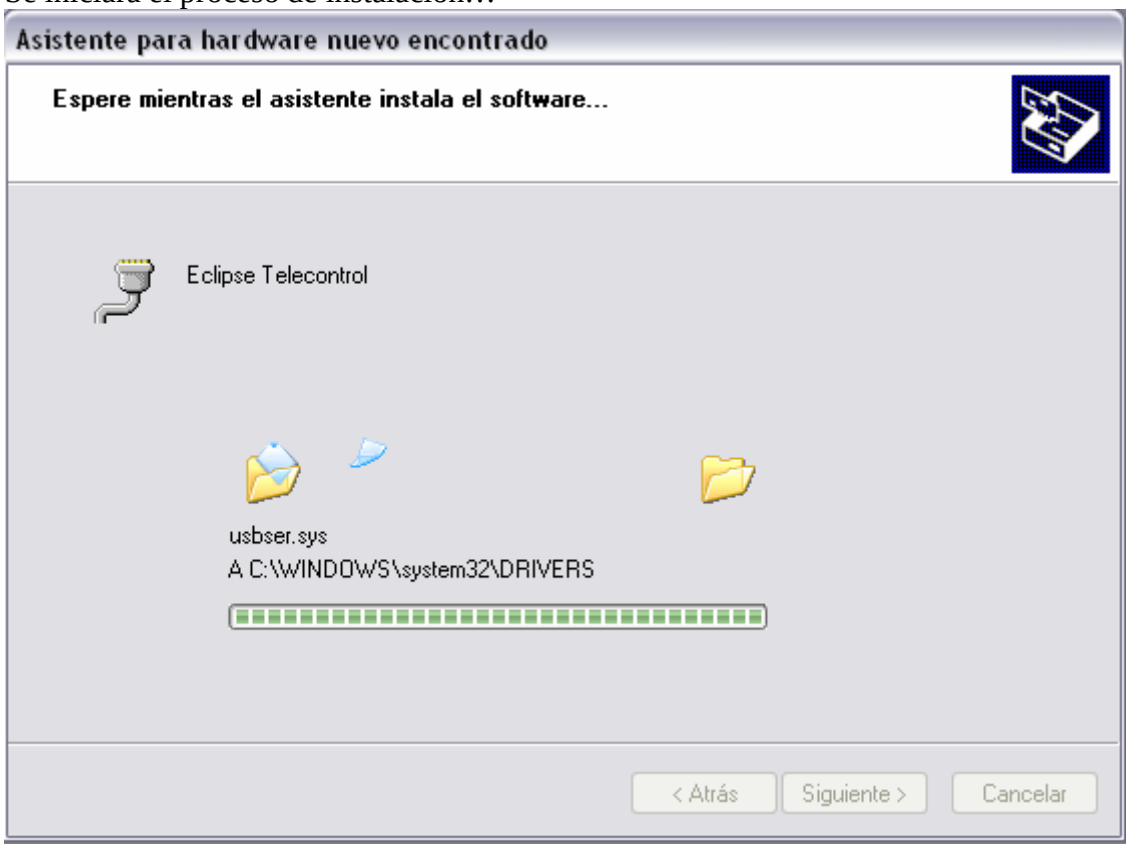
Haremos *click* en “No por el momento” y acto seguido en “Instalar desde una lista o ubicación específica”.



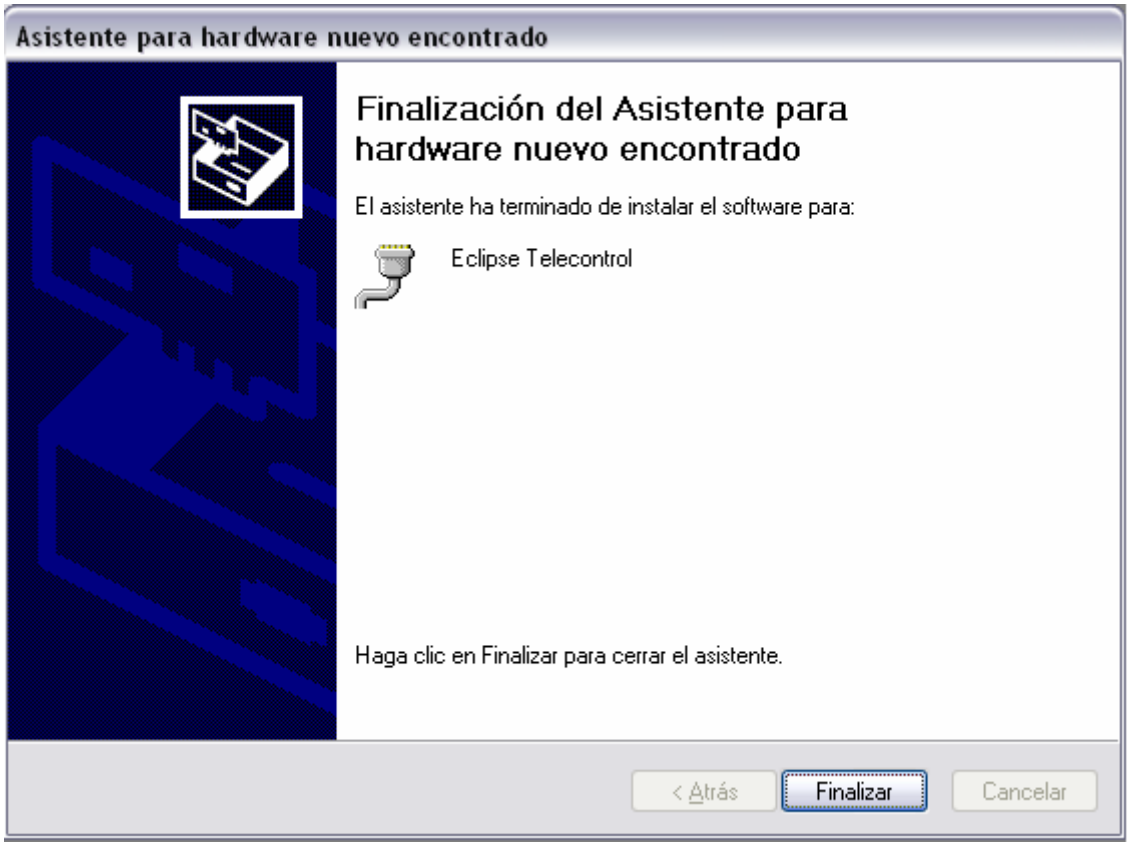
Se seleccionará el directorio que contiene el archivo *.inf* del driver y se hará *click* en "Aceptar".



Se iniciará el proceso de instalación…

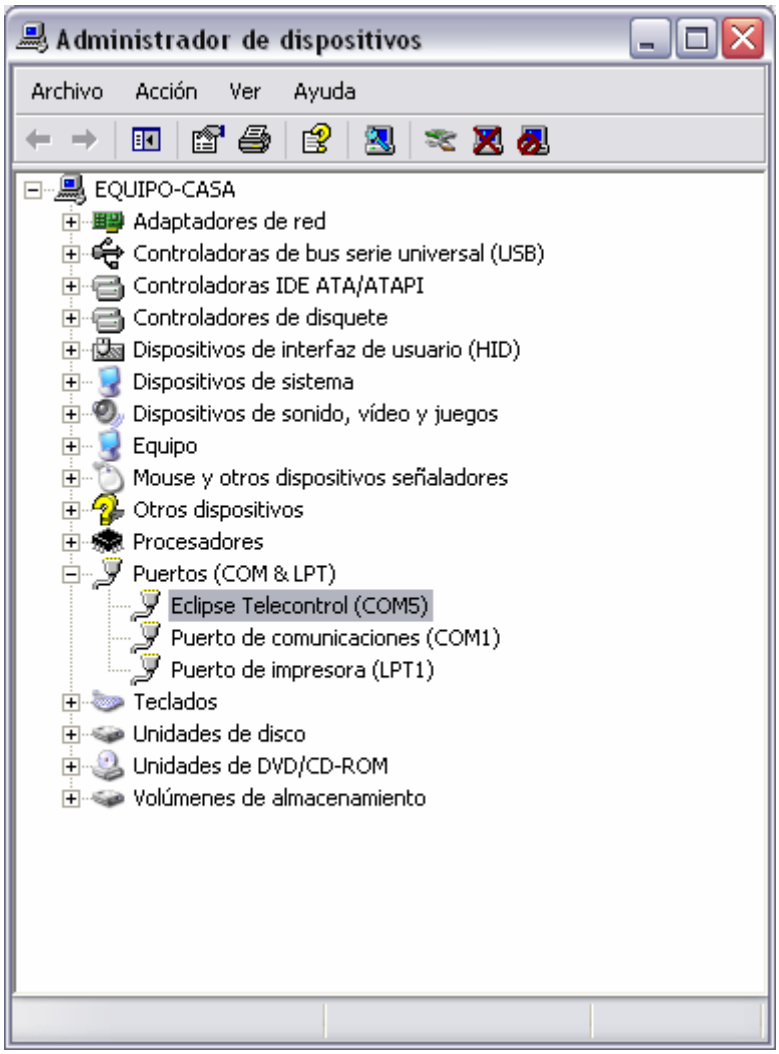


Y finalmente se mostrará el mensaje de fin del proceso.

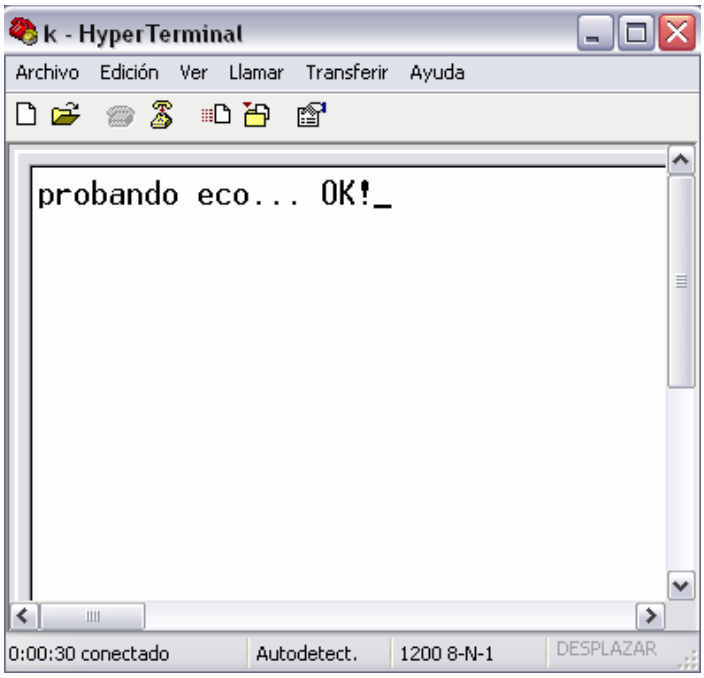


En este mismo momento el hardware se encontrará instalado y listo para funcionar. Este proceso solo será necesario llevarlo a cabo la primera vez que se conecte el dispositivo. De aquí en adelante y, al no ser que se desinstale intencionadamente el driver, cada vez que se conecte el dispositivo al sistema se cargará el driver de forma automática y transparente para el usuario.

Haciendo *click* en el botón derecho en "Mi PC" y posteriormente en "Administrador de dispositivos" se podrá ver como el dispositivo se encuentra instalado correctamente en el sistema como un puerto serie virtual:



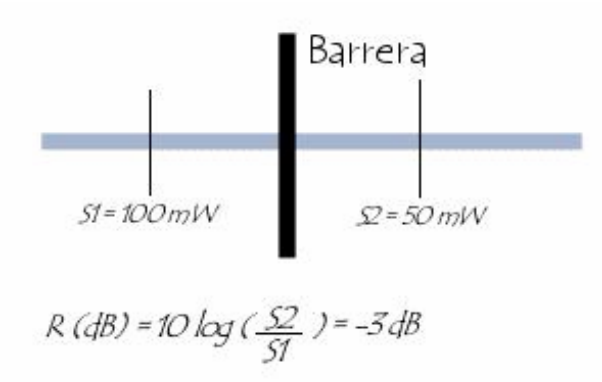
Una vez conectado el dispositivo e instalado el driver se procederá a llevar a cabo la prueba de funcionamiento del mismo. Para ello, se abrirá el *Hyperterminal* y se seleccionará el puerto serie virtual creado. En el administrador de dispositivos, como muestra la captura anterior, es posible observar el número asignado por el sistema. A la hora de configurar el *Hyperterminal* se tendrán en cuenta para ello los parámetros (baudios, paridad, bits de datos, etc.) especificados durante la modificación de la función *usb_cdc_init()* en el archivo *usb_cdc.h*. Una vez abierto el *Hyperterminal* será posible teclear sobre el, observando como efectivamente se produce el efecto de ECO de los caracteres teclados sobre el mismo. Quedará de este modo probado y demostrado el funcionamiento del bus USB sobre la placa.



