COMPILADOR C CCS Y SIMULADOR PROTEUS PARA MICROCONTROLADORES PIC

Incluye CD

Eduardo García Breijo

Alfaomega

marcombo
ediciones técnicas

COMPILADOR C CCS

Y

SIMULADOR PROTEUS

PARA

MICROCONTROLADORES PIC

Eduardo García Breijo

Alfaomega

marcombo
ediciones tecnicas

Datos catalográficos

García, Eduardo
Compilador C CCS y simulador PROTEUS para Microcontroladores PIC
Primera Edición

Alfaomega Grupo Editor, S.A. de C.V., México

ISBN: 978-970-15-1397-2

Formato: 17 x 23 cm Páginas: 276

Compilador C CCS y simulador PROTEUS para Microcontroladores PIC
Eduardo García Breijo
ISBN: 978-84-267-1495-4, edición original publicada por MARCOMBO, S.A., Barcelona, España
Derechos reservados © MARCOMBO, S.A.

Primera edición: Alfaomega Grupo Editor, México, junio de 2008.

© 2008 Alfaomega Grupo Editor, S.A. de C.V.
Pitágoras 1139, Col. Del Valle, 03100, México D.F.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **libreriapitagoras@alfaomega.com.mx**

ISBN: 978-970-15-1397-2

Derechos reservados:
Esta obra es propiedad intelectual de su autor y los derechos de publicación en lengua española han sido legalmente transferidos al editor. Prohibida su reproducción parcial o total por cualquier medio sin permiso por escrito del propietario de los derechos del copyright.

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro y en el CD-ROM adjunto, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Pitágoras 1139, Col. Del Valle, México, D.F. – C.P. 03100.
Tel.: (52-55) 5089-7740 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: libreriapitagoras@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Carrera 15 No. 64 A 29 – PBX (57-1) 2100122
Fax: (57-1) 6068648 – E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – General del Canto 370-Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Paraguay 1307 P.B. "11", Capital Federal, Buenos Aires, C.P. 1057 – Tel.: (54-11) 4811-7183 / 8352, E-mail: info@alfaomegaeditor.com.ar

Índice analítico

1. ISIS de PROTEUS VSM 1
1.1 Introducción 1
1.2 Captura electrónica: entorno gráfico ISIS 2
1.3 Depuración de los sistemas basados en PICmicro 12
2. Compilador CCS C 23
2.1 Introducción 23
2.2 Estructura de un programa 24
2.3 Tipos de datos 24
2.4 Las constantes 25
2.5 Variables 26
2.6 Operadores 27
2.6.1 Asignación 27
2.6.2 Aritméticos 27
2.6.3 Relacionales 28
2.6.4 Lógicos 28
2.6.5 De bits 28
2.6.6 Punteros 28
2.7 Funciones 29
2.8 Declaraciones de control 30
2.8.1 IF-ELSE 31
2.8.2 SWITCH 33
2.8.3 FOR 34
2.8.4 WHILE / DO-WHILE 35
2.8.5 Otros 37
2.9 Comentarios 37
2.10 Directivas y funciones (Preprocessor commands y built-in functions) 38
2.10.1 Directivas 38
2.10.2 Funciones 40
2.11 Librerías, drivers y ejemplos 43
2.12 Entorno de trabajo de CCS C Compiler 43
2.12.1 Introducción 43
2.12.2 Entorno de trabajo 44
3. La gestión de los puertos 55
3.1 Introducción 55
3.2 Gestión de puertos en C 57
3.2.1 A través de la RAM 57

3.2.2 A través de las directivas 60
3.2.3 Con punteros 62
3.3 Entradas y salidas 65
3.3.1 LCD 65
3.3.2 LCD gráfico 70
3.3.3 Teclado (keypad 3x4) 75
4. Las interrupciones y los temporizadores 83
4.1 Introducción 83
4.2 Interrupciones 83
4.2.1 Interrupciones en C 88
4.3 TIMER0 94
4.3.1 TIMER0 en C 95
4.4 TIMER1 y TIMER2 99
4.4.1 TIMER1 y TIMER2 en C 102
5. Convertidor Analógico – Digital 117
5.1 Introducción 117
5.2 Módulo Convertidor (gama media) 118
5.2.1 Registros FSR 119
5.2.2 Proceso de conversión 122
5.2.3 Efecto del modo SLEEP y RESET en el módulo AD 124
5.3 Módulo AD en C 125
6. Módulo CCP – Comparador, Captura y PWM 137
6.1 Introducción 137
6.2 Modo Captura 139
6.3 Modo Comparación 140
6.4 Modo PWM 141
6.5 Módulo CCP en C 143
7. Transmisión serie 167
7.1 Introducción 167
7.2 El módulo USART/SCI 168
7.2.1 Introducción 168
7.2.2 El módulo USART en C 174
7.2.3 La norma RS232 180
7.3 Puerto serie síncrono (SSP) 190
7.3.1 Interfaz Inter-Circuitos (I2C) 190
8. Gama Alta – PIC18 213
8.1 Introducción 213
8.2 Organización de la memoria 214
8.2.1 Arquitectura HARDVARD 215
8.2.2 Memoria de Programa 215
8.2.3 Contador de Programa 216
8.2.4 Memoria de Configuración 217

8.2.5 Pila 217
8.2.6 Memoria de Datos 218
8.2.7 Memoria EEPROM 219
8.2.8 Modos de Direccionamiento 220
8.2.9 Interrupciones 220
8.2.9.1 Registros de salvaguarda 222
8.2.10 Registro W 223
8.2.11 Oscilador 223
8.2.12 Unidades Funcionales 223
8.2.12.1 Puertos de entrada/salida 224
8.2.12.2 Temporizadores 225
8.2.12.3 Convertidor Analógico-Digital 226
8.2.12.4 Canal de Comunicación Serie (EUSART) 227
8.2.12.5 Modulo Master SSP (MSSP) 228
8.2.12.6 Modulo de Compración/Captura/PWM (CCP) 228
8.2.12.7 Modulo Comparador 230
8.2.12.8 Modulo de referencia 230
8.2.12.9 Modulo detector de Alto/Bajo Voltaje 230
9. RTOS – Real Time Operating System 239
9.1 Introducción 239
9.2 RTOS en C 240
10. USB – Universal Serial Bus 251
10.1 Introducción 251
10.1.1 Migración de RS232 a USB 252
10.1.1.1 USB CDC (Communication Device Class) 252
10.2 USB con ISIS y CCS C 253
10.2.1 USB en ISIS 253
10.2.2 USB en CCS C 254

Introducción

El estudio de los microcontroladores PIC no consiste sólo en dominar su arquitectura interna o el código maquina sino también en conocer programas auxiliares que facilitan el diseño de los sistemas donde intervienen.

Entre los muchos programas para el desarrollo de sistemas con PICmicro® destacan, por su potencia, el PROTEUS VSM de ©Labcenter Electrónics y el compilador C de ©Custom Computer Services Incorporated (CCS).

El programa PROTEUS VSM es una herramienta para la verificación vía software que permite comprobar, prácticamente en cualquier diseño, la eficacia del programa desarrollado. Su combinación de simulación de código de programación y simulación mixta SPICE permite verificaciones analógico-digitales de sistemas basados en microcontroladores. Su potencia de trabajo es magnífica.

Por otra parte, tenemos el compilador C de CCS, ya que después de conocer y "dominar" el lenguaje ensamblador es muy útil aprender a programar con un lenguaje de alto nivel como el C. El compilador CCS C permite desarrollar programas en C enfocado a PIC con las ventajas que supone tener un lenguaje desarrollado específicamente para un microcontrolador concreto. Su facilidad de uso, su cuidado entorno de trabajo y la posibilidad de compilar en las tres familias de gamas baja, media y alta, le confieren una versatilidad y potencia muy elevadas.

Al escribir este libro se plantean muchas dudas, sobre todo a la hora de concretar el temario. Escribir profusamente sobre los PIC o sobre el PROTEUS o sobre el CCS C supone, casi seguro, escribir un libro para cada uno de estos temas. Por ello, el planteamiento ha sido diferente, desarrollar los conocimientos básicos necesarios para manejar cada programa, apoyarlo con el mayor número de ejercicios y dejar al lector la posterior ampliación de conocimientos. Así lo he decido en base a la experiencia que me da estar impartiendo clases sobre PIC en la carrera de Ingenieros Técnicos Industriales, especialidad de Electrónica Industrial, de la Universidad Politécnica de Valencia.

Con estas premisas espero que el libro sirva a lector para aumentar sus conocimientos sobre el PIC o para iniciarlos en el caso de los que desconozcan este mundo.

Capítulo 1

ISIS de PROTEUS VSM

1.1 Introducción

El entorno de diseño electrónico *PROTEUS VSM* de *LABCENTER ELECTRONICS* (*www.labcenter.co.uk*) ofrece la posibilidad de simular código microcontrolador de alto y bajo nivel y, simultáneamente, con la simulación en modo mixto de *SPICE*. Esto permite el diseño tanto a nivel hardware como software y realizar la simulación en un mismo y único entorno. Para ello, se suministran tres potentes subentornos como son el *ISIS* para el diseño gráfico, *VSM* (*Virtual System Modelling*) para la simulación y el *ARES* para el diseño de placas (figura 1).

Figura 1. Entorno de trabajo PROTEUS (fuente: Labcenter Electronics)

1.2 Captura electrónica: entorno gráfico ISIS

ISIS es un potente programa de diseño electrónico que permite realizar esquemas que pueden ser simulados en el entorno *VSM* o pasados a un circuito impreso ya en el entorno *ARES*.

Posee una muy buena colección de librerías de modelos tanto para dibujar, simular o para las placas. Además, permite la creación de nuevos componentes, su modelización para su simulación e, incluso, la posibilidad de solicitar al fabricante (*Labcenter Electronics*) que cree un nuevo modelo.

Sin entrar profundamente en como utilizar dicho programa (requeriría un libro sólo para ello), a continuación se explican las bases para dibujar cualquier circuito electrónico. El programa *ISIS* posee un entorno de trabajo (figura 2) formado por distintas barras de herramientas y la ventana de trabajo.

Figura 2. El entorno de trabajo del programa ISIS

Varios de estos menús también se pueden utilizar con la ayuda del botón derecho del ratón. Al pulsarlo en cualquier parte del entorno de trabajo aparece un menú contextual donde se pueden ir obteniendo los distintos submenús de trabajo (figura3).

Figura 3. Submenús de trabajo del botón derecho del ratón

Para dibujar, lo primero es colocar los distintos componentes en la hoja de trabajo. Para ello, se selecciona el modo componentes (figura 4) y, acto seguido, realizar una pulsación sobre el botón *P* de la ventana de componentes y librerías (figura 5).

Figura 4. Modo componentes

Figura 5. Boton "pick"

Tras activar el botón *P* se abre la ventana para la edición de componentes (figura 6) donde se puede buscar el componente adecuado y comprobar sus características.

Al localizar el componente adecuado se realiza una doble pulsación en él, de tal forma que aparezca en la ventana de componentes y librerías (figura 7). Se puede realizar esta acción tantas veces como componentes se quieran incorporar al esquema. Una vez finalizado el proceso se puede cerrar la ventana de edición de componentes.

Figura 6. Ventana para la edición de componentes

Figura 7. Los componentes añadidos

Para situar un componente en el esquema tan sólo debemos seleccionarlo de la lista. Al hacerlo se puede comprobar su orientación (tal como se representará en el esquema) en la ventana de edición (figura 8). Si deseamos modificar la rotación o la reflexión del componente podemos acceder a ello a través de la barra de herramientas correspondiente (figura 9).

Haciéndolo de esta forma, "todos" los componentes de la lista tendrán la misma orientación (si se desea orientar un único componente deberemos hacerlo una vez ya situado en el esquema).

Figura 8. Selección y orientación del componente

Figura 9. Barra de rotación y reflexión

Ahora sólo falta realizar una pulsación sobre la ventana de trabajo y se colocará el componente. El cursor del ratón se convierte en un lápiz blanco (figura 10). Se pueden colocar varios componentes del mismo tipo simplemente realizando varias pulsaciones. Para terminar de colocar un componente se debe seleccionar otro componente de la lista o pasar a otro modo de trabajo.

Figura 10. Cursor en el modo de colocación

Es importante activar la herramienta de referencia automática (*Real Time Annotation*). De esta forma, los componentes tendrán una referencia distinta y de forma consecutiva; en los circuitos integrados con varios componentes encapsulados también se referenciarán según dicho encapsulado (U1A, U1B, etc.). Esta herramienta se activa o desactiva desde la opción de menú ***TOOLS → Real Time Annotation***.

Una vez situados los componentes en el área de trabajo se pueden mover, al pasar por encima del componente el cursor se convierte en una mano (figura 11) y al realizar una pulsación, el cursor se transforma en una mano con una cruz, indicando que se puede mover el componente (quedan seleccionados al ponerse en rojo) y se puede arrastrar (atención: si se vuelve a realizar otra pulsación del botón izquierdo se editan las características del componente). También se puede cambiar su orientación utilizando los comandos de rotación y reflexión a través de una pulsación del botón derecho del ratón (figura 12) y se pueden eliminar con dos pulsaciones con el botón derecho sobre ellos (o con el botón derecho y el comando *Delete Objet*).

Figura 11. El cursor en modo de selección y mover

Figura 12. Menú contextual de un componente activado por el botón derecho del ratón

Todas estás acciones se pueden realizar individualmente o de forma colectiva, es decir, se pueden agrupar varios componentes a través de pulsaciones consecutivas sobre ellos (manteniendo la tecla <Control> pulsada) o dibujando una ventana con el botón izquierdo y arrastrándola sobre los mismos (figura 14).

Figura 13. Los componentes en el área de trabajo

Una vez seleccionado el conjunto de componentes (se marcan todos en rojo) debe mos utilizar la herramienta de grupo (figura 15), que también aparece tras pulsar el botón derecho. Con esta herramienta se pueden copiar, mover, rotar o eliminar los componentes seleccionados.

Figura 14. Selección de varios componentes

Figura 15. La herramienta de grupo

Para unir los componentes con cables hay que situarse en los extremos de los terminales, el cursor se convierte en un lápiz verde (figura 16). Ahora se pueden ejecutar dos acciones o ir marcando el camino hasta el destino con distintas pulsaciones (figura 17) o realizar, directamente, una pulsación en el destino y dejar que *ISIS* realice el camino. Para ello, debe estar activada la herramienta *TOOLS → Wire Autorouter.*

Figura 16. Inicio de cable

Figura 17. Circuito "a mano"

Las uniones entre cables se pueden realizar de forma automática. Para ello, mientras se traza un camino debemos realizar una pulsación sobre el cable objeto de la unión eléctrica (figura 18). También se pueden realizar de forma manual mediante el modo de unión (figura 19); en este modo tan sólo hay que ir haciendo pulsaciones sobre los puntos donde deseamos realizar la unión.

Figura 18. Unión eléctrica entre cables

Figura 19. Modo de unión

Se puede modificar el trazado de los cables. Para ello, se realiza una pulsación sobre el cable, en ese instante el cursor se convierte en una doble flecha (figura 20) y se puede arrastrar el ratón para modificar el cable.

Figura 20. Mover los cables

También se pueden utilizar buses para las uniones multicable. Los buses permiten conectar varios terminales entre sí utilizando un único elemento (figura 21); en este caso, el cursor se convierte en un lápiz azul (figura 22). Pero para distinguir los distintos cables que forman parte del bus y distribuirlos en la entrada y en la salida se deben etiquetar mediante *labels*. En el caso de los cables se indicará una etiqueta única *LCD0, LCD1*, etc., y al bus una etiqueta conjunta según el formato *LCD[0..3]* que indique el nombre y la cantidad de cables que lo forman.

Figura 21. Cableado por bus

Figura 22. Cursor en modo de trazado de bus

El etiquetado también permite unir cables virtualmente. Para ello, tan sólo es necesario que los dos cables se llamen igual aunque no estén conectados entre si. Para etiquetar cables o buses se utiliza el modo *label* (figura 23). Al activar este modo y realizar una pulsación sobre un cable o bus se abre una ventana donde podemos introducir la etiqueta, además de seleccionar posición, orientación y estilo (figura 24).

Figura 23. Modo label

Figura 24. Ventana de edición de etiquetas

Otro modo de unión virtual es a través de terminales. Al activar el modo terminal (figura 25) se pueden seleccionar distintos tipos de terminales, entre ellos el utilizado por defecto (*default*). Al utilizar este terminal en varios componentes y darle el mismo nombre en todos ellos se consigue una unión eléctrica.

Figura 25. Modo terminal

Figura 26. Unión eléctrica a través de terminales

Mediante este modo también se pueden colocar las masas y alimentaciones del circuito utilizando las opciones *Ground* y *Power* (figura 26). De esta forma se puede finalizar el circuito (figura 27).

Figura 27. Circuito cableado

Tan sólo queda modificar las características de los componentes que lo requieran, por ejemplo modificando el valor de los componentes pasivos. Para ello, se selecciona un componente realizando una pulsación con el botón derecho, aparece el menú contextual y se selecciona la opción *EDIT PROPERTIES*; también se puede utilizar el modo edición (figura 28) en el cual tan sólo es necesario hacer una pulsación con el botón izquierdo sobre el componente; en este modo el cursor se convierte en una flecha (figura 29). Al ejecutar esta acción se abre la ventana de edición donde se pueden cambiar las características de los componentes (figura 30), por ejemplo la resistencia de 10K a 180 ohm. También se puede editar directamente la referencia o el valor de un componente si la pulsación se realiza encima de estos elementos.

Figura 28. Modo edición

Figura 29. Cursor en modo edición

Figura 30. Ventana de edición de un componente

Con esto quedaría finalizado el circuito electrónico (figura 31). Pero en el caso de los sistemas basados en un microcontrolador aún quedan por modificar las características del mismo microcontrolador.