4.3.- Tramas de datos.

La transmisión por radiofrecuencia tiene asociados ciertos problemas inherentes a la misma que hacen tener en cuenta una serie de consideraciones para asegurar la correcta transmisión.

Cuando una onda de radio se topa con un obstáculo, parte de la energía que posee es absorbida por el mismo y transformada en otro tipo de energía. El resto de la energía parte de ella puede ser reflejada y la otra parte traspasará el objeto y seguirá propagándose en su trayectoria. Sin embargo ello habrá conllevado la reducción de su potencia, es lo que se conoce como atenuación. La atenuación se da cuando la energía de una señal se reduce en el momento de la transmisión. La atenuación se mide en belios (Símbolo B) y equivale al logaritmo en base 10 de la intensidad de salida de la transmisión, dividida por la intensidad de entrada. Por lo general suelen usarse sin embargo los decibelios (símbolo dB) como unidad de medida, equivaliendo cada decibelio a un décimo de belio.



El problema de la atenuación se ve aumentado cuando sube la frecuencia de transmisión o la distancia entre los dos elementos del enlace radioeléctrico. Del mismo modo, se ve también afectado directamente por los obstáculos en el camino y en este caso el valor de la atenuación depende considerablemente del material del mismo. Los materiales metálicos tienden a reflejar la señal mientras que el agua la absorbe.

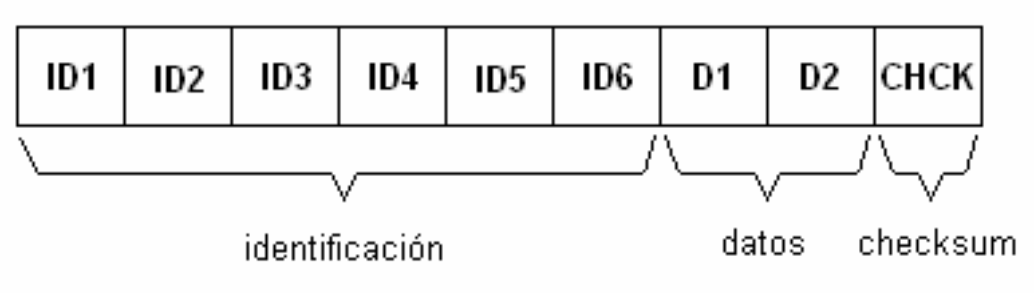
Por otra parte también está presente el problema de las interferencias. Las interferencias se producen cuando existen diversas fuentes (intencionadas o no) de emisión electromagnética que pueden perturbarse entre sí. La existencia de transformadores, aparatos electrónicos, motores u otros dispositivos eléctricamente ruidosos en las inmediaciones del enlace de radiofrecuencia pueden perturbar la información transmitida.

Un tercer problema de la transmisión electromagnética surge directamente de la ausencia de líneas eléctricas físicas para la conexión entre ambos puntos del enlace. El hecho de no haber una conexión punto a punto y transmitirse la información en modo *broadcast* hace posible a terceros dispositivos no implicados directamente en la comunicación la captura de los datos transmitidos. Del mismo modo, uno de los elementos implicados en la comunicación podría por error interpretar datos recibidos provenientes de una fuente diferente a la esperada.

Por estos motivos deben tomarse ciertas precauciones durante la transmisión de información a través de radiofrecuencia. Por una parte hay que proveer al sistema de un método de comunicación fiable de modo que el receptor sea capaz de reconocer que la información recibida proviene realmente del emisor esperado y descartar aquellas que no provengan de él. Del mismo modo el emisor debe asegurarse de que los datos transmitidos lleguen a su destinatario de forma íntegra e inalterada.

Se trata de una comunicación unidireccional sin conexión en la que el emisor no sabe si el receptor ha recibido correctamente la información ya que no recibe realimentación del proceso por parte del otro extremo. Por ello y en aras de subsanar los problemas descritos, la comunicación del sistema se basa en tramas.

Cada vez que se pulsa una tecla del mando emisor, éste compone una trama de datos compuesta por 9 bytes consecutivos y los envía al aire a través del módulo de radiofrecuencia. El emisor envía la información a través de dicho módulo, conectado a la *USART* del microcontrolador que lo gobierna. De forma análoga el receptor recibe la información también haciendo uso de la *USART* a través de las patillas c correspondientes. Para que el intercambio de datos se lleve a cabo de forma satisfactoria entre ambos éstos deben estar configurados para trabajar a la misma velocidad de transmisión, medida en baudios, o bits por segundo.

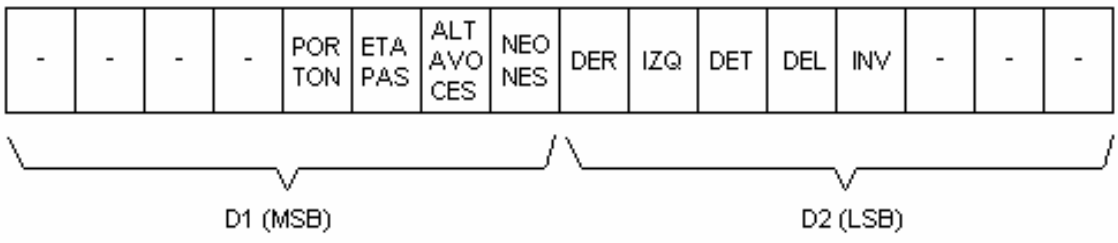


Dicha trama o paquete de datos suponen la unidad mínima de información transmitida entre el emisor y el receptor. Los seis primeros bytes de la trama son bytes de identificación. En ellos queda codificada la dirección del receptor, el cual solo tendrá en cuenta aquellas tramas en las cuales estos seis primeros bytes recibidos se correspondan con los bytes de identificación que tiene programados. De este modo se descartarán aquellas tramas de datos que no vallan destinadas al receptor en cuestión.

Los bytes 7 y 8 contienen la información de las pulsaciones del mando propiamente dichas. En estos 16 bits se encuentra codificado el estado de todos los pulsadores del mando en el momento de la transmisión de la trama.

Finalmente, el último byte contiene el *checksum* o suma de los valores de los dos bytes de datos. De este modo permite la detección de errores durante la transmisión, cuando el receptor recibe la trama, si ha pasado los bytes de identificación, tras guardar los datos recibidos en dos bytes consecutivos (bytes 7 y 8) comprueba que el siguiente y último byte sea igual al checksum calculado localmente por el microcontrolador de los dos bytes de datos. En caso de ser iguales, la trama se da por válida y se procede en consecuencia. En caso de no coincidir el checksum se descarta la trama y no se tiene en cuenta, entendiéndose que ha habido corrupción de datos durante la transmisión de la misma.

Bytes de datos:



Como se ha mencionado los bytes 7 y 8 de la transmisión contienen codificado el estado de todos los pulsadores del mando emisor en el momento de emisión de la trama. Cada vez que se pulsa un pulsador en el mando éste envía una serie de tramas (la misma trama repetida varias veces, por si hay pérdida de datos) que contienen el estado de los interruptores.

Byte D1:

PORTON (bit 3): Si INV=0, indica que el portón debe subir. Si INV=1, debe bajar.
ETAPAS (bit 2): Si INV=0, las etapas salen. Si INV=1, entran.
ALTAVOCES (bit 1): Si INV=0, la plataforma de altavoces sube. Si INV=1, baja.
NEONES (bit 0): Si INV=0, los neones se encienden. Si INV=1 se apagan.

Byte D2:

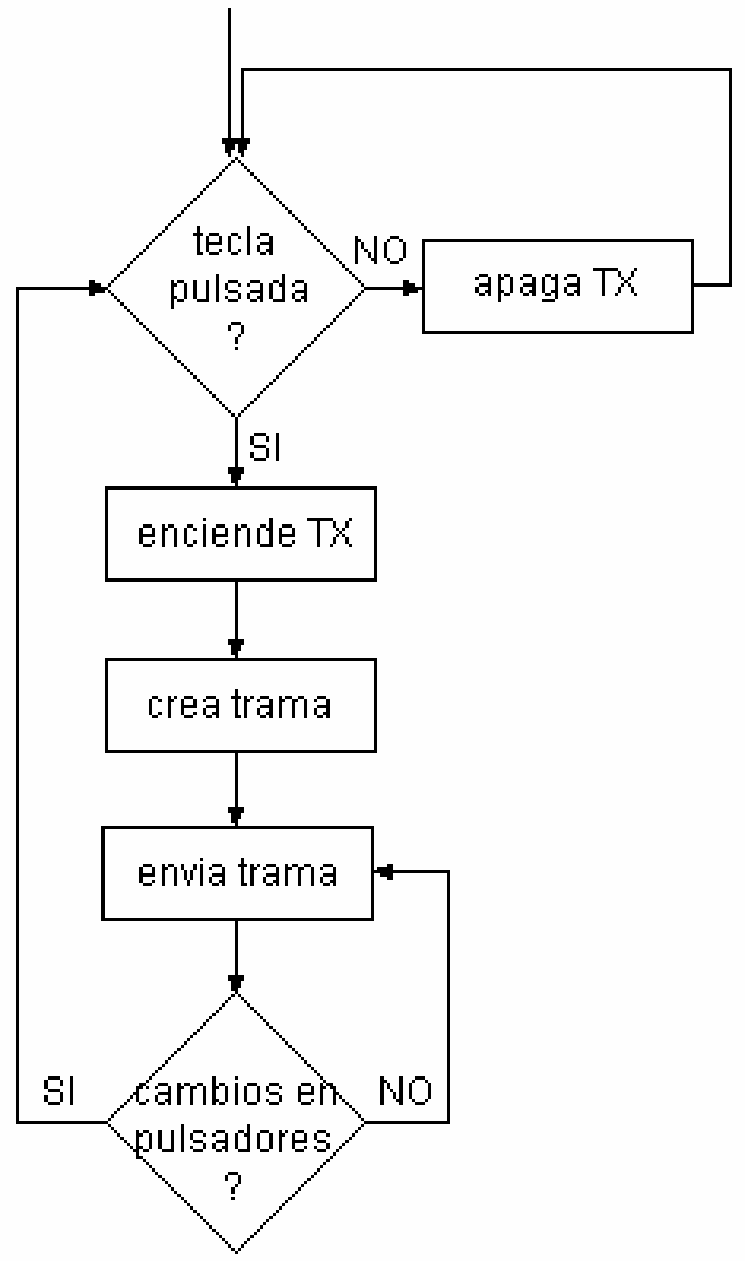
DER (bit 8): Si INV=0 el coche se levanta de la derecha. Si INV=1, parte derecha baja.
IZQ (bit 7): Si INV=0 el coche se eleva de su parte izquierda. Si INV=1, desciende.
DET (bit 6): Si INV=0 el coche se levanta de atrás. Si INV=1, baja.
DEL (bit 5): Si INV=0 el coche se eleva de delante. Si INV=1, desciende.
INV (bit 4): Bit de inversión. Controla el sentido de actuación de los elementos cuyo botón está pulsado.

4.4.- Firmware

Se procederá a continuación a tratar todo lo relativo al firmware de ambos dispositivos que componen el proyecto. Primeramente se analizará el firmware del microcontrolador PIC 16F628, que da vida al emisor, y posteriormente el que será grabado en el PIC 18F2550 sobre el cual se delega el funcionamiento del receptor.

4.4.1.- Emisor

El diagrama de flujo de la función principal del firmware del emisor queda representado por la siguiente figura.



Una vez encendido el emisor entra en un bucle en el cual se llevan a cabo una serie de tareas programadas en un orden preestablecido bien definido. Primeramente se comprueba que si hay alguna tecla pulsada. En caso negativo, se apaga el módulo de radiofrecuencia y se comprueba de nuevo dicha condición hasta que ésta sea cierta. Puede pensarse en un primer momento que es absurdo el hecho de apagar el transmisor cada vez que se comprueba que no existe ninguna tecla pulsada y por lo tanto no hay tramas que enviar, sin embargo la inclusión de la instrucción de apagado dentro de esta parte del bucle tiene sentido cuando se viene de un estado anterior en el cual se deja de pulsar un botón y se encuentra el emisor encendido, en cuyo caso la primera pasada por el bucle apagaría dejaría de alimentar el transmisor.

En el caso de que haya alguna tecla pulsada se enciende el módulo de radiofrecuencia, se crea la trama pertinente con el estado de los pulsadores y se envía.

Posteriormente se leen de nuevo los pulsadores y se comprueba si ha habido cambios. En caso de que no los haya habido significa que se está manteniendo el pulsador, con lo cual se vuelve a enviar la trama previamente creada hasta que deje de pulsarse el botón o se cambie el estado de otros, en cuyo caso la condición de cambio de pulsadores será positiva y volverá al estado inicial en el cual se comprueba si hay algún pulsador activado.

- Caso 1: Ningún pulsador se encuentra activado. El programa entra en el bucle derivado del primer condicional, manteniendo el módulo TX en estado apagado.