Figura 31. El esquema completo

En el caso de los microcontroladores, la ventana de edición aporta mucha información (figura 32). Tal vez lo más importante es que permite cargar en el micro controlador el archivo de programa (*.*HEX*) generado en la compilación; también se puede modificar la frecuencia de reloj (por lo tanto no es necesario el uso de cristales externos en la simulación), cambiar la palabra de configuración y otras propiedades avanzadas.

Figura 32. Ventana de edición de un micro

1.3 Depuración de los sistemas basados en PICmicro

La característica más importante del *PROTEUS VSM* es la capacidad de depurar programas fuente de distintos lenguajes de programación. Además de aceptar el archivo de programación *Intel Hex* (*HEX*), también admite ficheros *IAR UBROF* (*D39*), *ByteCraft COD* (*COD*), *Microchip Compatible COF* (*COF*) y *Crownhill Proton Plus* (*BAS*). Al utilizar estos archivos se puede abrir una ventana de código fuente llamada ***SOURCE CODE*** mediante la cual se puede seguir el programa fuente línea a línea de código.

Además permite visualizar elementos internos del *PIC* como son la memoria de programa, la memoria de datos *RAM* o la *EEPROM*, los registros especiales (*FSR*) y la pila (*Stack*).

Además, el entorno *PROTEUS VSM* permite compilar programas fuente en código ensamblador directamente. Para ello, se utiliza el comando ***SOURCE*** (figura 33).

Figura 33. Generador de código de ficheros fuente

En el caso del compilador *CCS C*, después de compilar se generan, entre otros, los archivos **.HEX* y **.COF*, los cuales se pueden utilizar para trabajar con el entorno *PROTEUS VSM*. Para ejecutar el programa desde *ISIS* se debe abrir la ventana de edición del microcontrolador (figura 32) y en el ítem ***PROGRAM FILE*** se puede indicar el fichero de código fuente utilizado.

Además, en esta ventana se puede indicar la frecuencia de trabajo con la opción ***PROCESSOR CLOCK FREQUENCY*** (debemos observar que para la simulación no es necesario colocar elementos externos de oscilación en el PIC, tan sólo hacen falta en caso de realizar la placa). En la opción ***ADVANCED PROPERTIES*** podemos habilitar o configurar muchos más elementos: configurar el *wacthdog*, habilitar avisos de desbordamiento de pila, accesos no correctos a memoria, etc.

Una vez cargado el microcontrolador con el programa fuente, se puede proceder a la simulación del circuito empleando la barra de simulación (figura 34). Esta barra se compone de la opción *MARCHA*, *PASO A PASO*, *PAUSA* y *PARADA*.

Figura 34. Barra de simulación

Con la opción *MARCHA* la simulación se inicia (el botón se vuelve verde) y funciona en modo continuo. La simulación **NO** es en tiempo real y dependerá de la carga de trabajo del PC. En la barra de estado se indica la carga de la *CPU* del PC (a mayor carga menos real será la simulación) y el tiempo de ejecución; este tiempo indica el tiempo que tardaría, en la realidad, el circuito en realizar un proceso (por ejemplo esto implica que, dependiendo de la carga de trabajo de la *CPU*, un tiempo de 1 s en el circuito puede significar varios minutos de simulación).

Figura 35. Barra de estado en la simulación

La opción *STOP* para totalmente la simulación mientras que PAUSE la para de forma momentánea permitiendo hacer uso de las herramientas de depuración.

La opción *PASO a PASO* permite trabajar en tramos de tiempo predefinidos y, además, permite utilizar las herramientas de depuración. Esta opción está ligada a la configuración de la animación (figura 36): ***SYSTEM → SET ANIMATION OPTIONS → SINGLE STEP TIME*** donde se puede definir el incremento de tiempo que se desea que pase cada vez que se pulsa esta tecla.

Figura 36. Set animation options

En este cuadro de diálogo también se pueden cambiar los siguientes parámetros:

- ***FRAMES PER SECOND***: numero de veces que la pantalla de *ISIS* se refresca en un segundo (por defecto 20).
- ***TIMESTEP PER FRAME***: indica el tiempo de simulación por cada uno de los *frames*; lo ideal es que sea el valor inverso del escogido en la opción *FRAMES PER SECOND*.
- ***ANIMATIONS OPTIONS***: permite habilitar la visualización de las sondas de tensión y corriente, mostrar los niveles lógicos en los pines, mostrar el nivel de tensión en los cables mediante colores o mostrar la dirección de la corriente en los cables mediante flechas.
- ***VOLTAGE/CURRENT RANGES***: permite determinar el umbral de tensión (±V) y corriente para utilizar en la visualización de las correspondientes *ANIMATIONS OPTIONS*.

En este punto se puede simular (y animar) un sistema con el *PICmicro* (figura 37). Lo más interesante de la simulación con microcontroladores es la utilización de las herramientas de depuración. Es decir, visualizar mediante ventanas las distintas partes internas del microcontrolador: memoria de programa, memoria de datos, pila, etc. La mayor parte de estas ventanas sólo se pueden visualizar durante una *PAUSA*.

Figura 37. Una simulación en marcha

Desde el menú *DEBUG* (figura 38) también se puede iniciar la simulación pero pensando en la depuración. Con la opción *START/RESTAR DEBUGGING* se puede iniciar la simulación pero haciendo una pausa para ver las distintas ventanas de depuración. También se puede ejecutar el programa directamente con la opción *EXECUTE, EXECUTE WITHOUT BREAKPOINT* o *EXECUTE FOR SPECIFIED TIME* que permite ejecutar directamente un programa, ejecutarlo sin puntos de ruptura (en el caso de tenerlos) y ejecutarlo en un tiempo concreto.

Figura 38. El menú DEBUG antes de la simulación

Desde esta ventana también se puede reinicializar la memoria *EEPROM* del microcontrolador mediante ***RESET PERSISTENT MODEL DATA.*** Al producirse una pausa, el menú ***DEBUG*** se modifica (figura 39) mostrando las correspondientes herramientas de depuración.

Figura 39. El menú DEBUG en una pausa

Estas herramientas son (figura 40):

- *SIMULATION LOG*: Mensajes resultantes de la simulación.
- *WATCH WINDOWS*: Ventana de visualización de posiciones de memoria. Permite añadir la que el usuario desea ver.
- *PIC CPU REGISTERS*: Muestra los registros *FSR* del *PIC*.
- *PIC CPU DATA MEMORY*: Muestra la memoria de datos (*RAM*).
- *PIC CPU EPROM MEMORY*: Muestra la memoria de datos (*EPROM*).
- *PIC CPU PROGRAM MEMORY*: Muestra la memoria de programa.
- *PIC CPU STACK*: Muestra la pila.

Figura 40. Ventanas de depuración

La ventana *WATCH WINDOW* es la más versátil puesto que se pueden añadir variables y modificar su visualización. Al pulsar el botón derecho sobre la ventana se abre un menú contextual (figura 41). Con *ADD ITEMS (name/address)* se añade la variable a visualizar directamente con el nombre predefinido (figura 42) en el *PIC* o, en el caso de variables propias del programador, se pueden visualizar por dirección (figura 43), donde se le indica el nombre, la dirección en hexadecimal, el tipo de dato (*byte, word,* etc.) y el formato de visualización (binario, decimal, etc.). El tipo de dato y el formato también se puede cambiar desde *DATA TYPE* y *DISPLAY FORMAT*.

Figura 41. Menú contextual de WATCH WINDOWS

Con *WATCHPOINT CONDITION* se pueden habilitar puntos de ruptura mediante condiciones sobre las distintas variables (figura 44); se indica la variable, la máscara de la condición (*NONE, AND, OR, XOR*) y el tipo de condición (*NONE, ON CHANGE, EQUALS*, etc.).

Figura 42. Add by Name

Figura 43. Add by Address

Figura 44. Puntos de ruptura

Hay una ventana de depuración que sólo se visualiza si se ha incorporado un fichero *COD* o *COF* al microcontrolador, se trata de la ventana *CPU SOURCE CODE* (figura 45). Con esta ventana se puede seguir la simulación línea a línea del archivo de código fuente.

En esta ventana (también en el menú *DEBUG*) están disponibles unos botones de control (figura 46).

PIC CPU Source Code - U1

EX_LCDKB.c

```
0000 #include <16F876.h>
---- #fuses XT,NOWDT,NOPROTECT,NOLVP
0016 #use delay(clock= 4000000)
----
---- #include <lcd_mio.c>
---- #include <kbd_MIO.c>
----
----
0172 void main() {
----    char k;
----    int  v,x;
----
0177    lcd_init();
0178    kbd_init();
0179    port_b_pullups(TRUE);
----
0186    lcd_putc("\fListo...\n");
----
----    while (TRUE) {
0196       k=kbd_getc();
0199       v=k;
019B       x=k-48; //0x30
019E       if(k!=0)
01A1         if(k=='*')
01A5           lcd_putc('\f');
01A8         else
01A9             lcd_putc(k);              //imprime caracter
---- //          printf(lcd_putc,"%c",k); //imprime caracter
---- //          printf(lcd_putc,"%c",v); //imprime caracter
---- //          printf(lcd_putc,"%u",k); //imprime valor ascii
---- //          printf(lcd_putc,"%u",v); //imprime valor ascii
---- //          printf(lcd_putc,"%u",x); //imprime valor numerico
01AC    }
---- }
----
```

Figura 45. La ventana CPU Source code

Figura 46. Los controles para la simulación

- Simulación en modo continuo, no permite ver las ventanas de depuración.
- Permite ejecutar una instrucción; si es una subrutina o una función la ejecuta directamente.
- Permite ejecutar una instrucción; si es una subrutina o una función entra en ella.
- Trabaja en modo continuo hasta que encuentra un retorno de cualquier subrutina.
- Trabaja en modo continuo hasta que se encuentra con un punto de ruptura.
- Habilita o deshabilita los puntos de ruptura.

Hay una ventana de diagnóstico que facilita la depuración, almacenando los errores, mensajes de diagnóstico y avisos producidos durante el proceso de simulación (figura 47). En la barra de estado (zona inferior del área de trabajo) se muestra un aviso (figura 48); con una pulsación en el aviso aparece la ventana de diagnosis.

Figura 47. Mensajes de diagnóstico de la simulación

Se pueden configurar las opciones de esta herramienta desde la opción *DEBUG→ CONFIGURE DISGNOSTIC* (figura 49). En la ventana se muestran los componentes del esquema susceptibles de un diagnóstico en la simulación y las diferentes posibilidades de diagnóstico y el tiempo de diagnóstico (figura 50).

Figura 48. Mensajes de la herramienta de diagnóstico

Figura 49. La configuración de diagnósticos

Figura 50. Opciones de configuración

Tras la simulación aparecen los diferentes resultados del análisis; en el ítem *SOURCE* aparece indicado el dispositivo fuente del análisis y tras una pulsación se puede acceder a él (figura 51).

Figura 51. Resultado del diagnóstico

El listado de nodos y patillas se pueden encontrar con la ayuda del *DESIGN EXPLORER* (a ésta opción podemos acceder a través de distintas opciones: comando *DESIGN*, botón derecho, etc.). En su ventana de trabajo se muestran todos los nodos y patillas que forman el circuito (figura 52).

Figura 52. La ventana Design Explorer

Capítulo 2

Compilador CCS C

2.1 Introducción

El *Compilador C* de *CCS* ha sido desarrollado específicamente para *PIC MCU*, obteniendo la máxima optimización del compilador con estos dispositivos. Dispone de una amplia librería de funciones predefinidas, comandos de preprocesado y ejemplos. Además, suministra los controladores (*drivers*) para diversos dispositivos como LCD, convertidores AD, relojes en tiempo real, EEPROM serie, etc. Las características generales de este compilador y más información adicional se pueden encontrar en la dirección **http://www.ccsinfo.com**.

Un compilador convierte el lenguaje de alto nivel a instrucciones en código máquina; un *cross-compiler* es un compilador que funciona en un procesador (normalmente en un PC) diferente al procesador objeto. El compilador *CCS C* es un *cross-compiler*. Los programas son editados y compilados a instrucciones máquina en el entorno de trabajo del PC, el código máquina puede ser cargado del PC al sistema PIC mediante el *ICD2* (o mediante cualquier programador) y puede ser depurado (puntos de ruptura, paso a paso, etc.) desde el entorno de trabajo del PC.

El *CCS C* es *C* estándar y, además de las directivas estándar (*#include*, etc.), suministra unas directivas específicas para PIC (*#device*, etc.); además incluye funciones específicas (*bit_set()*, etc.). Se suministra con un editor que permite controlar la sintaxis del programa.

NOTA
En el manual de CCS se da mucha más información de la que a continuación se va a dar. En este capítulo sólo se describirán los elementos más básicos y esenciales para comenzar a programar.

2.2 Estructura de un programa

Para escribir un programa en C con el *CCS C* se deben tener en cuenta una serie de elementos básicos de su estructura (figura 1).

- **DIRECTIVAS DE PREPROCESADO**: controlan la conversión del programa a código máquina por parte del compilador.
- **PROGRAMAS o FUNCIONES**: conjunto de instrucciones. Puede haber uno o varios; en cualquier caso siempre debe haber uno definido como principal mediante la inclusión de la llamada *main()*.
- **INSTRUCCIONES**: indican como debe comportar el PIC en todo momento.
- **COMENTARIOS**: permiten describir lo que significa cada línea del programa.

Figura 1. Estructura básica de un programa

2.3 Tipos de datos

CCS C acepta los siguientes tipos de datos:

Tipo	Tamaño	Rango	Descripción
Int1 Short	1 bit	0 a 1	Entero de 1 bit

Tipo	Tamaño	Rango	Descripción
Int **Int8**	8 bit	0 a 255	Entero
Int16 **Long**	16 bit	0 a 65.535	Entero de 16 bit
Int32	32 bit	0 a 4.294.967.295	Entero de 32 bit
Float	32 bit	$\pm 1.175 \times 10^{-38}$ a $\pm 3.402 \times 10^{-38}$	Coma flotante
Char	8 bit	0 a 255	Carácter
Void	-	-	Sin valor
Signed Int8	8 bit	-128 a +127	Entero con signo
Signed Int16	16 bit	-32768 a + 32767	Entero largo con signo
Signed Int32	32 bit	-2^{31} a $+(2^{31}-1)$	Entero 32 bit con signo

2.4 Las constantes

Las constantes se pueden especificar en decimal, octal, hexadecimal o en binario:

123	Decimal
0123	Octal (**0**)
0x123	Hexadecimal(**0x**)
0b010010	Binario (**0b**)
'x'	Carácter
'\010'	Carácter octal
'\xA5'	Carácter hexadecimal

Además, se pueden definir constantes con un sufijo:

Int8	127U
Long	80UL
Signed INT16	80L
Float	3.14F
Char	Con comillas simples 'C'

También se definen caracteres especiales, algunos como:

\n	Cambio de linea
\r	Retorno de carro

\t	Tabulación
\b	Backspace

2.5 Variables

Las variable se utilizan para nombrar posiciones de memoria *RAM*; se deben declarar, obligatoriamente, antes de utilizarlas; para ello se debe indicar el nombre y el tipo de dato que se manejará. Se definen de la siguiente forma:

TIPO NOMBRE_VARIABLE [=VALOR INICIAL]

TIPO hace referencia a cualquiera de los tipos de datos vistos en el punto 2.3. El *NOMBRE_VARIABLE* puede ser cualquiera y el valor inicial es opcional. Veamos un ejemplo:

```
float temp_limit=500.0;
```

Las variables definidas en un programa pueden ser de tipo *LOCAL* o *GLOBAL*. Las variables locales sólo se utilizan en la función donde se encuentran declaradas; las variables globales se pueden utilizar en todas las funciones del programa. Ambas deben declararse antes de ser utilizadas y las globales deben declararse antes de cualquier función y fuera de ellas. Las variable globales son puestas a cero cuando se inicia la función principal *main()*.

```
#include <16f876.h>
#USE DELAY (CLOCK=4000000)
int16 counter;          // Variable global
void FUNCION(void)
{
   char K, kant='0';    // Variables locales
}
void main( )
{
   int8 temp;           // Variable local
}
```

Las variables pueden ser definidas con:

- **AUTO**: (usada por defecto, no hace falta que se indique) donde la variable existe mientras la función esta activa. Estas variables no se inicializan a cero. Su valor se pierde cuando se sale de la función.

- STATIC: Una variable local se activa como global, se inicializa a cero y mantiene su valor al entrar y salir de la función.
- EXTERN: Permite el uso de variables en compilaciones múltiples.