- Caso 2: Se pulsa un botón y acto seguido se suelta. Se enciende el emisor, crea trama, envía y sigue enviando hasta que éste deje de pulsarse. Cuando ello suceda, el programa volverá al punto inicial de comprobación del que no saldrá hasta que se pulse otra tecla.

- Caso 3: se pulsa un botón y después se cambia el estado de otros pulsadores manteniendo pulsado el otro. En este caso inicialmente el flujo del programa evoluciona igual que en el caso anterior, reenviando la misma trama en el último bucle hasta que hayan cambios en los pulsadores. Cuando ello suceda, el primer condicional dará positivo con lo cual sin apagar el transmisor, se volverá a crear una trama nueva para posteriormente volver a entrar en el bucle de reenvío de la misma hasta que hayan novedades en el estado de los pulsadores. Se analizará el código fuente del firmware en las siguientes páginas.

```
1  #include <16F628.h>
2
3  #fuses XT,NOWDT,NOPROTECT,NOLVP,BROWNOUT
4  #use delay(clock=4000000)
5  #use rs232(baud=9600,XMIT=PIN_B2,RCV=PIN_B7,BITS=8,PARITY=N)
```

La primera línea del programa muestra la inclusión del archivo de cabecera del microcontrolador bajo uso así como la especificación de la configuración del mismo.

Se especifica también la frecuencia de funcionamiento del oscilador de cuarzo, necesaria para poder hacer uso de la USART del PIC cuya configuración se detalla en la siguiente línea. Se inicia la USART a una velocidad de 9600 baudios, indicando los pines de entrada y salida (aunque el de entrada no se utilizará), el número de bits de la palabra de transmisión, y se desactiva la comprobación de paridad.

```
8   // Identificador del dispositivo a controlar.
9   #define ID1 'P'
10  #define ID2 'F'
11  #define ID3 'C'
12  #define ID4 'J'
13  #define ID5 'C'
14  #define ID6 'R'
15  #define LONG_TRAMA  9
16
17  // Línea de habilitación de transmisor.
18  #define TX433N_CS   PIN_A4
```

De las líneas 8 a 18 se encuentran las definiciones del programa. Ello permite parametrizar ciertos aspectos del código mejorando la legibilidad, mantenimiento y posibilidad de cambios posteriores. Se define un alias para cada byte de identificación así como otro para la longitud de la trama y el pin dedicado al apagado y encendido del módulo de radiofrecuencia.

```
22  void TX433N_on();
23  void TX433N_off();
24  void TX433N_envia_trama(byte *trama, int longitud);
25  void crea_trama(byte *trama, long int pulsadores);
26  long int lee_interruptores();
```

Las líneas 22 a 26 definen todas las funciones contenidas en el firmware, cuya implementación se mostrará posteriormente.

A continuación se analizará el *main* o función principal del programa, correspondiente a la implementación del diagrama de flujo analizado al inicio de este apartado:

```
void main()
    {
    // Define el esqueleto de la trama. Los 6 primeros bytes
    // (identificación) permanecen invariantes. Los tres últimos
    // cambiarán según el estado de los pulsadores.
    byte trama[LONG_TRAMA] = {ID1,   // byte de identificación 1.
                              ID2,   // byte de identificación 2.
                              ID3,   // byte de identificación 3.
                              ID4,   // byte de identificación 4.
                              ID5,   // byte de identificación 5.
                              ID6,   // byte de identificación 6.
                              0x00,  // byte de datos D1.
                              0x00,  // byte de datos D2.
                              0x00}; // checksum (D1+D2).
    long int interruptores_ant;
    long int interruptores;
    // Configuración de puertos y desactivación del módulo RF.
    set_tris_a(0xDF);
    set_tris_b(0xEC);
    output_low(PIN_B0);

    interruptores = lee_interruptores();
    interruptores_ant = interruptores;

    while(TRUE)
        {
        // Si hay algún interruptor pulsado
        if(interruptores != 0)
            {
            TX433N_on();
            crea_trama(trama, interruptores);
            // Mientras el estado de los pulsadores no cambie, reenvía
            // continuamente la misma trama.
            do
                {
                TX433N_envia_trama(trama, LONG_TRAMA);
                interruptores_ant = interruptores;
                interruptores = lee_interruptores();
                }
            while(interruptores_ant == interruptores);
            }
        else
            {
            // Si no se ha pulsado, el transmisor permanece apagado
            // y volvemos a leer los interruptores.
            TX433N_off();
            interruptores = lee_interruptores();
            }
        }
    }
```

En las líneas 35 a 43 se crea el array de bytes destinado a servir de esqueleto durante la construcción de nuevas tramas. Los primeros 6 bytes contienen los bytes de identificación o direccionamiento del receptor, cuya finalidad ya ha sido indicada en apartados anteriores. Los tres bytes posteriores se corresponden con el primer byte de datos (maletero), segundo byte de datos (ruedas) y el checksum de ambos.

En las líneas 44 y 45 se declaran dos variables que contendrán el estado de los interruptores en el momento actual y anterior con la finalidad de poder llevar a cabo comparaciones entre ellas y determinar cuando se han producido cambios de estado en las pulsaciones.

El código comprendido entre las líneas 54 a 79 supone el bucle principal del programa.

Primeramente se comprueba si hay algún interruptor pulsado (línea 57). Posteriormente y en caso afirmativo, se enciende el transmisor y se crea una trama partiendo del esqueleto base previamente definido y el estado actual de los pulsadores recién leído de los puertos del microcontrolador.

El bucle do-while comprendido entre las líneas 59 y 63 asegura que la trama se reenvíe de forma continua mientras el estado de los pulsadores se mantenga.

Tras ello, el código comprendido entre las líneas 71 y 77 de la función main se ejecuta si no hay ningún pulsador activado. En el caso de que esto suceda, se desactiva el módulo emisor y se procede a una nueva lectura de los interruptores con el fin de poder seguir con el bucle de comprobaciones.

Función lee_interruptores():

```
87  // Lee el estado de los interruptores directamente a través de los
88  // puertos de entrada.
89  long int lee_interruptores()
90      {
91      long int interruptores;
92      byte *acceso_byte;
93      acceso_byte = &interruptores;
94
95      // LSB = ruedas. Lectura de PORTB.
96      *acceso_byte = input_b() & 0xF9;        // LSB = ruedas
97
98      // MSB = maletero. (la llamada input_a() resetea RA4...)
99      acceso_byte++;
100     *acceso_byte=0;
101     if(input(PIN_A0) != 0) *acceso_byte = *acceso_byte | 0x1;
102     if(input(PIN_A1) != 0) *acceso_byte = *acceso_byte | 0x2;
103     if(input(PIN_A2) != 0) *acceso_byte = *acceso_byte | 0x4;
104     if(input(PIN_A3) != 0) *acceso_byte = *acceso_byte | 0x8;
105
106     return interruptores;
107     }
108
```

Esta función obtiene el estado actual de los interruptores conectados al puerto A y puerto B del microcontrolador y coloca el resultado de la lectura en la variable global "interruptores".

Dicha variable global es de tipo "long int", esto es, de tipo entero de dos bytes de tamaño. El byte de menos peso (LSB, Less Significant Byte) contiene el estado de los pulsadores relativos a la suspensión neumática de las ruedas y el byte más significativo contiene el estado de los pulsadores destinados a controlar el maletero.

Para poder acceder a ambos bytes por separado se crea una variable de tipo puntero a byte llamada "acceso_byte" a la cual primeramente se le asigna la dirección de la variable "interruptores", haciendo posible de este modo la escritura del byte de menos peso de la misma con el estado del puerto B. Se utiliza la operación lógica AND (&) con la máscara 0xF9 para aislar los bits a tener en cuenta de aquellos cuyo valor no tiene ningún significado y por lo tanto no queremos obtener.

Posteriormente se incrementa la variable "acceso_byte" y se accede al byte de mayor peso de la variable "interruptores". En ella se guarda el estado de los interruptores asociados al maletero, se ha utilizado la consulta de estado bit a bit de forma independiente en lugar de proceder de manera análoga a como se ha llevado a cabo con el puerto B debido a que durante las pruebas se observó que la llamada a la función de consulta de estado del puerto A, el prototipo de la cual es "input_a()", resetea el valor de todos los bits que se encuentran a 1 en dicho puerto. Esto puede ser debido a un modo de funcionamiento interno diferente de este tipo de puerto (el cual posee una naturaleza y una electrónica interna distinta ya que permite ser configurado como analógico o digital), o a un *bug* del compilador CCS.

Una vez leídos los valores de ambos bytes, se devuelve el valor de 16 bits obtenido.

Función TX433_on():

```
110     // En caso de que esté apagado, enciende el emisor y
111     // espera un tiempo de estabilización de 50 milisegundos.
112     void TX433N_on()
113         {
114         if(input(TX433N_CS) != 1)
115             {
116             output_high(TX433N_CS);
117             delay_ms(50);
118             }
119         }
```

Enciende el emisor. En caso de que esté encendido no hace nada y en caso de que esté apagado lo enciende, esperando tras ello 50 ms con el fin de permitir la estabilización de tensiones en el mismo antes de comenzar a enviar las tramas de datos.

Función TX433N_off():

```
122     // Apaga el transmisor.
123     void TX433N_off()
124         {
125         output_low(TX433N_CS);
126         }
```

Envía la trama compuesta por el array de bytes cuya referencia es pasada como primer parámetro, de longitud “longitud”, pasado como segundo parámetro de tipo entero.

Esta función está constituida por un bucle for, que recorre todo el array de datos desde principio a fin enviando por la USART del microcontrolador cada uno de ellos de forma sucesiva uno a uno.

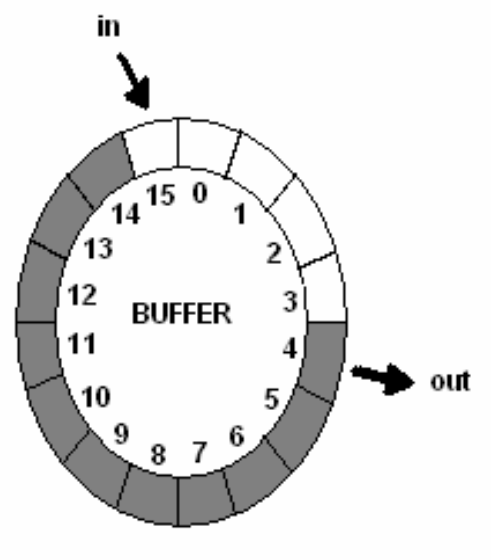
4.4.2.- Receptor

Antes de comenzar a analizar el firmware del receptor se explicará conceptualmente el funcionamiento del mismo. El código del aparato receptor es mucho más complejo que el del emisor, y en él entran en juego las interrupciones, los buses i2c, las temporizaciones y el protocolo USB, también presente en el proyecto y mediante el cual es posible la carga dinámica de secuencias preprogramadas.

Primeramente es necesario disponer de un mecanismo de recepción y almacenaje. Los datos llegan a través del receptor de radiofrecuencia conectado a la patilla de entrada de la USART. Cada vez que se recibe un byte se genera una interrupción de atención que se encarga de manejar el evento y actuar en consecuencia. El tratamiento del dato en la propia función de interrupción es posible, sin embargo las interrupciones deben ser funciones de código rápidas, ligeras y que supongan una carga computacional ligera. Téngase en cuenta que la ejecución de una interrupción supone romper el flujo natural del programa para pasar a ejecutar el código del manejador asociado, hasta la finalización del cual no se retomará el flujo de ejecución interrumpido. Del mismo modo, durante la ejecución del manejador de interrupción puede llegar una nueva interrupción. En el mundo asíncrono e impredecible de las interrupciones deben tenerse en cuenta todas las posibilidades dentro del abanico de opciones posibles durante la ejecución del programa.

Todos estos impedimentos llevan a concluir que debe utilizarse un método de almacenaje temporal de la información, de forma que se delega en una función externa el tratamiento de dicha información recibida, y la interrupción únicamente coloca en dicho almacén los datos recibidos de forma consecutiva según los va recibiendo. Si se incorpora la rutina de tratamiento de datos dentro de la propia interrupción se bloqueará la USART. Hágase la prueba si así se desea. El motivo de ello es que mientras está tratando un carácter recibido recibirá el siguiente, saltará de nuevo la interrupción y se saturarán los registros de recepción (únicamente tiene 2 registros de un byte cada uno).