2.6 Operadores

2.6.1 Asignación

+=	Asignación de suma (x+=y es lo mismo que x=x+y)
-=	Asignación de resta (x-=y es lo mismo que x=x-y)
*=	Asignación de multiplicación (x*=y es lo mismo que x=x*y)
/=	Asignación de división (x/=y es lo mismo que x=x/y)
%=	Asignación del resto de la división (x%=y es lo mismo que x=x%y)
<<=	Asignación de desplazamiento a la izquierda (x<<=y es igual que x=x<<y)
>>=	Asignación de desplazamiento a derecha (x>>=y es igual que x=x>>y)
&=	Asignación AND de bits (x&=y es lo mismo que x=x&y)
|=	Asignación OR de bits (x|=y es lo mismo que x=x|y)
^=	Asignación OR EXCLUSIVA de bits (x^=y es lo mismo que x=x^y)

2.6.2 Aritméticos

+	Suma
-	Resta
*	Multiplicación
/	División
%	Módulo, resto de una división entera
--	Incremento
++	Decremento
sizeof	Determina el tamaño, en bytes, de un operando

En las operaciones de decremento e incremento, en función de la posición del operador, se consigue un preincremento (++A) o un postincremento (A++).

```
a=3;
b=4*a++;          // b=4 y a=4;

a=3
b=4*++3;          // b=16 y a=4;
```

2.6.3 Relacionales

<	Menor que
>	Mayor que
>=	Mayor o igual que
<=	Menor igual que
==	Igual
!=	Distinto
?:	Expresión condicional

2.6.4 Lógicos

!	NOT
&&	AND
||	OR

2.6.5 De bits

~	Complemento a 1
&	AND
^	OR EXCLUSIVA
|	OR
>>	Desplazamiento a derechas
<<	Desplazamiento a izquierdas

2.6.6 Punteros

&	Dirección
*	Indirección
->	Puntero a estructura

Orden de precedencia de los operadores:

Expresiones en orden de precedencia descendente					
(expr)					
!expr	~expr	++expr	expr++	--expr	expr--
(type)expr	*expr	&value	sizeof(type)		
expr*expr	expr/expr	expr%expr			
expr+expr	expr-expr				
expr<<expr	expr>>expr				
expr<expr	expr<=expr	expr>expr	expr>=expr		
expr==expr	expr!=expr				
expr&expr					
expr^expr					
expr | expr					
expr && expr					
expr || expr					
expr ? expr:expr					
lvalue = expr	lvalue+=expr	lvalue-=expr			
lvalue*=expr	lvalue/=expr	lvalue%=expr			
lvalue>>=expr	lvalue<<=expr	lvalue&=expr			
lvalue^=expr	lvalue|=expr	expr, expr			

2.7 Funciones

Las funciones son bloques de sentencias; todas las sentencias se deben enmarcar dentro de las funciones. Al igual que las variables, las funciones deben definirse antes de utilizarse.

Una función puede ser invocada desde una sentencia de otra función. Una función puede devolver un valor a la sentencia que la ha llamado. El tipo de dato se indica en la definición de la función; en el caso de no indicarse nada se entiende que es un *int8* y en el caso de no devolver un valor se debe especificar el valor *VOID*. La función, además de devolver un valor, puede recibir parámetros o argumentos.

La estructura de una función es:

```
Tipo_Dato        Nombre_Funcion (tipo param1, param2,....)
{
   (sentencias);
}
```

```
float trunca (float a) {
float b;
   b=floor(a);
   a=a-b;
   a=a*100;
   a=floor(a);
   a=a*0.01;
   a=b+a;
   return(a);
}
```

La forma de devolver un valor es mediante la sentencia *RETURN*:

```
return (expresión);
return expresión;
```

Donde expresión debe manejar el mismo tipo de dato que el indicado en la definición de la función. En el caso de no devolver nada se finaliza con *RETURN*, al encontrar esta sentencia el compilador vuelve a la ejecución de la sentencia de llamada. También se puede finalizar la función sin *RETURN*, tan sólo con la llave de cierre "}".

Las funciones pueden agruparse en ficheros de librerías *<fichero.h>*, que se pueden utilizar mediante la directiva *#include <fichero.h>*.

2.8 Declaraciones de control

Las declaraciones son usadas para controlar el proceso de ejecución del programa. Las que admite *CCS* son:

- *If-Else*.
- *While*.

- *Do-While.*
- *For.*
- *Switch-Case.*
- *Return.*
- *Break, Continue* y *Goto.*

2.8.1 IF-ELSE

Con la ayuda de *IF-ELSE* se pueden tomar decisiones.

```
if (expresión)
        sentencia_1;
[else
        sentencia_2;]
```

NOTA
Los elementos que se encuentran entre corchetes [ ] son opcionales.

Primero se evalúa la *EXPRESIÓN* y si es cierta (*TRUE* o 1) ejecuta la *SENTENCIA_1*, en el caso contrario (*FALSE* o 0) ejecuta la *SENTENCIA_2*.

Pueden anidarse los *IF-ELSE* dando lugar a los *ELSE-IF*; esto permite tomar decisiones múltiples.

```
if (expresión_1)
        sentencia_1;
[else if (expresión_2)
        sentencia_2;]
[else
        sentencia_3;]
```

En este caso las *EXPRESIONES* se evalúan en orden, si alguna de ellas es cierta la *SENTENCIA* asociada a ella se ejecutará y se termina la función. En caso contrario se ejecuta la *SENTENCIA* del *ELSE*. En ambos casos si existen varias sentencias para ejecutar se deben utiliza las llaves { }:

```
if (expresión)
{
sentencia_1;
.....
sentencia_n;
}
[else
{
sentencia_1;
.....
sentencia_n;
}]
```

Ejemplos:

2.8.2 SWITCH

Switch es un caso particular de una decisión múltiple

```
switch (expresion)
{
case constante 1:
sentencias;
break;
case constante 2:
sentencias;
break;
...
[default:
sentencias;]
}
```

Evalúa la expresión y en orden a la *CONSTANTE* adecuada realiza las sentencias asociadas. Si ninguno de los *CASE* corresponde a la *CONSTANTE* se ejecuta *DEFAULT* (este comando es opcional).

El comando *BREAK* provoca la salida de *SWITCH*, de lo contrario se ejecuta el siguiente *CASE*.

NOTA
No pueden existir dos *CASE* con la misma *CONSTANTE*.

Ejemplo:

2.8.3 FOR

Se usa para repetir sentencias.

```
for (inicialización ; condición de finalización ; incremento )
{
        sentencias;
}
```

En las expresiones del *FOR* la inicialización es una variable a la cual se le asigna un valor inicial con el que controlar el bucle. La condición de finalización sirve para evaluar *ANTES* de ejecutar las sentencias si es cierta o no, en el caso de ser cierta se ejecutan las sentencias y en caso contrario se sale del *FOR*. Por último, la expresión de incremento o decremento modifica la variable de control *DESPUÉS* de ejecutar el bucle.

NOTA
Se pueden anidar bucles *FOR* utilizando distintas variables de control.

Si se ejecuta la siguiente expresión se consigue un *BUCLE SIN FIN*:

```
For( ; ; )
{
        sentencias;
}
```

Ejemplo:

2.8.4 WHILE / DO-WHILE

WHILE se utiliza para repetir sentencias.

```
while (expresión)
{
        sentencias;
}
```

La expresión se evalúa y la sentencia se ejecuta mientras la expresión es verdadera, cuando es falsa se sale del *WHILE*.

DO-WHILE se diferencia del *WHILE* y del *FOR* en la condición de finalización, la cual se evalúa al final del bucle, por lo que las sentencias se ejecutan al menos una vez.

```
Do
{
        sentencias;
}
while (expresion);
```

Si se ejecutan las siguientes expresiones se consigue un *BUCLE SIN FIN*:

```
While(1)
{
        sentencias;
}
```

```
Do
{
        sentencias;
}
While (1)
```

Ejemplos:

2.8.5 Otros

- **Return:** se emplea para devolver datos en las funciones.
- **Break:** permite salir de un bucle, se utiliza para *While, For ,Do* y *Switch.*
- **Goto:** provoca un salto incondicional.

2.9 Comentarios

Los comentarios en el programa facilitan la compresión de las distintas expresiones tanto para el programador como para quién tiene que interpretar dicho programa. No afectan a la compilación por lo que pueden ser tan extensos como el programador quiera. Se pueden colocar en cualquier parte del programa y con dos formatos:

- Utilizando //. Al colocar estos signos se comienza el comentario y finaliza en el final de la línea.

```
// Comentario que terminará al final de esta línea.
```

- Utilizando /* y */. Se debe utilizar al inicio y al final de comentario, pero no pueden repetirse dentro del mismo comentario.

```
/* Este comentario no finaliza al final de esta línea
finaliza cuando se cierre el comentario */
```

2.10 Directivas y funciones (Preprocessor commands y built-in functions)

2.10.1 Directivas

Las directivas de pre-procesado comienzan con el símbolo # y continúan con un comando específico. La sintaxis depende del comando. Algunos comandos no permiten otros elementos sintácticos en la misma expresión. Muchas de las directivas utilizadas por *CCS* son extensiones del C estándar.

Estándar C	#DEFINE ID STRING	#IF expr	#NOLIST
	#ELSE	#IFDEF id	#PRAGMA cmd
	#ENDIF	#LIST	#UNDEF id
	#ERROR	#INCLUDE "FILENAME"	
Cualificadores	#INLINE	#INT_GLOBAL	#SEPARATE
	#INT_DEFAULT	#INT_xxx	
Identificadores	__DATE__	__LINE__	__PCH__
	__DEVICE__	__PCB__	__TIME__
	__FILE__	__PCM__	__FILENAME__
RTOS	#TASK	#USE RTOS	
Especificación Dispositivos	#DEVICE CHIP	#ID "filename"	#FUSES options
	#ID CHECKSUM	#ID NUMBER	#SERIALIZE
Librerías predefinidas	#USE DELAY CLOCK	#USE FIXED_IO	#USE RS232
	#USE FAST_IO	#USE I2C	#USE SPI
	#USE STANDARD_IO		
Control de memoria	#ASM	#BYTE id=id	#ROM
	#BIT id=id.const	#ENDASM	#TYPE
	#BIT id=const.const	#FILL_ROM	#ZERO_RAM
	#BUILD	#LOCATE id=const	
	#BYTE id=const	#RESERVE	
Control de compilador	#CASE	#OPT n	#PRIORITY
	#ORG	#IGNORE_WARNINGS	

A lo largo del presente libro se irán viendo varias directivas en su ámbito de aplicación particular.

Como ejemplo se pueden comentar:

#DEVICE chip, permite definir el *PIC* con el que se realizará la compilación.

```
#device PIC16F84
```

#FUSES options, la cual permite definir la palabra de configuración para programar un *PIC*. Por ejemplo, en el *PIC16F84* las opciones posibles son:

LP, XT, HS, RC, NOWDT, WDT, NOPUT, PUT, PROTECT, NOPROTECT.

```
#device PIC16F84
#fuses XT, NOWDT, PUT, NOPROTECT
```

#INCLUDE "filename", permite incluir fichero en el programa.

```
#include <16F84.h>
#fuses XT, NOWDT, PUT, NOPROTECT
```

#USE DELAY (CLOCK=SPEED), permite definir las frecuencia del oscilador del *PIC*, el compilador lo utiliza para realizar cálculos de tiempo. Se puede utilizar M, MHZ, K y KHZ para definir la frecuencia.

```
#INCLUDE <16F877.h>
#use delay(clock=4000000)
```

#ASM y #ENDASM, permiten utilizar código ensamblador en el programa en C. Se utilizan al inicio y al final del bloque ensamblador.

```
#asm
bsf TATUS,RP0
movlw  0x8
movwf PORTB
bcf STATUS,RP0
#endasm
```

2.10.2 Funciones

CCS suministra una serie de funciones predefinidas para acceder y utilizar el PIC y sus periféricos. Estas funciones facilitan la configuración del PIC sin entrar en el nivel de los registros especiales. Las funciones se clasifican por bloques funcionales.

E/S RS232	ASSERT( )	GETCH( )	PUTC( )	
	FGETC( )	GETCHAR( )	PUTCHAR( )	
	FGETS( )	GETS( )	PUTS( )	
	FPRINTF( )	KBHIT( )	SET_UART_SPEED( )	
	FPUTC( )	PERROR( )	SETUP_UART( )	
	FPUTS( )	PRINTF( )		
E/S BUS	SETUP_SPI( )	SPI_DATA_IS_IN( )		SPI_WRITE( )
SPI 2-HILOS	SPI_XFER( )	SPI_READ( )		
E/S DISCRETAS	GET_TRISx( )	INPUT_K( )	OUTPUT_FLOAT( )	SET_TRIS_B( )
	INPUT( )	INPUT_STATE( )	OUTPUT_G( )	SET_TRIS_C( )
	INPUT_A( )	INPUT_x( )	OUTPUT_H( )	SET_TRIS_D( )
	INPUT_B( )	OUTPUT_A( )	OUTPUT_HIGH( )	SET_TRIS_E( )
	INPUT_C( )	OUTPUT_B( )	OUTPUT_J( )	SET_TRIS_F( )
	INPUT_D( )	OUTPUT_BIT( )	OUTPUT_K( )	SET_TRIS_G( )
	INPUT_E( )	OUTPUT_C( )	OUTPUT_LOW( )	SET_TRIS_H( )
	INPUT_F( )	OUTPUT_D( )	OUTPUT_TOGGLE( )	SET_TRIS_J( )
	INPUT_G( )	OUTPUT_DRIVE( )	PORT_A_PULLUPS( )	SET_TRIS_K( )
	INPUT_H( )	OUTPUT_E( )	PORT_B_PULLUPS( )	
	INPUT_J( )	OUTPUT_F( )	SET_TRIS_A( )	
E/S PUERTO	PSP_INPUT_FULL( )		PSP_OVERFLOW( )	
PARALELO ESCLAVO	PSP_OUTPUT_FULL( )		SETUP_PSP( )	
E/S BUS I2C	I2C_WRITE( )	I2C_SlaveAddr( )	I2C_ISR_STATE( )	
	I2C_POLL( )	I2C_START( )		
	I2C_READ( )	I2C_STOP( )		
CONTROL PROCESOS	CLEAR_INTERRUPT( )	GOTO_ADDRESS( )	RESET_CPU( )	
	DISABLE_INTERRUPTS( )	INTERRUPT_ACTIVE( )	RESTART_CAUSE( )	

CONTROL PROCESOS	ENABLE_INTERRUPTS( )	JUMP_TO_ISR	SETUP_OSCILLATOR( )	
	EXT_INT_EDGE( )	LABEL_ADDRESS( )	SLEEP( )	
	GETENV( )	READ_BANK( )	WRITE_BANK( )	
MANEJO BIT-BYTE	BIT_CLEAR( )	MAKE8( )	_MUL( )	SHIFT_LEFT( )
	BIT_SET( )	MAKE16( )	ROTATE_LEFT( )	SHIFT_RIGHT( )
	BIT_TEST( )	MAKE32( )	ROTATE_RIGHT( )	SWAP( )
OPERADORES MAT.	ABS( )	COSH( )	LABS( )	SIN( )
	ACOS( )	DIV( )	LDEXP( )	SINH( )
	ASIN( )	EXP( )	LDIV( )	SQRT( )
C ESTÁNDAR	ATAN( )	FABS( )	LOG( )	TAN( )
	ATAN2( )	FLOOR( )	LOG10( )	TANH( )
	CEIL( )	FMOD( )	MODF( )	
	COS( )	FREXP( )	POW( )	
TENSIÓN DE REFERENCIA	SETUP_VREF( )	SETUP_LOW_VOLT_DETECT( )		
A/D CONVERSIÓN	SET_ADC_CHANNEL( )		SETUP_ADC_PORTS( )	
	SETUP_ADC( )	READ_ADC( )		
CARACTERES C ESTÁNDAR	ATOF( )	ISLOWER(char)	STRCMP( )	STRRCHR( )
	ATOI( )	ISPRINT(x)	STRCOLL( )	STRSPN( )
	ATOI32( )	ISPUNCT(x)	STRCPY( )	STRSTR( )
	ATOL( )	ISSPACE(char)	STRCSPN( )	STRTOD( )
	ISALNUM( )	ISUPPER(char)	STRLEN( )	STRTOK( )
	ISALPHA(char)	ISXDIGIT(char)	STRLWR( )	STRTOL( )
	ISAMOUNG( )	ITOA( )	STRNCAT( )	STRTOUL( )
	ISCNTRL(x)	SPRINTF( )	STRNCMP( )	STRXFRM( )
	ISDIGIT(char)	STRCAT( )	STRNCPY( )	TOLOWER( )
	ISGRAPH(x)	STRCHR( )	STRPBRK( )	TOUPPER( )
TIMERS	GET_TIMER0( )	SET_RTCC( )	SETUP_TIMER_0 ( )	
	GET_TIMER1( )	SET_TIMER0( )	SETUP_TIMER_1 ( )	
	GET_TIMER2( )	SET_TIMER1( )	SETUP_TIMER_2 ( )	
	GET_TIMER3( )	SET_TIMER2( )	SETUP_TIMER_3 ( )	

TIMERS	GET_TIMER4( )	SET_TIMER3( )	SETUP_TIMER_4 ( )	
	GET_TIMER5( )	SET_TIMER4( )	SETUP_TIMER_5 ( )	
	GET_TIMERx( )	SET_TIMER5( )	SETUP_WDT ( )	
	RESTART_WDT( )		SETUP_COUNTERS( )	
MEMORIA C ESTANDAR	CALLOC( )	MEMCMP( )	OFFSETOFBIT( )	
	FREE( )	MEMCPY( )	REALLOC( )	
	LONGJMP( )	MEMMOVE( )	SETJMP( )	
	MALLOC( )	MEMSET( )		
	MEMCHR( )	OFFSETOF( )		
CADENAS ESTÁNDAR	STRXFRM( )	MEMCHR( )	MEMCMP( )	
	STRCAT( )	STRCHR( )	STRCMP( )	
	STRCOLL( )	STRCSPN( )	STRICMP( )	
	STRCOLL( )	STRCSPN( )	STRICMP( )	
	STRLEN( )	STRLWR( )	STRNCAT( )	
	STRNCMP( )	STRNCPY( )	STRPBRK( )	
	STRRCHR( )	STRSPN( )	STRSTR( )	
	STANDARD STRING FUNCTION( )			
MODULO CCP	SET_POWER_PWM_OVERRIDE( )		SETUP_CCP2( )	
	SET_POWER_PWMX_DUTY( )		SETUP_CCP3( )	
	SET_PWM1_DUTY( )		SETUP_CCP4( )	
	SET_PWM2_DUTY( )		SETUP_CCP5( )	
	SET_PWM3_DUTY( )		SETUP_CCP6( )	
	SET_PWM4_DUTY( )		SETUP_POWER_PWM( )	
	SET_PWM5_DUTY( )		SETUP_POWER_PWM_PINS( )	
	SETUP_CCP1( )			
EEPROM INTERNA	ERASE_PROGRAM_EEPROM( )		SETUP_EXTERNAL_MEMORY( )	
	READ_CALIBRATION( )		WRITE_CONFIGURATION_ MEMORY( )	
	READ_EEPROM( )		WRITE_EEPROM( )	
	READ_EXTERNAL_MEMORY( )		WRITE_EXTERNAL_MEMORY( )	
	READ_PROGRAM_EEPROM( )		WRITE_PROGRAM_EEPROM( )	
	READ_PROGRAM_MEMORY( )		WRITE_PROGRAM_MEMORY( )	
C ESTÁNDAR (ESPECIALES)	BSEARCH( )	RAND( )	SRAND( )	QSORT( )
RETARDOS	DELAY_CYCLES( )		DELAY_US( )	DELAY_MS( )

- **.PJT:** fichero de proyecto; contiene toda la información relacionada con el proyecto.
- **.LST:** muestra un listado con el código C y el código ensamblador asociado para cada línea de código.
- **.SYM:** muestra las posiciones y valores de los registros y las variables del programa.
- **.STA:** muestra una estadística de la utilización de la *RAM*, *ROM* y la *PILA*.
- **.TRE:** muestra un árbol del programa donde se especifican las funciones y sus llamadas, con la *ROM* y *RAM* usada en cada una de ellas.
- **.HEX:** fichero estándar para la programación del *PIC*.
- **.COF:** fichero binario que incluye el código máquina y la información para la depuración correspondiente.