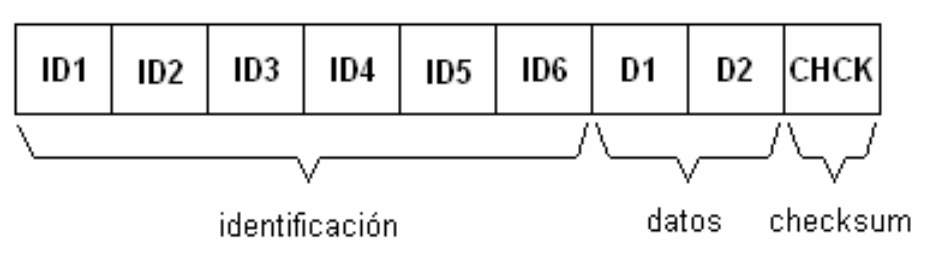
Cuando se saturan los registros, la USART se bloquea completamente. Para evitar eso se hace uso de un buffer circular, o buffer en anillo.



El funcionamiento es conceptualmente sencillo. Se dispone de una zona de memoria dedicada, el buffer propiamente dicho, destinada a almacenar información. Existen dos punteros que apuntan respectivamente hacia dos posiciones independientes dentro de este buffer. Uno de ellos está destinado al almacenaje de datos, e indica la posición del siguiente hueco libre en el buffer. El puntero de salida, sin embargo, apunta hacia la dirección del primer elemento extraíble del mismo. Cuando se llega a la última dirección del buffer, se vuelve a comenzar desde la primera. De ahí viene el nombre "buffer circular" o "buffer en anillo". La naturaleza de este tipo de buffer ofrece una metodología de almacenaje FIFO (First In, First Out), en la cual los elementos se irán extrayendo por orden de llegada. Una variable dedicada guarda el número de elementos actuales en el buffer. Si el buffer está vacío no se efectúa ninguna operación de lectura (o se devuelve un valor nulo, o no válido, depende de la implementación concreta) y si el buffer está lleno se descarta el dato y no se almacena o se devuelve un mensaje de error (nuevamente, depende de la implementación en cuestión).

El uso de un buffer intermedio de almacenaje de datos resuelve el problema de recepción y tratamiento de la información, liberando al manejador de interrupción de la elevada carga computacional que ello supone y permitiendo almacenar los datos recibidos para que la función de tratamiento opere con ellos durante el flujo natural de ejecución del programa.

En el buffer circular se dispondrá por tanto de los datos recibidos por la USART, es decir por el módulo receptor de radiofrecuencia, en estricto orden de llegada. Existirá, por tanto, una función del programa cuyo objetivo será analizar la secuencia de datos recibidos y reconocer en ella tramas provenientes del receptor. Recuérdese el formato de la trama, ya tratado con anterioridad:

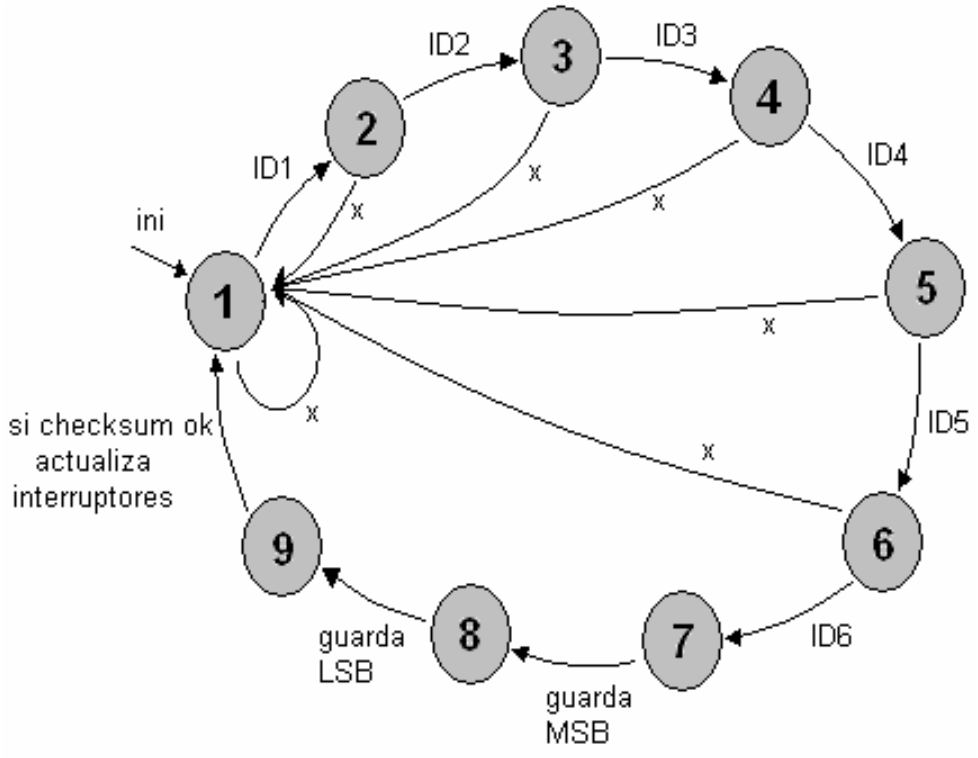


Para el reconocimiento de datos se ha llevado a cabo la implementación de una máquina de estados finitos. Primeramente se describirá que es una máquina de estados, para posteriormente y una vez comprendido, entender como se aplica la máquina de estados para el reconocimiento de tramas de datos preestablecidas.

Una máquina de estados finitos (FSM = Finite State Machine) se basa en la idea de que hay un número de estados acotado y bien definido dentro de un sistema informático. Reduciéndose al ejemplo más elemental, cuando una aplicación apaga o enciende un LED existen dos estados, un estado es cuando dicho LED está apagado y el otro es cuando está encendido. Las máquinas de estado requieren una variable global de estado, la cual contiene el estado actual en el cual se halla el sistema. La variable de estado puede modificarse dentro de las propias funciones de la máquina de estados o desde funciones externas. Dependiendo del estado, conocido en todo momento, se dirige el flujo del programa hacia unos módulos de ejecución u otros dependiendo de las condiciones concretas de las que ello dependa.

Las máquinas de estados tienen una característica muy importante: siempre se encuentran en un estado conocido. Esto es, no existe indeterminación, en caso de incertidumbre o según la situación lo requiera siempre se vuelve a un estado inicial conocido.

El primer paso a la hora de implementar un reconocimiento de tramas de datos en base a una máquina de estados es identificar el conjunto de estados que la definen. En este caso concreto se pueden identificar varios estados. El primer de ellos será el estado inicial. Siempre habrá un estado inicial, punto de partida para los demás estados. El segundo estado será aquel en el que se ha leido correctamente el primer byte de la secuencia y éste coincide con el primer byte de identificación de trama. El siguiente estado será aquel en el que se han leido consecutivamente dos bytes correctos de identificación. Así sucesivamente. En el caso de que se rompa el flujo de lecturas correctas, se vuelve al estado inicial. El siguiente gráfico muestra los estados que se pueden identificar en el sistema de reconocimiento, así como las transiciones entre ellos:



Los círculos representan los estados. Las flechas las transiciones posibles entre ellos, y la información que acompaña a cada flecha indica la condición para que dicha transición suceada.

Estado 1: Estado inicial. Estado por defecto punto de partida para los demás estados.

Estado 2: Se llega tras leer el primer byte y cerciorarse de que éste coincide con el primer byte de identificación del dispositivo. Si estando en el estado dos se recibe otro byte y este coincide con el segundo byte de identificación del dispositivo, se pasa al estado 3. En caso contrario, se vuelve al estado 1.

Estado 3: De manera análoga al estado dos, se han leido dos bytes consecutivos.

Estado 4: Se han leído 3 bytes de identificación correctos.
Estado 5: Se han leído 4 bytes de identificación correctos.

Estado 6: Se han leído 5 bytes de identificación correctos.

Estado 7: Se han leido 6 bytes y los 6 coinciden. Por lo tanto el dispositivo está diseccionado por el emisor. El siguiente byte recibido se guardará como primer byte de datos y se pasará al estado siguiente.

Estado 8: Se está esperando el segundo byte de datos. Cuando se reciba se pasará al estado 9.

Estado 9: Esperando el último byte de la trama, correspondiente al checksum. Cuando se reciba se hará la suma y se comprobará que no hay errores. En caso de haber errores se descartará la información recibida y en caso de resultar satisfactoria la comprobación se actualizarán las variables globales del sistema pertinentes, reflejando la información recibida. En cualquiera de los casos, luego se pasará al estado 1.

La máquina de estados mantiene en todo momento la información recibida en una variable global, que contiene el estado de los interruptores del emisor en todo momento. La máquina de estados actualiza dicha variable en función de los datos recibidos, cada vez que reconoce una trama enviada por el emisor. Esta variable global, que contiene el estado de los pulsadores, está disponible para que otras funciones del programa operen a conveniencia.

Existirá pues, otra función que cada vez que es llamada actualiza el estado de los relés. Dicha función se encuentra dentro del bucle principal. El bucle principal también comprueba cíclicamente el estado del conector USB. Si se detecta conexión, se rompe la ejecución del bucle y se entra en modo programación. Todo ello queda delegado en otra función encargada de ello.

A continuación pasa a describirse el código fuente del receptor, mediante lo cual se comprenderán ciertos aspectos y se hará mucho más comprensible los conceptos explicados aquí introducidos.

Se han extraido las instrucciones al preprocesador, #defines, configuración de pines de dispositivos, etc. en un archivo auxiliar llamado *config.h*. Se analizará previamente este archivo.

```
1   #ifndef CONFIG_H
2   #define CONFIG_H
3
4
5   /*
6   |
7   | Archivo de configuración. Reune todas las variables que
8   | conforman la configuración del firmware del dispositivo.
9   |
10  */
11
12
13  // Configuración del microcontrolador
14  #fuses HSPLL,NOWDT,NOPROTECT,NOLVP,USBDIV,PLL5,CPUDIV1,VREGEN
15
16
17  // Frecuencia del oscilador de cuarzo
18  #use delay(clock=20000000)
19
```

Configuración de los fuses del microcontrolador. Debido a que se trata de un cristal de cuarzo de 20 MHz hay que colocar la cadena “HSPLL”, que junto con la cadena “PLL5” configuran el dispositivo para obtener los 48 MHz internos necesarios para hacer uso del USB. La sentencia #use especifica la velocidad del cristal.

```
20      #define RX_BUFFER_SIZE  200
21
22      #define USE_INTERRUPTS
23
24
25      // Identificador del receptor
26      #define ID1 'P'
27      #define ID2 'F'
28      #define ID3 'C'
29      #define ID4 'J'
30      #define ID5 'C'
31      #define ID6 'R'
32
```

RX_BUFFER_SIZE define la capacidad en bytes del buffer circular previamente descrito. ID1…ID6 especifican los bytes de identificación del receptor. Obviamente para que el sistema funcione deberán ser los mismos que se especificaron en el mando emisor.

```
34      // Tiempo (ms) que estará activa la sirena antes de subir y bajar
35      // el portón del maletero (> 1s)
36      #define MSEGS_SIRENA        2000
37
38
39      // Pinout receptor RF RWS433N
40      #define RWS433N_PINOUT
41      #define RS232_TX            PIN_C6
42      #define RS232_RX            PIN_C7
43
44      #use rs232(baud=9600,XMIT=RS232_TX,RCV=RS232_RX,BITS=8,PARITY=N,ERRORS)
45
46
47      // Pinout rtc DS1307 y memoria 24LC1025
48      #define DS1307_PINOUT
49      #define I2C_SDA             PIN_B0
50      #define I2C_SCL             PIN_B1
51
52      #define MEM_24LC1025_PINOUT
53      #define MEM_24LC1025_WP     PIN_B2
54
55      #use i2c(sda=I2C_SDA,scl=I2C_SCL,slow)
```

MSEGS_SIRENA define el tiempo (en milisegundos) que deberá estar la sierena incorporada en el vehículo sonando antes de proceder a abrir o cerrar el portón del maletero. Esto tiene como finalidad avisar de forma acústica a los espectadores o

personas cercanas que se encuentren en las inmediaciones y evitar accidentes con las partes móviles del vehículo.

Desde las líneas 40 a 42 se define el pinout del módulo receptor de radiofrecuencia. Las líneas 48 a 50 definen la conexión del timer RTC DS1307, y las líneas 52 y 53 lo relativo a la memoria EEPROM (las líneas i2c son las mismas que las del timer).

Se configura la USART del pic a 9600 badios, con un tamaño de palabra de 8 bits y reseteando automáticamente la USART en caso de saturación. Ello se consigue especificando la cadena “ERRORS”.

Se configura el bus i2c, que conecta la memoria y el timer con el PIC, según las patillas especificadas anteriormente y se inicializa a 100 KHz (parámetro “show”).

```
58      // Pinout etapa de relés
59      #define RELES_PINOUT
60      #define RELE_DEL_IZQ_SUBE   PIN_B7
61      #define RELE_DEL_IZQ_BAJA   PIN_B6
62      #define RELE_DEL_DER_SUBE   PIN_B5
63      #define RELE_DEL_DER_BAJA   PIN_B4
64      #define RELE_TRAS_IZQ_SUBE  PIN_B3
65      #define RELE_TRAS_IZQ_BAJA  PIN_C0
66      #define RELE_TRAS_DER_SUBE  PIN_C1
67      #define RELE_TRAS_DER_BAJA  PIN_C2
68      #define RELE_PORTON         PIN_A5
69      #define RELE_ALTAVOCES      PIN_A4
70      #define RELE_ETAPAS         PIN_A3
71      #define RELE_NEONES         PIN_A2
72      #define RELE_SIRENA         PIN_A1
73      #define RELE_CIERRE         PIN_A0
74
75      // Dirección inicial de la secuencia grabada en memoria
76      #define DIR_SEC_INI         0x00000000
77
78      #endif
79
80
```

Finalmente se indica el pinout (conexiones físicas) del microcontrolador con los relés que manejan los actuadores del vehículo.

Una vez se ha descrito el archivo *config.h* pasará a analizarse el archivo principal del firmware del receptor propiamente dicho:

```
1  #include <18F2550.h>
2  #include "config.h"
3  #include "DS1307.h"
4  #include "24LC1025.h"
```

Primeramente se incluye el archivo previamente estudiado, con las definiciones y opciones que éste incorpora. Seguidamente se incluye el archivo de cabecera del driver del timer RTC DS1307, ya desarrollado y estudiado en capítulos anteriores, y se repite el proceso con el archivo relativo al driver de la memoria EEPROM 24LC1025.

```
56  // Suspensión neumática.
57  void rueda_del_izq_sube();
58  void rueda_del_izq_baja();
59  void rueda_del_izq_reposo();
60  void rueda_del_der_sube();
61  void rueda_del_der_baja();
62  void rueda_del_der_reposo();
63  void rueda_tras_izq_sube();
64  void rueda_tras_izq_baja();
65  void rueda_tras_izq_reposo();
66  void rueda_tras_der_sube();
67  void rueda_tras_der_baja();
68  void rueda_tras_der_reposo();
```

Definición de los prototipos de las funciones que controlan las ruedas de forma independiente.

```
70  // Maletero.
71  void neones_on();
72  void neones_off();
73  void etapas_entra();
74  void etapas_sale();
75  void altavoces_sube();
76  void altavoces_baja();
77  void porton_abre();
78  void porton_cierra();
```

Definiciones de las funciones de control de los actuadores del maletero.

```
80  // Inicialización de los actuadores.
81  void reles_ini();
```

Definición de la función de inicialización de los actuadores. Coloca todos los actuadores en una posición conocida predefinida de antemano.

```
84  // Buffer circular.
85  int rx_in_pos;
86  int rx_out_pos;
87  int rx_num_elem;
88  unsigned char rx_buffer[RX_BUFFER_SIZE];
89  void rx_buffer_put(unsigned char dato);
90  unsigned char rx_buffer_get();
91  byte rx_buffer_vacio();
92  void rx_buffer_ini();
```

Definición de prototipos de las funciones relativas al buffer circular, posteriormente implementado.

```
95  // Máquina de estados finitos.
96  void maquina_estados();
97  void maquina_estados_ini();
98  unsigned long int pulsadores;
99  unsigned long int pulsadores_ant;
100 unsigned int estado;
101 byte novedades;
102 unsigned char rx_input;
103 unsigned char dato_D1;
104 unsigned char dato_D2;
105 byte *acceso_byte;
```

Definición de las funciones relativas a la máquina de estados.

```
107 // USB
108 void gestiona_usb();
```

Función en la que se delega el control de todo lo relativo a la gestión de la conectividad USB y grabación de la secuencia recibida. Comprueba el estado de la conexión, en caso de estar conectado el cable al dispositivo, graba la memoria EEPROM cuando recibe la secuencia de instrucciones a guardar.

```
110 // Establece el estado de los actuadores en base
111 // al estado de la variable global "pulsadores".
112 void activa_reles_pulsadores();
```

Analiza la variable global que contiene el estado de los pulsadores del mando en tiempo (casi) real y actualiza el estado de los relés que manejan los actuadores acorde a la información contenida en dicha variable. Dicho de otro modo, materializa en los actuadores el estado de los pulsadores del mando remoto.

```
114 // Secuencias
115 void reproduce_secuencia();
116 byte ejecuta_orden_secuencia(byte *orden);
117 byte checksum_orden_secuencia_ok(byte *orden);
118 byte orden_secuencia_es_bucle(byte *orden);
119 void lee_orden(int32 dir_primer_byte, byte *orden);
120 void indicador_fin_secuencia();
121 byte ejecuta_orden_secuencia(byte *orden);
```

Definición de los prototipos de las funciones relativas a la reproducción de la secuencia grabada en la memoria EEPROM.

A partir aquí comienza la implementación de las funciones y el código propiamente dicho.

```
124     // Manejador de interrupción de
125     // recepción de la USART.
126     #int_rda
127
128     void rx_input_manejador()
129         {
130         rx_buffer_put(getc());
131         }
```

Rutina de atención de interrupción de la USART. Como puede verse solo consta de una línea. La atención debe ser rápida y retornar de forma lo más inmediatamente posible, dejando el procesamiento de los datos recibidos a funciones externas, como ya se ha explicado anteriormente. Esta línea tan solo introduce en el buffer circular el carácter recibido por el módulo receptor de radiofrecuencia.

```
135     void main()
136         {
137         reles_ini();
138         rx_buffer_ini();
139         maquina_estados_ini();
140
141         // Habilitación de interruptciones.
142         enable_interrupts(INT_RDA);
143         enable_interrupts(GLOBAL);
144
145         while(true)
146             {
147             // comprueba el buffer de entrada entrada.
148             maquina_estados();
149             // coloca los relés según el estado
150             // de los pulsadores del emisor.
151             activa_reles_pulsadores();
152             // Comprueba si se ha conectado el dispositivo
153             // al PC y gestiona todo el proceso.
154             gestiona_usb();
155             }
156         }
```

La función “main” del firwmare. Se trata de una función repetitiva, sencilla, y que oculta la complejidad de la infraestructura software en el interior de las funciones que llama. Conceptualmente el funcionamiento es sencillo, primeramente se inicializan

los relés, el buffer y la máquina de estados. Tras ello se habilitan las interrupciones y posteriormente se pasa al bucle principal de la aplicación.

Este bucle primeramente lanza la función “maquina_estados()” la cual lee el buffer circular y, en caso de haber datos, comprueba si hay alguna trama que reconozca. En caso afirmativo actualiza la variable global “pulsadores”, accesible desde el resto de funciones. Esta función es la que implementa la máquina de estados anteriormente descrita y en la que se delega todo el proceso de reconocimiento de la información transmitida por el emisor.

Posteriormente se llama a la función “activa_reles_pulsadores()” la cual analiza la variable global “pulsadores”, que contiene el estado de los pulsadores del emisor en todo momento gracias a la función “maquina_estados()”, y activa o desactiva los relés según el contenido de esta variable. También lanza la reproducción de la secuencia grabada en el caso de que así se indique en la variable mediante el reconocimiento de la combinación de teclas.

Finalmente, se llama a la función “gestiona_usb()”. Como se ha indicado anteriormente en ella se delega toda la gestión de las funcines USB. Primeramente comprueba si está conectado el cable, en caso negativo no hace nada. En caso afirmativo espera hasta que recibe secuencialmente los datos que va grabando en la memoria EEPROM. Si se desconecta el cable, se cancela el proceso de grabación y finaliza la ejecución de la función volviendo el flujo del programa de nuevo al bucle principal.

```
158
159     // Inicializa los puertos del PIC como salida
160     // y desactiva todos los relés.
161     void reles_ini()
162         {
163         set_tris_a(0x00);
164         set_tris_b(0x00);
165         set_tris_c(0x00);
166         output_a(0x00);
167         output_b(0x00);
168         output_c(0x00);
169         }
170
```

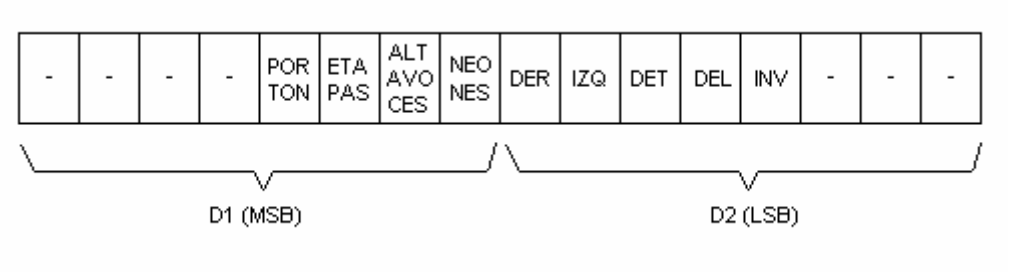
Inicializa los relés. Esta función se llama al encender del receptor, y configura los puertos del micrcontrolador como salidas para, posteriormente, colocar a nivel bajo sus salidas y con ello desactivar todos los elementos conectados a todos los relés. De ello se deduce que nada mas conectar el receptor todos los actuadores del coche pasarán a posición de reposo.

```
173  void activa_reles_pulsadores()
174      {
175      if(novedades)
176          {
177          if(pulsadores & 0x0F00) reproduce_secuencia();
178
179          if(pulsadores & 0x0008) // INV pulsado
180              {
181              // suspensión
182              if(pulsadores & 0x0050) rueda_del_izq_baja();
183              else rueda_del_izq_reposo();
184              if(pulsadores & 0x0090) rueda_del_der_baja();
185              else rueda_del_der_reposo();
186              if(pulsadores & 0x0060) rueda_tras_izq_baja();
187              else rueda_tras_izq_reposo();
188              if(pulsadores & 0x00A0) rueda_tras_der_baja();
189              else rueda_tras_der_reposo();
190              // maletero
191              if(pulsadores & 0x0100) neones_off();
192              if(pulsadores & 0x0200) etapas_entra();
193              if(pulsadores & 0x0400) altavoces_baja();
194              if(pulsadores & 0x0800) porton_cierra();
195              }
196          else                    // INV no pulsado
197              {
198              // suspensión
199              if(pulsadores & 0x0050) rueda_del_izq_sube();
200              else rueda_del_izq_reposo();
201              if(pulsadores & 0x0090) rueda_del_der_sube();
202              else rueda_del_der_reposo();
203              if(pulsadores & 0x0060) rueda_tras_izq_sube();
204              else rueda_tras_izq_reposo();
205              if(pulsadores & 0x00A0) rueda_tras_der_sube();
206              else rueda_tras_der_reposo();
207              // maletero
208              if(pulsadores & 0x0100) neones_on();
209              if(pulsadores & 0x0200) etapas_sale();
210              if(pulsadores & 0x0400) altavoces_sube();
211              if(pulsadores & 0x0800) porton_abre();
212              }
213          novedades = FALSE;
214          }
215      }
```

Esta función coloca los relés en la posición adecuada según el valor de la variable global “interruptores”. Primeramente comprueba si el estado de las teclas contiene la combinación de teclas la cual lanza la reproducción de la secuencia. En cuyo caso delega el control en la función de reproducción.

En caso contrario, comprueba si el botón de inversión de función se encuentra presionado (INV=1). En este caso analiza el estado de los pulsadores y establece el estado de los relés de cada rueda en la posición adecuada.

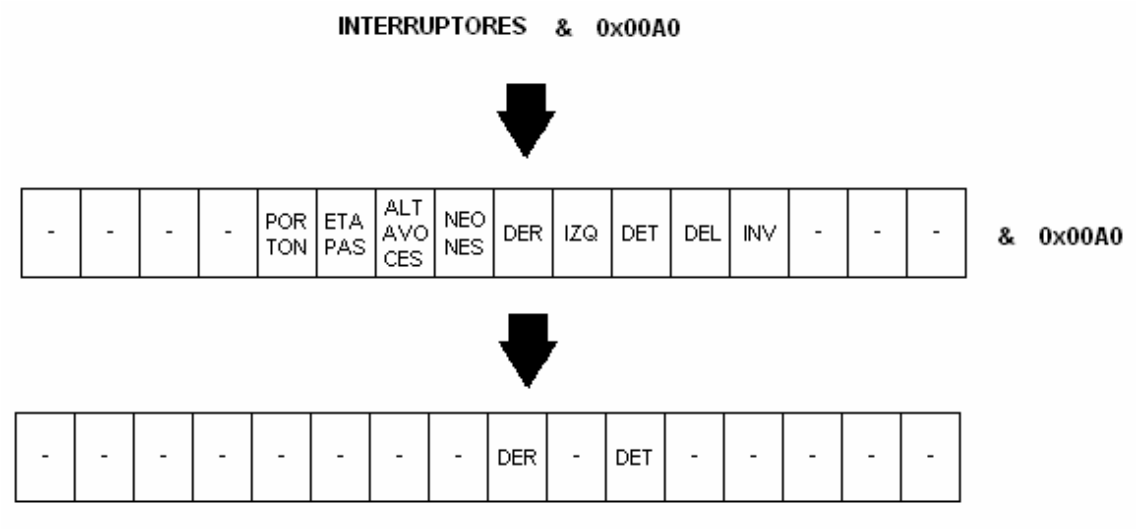
Recuérdese el significado de cada bit dentro de la variable global "interruptores":



El uso de máscaras de bits y operaciones lógicas permite aislar los bits deseados dentro de una variable y actuar en consecuencia según su estado. A continuación se muestra cuando deberá subir cada rueda:

Delantera izquierda = subir de delante ó subir lateral izquierdo.
Delantera derecha = subir de delante ó subir lateral derecho.
Trasera izquierda = subir de detrás o subir de la izquierda.
Trasera derecha = subir de detrás o subir de la derecha.