2.12.2 Entorno de trabajo

El entorno de trabajo del *CCS* en *PCW* y *PCWH* permite compilar y también suministra una gran variedad de herramientas auxiliares. En la figura 2 se muestra los distintos elementos básicos del entorno de trabajo. Existen dos formas de iniciar una sesión: abriendo un fichero de código fuente o creando un proyecto.

Figura 2. Entorno de Trabajo

Para abrir un fichero fuente directamente se realiza una pulsación sobre el icono para el manejo de ficheros (figura 3) y aparece un menú donde podemos crear, abrir, guardar o cerrar ficheros. Con el comando *NEW* podemos crear un fichero fuente, un proyecto, un fichero *RTF* o un fichero de diagrama de flujo.

Figura 3. Los menús para el manejo de los ficheros

Con la opción *NEW* → *SOURCE FILE*, el programa pide el nombre del nuevo fichero y crea una nueva ventana en blanco donde podemos empezar a escribir (Figura 4).

Figura 4. Fichero fuente nuevo

Si se ejecuta el comando *PROJECT WIZARD*, tras pedir el nombre del nuevo proyecto aparece la ventana de configuración con dos pestañas, una para configurar las distintas opciones que se muestran en la barra de la izquierda (figura 5) y otra donde se muestra el código resultante de la configuración (figura 6). Recorriendo las distintas opciones (*general, communications*, etc.) se llega a obtener el código de configuración deseado (figura 7), tras lo cual ya podemos empezar a escribir el resto del código del programa. Debemos observar como se incluye un fichero de cabecera **.h* donde se encuentra la configuración del dispositivo (figura 8).

Figura 5. Ventana de configuración de las opciones

Figura 6. Ventana de configuración con el código resultante

```
prueba_1.c
1   #include "c:\ccs\prueba_1.h"
2   #include <LCD.C>
3     #include <math.h>
4
5
6   void main()
7   {
8
9     setup_adc_ports(AN0);
10    setup_adc(ADC_CLOCK_INTERNAL);
11    setup_spi(FALSE);
12    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_1);
13    setup_timer_1(T1_DISABLED);
14    setup_timer_2(T2_DISABLED,0,1);
15    lcd_init();
16
17    [illegible]
18
19  }
```

Figura 7. El código después de una configuración

```
prueba_1.c   prueba_1.h
1   #include <f876.h>
2   #device adc=10
3
4   #fuses NOWDT                    [illegible]
5   #fuses HS                       [illegible]
6   #fuses PUT                      [illegible]
7   #fuses NOPROTECT                [illegible]
8   #fuses NOBROWNOUT               [illegible]
9   #fuses NOLVP                    [illegible]
10  #fuses NOCPD                    [illegible]
11  #fuses NOWRT                    [illegible]
12  #fuses NODEBUG                  [illegible]
13
14  #use delay(clock=[illegible])
15  #use rs232(baud=[illegible], parity=N, xmit=PIN_C6, rcv=PIN_C7, bits=[illegible])
16
```

Figura 8. El fichero de cabecera con la configuración del PIC

2.12.2.1 El primer programa

La opción del *PROJECT WIZARD* es muy cómoda pero para comenzar a trabajar con *CCS C*, se recomienda iniciar los ficheros de código fuente directamente hasta que el programador adquiera los conocimientos básicos para manejar esta opción.

Así pues abrimos un fichero fuente nuevo donde se escribirá un programa para encender y apagar un *led* durante 1 segundo. El *led* se conectará a la patilla *RB7* de un *PIC16F876* trabajando a una frecuencia de 4 MHz. En los siguientes temas se irán explicando cada una de las sentencias utilizadas, ahora lo interesante es manejar el entorno de trabajo y no tanto lo que hace cada sentencia.

Lo primero es utilizar el fichero de cabecera donde se especifican las características del microcontrolador *PIC*:

```
#include <16F876.h>
```

Este fichero lo suministra *CCS* y lo incorpora en el directorio de dispositivos (*devices*). El compilador tiene una ruta de búsqueda para los ficheros ***#include***; esta ruta se puede modificar en el caso de querer incluir ficheros que se encuentren en otros directorios. Con el comando *OPTIONS → PROJECTS OPTIONS → INCLUDE FILES* se accede a una ventana (figura 9) donde se puede añadir, eliminar o modificar el orden de búsqueda de los ficheros **#include** (también podemos observar que se pueden configurar los ficheros de trabajo –*FILES*– o los ficheros de salida –*OUTPUT FILES*–).

Figura 9. Ruta de búsqueda de los ficheros #include

A continuación se definen, mediante las correspondientes directivas, la velocidad del *PIC* y el puerto utilizado. Es importante definir la velocidad inmediatamente después del *PIC* ya que muchos *drivers* (como el *LCD*) la necesitan para configurarse.

```
#use delay (clock = 4000000)
# byte puerto_b = 0x06
```

Ahora se puede describir la función principal *MAIN ()*. Los cambios de color, letra, etc., se puede configurar desde la opción *OPTIONS → EDITOR PROPERTIES...*

Al escribir el programa (figura 10) podemos observar como aparece un árbol de funciones a la izquierda de la ventana de programa; esto permite expandir o contraer las funciones y declaraciones de control para optimizar la visualización de los programas más complejos (figura 11).

Figura 10. El programa

Figura 11. Contrayendo el árbol

En el editor de texto se puede pulsar el botón derecho sobre cualquier línea (figura 12); en el caso de los ficheros *#include* permite abrirlos en una pestaña adicional.

Figura 12. Las opciones del botón derecho

Como ayuda para escribir el programa, *CCS* ofrece el comando ***VIEW*** (figura 13) que permite visualizar las interrupciones (*Valid Interrupts*), fusibles de configuración (*Valid Fuses*), hojas de características (*Data Sheet*) y una ventana completa donde se describe el *PIC* (*Device Table Editor*) mediante distintas pestañas (esta opción también es accesible desde la opción ***TOOLS → DEVICE EDITOR*** (ver figura 15).

Figura 13. Comando VIEW

Se puede proceder a la compilación, que se puede hacer con el comando ***COMPILE → COMPILE*** o directamente con la tecla de función **<F9>**. Durante la compilación aparece una ventana donde se informa del proceso de compilación y si hay errores (figura 14). Tras la compilación aparece una ventana con los mensajes de error si los hubiese o el porcentaje de utilización de la memoria *RAM* y *ROM* si la compilación ha sido correcta (figura 15).

NOTA

¡ATENCIÓN! Si se escribe un fichero fuente y a continuación se abre o se crea un segundo fichero fuente, al compilar este último se compilará el primero. Siempre se compila siempre el PRIMER fichero abierto.

Figura 14. Ventana de compilación

Figura 15. Mensajes de compilación

Tras la compilación se obtiene, entre otros, el fichero *HEX* para programar o simular el *PIC*. En ***OPTIONS → PROJECT OPTIONS → OUPUT FILES***, se pueden configurar los ficheros de salida (figura 16).

Figura 16. Ficheros de salida

En la parte izquierda del fichero fuente aparecen unas ventanas auxiliares (*Identifiers, Projects, Files*) en las que se pueden observar la estructura de fichero del programa compilado (figura 17). Haciendo una pulsación en cualquiera de ellos se abre una pestaña con su contenido.

Figura 17. Ventana auxiliar para ficheros

En la barra estándar –figura 18– (para activarla: *OPTIONS → TOOLBAR... → TOOLBARS*, figura 19), también aparecen distintos comandos entre los que se encuentran la visualización de los ficheros de salida.

Figura 18. Barra estándar

Figura 19. Activación barras de herramientas

Hay un fichero de salida especialmente útil para la simulación con el *PROTEUS VSM*, se trata del fichero *RAM SYMBOL MAP* (*.*SYM*) donde aparecen todas las variables de la memoria *RAM* y sus correspondientes direcciones. Por ejemplo, si en un programa existe una variable *FLOAT* llamada *TEMP*, se puede consultar su dirección a través de este fichero (figura 20) para utilizarla en el *WATCH* del *PROTEUS* (figura 21 y figura 22).

Figura 20. Fichero de salida SYM

Figura 21. Configuración del WATCH en el PROTEUS

Figura 22. Ventana de WATCH con la variable

Capítulo 3

La gestión de los puertos

3.1 Introducción

Los microcontroladores *PIC* tienen terminales de entrada/salida divididos en puertos, que se encuentran nombrados alfabéticamente A, B, C, D, etc. Cada puerto puede tener hasta 8 terminales que, de forma básica, se comportan como una entrada/salida digital. Según las características del *PIC*, cada puerto puede tener, además, asignado un bloque funcional: convertidor *AD*, *USART*, *I2C*, etc.

Por ejemplo, en la familia PIC16F87X (figura 1), pueden llegar hasta 5 puertos en el PIC16F877 donde se pueden encontrar bloques de *TIMERS*, *CCP*, *MSSP*, *USART*, *PSP* y convertidores AD.

Características Gama Media	PIC16F873	PIC16F874	PIC16F876	PIC16F877
Frecuencia de trabajo	DC-20MHz	DC-20MHz	DC-20MHz	DC-20MHz
Reset (y delays)	POR,BOR (PWRT,OST)	POR,BOR (PWRT,OST)	POR,BOR (PWRT,OST)	POR,BOR (PWRT,OST)
Memoria de Programa FLASH (palabras de 14-bits)	4K	4K	8K	8K
Memoria Datos (bytes)	192	192	368	368
Memoria Datos EEPROM	128	128	256	256
Interrupciones	13	14	13	14
Puertos E/S	Ports A,B,C	Ports A,B,C,D,E	Ports A,B,C	Ports A,B,C,D,E
Temporizadores	3	3	3	3
Módulos CCP	2	2	2	2
Comunicaciones serie	MSSP,USART	MSSP,USART	MSSP,USART	MSSP,USART
Comunicaciones Paralelo		PSP		PSP
Modulo AD de 10 bits	5 CANALES	8 CANALES	5 CANALES	8 CANALES
Número de Instrucciones	35	35	35	35

Figura 1. Características de la familia PIC16F87X

Considerando a los puertos como entradas/salidas digitales, los puertos se caracterizan por ser independientes, es decir, se puede programar cada terminal del puerto para que se comporte como una entrada o una salida digital (figura 2). La

habilitación como entrada o salida se realiza a través del registro *TRISx* (TRISA: 85h, TRISB: 86h, TRISC: 87h, TRISD: 88h o TRISE: 89h en el *BANCO 1* de la memoria *RAM*).

NOTA
Un valor 0 en estos registros indica que el terminal correspondiente del puerto es de salida, mientras que un valor 1 indica que será de entrada.

La gestión del bus de datos se realiza a través de los registros *PORTx* (PORTA: 05h, PORTB: 06h, PORTC: 07h, PORTD: 08h o PORTE: 09h en el *BANCO 0* de la memoria *RAM*).

Figura 2. Estructura básica de un terminal

También existen algunos terminales que poseen unas características especiales, por ejemplo:

- En el puerto A, el terminal *RA4* tiene salida en drenador abierto lo que obliga a utilizar una resistencia de *pull-up* en el caso de funcionar como salida. Este terminal tiene entrada en *trigger-schmitt* lo que permite su utilización como entrada de contador de eventos externos en conjunción con un modulo temporizador (*TIMER*).
- En el puerto B, los terminales tienen una resistencia de *pull-up* interna que se puede habilitar a través del bit *RBPU* del registro *OPTION_REG* (81h, 181h). Si dicho bit es 1, todas las resistencias de *pull-up* estarán deshabilitadas, si es un 0 estarán habilitadas sólo en el caso de que el terminal funcione como entrada (figura 3).

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
Bit7							Bit0

R=Bit a leer
W=Bit a escribir
U=Bit no implementado se lee como 0
-n=valor para un POR reset

Figura 3. Registro OPTION_REG

Bit 7: **RPBU**: Habilita las resistencias de pull-up.
1=Las deshabilita.
0=Las habilita todas.

Las características eléctricas de los puertos delimitan su utilización para manejar cargas de forma directa.

Máxima corriente de salida a nivel alto por un pin I/O	25 mA
Máxima corriente de salida a nivel bajo por un pin I/O	20 mA
Máxima corriente de salida a nivel alto por el puerto A	80 mA
Máxima corriente de salida a nivel bajo por el puerto A	50 mA
Máxima corriente de salida a nivel alto por el puerto B	150 mA
Máxima corriente de salida a nivel bajo por el puerto B	100 mA

Figura 4. Características eléctricas de los puertos de un PIC16F84

Estos niveles de tensión permiten trabajar con cargas de bajo consumo como *leds*, *displays* de 7 segmentos o *LCD*, pero para activar cargas de mayor consumo es necesaria la utilización de transistores.

3.2 Gestión de puertos en C

En lenguaje C se pueden gestionar los puertos de dos formas:

- Se declaran los registros *TRISX* y *PORTX* definiendo su posición en la memoria *RAM* como variables de C.
- Utilizando las directivas específicas del compilador (*#USE FAST_IO*, *#USE FIXED_IO*, *#USE STANDARD_IO*).

3.2.1 A través de la RAM

Se definen los registros *PORTx* y *TRISx* como bytes y se sitúan en la posición correspondiente de la memoria *RAM*. La directiva utilizada de C es *#BYTE*:

#BYTE variable=constante;

```
#BYTE TRISA = 0x85          // Variable TRISA en 85h.
#BYTE PORTA = 0x05          // Variable PORTA en 05h.
#BYTE TRISB = 0x86          // Variable TRISB en 86h.
```

```
#BYTE PORTB = 0x06          // Variable PORTB en 06h.
#BYTE TRISC = 0x87          // Variable TRISC en 87h.
#BYTE PORTC = 0x07          // Variable PORTC en 07h.
.......
```

Una vez definidas estas variables se pueden configurar y controlar los puertos a través de los comandos de asignación.

```
TRISA = 0xFF;          // 8 terminales de entrada
TRISB = 0x00;          // 8 terminales de salida
TRISC = 0x0F;          // 4 terminales de mayor peso de salida, 4 terminales de
                       // menor peso de entrada
```

Escritura en los puertos:

```
PORTC = 0x0A;          // salida del datos 00001010 por el puerto C
```

Lectura de puertos:

```
valor = PORTA;         // Asigna el dato del puerto A a la variable valor.
```

Manejo de sentencias:

```
TRISD=0x0F;
if (PORTD & 0x0F) PORTD |= 0xA0;      // comprueba los 4 terminales de
                                      // menor peso del puerto D y si son
                                      // 1111 saca por los 4 terminales de
                                      // mayor peso el dato 1010.
```

Existen unas funciones de C que permiten trabajar bit a bit con los registros o variables definidas previamente. Estas funciones son las siguientes:

bit_clear (var,bit); // Pone a 0 el bit específico (0 a 7) de la variable.

bit_set (var , bit); // Pone a 1 el bit específico (0 a 7) de la variable.

bit_test (var , bit); // Muestra el bit especifico (0 a 7) de la variable.

swap (var); // Intercambia los 4 bits de mayor peso por los 4 de // menor peso de la variable

```
bit_set (PORTC , 4);                    // "saca" un 1 por el terminal RC4
if (bit_test(PORTB,0)==1) bit_clear(PORTB,1);   //si RB0 es 1 borra RB1
```

Se puede declarar un bit de un registro con una variable mediante la directiva ***#BIT***, lo que permite trabajar directamente con el terminal:

#BIT nombre = posición.bit

```
#BIT RB4 = 0x06.4      // PORTA=0x06
RB4 = 0;
```

Ejemplo 1: Se configuran los terminales *RB1* como salida y el *RB0* como entrada (con resistencia de *pull-up*). La salida debe tener el mismo valor que la entrada. Se utiliza un interruptor en la entrada y un *led* en la salida (figura 5). Componentes *ISIS: PIC16F876, RES, LED-BLUE* y *SW-SPST-MOM*.

Figura 5. El esquema del ejemplo 1

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay( clock = 4000000 )          // Reloj de 4 MHz
#BYTE TRISB = 0x86              // TRISB en 86h.
#BYTE PORTB = 0x06              // PORTB en 06h.
#BYTE OPTION_REG = 0x81         // OPTION_REG en 81h.
void main() {
        bit_clear(OPTION_REG,7);        // Habilitación Pull-up
        bit_set(TRISB,0);               // B0 como entrada
        bit_clear(TRISB,1);             // B1 como salida
        bit_clear(PORTB,1);             // Apaga el LED
        while (1) {
           if (bit_test(portb,0) == 1 )        // Si RB0 es 1, apaga el LED
                 bit_clear(portb,1);
           else
                 bit_set(portb,1);             // Si RB0 = 0, enciende el LED
        }
}
```

Figura 6. El programa del ejemplo 1

3.2.2 A través de las directivas

El compilador ofrece funciones predefinidas para trabajar con los puertos. Estas funciones son:

output_X (valor);	// Por el puerto correspondiente saca el valor (0-255).
input_X();	// Se obtiene el valor en el puerto correspondiente.
set_tris_X(valor);	// Carga el registro *TRISx* con el valor (0-255).
port_b_pullups (valor);	// Mediante valor = *TRUE* o valor = *FALSE* habilita // o deshabilita las resistencias de *pull-up* en *PORTB*.
get_trisX()	// Devuelve el valor del registro *TRISx*

Donde la X es la inicial del puerto correspondiente (A, B, C,...).

```
Output_A(0xFF);      // Saca por el puerto A el valor 11111
Valor=Input_B();     // Lee el valor del puerto B
Set_tris_C(0x0F);    // Configura el puerto C: C0-C3 entradas, C4-C7 salidas
```

Existen una serie de funciones asociadas a un terminal o pin*. El parámetro pin* se define en un fichero *include* (por ejemplo, 16F876.h) con un formato del tipo ***PIN_Xn***, donde X es el puerto y n es el número de pin.

#define PIN_A0 40

#define PIN_A1 41

Las funciones son:

output_low (pin*);	// Pin a 0.
output_high (pin*);	// Pin a 1.
output_bit (pin* , valor);	// Pin al valor especificado.
output_toggle(pin*);	// Complementa el valor del pin.
output_float (pin*);	// Pin de entrada, quedando a tensión flotante... // (simula salida en drenador abierto)
input_state(pin*);	// Lee el valor del pin sin cambiar el sentido del // terminal.
input(pin*);	// Lee el valor del pin.

Las funciones ***output_x()*** e ***input_x()*** dependen de la directiva tipo ***#USE *_IO*** que esté activa. Directivas:

#USE FAST_IO (PUERTO) [PUERTO: A...]

Con la función *output_x()* se saca el valor al puerto y con la función *input_x()* se lee el puerto. La directiva no modifica previamente el registro *TRIS* correspondiente.

Hay que asegurarse de que los registros *TRIS* están correctamente definidos. Entonces, el ejemplo 1 quedaría :

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay( clock = 4000000 )
#use fast_io(B) ←
void main() {
        port_b_pullups (TRUE);
        set_tris_B(0x01);
        output_low(PIN_B1);
   while (1)
   {
   if (input(PIN_B0) == 1 )
     output_low(PIN_B1);
   else
     output_high(PIN_B1);
   }
}
```

#USE STANDARD_IO (PUERTO) [PUERTO: A...]

Con la función *output_x()* el compilador se asegura de que el terminal, o terminales correspondientes, sean de salida mediante la modificación del *TRIS* correspondiente. Con la función *input_x()* ocurre lo mismo pero asegurando el terminal (terminales) como entrada. Es la directiva por defecto. Entonces, el ejemplo 1 quedaría:

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay( clock = 4000000 )
#use standard_io(B) ←
void main() {
    port_b_pullups (TRUE);
   output_low(PIN_B1);
   while (1)
   {
   if (input(PIN_B0) == 1 )
     output_low(PIN_B1);
   else
     output_high(PIN_B1);
   }
}
```

#USE FIXED_IO (PUERTO_OUTPUTS=pin* , ...) [PUERTO: A...]

El compilador se encarga de generar el código para definir los puertos de acuerdo con la información que indica la directiva (donde sólo se indican los terminales de

salida), sin tener en cuenta si la operación es de entrada o de salida. Entonces, el ejemplo 1 quedaría:

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay( clock = 4000000 )
#use fixed_io(b_outputs=pin_b1) ←
void main() {
   port_b_pullups (TRUE);
   output_low(PIN_B1);
   while (1)
   {
   if (input(PIN_B0) == 1 )
     output_low(PIN_B1);
   else
     output_high(PIN_B1);
   }
}
```

3.2.3 Con punteros

En C se puede acceder a la memoria de datos mediante punteros. Los punteros se deben definir como *INT*:

```
#define TRISA (int*) 0x85
#define PORTA (int*) 0x05
```

El registro es manejado mediante la utilización del operando *:

```
int valor
valor = *porta
```

Los terminales se pueden leer o escribir utilizando operadores lógicos:

```
*porta |= 0b00000001;          // RA0 = 1
*porta &= 0b11111101;          // RA2 = 0
if (*porta & 0b00000001)...    // Lee el valor de RA0
```

Entonces, el ejemplo 1 quedaría:

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay( clock = 4000000 )
#define TRISB (int*) 0x86 ←
#define PORTB (int*) 0x06 ←
#define OPTION (int*) 0x81 ←
```

```
void main() {
   *option &= 0b01111111;        // Pone a 0 el bit 7 del OPTION_REG
                                 // (pull-up habilitado)
   *trisb = 0x01;                // RB0 entrada, RB1 salida
   *portb = 0x00;                // Apaga el LED
   while (1)
   {
   if (*portb & 0x01 )           // Lee el RB0 y si es 1..
      *portb=0x00;               // Apaga el LED (RB1 = 0)
   else
      *portb=0x02;               // Si es 0 enciende el LED (RB1 = 1)
   }
}
```

Ejemplo 2: Realizar un contador de 0 a 99 con un doble *display* de 7 segmentos de cátodo común. La cuenta debe ser continua y de 0 a 9 el digito de las decenas debe estar apagado. Componentes *ISIS*: *PIC16F876, RX8* y *7SEG-MPX2-CA-BLUE*.

Figura 7. El esquema del ejemplo 2

```
#include <16F876.h>
#USE DELAY(CLOCK = 4000000)
#FUSES XT,NOWDT,NOPROTECT,NOPUT
#USE fast_IO (B)
#USE fast_IO (A)
byte CONST DISPLAY[10] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f};
```

```
main(){
    byte ud=0,dec=0;
    SET_TRIS_B(0x00);
    SET_TRIS_A(0x00);
    OUTPUT_B(0);
    for( ;; ) {
        for (dec=0;dec<10;dec++)        {        // Cuenta digito decenas
            for (ud=0;ud<10;ud++) {
                OUTPUT_A(0x02);          // cat_D = apagado, cat_U = encendido
                OUTPUT_B(DISPLAY[ud]);  // Digito unidades
                delay_ms(50);                    // Para evitar parpadeos
                if (dec==0) output_a(0x03);      // Si decenas = 0,
                                                 // cat_D = apagado
                else output_a(0x01);             // Si decenas > 0,
                                                 // cat_D = encendido
                OUTPUT_B(DISPLAY[dec]);          // Digito decenas
                delay_ms(50);                    // Para evitar parpadeos
            }
        }
    }
}
```

Figura 8. El programa del ejemplo 2

Los terminales de los dos *displays* son comunes por lo que el dato es común; para que aparezca el digito sólo en las unidades, o sólo en las decenas, se debe apagar el otro *display* mediante el terminal de cátodo. Es decir, si se desea visualizar las unidades se pasa el código "10" al *display* y si son las decenas se pasa el "01" (con un 1 el *display* está apagado y con un 0 está encendido). La alternancia entre los dos cátodos debe ser tan rápida que el ojo no se de cuenta del parpadeo. En el caso que las decenas sean cero, su *display* se apagará.

CONST DISPLAY[10] = {0x3f,0x06,0x5b,0x4f,0x66,0x6d,0x7d,0x07,0x7f,0x6f} permite visualizar de 0 a 9 mediante la combinación de dígitos típica de los *displays* de 7 segmentos (figura 9). Por ejemplo, en el 0 se encienden **a, b, c, d, e** y **f**, lo que significa 111111, 0x3F en hexadecimal.

Figura 9. Los 7 segmentos del display

3.3 Entradas y salidas

3.3.1 LCD

Se acostumbran ha utilizar *LCD* del tipo HD44780, con un número de líneas variable y un número de caracteres por línea también variable (por ejemplo, con 2 x 16 se trabaja con dos líneas de 16 caracteres cada una) (ver figura 10).

Figura 10. El patillaje de un LCD típico

El bus de datos es de 8 bits, aunque también existe la posibilidad de trabajar con 4 bits (con un menor número de caracteres). El compilador C incluye un fichero (*driver*) que permite trabajar con un *LCD*. El archivo es *LCD.C* y debe llamarse como un *#include*. Este archivo dispone de varias funciones ya definidas:

lcd_init ();

Es la primera función que debe ser llamada.

Borra el LCD y lo configura en el formato de 4 bits, con dos lineas y con caracteres de 5 x 8 puntos, en modo encendido, cursor apagado y sin parpadeo.

Configura el LCD con un autoincremento del puntero de direcciones y sin desplazamiento del *display* real.

lcd_gotoxy (byte x , byte y);

Indica la posición de acceso al LCD. Por ejemplo, (1,1) indica la primera posición de la primera linea y (1,2) indica la primera posición de la segunda línea.

lcd_getc (byte x , byte y);

Lee el carácter de la posición (x,y).

lcd_putc (char s);

S es una variable de tipo *char*. Esta función escribe la variable en la posición correspondiente. Si, además, se indica:

\f se limpia el LCD.

\n el cursor va a la posición (1,2).

\b el cursor retrocede una posición.

El compilador de C ofrece una función más versátil para trabajar con el *LCD*:

printf (string)

printf (cstring, values...)

printf (fname, cstring, values...)

String es una cadena o un array de caracteres, *values* es una lista de variables separadas por comas y *fname* es una función.

El formato es **%nt**, donde **n** es opcional y puede ser:

1-9: para especificar cuantos caracteres se deben especificar.

01-09: para indicar la cantidad de ceros a la izquierda.

1.1-9.9 para coma flotante.

t puede indicar:

c	Carácter.
s	Cadena o carácter.
u	Entero sin signo.
d	Entero con signo.
Lu	Entero largo sin signo.
Ld	Entero largo con signo.
x	Entero Hexadecimal (minúsculas).
X	Entero Hexadecimal (mayúsculas).
Lx	Entero largo Hexadecimal (minúsculas).
LX	Entero largo Hexadecimal (mayúsculas).
f	Flotante con truncado.
g	Flotante con redondeo.
e	Flotante en formato exponencial.
w	Entero sin signo con decimales insertados. La 1ª cifra indica el total, la 2ª el número de decimales.

A continuación, mostramos unos ejemplos de los distintos formatos:

Formato	Valor = 0x12	Valor = 0xFE
%03u	018	254
%u	18	254
%2u	18	*
%5	18	254
%d	18	-2
%x	12	fe
%X	12	FE
%4X	0012	00FE
%3.1w	1.8	25.4

Ahora, mostramos una serie de ejemplos de aplicación:

```
byte x,y,z;
printf("Hola");
printf("Valor=>%2x\n\r",get_rtcc());
printf("%2u %X %4X\n\r",x,y,z);
printf(LCD_PUTC, "n=%u",n);
```

El *driver LCD.C* está pensado para trabajar con el *PORTD* o el *PORTB*. Por defecto, utiliza el *PORTD* a menos que le indiquemos lo contrario mediante:

#define use_portb_lcd TRUE, se comenta o descomentatal como se indica en la figura 11.

Por defecto, este *driver* usa siete terminales para la comunicación entre el *LCD* y el *PIC*. En la figura 11 se observa parte del fichero *LCD.C* donde se encuentran definidas las conexiones utilizadas y la posibilidad de cambiar de puerto.

```
// As defined in the following structure the pin connection is as follows:
//      D0  enable
//      D1  rs
//      D2  rw
//      D4  D4
//      D5  D5
//      D6  D6
//      D7  D7
//
// LCD pins D0-D3 are not used and PIC D3 is not used.
// Un-comment the following define to use port B
// #define use_portb_lcd TRUE
```

Figura 11. Extracto del fichero LCD.C

Se puede trabajar con otros puertos, por ejemplo el *PORTC*, modificando el fichero *LCD.C*. En la figura 12 se muestra parte del fichero *LCD.C* donde se definen los puertos de trabajo (el D o el B); modificando estas sentencias se puede trabajar con otro puerto (A, C, etc.).

```
#if defined use_portb_lcd
   #locate lcd = getenv("sfr:PORTB")   // This puts the entire structure over
                                       // the port
   #define set_tris_lcd(x) set_tris_b(x)
#else
   #locate lcd = getenv("sfr:PORTD")   // This puts the entire structure over
                                       // the port
   #define set_tris_lcd(x) set_tris_d(x)
#endif
```

Figura 12. Extracto del fichero LCD.C

Ejemplo 3: Realizar un menú de control mediante un pulsador. El programa debe mostrar un menú de 3 funciones. Mediante el pulsador se debe seleccionar uno de los 3 elementos y con otro ejecutar la función (en este caso encender un *led*). Disponemos de los componentes *ISIS*: *PIC16F876, RES, BUTTON* y *LM016L*.

Figura 13. El esquema del ejemplo 3

```
#include <16F876.h>
#fuses XT,NOWDT
#use delay(clock= 4000000)
#include <lcd.c>
#use standard_io(C)
#use standard_io(A)

  enum funciones {med,cal,ini};        // Asigna un  valor a cada elemento
                                       // med = 0, cal = 1 e ini = 2

void medir(void){                              // Función de medir
  //algoritmo correspondiente
  output_toggle(pin_C0);
 }

void calibrar(void){                           //Funcion de calibrar
  //algoritmo correspondiente
 output_toggle(pin_C1);
 }

void inicializar(void){                        // Funcion de inicializar
  //algoritmo correspondiente
 output_toggle(pin_C2);
 }


void run_func(int numfunc){            // Asignación de la función a realizar
                                       // viene dada por la variable "item"
   switch(numfunc) {
      case med:
         medir();
         break;
      case cal:
         calibrar();
         break;
      case ini:
         inicializar();
         break;
   }


void main() {
   char item;                          // Variables de funciones
```

```
   char n_menus = 3;                    // Número de funciones

//   bit_set(TRISA,0);
   lcd_init();

  while (1) {
   if (input(PIN_A0) == 1) {            // Detecta botón de selección
       item++;                          // Si pulsa aumenta la variable
       delay_ms(300);                   // Para evitar rebotes
       lcd_putc('\f');      }

   if (item > (n_menus-1)) {  // Si la variable supera el número de...
      item = 0; }             // funciones la inicializa

   switch (item) {

      case 0:
        lcd_gotoxy(1,1);
        printf(lcd_putc, "MEDIR");
        lcd_gotoxy(1,1);
        break;

      case 1:
        printf(lcd_putc, "CALIBRAR");
        lcd_gotoxy(1,1);
        break;

      case 2:
        printf(lcd_putc, "INICIALIZAR");
        lcd_gotoxy(1,1);
        break;
   }
  if (input(PIN_A1) == 1)      // Si se pulsa el botón de selección...
     {delay_ms(200);
     run_func(item);}          // se llama a la función correspondiente
  }
}
```

Figura 14. El programa del ejemplo 3

3.3.2 LCD gráfico

Se puede utilizar un LCD gráfico con una controladora KS0108 (como el de la figura 15), por ejemplo la *ASI-A-1286AS-LJ-EWS/W* de la casa ALL SHORE INDUSTRIES.

Figura 15. LCD gráfico con controladora KS0108

La distribución de patillas es la siguiente:

VSS: masa.

VDD: alimentación.

Vo: tensión de contraste.

D/I: entrada de datos/entrada de códigos de instrucción.

R/W: lectura/escritura.

E: enable.

DB0..DB7: datos de entrada.

CS1..CS2: chip select.

RST: reset.

El compilador C suministra varios *drivers* para este tipo de *LCD* gráficos, el *GLCD.C*, *GRAPHICS.C* o, el más específico, *HDM64GS12.C*.

La conexión definida en estos ficheros es la siguiente:

```
////////////////////////////////////////////////////////////////////////////
////                                                                    ////
//// LCD Pin connections:                                               ////
//// (These can be changed as needed in the following defines).         ////
//// VSS is connected to GND                                            ////
//// VDD is connected to +5V                                            ////
//// V0 - LCD operating voltage (Constrast adjustment)                  ////
//// D/I - Data or Instruction is connected to B2                       ////
//// R/W - Read or Write is connected to B4                             ////
//// Enable is connected to B5                                          ////
//// Data Bus 0 to 7 is connected to port d                             ////
```

```
//// Chip Select 1 is connected to B0                                  ////
//// Chip Select 2 is connected to B1                                  ////
//// Reset is connected to C0                                          ////
//// Negative voltage is also connected to the 20k Ohm POT             ////
//// Positive voltage for LED backlight is connected to +5V            ////
//// Negavtive voltage for LED backlight is connected to GND           ////
////                                                                   ////
////////////////////////////////////////////////////////////////////////////////
```

Las funciones definidas son:

gcd_init(mode)

Debe ser la primera función en invocarse. Enciende el *LCD*.

glcd_pixel(x,y,color)

Establece el color del píxel. Puede activarse o desactivarse.

glcd_fillScreen(color)

Rellena el *LCD* de un color determinado. Puede activarse o desactivarse.

glcd_update()

Escribe en la RAM del *LCD*; sólo es posible si está definido *FAST_GLCD*.

glcd_line(x1, y1, x2, y2, color)

Dibuja una línea desde el primer punto al segundo asignando color, el cual, a su vez, puede activarse o desactivarse.

glcd_rect(x1, y1, x2, y2, fill, color)

Dibuja un rectángulo con un vértice en (x1, y1) y el otro en (x2, y2). Puede ser rellenado o no y puede activarse un color o no.

glcd_bar(x1, y1, x2, y2, width, color)

Dibuja una barra desde el primer punto al segundo; se puede definir el número del rango de píxeles y puede activarse el color o no.

glcd_circle(x, y, radius, fill, color)

Dibuja un circulo con centro en (x, y) y con el radio especificado; puede rellenarse o no y puede activarse el color o no.

glcd_text57(x, y, textptr, size, color)

Escribe el texto empezando en (x, y); los caracteres son de 5 x 7 píxeles; se puede escalar el tamaño y puede activarse el color o no. Esta función envía los caracteres a la línea siguiente (se debe usar *#define GLCD_WIDTH* para definir el ancho de visualización).

Ejemplo 4: Visualizar en un *LCD* el estado de las entradas del *PUERTOA* (ver figura 16). Disponemos de los componentes *ISIS*: *PIC16F877, RESPACK8, LGM12641-BS1R* y *SW-SPST-MOM*.

Figura 16. El esquema del ejemplo 4