¿Como se reconoce cada caso mediante las máscaras de bits? Pongamos el ejemplo de la rueda trasera derecha. La rueda trasera derecha deberá elevarse cuando se esté dando la orden de elevar la parte trasera del coche (que supondrá la elevación de las dos ruedas traseras) o cuando se de la orden de elevar el lateral derecho (que supondrá la elevación de las dos ruedas derechas). Aplicando la máscara de bits conseguimos aislar los dos bits de la variable de 16 bits:



En caso de que el resultado de aplicar la máscara de bits sea distinto de cero deberá elevarse la rueda derecha, puesto que alguno de los dos casos estará sucediendo. Es decir, se estará dando la orden de elevación de la parte derecha o elevación de la parte trasera. En caso de que el resultado sea igual a cero es que ninguna de estas órdenes se está dando con lo cual se debe dejar la rueda en estado de reposo. Lo mismo

se aplica al caso de inversión, en el cual en lugar de subir los actuadores invierten su acción (INV=1).

Se trata el caso de cada rueda por separado y los casos de INV=0 e INV=1 de forma aislada. Posteriormente se procede del mismo modo con los actuadores del maletero, aplicando las máscaras adecuadas con la operación AND (&) y actuando en consecuencia. Con la finalidad de evitar sobrecargar al procesador se ha añadido al sistema una variable “novedades”. Cada vez que la máquina de estados cambia el valor de la variable “interruptores” pone “novedades” a TRUE, con lo cual se entra en la función y se actualizan todos los relés. En caso de no haber novedades es innecesario perder ciclos de reloj en reescribir constantemente el estado de todos los relés con el mismo valor.

La función mostrada en la siguiente página se corresponde con la máquina de estados reconocedora de tramas.

Cada vez que se inicia su ejecución inicia un bucle en el cual recorre todos los elementos del buffer circular. Extrae del buffer cada elemento de forma secuencial, uno a uno. Comprueba cada dato extraído actualizando el estado en función del mismo. E caso de reconocer la trama de identificación, guarda los dos bytes consecutivos (bytes de datos) y en el último estado los suma y comprueba con el checksum recibido como último byte de la trama. En caso de éxito, si el estado de los interruptores recibido no presenta diferencias con el actual no hace nada (significa que se está manteniendo el pulsador). En caso de cambio de estado, actualiza la variable global “pulsadores” con el estado de los pulsadores, obtenidos de dato_D1 y dato_D2, y coloca a TRUE la variable “novedades”.

```
218  void maquina_estados()
219      {
220      while(!rx_buffer_vacio())
221          {
222          rx_input = rx_buffer_get();
223          switch(estado)
224              {
225              case 1:
226                  if(rx_input==ID1) estado=2;
227                  else estado=1;
228                  break;
229              case 2:
230                  if(rx_input==ID2) estado=3;
231                  else estado=1;
232                  break;
233              case 3:
234                  if(rx_input==ID3) estado=4;
235                  else estado=1;
236                      break;
237              case 4:
238                  if(rx_input==ID4) estado=5;
239                  else estado=1;
240                  break;
241              case 5:
242                  if(rx_input==ID5) estado=6;
243                  else estado=1;
244                  break;
245              case 6:
246                  if(rx_input==ID6) estado=7;
247                  else estado=1;
248                  break;
249              case 7:
250                  dato_D1=rx_input;
251                  estado=8;
252                  break;
253              case 8:
254                  dato_D2=rx_input;
255                  estado=9;
256                  break;
257              case 9:
258                  if(rx_input==dato_D1+dato_D2)
259                      {
260                      acceso_byte = &pulsadores;
261                      *acceso_byte = dato_D2;
262                      *(acceso_byte+1) = dato_D1;
263                      if(pulsadores != pulsadores_ant)
264                          {
265                          pulsadores_ant = pulsadores;
266                          novedades = TRUE;
267                          }
268                      }
269                  estado=1;
270                  break;
271              }
272          // DEBUG
273          //printf("%u, %c\r\n", estado, rx_input);
274          }
275      }
```

```
278     void maquina_estados_ini()
279         {
280         estado = 1;
281         pulsadores = 0;
282         pulsadores_ant = 0;
283         novedades = FALSE;
284         }
```

Inicializa la máquina de estados. Coloca la máquina en el estado inicial y establece a 0 el estado de los pulsadores (ningún interruptor pulsado en el emisor hasta que se de muestre lo contrario al recibir alguna trama indicadora de ello).

```
287     void rx_buffer_put(unsigned char dato)
288         {
289         if(rx_num_elem < RX_BUFFER_SIZE)
290             {
291             rx_buffer[rx_in_pos] = dato;
292             rx_in_pos = (rx_in_pos+1)%RX_BUFFER_SIZE;
293             rx_num_elem++;
294             }
295         }
```

Esta función añade un elemento de tipo entero de ocho bits sin signo (byte) al buffer circular. Para ello primeramente comprueba si hay sitio. En caso negativo el dato se descarta (se pierde). En caso de que haya sitio disponible, se introduce en la primera posición del buffer libre, apuntada por rx_in_pos, y se aumenta en una unidad dicho apuntador. Para ello se lleva a cabo la operación módulo (%) con el tamaño del buffer. De este modo se consigue volver al inicio una vez pasada la última posición del array.

Posteriormente, se incrementa la variable indicadora del número de elementos y finaliza la función.

```
298     unsigned char rx_buffer_get()
299         {
300         unsigned char dato;
301         if(rx_num_elem == 0) return 0;
302         dato = rx_buffer[rx_out_pos];
303         rx_out_pos = (rx_out_pos+1)%RX_BUFFER_SIZE;
304         rx_num_elem--;
305         return dato;
306         }
```

Función complementaria a la anterior, extrae un elemento del buffer. Primeramente comprueba la disponibilidad de información, en caso de que no haya información disponible devuelve cero (hay que elegir un valor concreto que no tenga repercusión en el proceso que extrae los datos). Posteriormente se extra el dato del buffer, se actualiza el apuntador de lectura y se decrementa el número de elementos.

```
309  byte rx_buffer_vacio()
310       {
311       if(rx_num_elem == 0) return 1;
312       else return 0;
313       }
```

Devuelve 1 si el buffer está vacío, 0 en caso contrario.

```
316  void rx_buffer_ini()
317       {
318       rx_num_elem = 0;
319       rx_in_pos = 0;
320       rx_out_pos = 0;
321       }
```

Inicializa el buffer. Establece el número de elementos a 0 y inicializa los apuntadores a la primera posición del array.

```
324  // Eleva la suspensión de la rueda izquierda.
325  void rueda_del_izq_sube()
326       {
327       output_low(RELE_DEL_IZQ_BAJA);
328       output_high(RELE_DEL_IZQ_SUBE);
329       }
```

Eleva la suspensión de la rueda delantera izquierda. Para ello desactiva la electroválvula que acciona la bajada del coche y activa la que hace subir la suspensión.

```
331  // Desciende la suspensión de la rueda izquierda.
332  void rueda_del_izq_baja()
333       {
334       output_low(RELE_DEL_IZQ_SUBE);
335       output_high(RELE_DEL_IZQ_BAJA);
336       }
```

Desciende la suspensión de la rueda delantera izquierda, de forma análoga a la función anterior.

```
338  // Mantiene la posición de la suspensión de la rueda izquierda.
339  void rueda_del_izq_reposo()
340       {
341       output_low(RELE_DEL_IZQ_SUBE);
342       output_low(RELE_DEL_IZQ_BAJA);
343       }
```

Mantiene la posición de la rueda delantera izquierda. Para ello desactiva la electroválvula de subida y también la de bajada de la rueda.

```
345  // Eleva la suspensión de la rueda delantera derecha.
346  void rueda_del_der_sube()
347       {
348       output_low(RELE_DEL_DER_BAJA);
349       output_high(RELE_DEL_DER_SUBE);
350       }
351
352  // Desciende la suspensión de la rueda delantera derecha.
353  void rueda_del_der_baja()
354       {
355       output_low(RELE_DEL_DER_SUBE);
356       output_high(RELE_DEL_DER_BAJA);
357       }
358
359  // Mantiene la posición de la suspensión de la rueda delantera derecha.
360  void rueda_del_der_reposo()
361       {
362       output_low(RELE_DEL_DER_SUBE);
363       output_low(RELE_DEL_DER_BAJA);
364       }
```

Estas funciones llevan a cabo las mismas operaciones, pero sobre la rueda delantera derecha.

```
366  // Eleva la suspensión de la rueda trasera izquierda.
367  void rueda_tras_izq_sube()
368       {
369       output_low(RELE_TRAS_IZQ_BAJA);
370       output_high(RELE_TRAS_IZQ_SUBE);
371       }
372
373  // Desciende la suspensión de la rueda trasera izquierda.
374  void rueda_tras_izq_baja()
375       {
376       output_low(RELE_TRAS_IZQ_SUBE);
377       output_high(RELE_TRAS_IZQ_BAJA);
378       }
379
380  // Mantiene la posición de la suspensión de la rueda trasera izquierda.
381  void rueda_tras_izq_reposo()
382       {
383       output_low(RELE_TRAS_IZQ_SUBE);
384       output_low(RELE_TRAS_IZQ_BAJA);
385       }
```

Controles de la rueda trasera izquierda.

```
387  // Eleva la suspensión de la rueda trasera derecha.
388  void rueda_tras_der_sube()
389      {
390      output_low(RELE_TRAS_DER_BAJA);
391      output_high(RELE_TRAS_DER_SUBE);
392      }
393
394  // Desciende la suspensión de la rueda trasera derecha.
395  void rueda_tras_der_baja()
396      {
397      output_low(RELE_TRAS_DER_SUBE);
398      output_high(RELE_TRAS_DER_BAJA);
399      }
400
401  // Mantiene la posición de la rueda de la suspensión trasera derecha.
402  void rueda_tras_der_reposo()
403      {
404      output_low(RELE_TRAS_DER_SUBE);
405      output_low(RELE_TRAS_DER_BAJA);
406      }
```

Controles de la rueda trasera derecha.

```
408  // Enciende los neones.
409  void neones_on()
410      {
411      output_high(RELE_NEONES);
412      }
413
414  // Apaga los neones.
415  void neones_off()
416      {
417      output_low(RELE_NEONES);
418      }
```

Encendido y apagado de los neones, respectivamente.

```
420  // Oculta la plataforma de las etapas.
421  void etapas_entra()
422      {
423      output_low(RELE_ETAPAS);
424      }
425
426  // Muestra la plataforma de las etapas.
427  void etapas_sale()
428      {
429      output_high(RELE_ETAPAS);
430      }
```

Control del movimiento de las etapas.

```
// Eleva los altavoces.
void altavoces_sube()
     {
     output_high(RELE_ALTAVOCES);
     }

// Baja los altavoces.
void altavoces_baja()
     {
     output_low(RELE_ALTAVOCES);
     }
```

Control del movimiento vertical de la plataforma de los altavoces.

```
// Abre el portón del maletero.
void porton_abre()
     {
     output_high(RELE_SIRENA);
     delay_ms(MSEGS_SIRENA-1000);
     output_high(RELE_CIERRE);
     delay_ms(1000);
     output_high(RELE_PORTON);
     delay_ms(3000);
     output_low(RELE_SIRENA);
     output_low(RELE_CIERRE);
     }
```

Apertura del portón del maletero. Primeramente activa la sirena durante el tiempo definido por MSEG_SIRENA (definido en el archivo *config.h*). Acto seguido acciona la apertura de la cerradura servocontrolada mientras la sirena aún está sonando, tras lo cual posteriormente abre el portón.

```
// Cierra el portón del maletero.
void porton_cierra()
     {
     output_high(RELE_SIRENA);
     delay_ms(MSEGS_SIRENA);
     output_low(RELE_PORTON);
     delay_ms(3000);
     output_low(RELE_SIRENA);
     }
```