```
#include <16F877.h>
#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay(clock=20000000)
#include <HDM64GS12.c>
#include <graphics.c>
#use standard_io(a)

void main() {
   CHAR A5[ ]="A5";
   CHAR A4[ ]="A4";
   CHAR A3[ ]="A3";
   CHAR A2[ ]="A2";
   CHAR A1[ ]="A1";
   CHAR A0[ ]="A0";
   CHAR IN[ ]="PUERTO A";
   glcd_init(ON);
   glcd_text57(33, 30,A5, 1, 1);
   glcd_text57(49, 30,A4, 1, 1);
   glcd_text57(65, 30,A3, 1, 1);
   glcd_text57(81, 30,A2, 1, 1);
```

```
    glcd_text57(97, 30,A1, 1, 1);
    glcd_text57(113, 30,A0, 1, 1);
    glcd_text57(30,5,IN, 2, 1);

while(1){
    if (input_state(PIN_A5)==0)
        glcd_rect(32,40,46,60,1,1);
    else
        glcd_rect(32,40,46,60,1,0);
    glcd_rect(32,40,46,60,0,1);

    if (input_state(PIN_A4)==0)
        glcd_rect(48,40,62,60,1,1);
    else
        glcd_rect(48,40,62,60,1,0);
    glcd_rect(48,40,62,60,0,1);

    if (input_state(PIN_A3)==0)
        glcd_rect(64,40,78,60,1,1);
    else
        glcd_rect(64,40,78,60,1,0);
    glcd_rect(64,40,78,60,0,1);

    if (input_state(PIN_A2)==0)
        glcd_rect(80,40,94,60,1,1);
    else
        glcd_rect(80,40,94,60,1,0);
    glcd_rect(80,40,94,60,0,1);

    if (input_state(PIN_A1)==0)
        glcd_rect(96,40,110,60,1,1);
    else
        glcd_rect(96,40,110,60,1,0);
    glcd_rect(96,40,110,60,0,1);

    if (input_state(PIN_A0)==0)
        glcd_rect(112,40,126,60,1,1);
    else
        glcd_rect(112,40,126,60,1,0);
    glcd_rect(112,40,126,60,0,1);
    delay_ms(400);
    }
}
```

Figura 17. El programa correspondiente al ejemplo 4

3.3.3 Teclado (keypad 3x4)

Las entradas a través de un pulsador son muy habituales en los sistemas con microcontroladores para trabajar con una mayor información o información alfanumérica. Por ejemplo, se utilizan los teclados matriciales de 1x4, 3x4 o 4x4 (ver figura 18).

Figura 18. Diferentes tipos de teclados matriciales

El compilador de C incluye el *driver KBD.C* para manejar el teclado (3x4). Las funciones que incorporan son las siguientes:

kbd_init()

Inicializa el sistema, debe ser la primera función en el programa.

kbd_getc()

Devuelve el valor de la tecla pulsada en función de la tabla que tiene programada (ver figura 19).

```
// Keypad layout:
char const KEYS[4][3] = {{'1','2','3'},
                         {'4','5','6'},
                         {'7','8','9'},
                         {'*','0','#'}};
```

Figura 19. La definición de teclas en el archivo KBD.C

A través de la modificación de esta tabla podemos adecuar el resultado del programa a las distintas carátulas del teclado.

El archivo *KBD.C* está pensado para trabajar con el *PORTB* o el *PORTD* (ver figura 20). Activando o no la línea *#define use_portb_lcd TRUE* podemos seleccionar el *PORTB* (ver figura 21).

```
#if defined use_portb_kbd
   #byte kbd = 6
#else
   #byte kbd = 8
#endif
#endif

#if defined use_portb_kbd
   #define set_tris_kbd(x) set_tris_b(x)
#else
   #define set_tris_kbd(x) set_tris_d(x)
#endif
```

Figura 20. La configuración de puertos

```
//////////////////// the following defines the keypad layout on port B

// Uncomment the following define to use port B
// #define use_portb_kbd TRUE

// Make sure the port used has pull up resistors (on the LCD) on
// the ribbon cable.
```

Figura 21. La selección del PORTB

Las conexiones vienen dadas en el fichero pero se pueden modificar:

```
#define COL0 (1 << 5)
#define COL1 (1 << 6)
#define COL2 (1 << 7)

#define ROW0 (1 << 1)
#define ROW1 (1 << 2)
#define ROW2 (1 << 3)
#define ROW3 (1 << 4)
```

Figura 22. La asignación de patillas

Se puede trabajar con el *LCD* y el teclado a través de puertos separados o compartiendo el mismo puerto (con el ahorro de patillas que conlleva) (ver figuras 23 y 24). Compartir puerto suele conllevar problemas si se quiere trabajar con el teclado y las interrupciones *RB4/RB7*.

Figura 23. El LCD y el teclado en puertos distintos

Figura 24. El LCD y el teclado compartiendo los mismos puertos

Ejemplo 5: Introducir datos por el teclado y visualizarlos en el *LCD*. Cuando se pulsa la tecla "*" borrar el *LCD* (ver figura 24). Se dispone de los siguientes componentes *ISIS*: *PIC18F876, LM016L* y *KEYPAD-PHONE*.

```
#include <16F876.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#use delay(clock= 4000000)
#include <lcd.c>
#include <kbd.c>
```

```
void main() {
   char k;
   int  x;
   lcd_init();
   kbd_init();
   port_b_pullups(TRUE);
   lcd_putc("\fListo...\n");

   while (TRUE) {
      k=kbd_getc();
      x=k-48;                                  // Conversión numérica
      if(k!=0) {
        if(k=='*')
          lcd_putc('\f');
        else
           lcd_putc(k);                        // Imprime carácter
           delay_ms(500);

           printf(lcd_putc,"\f Car=%c",k);     // Imprime carácter
           delay_ms(500);

           printf(lcd_putc,"\f Car=%u",k);     // Imprime valor ASCII
           delay_ms(500);

           printf(lcd_putc,"\f Num=%u",x);     // Imprime valor numérico
           delay_ms(500);
           lcd_putc("\fListo...\n");
     }
   }
}
```

Figura 25. El programa del ejemplo 5

Debemos observar que el valor leído en el teclado y el visualizado en el *LCD* es un carácter ASCII. Si deseamos convertirlo a su valor numérico correspondiente deberemos restarle el valor 48 (30 en hexadecimal); esto se debe a que el carácter 0 en ASCII es 30h, el 1 es 31h, etc.

Ejemplo 6: Diseñar un sistema básico para el control de accesos; a través de un teclado de 3x4 introducir una clave de 3 dígitos que cuando sea correcta abra una puerta (con un pulso a un relé) y lo indique en una pantalla de *LCD*. Guardar la clave de acceso en la memoria *EEPROM* (figura 26). Se dispone de los siguientes componentes *ISIS*: *PIC18F876, KEYPAD-PHONE, RES, BD135, CELL y RELAY.*

Figura 26. El esquema del ejemplo 6

```
#include <16F876.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#use delay(clock= 4000000)
#use standard_io(a)

#include <lcd.c>
#include <kbd.c>
#include <stdlib.h>
#rom 0x2100={'7','2','3'}   // Posición 0,1 y 2 de la Eeprom con los datos…
                            // 7,2, y 3 respectivamente
void main() {
   char k;
   int i;
   char data[3], clave[3];  // Matrices para guardar clave y datos
   lcd_init();
   kbd_init();
   port_b_pullups(TRUE);
   while (TRUE) {
   i=0;              // posición de la matriz
   printf(lcd_putc,"\fpulsar tecla 1\n");  // Para primer dato
   while(i<=2){                  // Para tres datos
      k=kbd_getc();              // Lee el teclado
```

```
      if (k!=0)                  // Si se ha pulsado alguna tecla
         {data[i]=k;             // se guarda en la posición correspondiente
          i++;                   // de la matriz
          printf(lcd_putc,"\fpulsar tecla %u\n",i+1); // Siguiente dato
         }
    }
   for (i=0;i<=2;i++) {          // Pasa datos de eeprom a la matriz clave
     clave[i]=read_eeprom(i);}
   if ((data[0]==clave[0])&&(data[1]==clave[1])&&(data[2]==clave[2]))
      { printf(lcd_putc,"\fPuerta Abierta"); // Compara los datos y la clave
        output_high(PIN_A0);                 // Si es igual da pulso al relé
        delay_ms(500);
        output_low(PIN_A0);}
   else printf(lcd_putc,"\fPuerta Cerrada");  // Clave errónea
   delay_ms(1000);
   }
}
```

Figura 27.- Programa del Ejemplo 6

Ejemplo 7: Introducir los datos, a través de un teclado, de velocidad de un motor y generar una señal modulada en ancho de pulso proporcional al dato de la velocidad (figura 28). Se dispone de los siguientes componentes *ISIS*: *PIC18F876, KEYPAD-PHONE, RES, 2SK1058, CELL* y *MOTOR*.

Figura 28. El esquema del ejemplo 7

Se genera una señal modulada en ancho de pulso *PWM* (sin utilizar el modulo *CCP* de los *PIC*) donde el semiperiodo de señal a nivel alto está fijado por el valor introducido en el teclado. Para ello, se utiliza un registro de 8 bits para fijar el semiperiodo a nivel alto (*PWMH*) y el semiperiodo a nivel bajo (*PWML*) (ver la figura 29). Como el valor máximo del registro es 255, este debe coincidir con el valor máximo del teclado, es decir 9; por lo tanto, la relación entre el valor del teclado y el semiperiodo *PWMH* será:

PWMH= (255/9) x Tecla, aproximadamente **PWMH=28*Tecla.**

Figura 29. La modulación PWM

Según dicha ecuación cuando el valor de teclado sea 0 la salida será 0 de forma continua y el motor estará parado. Cuando el valor de teclado sea 9, el semiperiodo *PWMH* será de 252 (no llega a 255) y el motor estará casi a toda marcha. El semiperiodo a nivel bajo *PWML* se obtiene de restar el *PWMH* a 255.

```
#include <16f676.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#USE DELAY (CLOCK=4000000)
#INCLUDE <kbd.c>
#USE STANDARD_IO (a)

VOID MAIN()
{
   CHAR k,kant='0';      // k valor de teclado, k valor anterior de teclado
   char PWMH=0,PWML=0;   // Semiperiodo alto y bajo
   kbd_init();
   PORT_B_PULLUPS(TRUE);

WHILE (1) {                // Bucle infinito (siempre consulta el teclado)
      k=kbd_getc();        // Lee en ASCII el valor de la tecla pulsada
      if (k=='\0') k=kant;            // Si no se pulsa tecla (\0) se usa
                                      // el valor anterior
      if ((k=='*') || (k=='#')) k='0';        // Si se pulsa * o # se asigna
                                              // un valor cero.
```

```
    kant=k;                    // Se guarda tecla pulsada
     k=k-48;                   // Se convierte de ASCII a valor numérico
     PWMH=k*28;        // Proporción entre valor tecla y semiperiodo Alto.
     PWML=255-PWMH;            // Semiperiodo Bajo
    for(PWMH;PWMH>0;PWMH--){          // Obtención de la salida a nivel alto
     OUTPUT_HIGH(PIN_A0);}
    for(PWML;PWML>0;PWML--){          // Obtención de la salida a nivel bajo
    OUTPUT_LOW(PIN_A0);}
  }
}
```

Figura 30. El programa del ejemplo 7

Figura 31. La señal obtenida para el valor 9

Capítulo 4

Las interrupciones y los temporizadores

4.1 Introducción

Las interrupciones permiten a cualquier suceso interior o exterior interrumpir la ejecución del programa principal en cualquier momento. En el momento de producirse la interrupción, el *PIC* ejecuta un salto a la rutina de atención a la interrupción, previamente definida por el programador, donde se atenderá a la demanda de la interrupción. Cuando se termina de ejecutar dicha rutina, el *PIC* retorna a la ejecución del programa principal en la misma posición de la memoria de programa donde se produjo la interrupción.

El manejo de interrupciones permite realizar programas que no tienen que estar continuamente consultando sucesos internos o externos mediante técnicas de consulta o *polling*, las cuales provocan retardos o paradas en la ejecución del programa principal.

Los *TIMER* o temporizadores son módulos integrados en el *PIC* que permite realizar cuentas tanto de eventos internos como externos. Cuando la cuenta es interna se habla de temporización y cuando la cuenta es externa se habla de contador. Los *timers* están íntimamente ligados al uso de las interrupciones, pero no por ello se utilizan siempre de forma conjunta.

4.2 Interrupciones

Al producirse una interrupción, el *PIC* salta automáticamente a la dirección del vector de interrupción de la memoria de programa y ejecuta la porción de programa, correspondiente a la atención de la interrupción, hasta encontrar la instrucción *RETFIE*. Al encontrar dicha instrucción, abandona la interrupción y retorna a la

posición de memoria del programa principal desde la que saltó al producirse la interrupción.

Las fuentes de interrupción dependen del *PIC* utilizado. Por ejemplo, el *PIC16F84* tiene 4 fuentes de interrupción y la familia *PIC16F87X* tiene entre 13 y 14.

Los *PIC* de gama baja y media tienen un único vector de interrupción situado en la dirección 04h de programa (figura 1), mientras que los de gama alta tienen dos vectores de interrupción de distinta prioridad, alta y baja, situados en la posición 08h y 18h de la memoria de programa.

Figura 1. Memoria de programa (posición del vector de interrupción)

Al poseer un único vector de interrupción (dos en la gama alta), el *PIC* posee unos registros de control donde mediante la utilización de banderas, o *flags*, el usuario puede determinar que es lo que ha producido la interrupción; además, en estos registros, se pueden habilitar o no las distintas fuentes de interrupción (máscaras de interrupción) e, incluso, permite una habilitación general.

Cuando la habilitación general está activa y algunas, o todas, la particulares también lo están, los *flags* se activan en el caso de producirse algunas de las interrupciones, de tal manera que el programador puede, mediante el testeo de dichos *flags*, actuar de la forma más adecuada.

La familia *PIC16F87X* tiene hasta 14 fuentes de interrupción. Posee un registro de control, ***INTCON*** (figura 2), que permite la habilitación de interrupciones y el manejo de los *flags*. La habilitación general se activa mediante el bit *GIE* (*INTCON<7>*), el cual es desactivado en el reset; por lo tanto, hay que habilitarlo por programa. Existen 4 registros adicionales para la gestión de las interrupciones: ***PIR1, PIR2, PIE1*** y ***PIE2***.

Cuando se responde a una interrupción, el bit *GIE* es inhabilitado para evitar interrupciones sucesivas. La dirección de retorno del programa principal se almacena en la pila y el contador de programa se carga con la dirección 0004h. Una vez en la rutina de atención a la interrupción se puede determinar la fuente de la interrupción mediante el testeo de los diferentes *flags*. Los *flags* activos deben ser "borrados" antes de abandonar la rutina de interrupción para evitar reentradas erróneas.

Registro ITCON (dirección RAM: 0Bh/8Bh/10Bh/18Bh) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
Bit7							Bit0

Figura 2. Registro INTCON

bit 7: *GIE*: Bit de habilitación global de las interrupciones.

1 = Habilita todas las interrupciones no enmascararbles.

0 = Las deshabilita.

bit 6: *PEIE*: Bit de habilitación de las interrupciones de periféricos.

1 = Habilita todas las interrupciones no enmascarables de periféricos.

0 = Las deshabilita.

bit 5: *T0IE*: Bit de habilitación de la interrupción por desbordamiento del *TMR0*:

1 = Habilita la interrupción del *TMR0*.

0 = La deshabilita.

bit 4: *INTE*: Bit de habilitación de la interrupción externa *RB0/INT*.

1 = Habilita la interrupción del *RB0/INT*.

0 = La deshabilita.

bit 3: *RBIE*: Bit de habilitación de la interrupción por cambio en el *PORTB*.

1 = Habilita la interrupción del *PORTB*.

0 = La deshabilita.

bit 2: *T0IF*: Bit de *flag* de la interrupción del *TMR0*.

1 = El registro del *TMR0* se ha desbordado (debe borrarse por software).

0 = El registro del *TMR0* no se ha desbordado.

bit 1: *INTF*: Bit de *flag* de la interrupción del *RB0/INT*.

1 = Se ha producido una interrupción externa por *RB0/INT* (debe borrarse por software).

0 = No se ha producido la interrupción.

bit 0: **RBIF**: Bit de *flag* de la interrupción por cambio en *PORTB*.

1 = Al menos uno de los terminales *RB7:RB4* ha cambiado de estado (debe borrarse por software).

0 = No se ha producido cambio en dichas patillas.

Registro PIE1 (dirección RAM: 8Ch) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
Bit7							Bit0

Figura 3. Registro PIE1

bit 7: *PSPIE*: Bit de habilitación de interrupción por lectura/escritura del puerto paralelo esclavo:

1 = Habilita la interrupción por lectura/escritura del *PSP*.

0 = La deshabilita.

bit 6: *ADIE*: Bit de habilitación de interrupción por conversión A/D.

1 = Habilita la interrupción.

0 = La deshabilita.

bt 5: *RCIE*: Bit de habilitación de interrupción por recepción del *USART*.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 4: *TXIE*: Bit de habilitación de interrupción por transmisión del *USART*.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 3: *SSPIE*: Bit de habilitación de interrupción del puerto serie síncrono.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 2: *CCP1IE*: Bit de habilitación de interrupción del módulo *CCP1*.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 1: *TMR2IE*: Bit de habilitación de interrupción por igualación del *TMR2* y *PR2*.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 0: *TMR1IE*: Bit de habilitación de interrupción por desbordamiento del *TMR1*.

1 = Habilita la interrupción.

0 = La deshabilita.

Registro PIE2 (dirección RAM: 8Dh) [PIC16F87x]

U-0	R/W-0	U-0	R/W-0	R/W-0	U-0	U-0	R/W-0
			EEIE	BCLIE			CCP2IE
Bit7							Bit0

Figura 4. Registro PIE2

bit 7: No implementado. Se lee como 0.

bit 6: Reservado.

bit 5: No implementado. Se lee como 0.

bit 4: *EEIE*: Habilitación de interrupción por escritura en *EEPROM*.

1 = Habilita la interrupción.

0 = La deshabilita.

bit 3: *BCLIF*: *Flag* de interrupción por colisión en bus

1 = Colisión en el bus *SSP* o *I2C* en modo master.

0 = No hay colisión.

bit 2-1: No implementados. Se leen como 0.

bit 0: *CCP2IE*: Habilitación de interrupción del módulo *CCP2*.

1 = Habilita la interrupción.

0 = La deshabilita

Los registros *PIR1* y *PIR2* contienen los *flags* de las distintas interrupciones. El compilador C se encarga de generar el código necesario para leer y borrar dichos *flags*, tal y como podremos ver en el siguiente apartado.

4.2.1 Interrupciones en C

En el compilador C, la directiva habitual en el manejo de las interrupciones es *#INT_xxxx*. Especifica que la función que le sigue es una función de interrupción; además, no necesita más parámetros.

Las posibles directivas son las siguientes (en gris las posibles interrupciones en la familia *PIC16F87X*):

Interrupciones	Descripción
#INT_AD	Conversión AD completa.
#INT_ADOF	Conversión AD fuera de rango de tiempo.
#INT_BUSCOL	Colisión de bus.
#INT_BUTTON	Por botón (14000).
#INT_CANERR	Error en el modulo *CAN*.
#INT_CANIRX	Mensaje inválido en el bus *CAN*.
#INT_CANRX0	Bus *CAN* recibe un nuevo mensaje en *buffer* 0.
#INT_CANRX1	Bus *CAN* recibe un nuevo mensaje en *buffer* 1.
#INT_CANTX0	Bus *CAN* transmisión completa en *buffer* 0.
#INT_CANTX1	Bus *CAN* transmisión completa en *buffer* 1.
#INT_CANTX2	Bus *CAN* transmisión completa en *buffer* 2.
#INT_CANWAKE	Bus *CAN* evento de activación o *wake-up*.
#INT_CCP1	Unidad 1 de captura, comparación y *PWM*.
#INT_CCP2	Unidad 2 de captura, comparación y *PWM*.
#INT_CCP3	Unidad 3 de captura, comparación y *PWM*.
#INT_CCP4	Unidad 4 de captura, comparación y *PWM*.
#INT_CCP5	Unidad 5 de captura, comparación y *PWM*.
#INT_COMP	Comparador.
#INT_COMP1	Comparador 1.

Interrupciones	Descripción
#INT_COMP2	Comparador 2.
#INT_CR	Encriptación finalizada.
#INT_EEPROM	Escritura *EEPROM* finalizada.
#INT_EXT	Interrupción externa (*RB0*).
#INT_EXT1	Interrupción externa #1.
#INT_EXT2	Interrupción externa #2.
#INT_EXT3	Interrupción externa #3.
#INT_I2C	Interrupción *I2C* (14000).
#INT_IC1	Entrada captura #1.
#INT_IC2	Entrada captura #2.
#INT_IC3	Entrada captura #3.
#INT_LCD	Actividad *LCD*
#INT_LOWVOLT	Detectado bajo voltaje.
#INT_LVD	Detectado bajo voltaje.
#INT_OSC_FAIL	Fallo en oscilador.
#INT_OSCF	Fallo en oscilador.
#INT_PSP	Dato de entrada en puerto paralelo.
#INT_PWMTB	Base de tiempo *PWM*.
#INT_RA	Cambio de estado en A0-A5.
#INT_RB	Cambio de estado en B4-B7.
#INT_RC	Cambio de estado en C4-C7.
#INT_RDA	*RS232* dato recibido.
#INT_RDA0	*RS232* dato recibido en *buffer* 0.
#INT_RDA1	*RS232* dato recibido en *buffer* 1.
#INT_RDA2	*RS232* dato recibido en *buffer* 2.
#INT_RTCC	Desbordamiento del *Timer* 0 (*RTCC*).
#INT_PSP	Escritura/lectura del puerto paralelo.

Interrupciones	Descripción
#INT_COMP2	Comparador 2.
#INT_CR	Encriptación finalizada.
#INT_EEPROM	Escritura *EEPROM* finalizada.
#INT_EXT	Interrupción externa (*RB0*).
#INT_EXT1	Interrupción externa #1.
#INT_EXT2	Interrupción externa #2.
#INT_EXT3	Interrupción externa #3.
#INT_I2C	Interrupción *I2C* (14000).
#INT_IC1	Entrada captura #1.
#INT_IC2	Entrada captura #2.
#INT_IC3	Entrada captura #3.
#INT_LCD	Actividad *LCD*
#INT_LOWVOLT	Detectado bajo voltaje.
#INT_LVD	Detectado bajo voltaje.
#INT_OSC_FAIL	Fallo en oscilador.
#INT_OSCF	Fallo en oscilador.
#INT_PSP	Dato de entrada en puerto paralelo.
#INT_PWMTB	Base de tiempo *PWM*.
#INT_RA	Cambio de estado en A0-A5.
#INT_RB	Cambio de estado en B4-B7.
#INT_RC	Cambio de estado en C4-C7.
#INT_RDA	*RS232* dato recibido.
#INT_RDA0	*RS232* dato recibido en *buffer* 0.
#INT_RDA1	*RS232* dato recibido en *buffer* 1.
#INT_RDA2	*RS232* dato recibido en *buffer* 2.
#INT_RTCC	Desbordamiento del *Timer* 0 (*RTCC*).
#INT_PSP	Escritura/lectura del puerto paralelo.

Interrupciones	Descripción
#INT_SSP	Actividad en *SPI* o *I2C*.
#INT_SSP2	Actividad en *SPI* o *I2C Port* 2.
#INT_TBE	*RS232 buffer* de transmisión vacío.
#INT_TBE0	*RS232 buffer* 0 de transmisión vacío.
#INT_TBE1	*RS232 buffer* 1 de transmisión vacío.
#INT_TBE2	*RS232 buffer* 2 de transmisión vacío.
#INT_TIMER0	Desbordamiento del *Timer* 0 (*RTCC*).
#INT_TIMER1	Desbordamiento del *Timer* 1.
#INT_TIMER2	Desbordamiento del *Timer* 2.
#INT_TIMER3	Desbordamiento del *Timer* 3.
#INT_TIMER4	Desbordamiento del *Timer* 4.
#INT_TIMER5	Desbordamiento del *Timer* 5.
#INT_USB	Actividad en el *USB*.

Existe una directiva **#INT_DEFAULT** que implica que se utilizará la función que le acompaña si se activa una interrupción y ninguno de los *flags* está activo.

La directiva **#INT_GLOBAL** implica que la función sustituye todas las acciones que inserta el compilador al aceptarse una interrupción. Se ejecuta solamente lo escrito en dicha función. No se suele utilizar y si se hace debe hacerse con mucho cuidado.

Si se utilizan las directivas de interrupción, el compilador genera el código necesario para ejecutar la función que sigue a la directiva. Además genera el código necesario para guardar al principio y restituir al final el contexto; también borrará el *flag* activo por la interrupción.

El compilador C incluye funciones para el mejor manejo de las directivas de interrupción:

enable_interrupts (nivel);

nivel es una constante definida en un fichero de cabecera (16F87X.h –figura 5–) y genera el código necesario para activar las máscaras correspondientes, afectando a los registros **ITCON**, ***PIE1*** y ***PIE2***.

En el PIC16F876, los "niveles" permitidos son:

enable_interrupts (nivel);	ITCON(0Bh)	PIE1(8Ch)	PIE2(8Dh)
GLOBAL	11000000 C0h		
INT_RTCC INT_TIMER0	00100000 20h		
INT_EXT	00010000 10h		
INT_RB	00001000 08h		
INT_AD		01000000 40h	
INT_RDA		00100000 20h	
INT_TBE		00010000 10h	
INT_SSP		00001000 08h	
INT_CCP1		00000100 04h	
INT_TIMER2		00000010 02h	
INT_TIMER1		00000001 01h	
INT_EEPROM			00010000 10h
INT_BUSCOL			00001000 08h
INT_CCP2			00000100 04h

GLOBAL equivale a **GIE = PEIE = 1** y debe activarse de forma independiente. El resto activarán la máscara correspodiente.

disable_interrupts (nivel);

Realiza la función contraria a la anterior, inhabilita las máscaras de la interrupción correspondiente.

```
#define L_TO_H
#define H_TO_L

#define GLOBAL
#define INT_RTCC
#define INT_RB
#define INT_EXT
#define INT_AD
#define INT_TBE
#define INT_RDA
#define INT_TIMER1
#define INT_TIMER2
#define INT_CCP1
#define INT_CCP2
#define INT_SSP
#define INT_PSP
#define INT_BUSCOL
#define INT_EEPROM
#define INT_TIMER0
```

Figura 5. Parte del fichero include 16F87x.h

4.2.1.1 Interrupción exterior por RB0

Es una interrupción básica, común a la mayoría de los PIC. Permite generar una interrupción tras el cambio de nivel de alto a bajo o de bajo a alto en la entrada **RB0**.

La directiva utilizada es **#INT_EXT** y se debe acompañar de las siguientes funciones (afectan al bit 6 del registro **OPTION_REG**, ver figura 6).

ext_int_edge (H_TO_L);

La interrupción es por flanco de bajada (activa el *flag* **INTF**).

ext_int_edge (L_TO_H);

La interrupción es por flanco de subida (activa el *flag* **INTF**).

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
Bit7							Bit0

Figura 6. Registro OPTION_REG

bit 6: **INTEDG**: Bit de selección del flanco de interrupción de **RB0**.

1 = La interrupción es por flanco de subida del pin **RB0/INT**.

0 = La interrupción es por flanco de bajada del pin **RB0/INT**.

Ejemplo 1: *Encender y apagar, consecutivamente, un LED en la patilla RB7 cuando se produzca un cambio de nivel en la patilla RB0* (ver figura 7). *Componentes: ISIS: PIC16F876, RES, LED-GREEN y SW-SPDT-MOM.*

Figura 7. Ejemplo 1

```
#include <16F876.h>
#fuses XT,NOWDT,PUT,NOWRT
#use delay(clock= 4000000)
#use fast_io(B)
intl cambio=0;          // Variable de cambio
#INT_EXT                // Atención a interrupción por cambio en RB0
ext_isr(){              // Función de interrupción
   output_toggle(pin_B7);
}
void main() {
   set_tris_B(0x01);          // B0 como entrada, B7 como salida
   output_low(PIN_B7);        // Apaga LED
   port_b_pullups(TRUE);      // Pull-up para RB0
   enable_interrupts(int_ext);        // Habilita int. RB0...
   ext_int_edge(L_TO_H);              // por flanco de subida
   enable_interrupts(GLOBAL);         // Habilita int. general
while (1){              // Bucle infinito de espera
   }
}
```

Figura 8. Programa del ejemplo 1

4.3 TIMER0

El bloque funcional *TIMER0/WATCHDOG* es un contador (registro) de 8 bits, incrementado por hardware y programable. La cuenta máxima es de 255 (el incremento es constante e independiente).

- Contador: cuenta los eventos externos (a través del pin **RA4/TOCK1**).
- Temporizador: cuenta los pulsos internos de reloj.

Se puede insertar un *prescaler*, es decir, un divisor de frecuencia programable que puede dividir por 2, 4, 8, 16, 32, 64, 128 o 256. La frecuencia de conteo es una cuarta parte de la frecuencia de reloj (fosc/4). Posteriormente, con el uso del *prescaler* se puede dividir la frecuencia.

El bloque del *TIMER0* puede funcionar como *WATCHDOG*, lo que permite que durante el funcionamiento normal del microcontrolador, un desbordamiento (o *time-out*) del *Watchdog* provoque un reset (*Watchdog Timer Reset*). Para evitar el desbordamiento se debe, cada cierto tiempo y antes de que llegue al límite, ejecutar una instrucción *CLRWDT* que borra el *Watchdog* y que hace comenzar un nuevo conteo desde cero. Se basa en un oscilador *RC* interno, independiente del oscilador del microcontrolador y que no requiere ningún componente externo. El *Watchdog* cuenta incluso si el reloj conectado a **OSC1/CLKI** y/o **OSC2/CLKO** está parado, por ejemplo, por la ejecución de una instrucción *SLEEP* o por un defecto del cristal oscilador.

Los registros implicados en la configuración del *TIMER0/WDT* son los siguientes:

- **OPTION_REG**: configura el "hardware" del *TIMER0/WDT*.
- **INTCON**: permite trabajar con la interrupción del *TIMER0/WDT*.
- **TRISA**: habilita la patilla *RA4*.

Registro OPTION_REG (dirección RAM: 81h/181h) [PIC16F87x]

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
Bit7							Bit0

Figura 9. Registro OPTION_REG

bit 5: **T0CS:** Procedencia de las señales:

1 = *RA4/TOCK1*.

0 = Reloj interno.

bit 4: **T0SE:** Tipo de flanco en el *TOCK1/RA4*:

1 = Flanco descendente.

0 = Flanco ascendente.

bit 3: **PSA:** Asignación del divisor de frecuencias:

1 = *WDT*.

0 = *TMR0*.

bit 2:0: **PS2:PS1:PS0:** Determina el divisor de frecuencias a actuar según la siguiente tabla.

Valor	Rango TMR0	Rango WDT
000	1:2	1.1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

El tiempo de desbordamiento del *TIMER0* se calcula según la siguiente ecuación:

$$T = T_{CM} \cdot \text{Prescaler} \cdot (256 - \text{Carga TMR0})$$

Donde T_{CM} es el ciclo máquina que se puede calcular mediante la ecuación:

$$T_{CM} = 4/F_{OSC}$$

4.3.1 TIMER0 en C

La función para configurar el *TIMER0* es:

setup_timer_0 (modo);

Donde *modo* está definido en el fichero de cabecera (afecta a los bits 5:0 del *OPTION_REG*):

Setup_Timer_0(modo);	OPTION_REG(81h/181h)
RTCC_INTERNAL	**00000000** **00h**
RTCC_EXT_L_TO_H	**00100000** **20h**

Setup_Timer_0(modo);	OPTION_REG(81h/181h)
RTCC_EXT_H_TO_L	00110000 30h
RTCC_DIV_1	00001000 08h
RTCC_DIV_2	00000000 00h
RTCC_DIV_4	00000001 01h
RTCC_DIV_8	00000010 02h
RTCC_DIV_16	00000011 03h
RTCC_DIV_32	00000100 04h
RTCC_DIV_64	00000101 05h
RTCC_DIV_128	00000110 06h
RTCC_DIV_256	00000111 07h

Los distintos modos se pueden agrupar mediante el empleo de símbolo |.