Cierra el portón del maletero. Primeramente activa la señal acústica de aviso durante el tiempo definido por MSEGS_SIRENA, para posteriormente comenzar el descenso del portón y cortar la sirena tres segundos después.

```
468     void reproduce_secuencia()
469         {
470         int32 direccion = DIR_SEC_INI;
471         byte fin = FALSE;
472         byte orden[4];
473 
474         MEM_24LC1025_init();
475 
476         while(!fin)
477             {
478             lee_orden(direccion, orden);
479 
480             if(!checksum_orden_secuencia_ok(orden)) fin = true;
481             else if(orden_secuencia_es_bucle(orden)) direccion = DIR_SEC_INI;
482             else
483                 {
484                 fin = ejecuta_orden_secuencia(orden);
485                 direccion = direccion+4;
486                 }
487             }
488         }
```

Lleva a cabo la reproducción automática de la secuencia preprogramada a través del ordenador. Para ello se define una variable a la que primeramente se le asigna la dirección de inicio (definida en *config.h*). Se inicializa la memoria 24LC1025 y se entra en un bucle de ejecución de instrucciones. Dentro del bucle, como primer paso, se lee una orden, formada por 4 bytes, a partir de la dirección especificada. Posteriormente se comprueba el checksum de la orden y si éste no es válido se pone a TRUE la variable "fin", con lo cual se sale del bucle y la reproducción se da por terminada. En caso de que el checksum sea correcto se comprueba si la orden indica un bucle. En caso afirmativo se reinicia la dirección de lectura con la dirección inicial, y se espera a la siguiente pasada del bucle. Como tercer caso se tiene una ejecución de instrucción normal. En este caso, se ejecuta la instrucción mediante la función "ejecuta_orden_secuencia()" y posteriormente se incrementa la dirección en 4 de forma que en la siguiente pasada del bucle se leerán los próximos 4 bytes al llamar a la función "lee_orden()", ya que cada instrucción está compuesta por 4 bytes. El valor booleano que devuelve la función "ejecuta_orden_secuencia()" indica el motivo de la salida de la función. Si es FALSE significa final de ejecución de la instrucción. En caso de ser TRUE indica la pulsación del botón del emisor que indica la interrupción del proceso de reproducción, con lo cual la siguiente pasada saldrá del bucle y la reproducción de la secuencia se dará por concluida.

La función de lectura de órden de la memoria está implementada del siguiente modo:

```
512  void lee_orden(int32 dir_primer_byte, byte *orden)
513      {
514      *orden = MEM_24LC1025_read(dir_primer_byte);
515      *(orden+1) = MEM_24LC1025_read(dir_primer_byte+1);
516      *(orden+2) = MEM_24LC1025_read(dir_primer_byte+2);
517      *(orden+3) = MEM_24LC1025_read(dir_primer_byte+3);
518      }
```

Lee los cuatro bytes de la memoria EEPROM. Las órdenes se almacenan como un conjunto consecutivo de 4 bytes. Esta instrucción lee las 4 posiciones consecutivas a partir de la dirección especificada como parámetro.

La implementación de la función de comprobación de checksum se muestra a continuación. El motivo de la existencia de esta función es la comprobación de integridad de la orden. En la propia orden se encuentra implícito un valor de checksum el cual debe coincidir con el valor calculado. En caso negativo se toma la orden como errónea y se detendrá la ejecución del programa. Como se verá más adelante, la órden de fin de secuencia se aprovecha de esta propiedad introduciendo una orden vacía con un checksum inválido de forma que de esta manera se fuerza la detención de la ejecución.

```
492  byte checksum_orden_secuencia_ok(byte *orden)
493      {
494      byte checksum_calculado;
495      byte checksum_orden;
496
497      checksum_calculado = *orden+*(orden+1)+*(orden+2);
498      if(*(orden+3) & 0x80) checksum_calculado++;
499      checksum_calculado &= 0x7F;
500      checksum_orden = *(orden+3) & 0x7F;
501
502      if(checksum_calculado == checksum_orden) return TRUE;
503      else return FALSE;
504      }
```

Se trata de un checksum de siete bits, el cual se almacena en los siete bits de menos peso del byte más significativo de la orden. Primeramente se calcula el checksum de los tres bytes inferiores que la componen. En caso de que el bit de mayor peso del byte más significativo sea uno, se sumará una unidad al checksum calculado. Esto es debido a que este bit es parte de la información codificada en la orden y no del checksum (el cual, como se ha dicho, solo ocupa los siete bits de menos peso).

Acto seguido se aisla el checksum codificado en la propia orden a través de la máscara de bits y la operación lógica AND (linea 500) y se comprueba si coinciden. En caso afirmativo la orden es correcta y en caso negativo se asume un error de integridad, dando la orden por descartada.

La siguiente función comprueba si una orden dada representa un bucle en la secuencia:

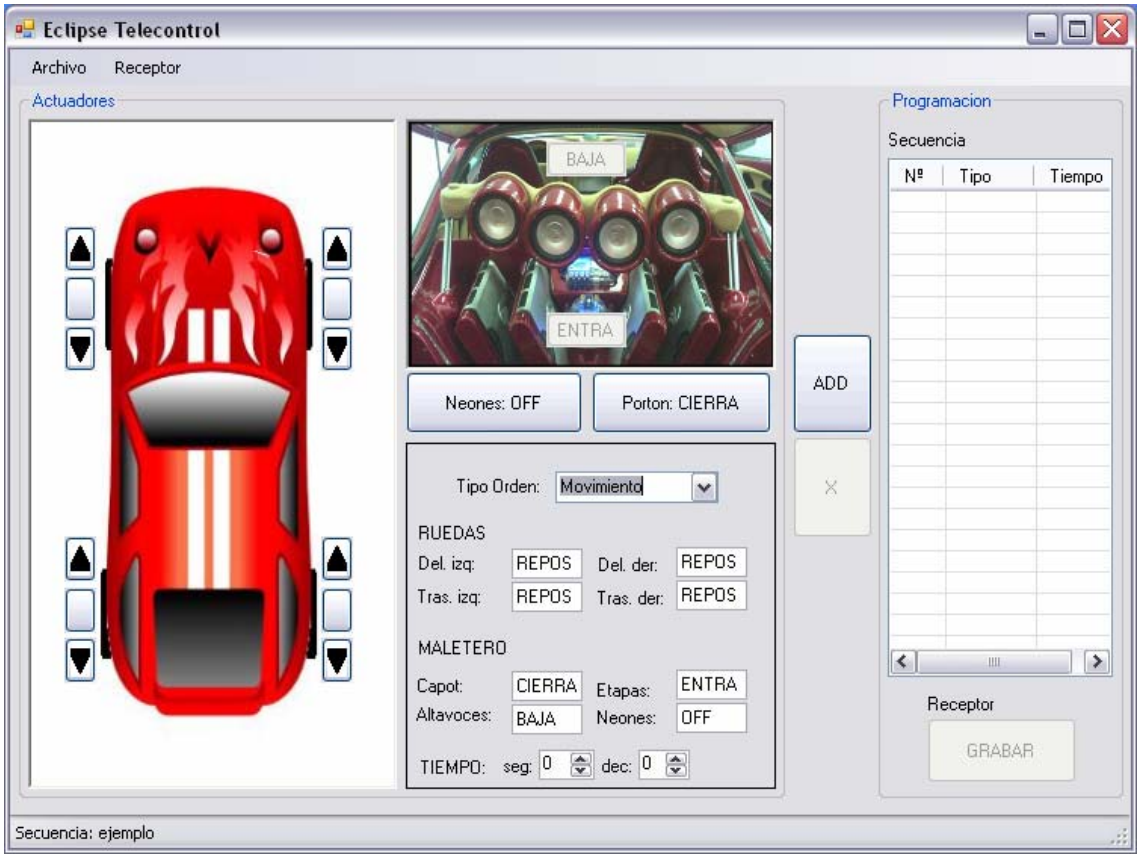
```
506     byte orden_secuencia_es_bucle(byte *orden)
507         {
508         // Si el bit 7 del segundo byte está a 1, indica un cierre de bucle.
509         if((*(orden+1) & 0x0080) != 0) return TRUE;
510         else return FALSE;
511         }
```

Para ello se aisla el bit de mayor peso del segundo byte de los cuatro que componen la orden. En caso de estar activado este bit la orden indica un cierre de bucle de la secuencia, en caso negativo se trata de una orden más de la lista.

5.- Eclipse Telecontrol. Grabación de secuencias.

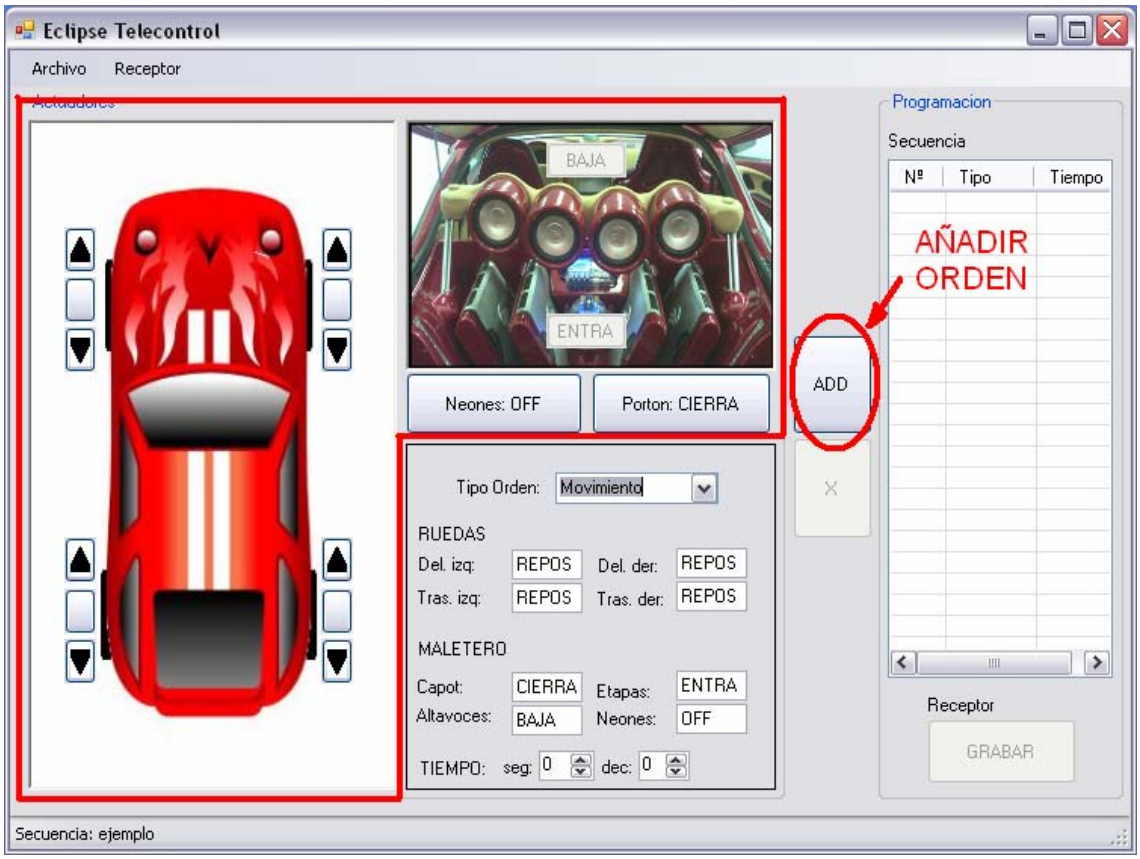
Con la finalidad de hacer posible la grabación de secuencias en el receptor, posteriormente reproducibles desde el mando emisor con tan solo pulsar una combinación preestablecida de teclas, se ha desarrollado un programa de ordenador en C# bajo Windows.

Dicho programa responde al nombre de “Eclipse Telecontrol” y tiene el siguiente aspecto:



El software desarrollado ofrece toda la funcionalidad necesaria para crear secuencias de órdenes en las cuales es posible controlar todos los actuadores del vehículo. Permite controlar la suspensión neumática de cada rueda haciendo posible

subir, bajar y mantener en reposo la misma de forma independiente a las otras 3. Del mismo modo, permite encender o apagar los neones, abrir o cerrar el portón, subir o bajar la plataforma de altavoces e introducir o sacar la plataforma de las etapas. Todo ello de forma gráfica e intuitiva para el usuario. Cada vez que se configuran las opciones de una nueva orden, haciendo *click* en el botón "ADD" se añade a la lista de órdenes de la secuencia de la derecha. Pulsando sobre el botón etiquetado con "X" es posible eliminar de la lista la última órden introducida, en caso de querer rectificar algún error.



Para especificar las órdenes se utilizarán los controles gráficos recuadrados en rojo. Una vez se han colocado todos los actuadores como se desea será posible añadirla a la lista. El recuadro inferior únicamente contiene un resumen de los parámetros configurados gráficamente. Este recuadro es únicamente de carácter informativo. El botón marcado en rojo es el mencionado boton "ADD", bajo el cual se encuentra el de rectificación de última orden marcado con "X".

Una vez especificada la secuencia a gusto del usuario se grabará en la EEPROM del receptor haciendo *click* en el botón "GRABAR". Como es obvio, para ello tendrá que estar conectado a través del cable USB. Puede apreciarse en la captura de pantalla este botón en la parte inferior derecha de la interfaz gráfica.

Obsérvese la siguiente imagen:

El programa da la opción de guardar en archivo las secuencias programadas. Los archivos guardados tendrán la extensión ".sec". Será posible también cargar un archivo de secuencias *.sec* previamente creado. De este modo se posibilita al usuario la opción de llevar a cabo la creación de secuencias en modo *off-line* (desconectado del receptor) y posteriormente poder grabar unas u otras de forma independiente a conveniencia. Asimismo, es posible cargar secuencias previamente creadas, modificarlas y volverlas a grabar de nuevo. Todo ello con el fin de ofrecer una flexibilidad considerable al usuario.

En el comboBox etiquetado como "Tipo de orden" es posible seleccionar que orden se desea introducir a continuación en el sistema. Existen cuatro tipos de órdenes disponibles: Movimiento, Espera, Bucle y Fin.

Movimiento: El tipo principal de órden. Es en el que se basa todo el sistema, y contiene el estado de la suspensión neumática independiente en las cuatro ruedas, el estado de los actuadores del maletero y los neones. Asimismo, tiene un atributo que indica el tiempo mediante el cual és efectiva la orden. Esto es, cuanto tiempo deberá mantenerse en el sistema el estado de los actuadores hasta que se lea la siguiente órden para cambiar el estado de los actuadores del coche con la órden posterior. Dicho de otro modo, durante cuanto tiempo deberá hacerse efectiva la orden y reflejarse en los relés el estado codificado en ella hasta pasar a la siguiente.

Espera: El segundo tipo de orden. Representa una espera, esto es, un tiempo en el cual los actuadores del vehículo permanecen inmóviles, manteniendo el estado de todos los actuadores en ese momento en el estado en el que se encontraran anteriormente.

Fin: Cuando se encuentra la orden "Fin" en el sistema, se interrumpe la reproducción de la secuencia y se da por finalizado el programa de movimientos.

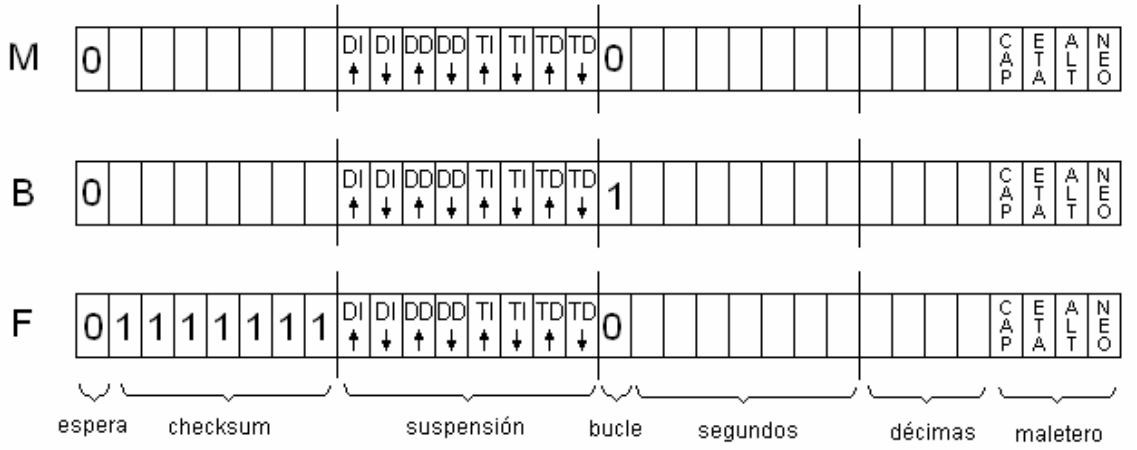
Bucle: permite establecer bucles en la programación de movimientos. Si el sistema encuentra una orden "Bucle", automáticamente volverá al principio de la secuencia. Esto se repetirá de forma indefinida de forma que la única forma que hay de interrumpir la secuencia será apretando las teclas correspondientes en el mando emisor para finalizar la misma (ya que no existe una finalización explícita como ocurría con la orden "Fin").

A continuación se puede ver la modelización UML simplificada (no mostrando los atributos ni funciones de las clases) de la aplicación.

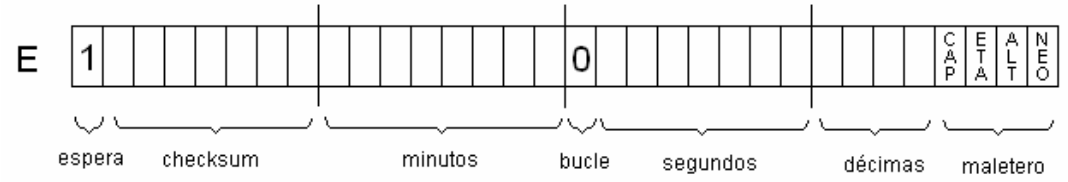
Se puede observar como existe una clase “Orden” de la cual derivan las demás en una estructura jerárquica. Aquí es necesario decir que cada órden del sistema se representará siempre mediante un conjunto de cuatro bytes. Así pues, un programa de receptor consistirá en una sucesión de bytes consecutivos en grupos de cuatro. Depende del tipo de órden, los bits de este conjunto de cuatro bytes tendrán un significado u otro:

En las órdenes tipo Movimiento, Bucle y Fin, el bit más significativo del byte más significativo se establece siempre en 0 (posteriormente se verá el motivo de esto). Los otros 7 bits de byte de mayor peso contienen siempre un checksum de 7 bits, que sirve para comprobar la integridad de los datos de la orden. Durante la reproducción de la secuencia en el receptor, si se lee una orden (como se ha indicado antes, formada siempre por cuatro bytes) el checksum no se corresponde con el calculado, se interrumpirá el proceso de reproducción de secuencia. El segundo byte comenzando por la izquierda contendrá el estado de las electroválvulas que controlan los pistones neumáticos de la suspensión independiente de las cuatro ruedas. Habrá por tanto un bit de control de subida y un bit de control de bajada por cada rueda. Los dos bits a cero significará reposo de la rueda (mantiene la posición) al no accionar ninguna de las dos electroválvulas. El tercer byte (siempre de izquierda a derecha) contiene un bit que indica si la orden es un bucle. En caso de estar a “1” el bit más significativo se entenderá que la orden indica un bucle con lo cual se volverá a la primera instrucción de la secuencia. Los otros 7 bits de este byte indican los segundos por los cuales la orden es efectiva, esto es, cuanto tiempo debe reflejarse en los relés la configuración indicada. El cuarto byte de la instrucción, o byte de menos peso, contiene en su nibble superior (4 bits superiores) las décimas de segundo que junto con los segundos forman la variable que indica el tiempo. En su nibble inferior contiene el estado de los actuadores del maletero.

Volvamos un poco para atrás. Como se indicaba, un checksum erroneo supone la detención de la reproducción de secuencia. Obsérvese la disposición de bits de la orden "Fin", la orden "Fin" se implementa como un caso particular de Movimiento en el cual el checksum es erróneo.



La Orden Espera merece un análisis diferente. Es posible que se deseen hacer esperas en las cuales los actuadores permanecen inmóviles durante minutos. Por ello, en el caso de la Orden de tipo Espera se incluye un campo que representa los minutos que, junto con los segundos y décimas, dotan al sistema de una mayor flexibilidad. En caso de Espera los actuadores de los neumáticos permanecen inamovibles, esto es, serán siempre cero. Por lo tanto se aprovecha el mapeado de bits de la suspensión neumática para codificar en ellos la variable de los minutos. Con el fin de distinguir este caso concreto de los anteriores, se establece el bit más significativo del byte más significativo en "1". Si este bit está a "1" se tratará de una espera y se dará al campo de bits que antes estaba dedicado a la suspensión el tratamiento adecuado teniendo en cuenta que ahora contiene los minutos de la espera.



Las órdenes de tiempo Movimiento y Espera tienen atributos comunes: el estado de los actuadores del maletero. Ello es debido a que los actuadores del maletero tienen dos únicas posiciones posibles. Por ejemplo, el maletero o está subido o está bajado. Para mantenerlo subido es necesario mantener la electroválvula del pistón de subida activo. Lo mismo ocurre con la plataforma de las etapas y de los altavoces, y con los neones, solo hay dos posiciones posibles. Cuando se lleva a cabo una espera (esto es, una órden que simplemente mantiene el estado de los actuadores inalterable durante cierto tiempo) es necesario también activar los actuadores del maletero exactamente en la misma posición en la que se encontraban antes, para mantener su estado. Esto no ocurre con la suspensión neumática, que controla la subida y bajada con electroválvulas

diferentes. Dicho de otro modo, para mantener en reposo la suspensión neumática simplemente no hay que accionar ninguna de las dos electroválvulas que la accionan. Por ello no es necesario incluirla en la orden “Espera”, simplemente cuando durante la reproducción en el receptor se encuentre una orden Espera se cortará la alimentación de todas las electroválvulas de la suspensión neumática mientras que para el maletero si que será necesario establecer los actuadores según indique.

Carece de sentido analizar toda la implementación de las funciones del programa cargador de secuencias ya que el código se encuentra lo suficiente autodocumentado y resultaría en información redundante. El código de la aplicación se incluye como anexo.

Junto a la aplicación se inclye también el driver de extensión *.inf* que será necesario suministrar al sistema durante la primera conexión del hardware a través del puerto USB. Este paso solamente será necesario llevarlo a cabo la primera vez que se conecte el dispositivo. Este archivo es el mismo que se estudió en el apartado anterior de la aplicación de prueba del USB CDC con lo cual no se tratará de nuevo. Como recordatorio se mostrarán las modificaciones llevadas a cabo en el mismo en su seccion inicial:

```
ClassGUID={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%JCR%
LayoutFile=layout.inf

[Manufacturer]
%JCR%=PFC

[PFC]
%Telecontrol%=Reader, USB\VID_04D8&PID_0000

[Reader_Install.NTx86]
;Windows2000

[DestinationDirs]
DefaultDestDir=12
Reader.NT.Copy=12

[Reader.NT]
Include=mdmcpq.inf
CopyFiles=Reader.NT.Copy
```

y al final del mismo:

```
ServiceType = 1 ; SERVICE_KERNEL_DRIVER
StartType = 3 ; SERVICE_DEMAND_START
ErrorControl = 1 ; SERVICE_ERROR_NORMAL
ServiceBinary = %12%\usbser.sys
LoadOrderGroup = Base

[Strings]
JCR = "JCR"
Telecontrol = "Eclipse Telecontrol"
Serial.SvcDesc = "Programador de secuencias (Eclipse Telecontrol)"
```

6.- Conclusiones y agradecimientos.

Es un hecho que la informática avanza a pasos agigantados. Este proyecto aúna la informática y la electrónica en un solo proyecto. Realmente aunque muchas veces se pase por alto, la informática y la electrónica van de la mano desde los comienzos de la era digital: en el momento que existe una lógica, un protocolo y un lenguaje elemental (binario) se puede hablar de computación, gestión de información y por tanto informática. Llevar a cabo este proyecto me ha permitido autoafirmarme en la idea de que la informática está presente en muchos aspectos de la vida cotidiana. Cualquier aparato electrónico que incorpore un microcontrolador está basándose en la informática para su funcionamiento.

Del mismo modo el hecho de realizar un proyecto destinado a una aplicación poco habitual como es el tuning de vehículos me ha permitido observar que, nuevamente, la tecnología tiene cabida en todo tipo de ambientes y aplicaciones.

Me gustaría agradecer la comprensión a mi familia y amigos en los momentos de tensión derivados ocasionalmente del desarrollo de este proyecto. Del mismo modo agradezco a la Universidad Politécnica de Valencia la formación recibida durante estos años, que me ha permitido llevar a cabo este proyecto de forma íntegra así como muchos otros, y en especial a Angel Rodas, por la paciencia y la ayuda que me ha prestado en todo momento cuando la situación así lo ha requerido.

7.- Anexos.

La carpeta etiquetada bajo el nombre “anexos” que acompaña a este documento posee los siguientes elementos:

- Datasheet de todos los componentes.
- El protocolo rs232.
- El protocolo i2c.
- El protocolo USB CDC
- “Microchip application notes” oportunas.
- Proyectos del diseño de hardware de emisor y receptor así como las respectivas simulaciones en Proteus.
- Proyectos del diseño del código de los microcontroladores. Se adjuntan dos proyectos en MPLab, del emisor y receptor respectivamente.
- Proyectos del diseño del firmware de pruebas
- Implementación de las librerías de la memoria 24LC1025 y el timer DS1307.
- Proyecto en Visual Studio 2008 C# del programa desarrollado para PC.