```
setup_timer_0 (RTCC_DIV_2 | RTCC_EXT_L_TO_H);
```

La función para configurar el *WDT* es:

setup_wdt (modo);

Donde *modo* está definido en el fichero de cabecera (afecta a los bits 3:0 del *OPTION_REG*):

Setup_wdt(modo);	OPTION_REG(81h/181h)
WDT_18MS	00001000 08h
WDT_36MS	00001001 09h
WDT_72MS	00001010 0Ah

Setup_wdt(modo);	OPTION_REG(81h/181h)
WDT_144MS	00001011 0Bh
WDT_288MS	00001100 0Ch
WDT_576MS	00001101 0Dh
WDT_1152MS	00001110 0Eh
WDT_2304MS	00001111 0Fh

Para activar el *Watchdog* se deben utilizar los bits de configuración mediante la directiva **#FUSES**:

#fuses WDT Activado.

#fuses NOWDT Desactivado.

El compilador C suministra una serie de funciones para leer o escribir en el *TIMER0/WDT*. Para escribir un valor en el registro :

set_timer0 (valor);

valor: entero de 8 bits.

Para leer el valor actual del registro:

valor = get_timer0 ();

valor: entero de 8 bits.

También permite realizar la puesta a cero del *Watchdog* (como *CLRWDT*):

restart_wdt ();

Ejemplo 2: *Generar una señal cuadrada de 1KHz utilizando la interrupción del TIMER0* (ver figura 10). *Componentes ISIS: PIC16F876 e Instrumentos ISIS: OSCILLOSCOPE y COUNTER TIMER.*

Para generar una señal de 1 Khz se necesita un semiperiodo de 500 µs. Según la ecuación de desbordamiento del K, utilizando un cristal de 4 MHz y un *prescaler* de 2:

$$T = T_{CM} \cdot \text{Prescaler} \cdot (256 - \text{Carga TMR0})$$

$$500\ \mu s = (4/4000000)\cdot 2\cdot(256-x)$$

donde x = 6, es decir, se debe cargar el *TIMER0* con el valor 6. Pero esta relación sólo se cumple si se trabaja en ensamblador. Al trabajar en C, el compilador genera líneas de código que aumentan el tiempo de ejecución del programa y, por ello, es necesario ajustar el valor final. En este caso se ha utilizado un valor de carga de 29 (0x1D).

Figura 10. Ejemplo 2

```
#INCLUDE <16F876.h>
#use delay(clock=4000000)
#fuses XT,NOWDT
#use standard_io(B)
#int_TIMER0
void TIMER0_isr(void) {
   output_toggle(PIN_B0);
   set_timer0 (0x1B);  }                    // Se recarga el timer0
void main() {
  setup_timer_0(RTCC_INTERNAL|RTCC_DIV_2);   // Configuración timer0
  set_timer0 (0x1B);                         // Carga del timer0
  enable_interrupts(INT_TIMER0);        // Habilita interrupción timer0
  enable_interrupts(global);            // Habilita interrupción general
  while (1);          // bucle infinito
}
```

Figura 11. Programa del Ejemplo 2

El compilador se encarga al entrar en la interrupción de inhabilitar las interrupciones y al salir de borrar los *flags*, por lo que no es necesario hacerlo por programa.

Se puede observar la señal con el osciloscopio digital y activando los cursores en el menú de *Trigger* (figura 12) y la medida es de 998.76 µs.

Figura 12. Osciloscopio digital

4.4 TIMER1 y TIMER2

El modulo *TIMER1* es otro temporizador/contador con las siguientes características:

- Trabaja con 16 bits (con dos registros de 8 bits: **TMR1H** y **TMR1L**).
- Ambos registros se pueden leer y escribir.
- Interrupción por desbordamiento de FFFFh a 0000h.
- *Reset* por disparo del módulo **CCP**.
- Controlado por el registro **T1CON** (ver figura 13). Con el bit **TMR1ON** (**T1CON<0>**) se puede habilitar o deshabilitar.

Registro T1CON (dirección RAM: 10h)[PIC16F87x]

bit 7:6: No implementados: Se leen como 0.

bit 5:4: **T1CKPS1:T1CKPS0**: Selección del valor del *prescaler* del reloj del *TMR1*:

11= *Prescaler* a 1:8.

10= *Prescaler* a 1:4.

01= *Prescaler* a 1:2.

00= *Prescaler* a 1:1.

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
		T1CKPS1	T1CKPS0	T1OSCEN	T1SYNC	TMR1CS	TMR10N
Bit7							Bit0

Figura 13. Registro T1CON

bit 3: **T1OSCEN**: Bit de habilitación del oscilador del *TMR1*:

1 = Oscilador habilitado.

0 = Oscilador parado.

bit 2: **T1SYNC**: Sincronización de la entrada de reloj externa.

Si *TMR1CS* = 1:

1 = No sincronizado con la entrada de reloj interno.

0 = Sincronización de la entrada del reloj externo.

Si *TMR1CS* = 0:

Este bit es ignorado. **TMR1** utiliza el reloj interno.

bit 1: **TMR1CS**: Bit de selección de la fuente de reloj para el *TMR0*:

1 = Reloj externo desde el pin *RC0/T10S0/T1CKI* (flanco de subida).

0 = Reloj interno ($F_{osc}/4$).

bit 0: **TMR1ON**: Bit de habilitación del *TMR1*:

1 = TMR1 habilitado.

0 = TMR1 parado.

El *TIMER1* puede operar en los siguientes modos:

- Como temporizador.
- Como contador síncrono.
- Como contador asíncrono.

El modo de operación se establece mediante el bit **TMR1CS** (**T1CON<1>**). En modo temporizador, el *TIMER1* se incrementa en cada ciclo de instrucción. En modo contador se incrementa por flanco de subida de la señal externa. Cuando

se habilita el oscilador interno del *TIMER1* mediante el bit **T1OSCEN**, las patillas **RC1/T1OSI** y **RC0/T1OSO/T1CKI** se configuran como entradas ignorando el valor de **TRISC<1:0>**. El *TIMER1* tiene un reset interno que puede ser generado por el módulo **CCP**. Las interrupciones del *TIMER1* se controlan a través de los registros **PIE1** y **PIR1**.

El tiempo de desbordamiento del *TIMER1* se calcula según la siguiente ecuación:

$$T = T_{CM} \cdot Prescaler \cdot (65536 - Carga\ TMR1)$$

Donde T_{CM} es el ciclo máquina que se puede calcular mediante la ecuación:

$$T_{CM} = 4/F_{OSC}$$

El *TIMER2* es un modulo temporizador con las siguientes características:

- Temporización de 8 bits (registro **TMR2**).
- Registro de periodo de 8 bits (**PR2**).
- Ambos registros se pueden leer o escribir.
- *Prescaler* programable por programa (1:1, 1:4, 1:16).
- *Postcaler* programable por programa (1:1 a 1:16).
- Interrupción controlada por **PR2**.
- El módulo *SSP* utiliza opcionalmente el *TIMER2* para generar una señal de reloj.

El *TIMER2* tiene un registro de control **T2CON** (figura 14). El *TIMER2* puede ser habilitado mediante el bit **TMR2ON** (**T2CON<2>**) para optimizar el consumo de potencia.

Registro T2CON (dirección RAM: 12h)[PIC16F87x]

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
	TOUTPS3	TOUTPS2	TOUTPS1	TOUTPS0	TMR2ON	T2CKPS1	T2CKPS0
Bit7							Bit0

Figura 14. Registro T2CON

bit 7: No implementado: Se lee como 0.

bit 6:3: **TOUTPS3:TOUTPS0**: Selección del valor del *postscaler* del *TMR2*.

0000 = *Postscaler* a 1:1.

0001 = *Postscaler* a 1:2.

.....

.....

1110 = *Postscaler* a 1:15.

1111 = *Postscaler* a 1:16.

bit 2: **TMR2ON**: Bit de habilitación del *TMR2*:

1 = *TMR2* habilitado.

0 = *TMR2* parado.

bit 1:0: **T2CKPS1:T2CKPS0**: Selección del valor del *prescaler* del *TMR2*.

00 = *Prescaler* a 1:1.

01 = *Prescaler* a 1:4.

1x = *Prescaler* a 1:16.

El *TIMER2* se puede emplear como base de tiempos para la modulación en ancho de pulso (*PWM*) mediante la utilización del módulo **CCP**. El *TIMER2* se puede leer o escribir y es borrado en el reset. La entrada de reloj (FOSC/4) tiene un *prescaler* de 1:1. 1:4 o 1:16 seleccionado mediante los bits **T2CKPS1:T2CKPS0** (**T2CON<1:0>**). La salida se obtiene a través de un *postscaler* (de 1:1 a 1:16) que permite generar la interrupción cuyo *flag* se encuentra en **TMR2IF**, (**PIR1<1>**). Los contadores de *prescaler* y *postscaler* son borrados cuando se escribe en el registro *TIMER2*, cuando se escribe en el registro **T2CON** o en cualquier reset. El *TIMER2* no se borra cuando se escribe en el **T2CON**.

El tiempo de desbordamiento del *TIMER2* se calcula según la siguiente ecuación:

$$T = T_{CM} \cdot [\text{Prescaler} \cdot (\text{Carga TMR2} + 1) \cdot \text{Postscaler}]$$

Donde T_{CM} es el ciclo máquina que se puede calcular mediante la ecuación:

$$T_{CM} = 4/F_{OSC}$$

4.4.1 TIMER1 y TIMER2 en C

La configuración del módulo *TMR1* en el compilador de C se realiza a través de la función:

setup_timer_1 (modo);

Donde *modo* está definido en el fichero de cabecera (afecta a los bits 5:0 del registro **T1CON**):

Setup_Timer_1(modo);	T1CON(10h)
T1_DISABLED	00000000 00h

Setup_Timer_1(modo);	T1CON(10h)
T1_INTERNAL	10000101 85h
T1_EXTERNAL	10000111 87h
T1_EXTERNAL_SYNC	10000011 83h
T1_CLK_OUT	00001000 08h
T1_DIV_BY_1	00000000 00h
T1_DIV_BY_2	00010000 10h
T1_DIV_BY_4	00100000 20h
T1_DIV_BY_8	00110000 30h

Los distintos modos se pueden agrupar mediante el empleo de símbolo |.

La lectura y escritura en el módulo *TMR1* se realiza a través de las siguientes funciones:

valor = get_timer1 ();

set_timer1 (valor);

donde *valor* es un entero de 16 bits.

La configuración del módulo *TMR2* en el compilador de C se realiza con la función:

setup_timer_2 (modo,periodo,postscaler);

donde:

- *periodo* es un valor entero de 8 bits (0-255) para el registro **PR2**;
- *postscaler* es el valor del *postscaler* (1 a 16). Afecta a los bits 6:3 del registro **T2CON**; y
- *modo* afecta a los bits 2:0 del registro **T2CON**.

Setup_Timer_2(modo,periodo,postscaler);	T2CON(12h)
T2_DISABLED	00000000 00h
T1_DIV_BY_1	00000100 04h
T1_DIV_BY_4	00000101 05h
T1_DIV_BY_16	00000110 06h

La lectura y escritura en el módulo *TMR2* se realiza con la ayuda de las siguientes funciones:

valor = get_timer2 ();

set_timer2 (valor);

donde *valor* es un entero de 8 bits.

Ejemplo 3: *Generar una función que permita realizar retardos de 1 segundo empleando el TIMER1. Componentes ISIS: PIC16F876.*

Figura 15. Ejemplo 3

Se calcula un periodo parcial de 0.5 segundos y se repite dos veces:

$$0.5 = 4/F_{osc}\cdot(65536-x)\cdot P$$

con F_{osc} = 4 MHz y *preescaler* 1:8,; el *TMR1* = 3036.

NOTA
Para observar mejor el periodo conectar el *Counter Timer* a **CE** y ponerlo en modo *TIME* (segundos).

```
#include <16f876.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#use standard_io(b)

int1 cont=0;

#int_TIMER1          // Interrupción TIMER1
void temp1s(void)    // Función
{
  if (cont==1)  output_toggle(PIN_B1);   // Cada 2 interrupciones de 0.5 s
  set_timer1 (3036);                     // recarga del TMR1
  cont++;
  }

main ()
{
  setup_timer_1 (T1_INTERNAL | T1_DIV_BY_8);
  set_timer1 (3036);                 // recarga del TMR1
  enable_interrupts(INT_TIMER1);     // habilita interrupción timer1
  enable_interrupts(global);         // habilita interrupción general

  while(1);
}
```

Figura 16. Programa del ejemplo 3

También se podría realizar sin interrupciones, esperando a que el *TMR1* se desbordase.

```
#include <16f876.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#use standard_io(b)
```

```
templs()
{
  int cont=0;

  output_toggle(PIN_B1);
  while (cont<2)                // Para contar 2 veces 0.5 seg
  {
  set_timer1 (3036);            // Inicializa el TMR1
  while (get_timer1()>=3036);     // Espera a que se desbcrde (0.5 s)
  cont++;
  }
}

main ()
{
   setup_timer_1 (T1_INTERNAL | T1_DIV_BY_8);

   while(1){
      templs();                 // Llamada a la función de temporización
   }
}
```

Figura 17. Programa del ejemplo 3 sin interrupciones

Ejemplo 4: *Medir el ancho de un pulso mediante el TIMER1 y la interrupción externa por RB0 (figura 18). Componentes ISIS: PIC16F876 y LM016L. Instrumentos: OSCILLOSCOPE y Generadores: PULSE.*

Figura 18. Ejemplo 4

Al medir el ancho de un pulso se necesita detectar su flanco de subida y después su flanco de bajada. Esto se puede realizar mediante la interrupción *RB0* ya que permite configurar el flanco de disparo.

Al producirse una interrupción, por ejemplo en el flanco de subida, se puede inicializar el valor del temporizador (*TIMER1*) en ese momento; se cambia la activación de la interrupción del *RB0* a flanco de bajada y cuando se vuelva a producir la siguiente interrupción por dicho flanco se guarda el valor del temporizador.

El ancho de pulso será la diferencia entre el valor del *TIMER1* en el flanco de subida y el flanco de bajada. El ancho de pulso máximo para una frecuencia de 4 MHz (ciclo máquina de 1 µs) es de 65,536 ms (un ciclo del *TIMER1*). El mínimo dependerá del tiempo que tarda un programa en gestionar la interrupción y los cálculos. En el ejemplo se puede conseguir medir anchos de pulso de entre 64,934 ms y 69 µs.

Modificar el fichero *LCD.C* para que se visualice a través del puerto C.

```
#include <16f876.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#include <lcd.c>

Int16 TFB;                    // Tiempo flanco de bajada
float AP;                     // Valor final del ancho de pulso
int1 nuevopulso=0;            // Entra otro pulso
int1 cambio=0;                // Cambio de flanco de disparo

#int_ext
void funcion_ext_int(){               // Función Interrupción
  if(cambio==0){                      // Flanco de Subida
    set_timer1(0);                    // Inicializa TMR1
    ext_int_edge(0,H_TO_L);           // Configurar para flanco bajada
    cambio=1;                         // Control de cambio de flanco
  } else {                            // Flanco de Bajada
    TFB=get_timer1();             // valor del TIMER1 para el flanco de bajada
    ext_int_edge(0,L_TO_H);           // Configurar para flanco subida
    cambio=0;                         // Control de cambio de flanco
    if(nuevopulso==0){                // Fin de pulso...
      nuevopulso=1;                   // Pulso a calcular
    }
  }
}

void main() {
  lcd_init();
```

```
  setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);  // Configuración TIMER1
  ext_int_edge(0,L_TO_H);                    // Configurar para flanco subida
  cambio = 0;                                       // Control de cambio de flanco
  enable_interrupts(int_ext);                // Habilitación interrupción RB0
  enable_interrupts(global);                 // Habilitación general
  do {
    if(nuevopulso==1){                       // ¿Pulso nuevo?
    AP = TFB*1.0;                      // Ancho de pulso en microsegundos de TIMER1.
                                       // a 4MHZ el T = 1us*Timer1
        printf(lcd_putc,"\nPulso = %6.1fuS ", AP);   // Visualiza medida
                                                     // en LCD
      }
      nuevopulso=0;                    // Pulso medido
    }
  } while (TRUE);                      // Bucle infinito
}
```

Figura 19. Programa del Ejemplo 4

El generador *PULSE* se utiliza para crear la señal de entrada. Con el botón derecho se pueden editar sus características (figura 20).

Figura 20. Propiedades de PULSE

Ejemplo 5: *Generar una señal cuadrada de 1 KHz utilizando la interrupción del TIMER2 (figura 21). Componentes ISIS: PIC16F876 e Instrumentos ISIS: OSCILLOSCOPE y COUNTER TIMER.*

Para generar una señal de 1 Khz se necesita un semiperiodo de 500 µs, según la ecuación del desbordamiento del *TIMER1*, utilizando un cristal de 4 MHz, un *pre-escaler* de 4 y un *postscaler* de 1:

$$T = T_{CM} \cdot [\text{Prescaler} \cdot (\text{Carga TMR2}+1) \cdot \text{Postscaler})$$

$$500\ \mu s = (4/4000000) \cdot [4 \cdot (X+1) \cdot 1]$$

donde x = 125; es decir, se debe cargar el *TIMER2* con el valor 125. Pero esta relación sólo se cumple si se trabaja en ensamblador. Al trabajar en C, el compilador genera líneas de código que aumentan el tiempo de ejecución del programa y, por ello, es necesario ajustar el valor final. En este caso se ha utilizado un valor de carga de 11.

Figura 21. Ejemplo 5

```
#INCLUDE <16F876.h>
#use delay(clock=4000000)
#fuses XT,NOWDT
#use standard_io(B)

#int_TIMER2
void TIMER2_isr(void) {
   output_TOGGLE(PIN_B0);            // para semiperiodo alto
   set_timer2(11); }                 // se recarga el TIMER0
```

```
void main() {
  setup_timer_2(T2_DIV_BY_4,124,1);   // configuración TIMER2
  enable_interrupts(INT_TIMER2);      // habilita interrupción TIMER0
  enable_interrupts(global);          // habilita interrupcion general

  while (1);          // bucle infinito
}
```

Figura 22. Programa del ejemplo 5

Figura 23. Señal de 1 KHz

Ejemplo 6: *Introducir los datos, a través del teclado, de la velocidad de un motor de corriente continua y generar una señal modulada en un ancho de pulso proporcional al dato de velocidad. Controlar la velocidad en rpm y visualizarla en un LCD* (figura 24). *Componentes ISIS: PIC18F876, KEYPAD-PHONE, RES, 2SK1058, CELL, LM016L y MOTOR-ENCODER.*

Funciona igual que el ejemplo 6 del tema de los puertos, pero ahora se le añade un contador de pulsos controlado por el *TIMER0*. Los pulsos proceden de un *encoder* que suministra el modelo del *MOTOR-ENCODER*. Este modelo permite obtener tres tipos de salidas: **Q1**, **Q2** e **IDX**.

Q1 y **Q2** permiten controlar el sentido de giro y la posición angular del motor (se configuran en la opción *PULSES PER REVOLUTION* del menú de edición del motor). La señal **IDX** suministra un pulso por revolución y es la que se usará para medir la velocidad del motor en este ejemplo; al suministrar un pulso por revolución se pueden contar los pulsos por minuto mediante el *TMR0* y visualizarlos en el *LCD*.

Figura 24. Ejemplo 6

Se puede utilizar el puerto B conjuntamente para el *LCD* y el teclado, pero se ha preferido utilizar el puerto C para el *LCD* (esto supone modificar el fichero *LCD.C* para que en lugar de utilizar el puerto D o E utilice el C), figura 25.

```
// Para seleccionar otro puerto

struct lcd_pin_map
          BOOLEAN enable
          BOOLEAN rs
          BOOLEAN rw
          BOOLEAN unused
          int     data
        lcd


#if defined __PCH__
#if defined use_portb_lcd
   #byte lcd = 0xF81
#endif
#else
#if defined use_portb_lcd
   #byte lcd = 0xF82
#else
   #byte lcd = 7
#endif
#endif

#if defined use_portb_lcd
   #define set_tris_lcd x  set_tris_b x
#else
   #define set_tris_lcd x  set_tris_d x
#endif
```

#byte lcd = 0xF82

#byte lcd = 7 **//on to portC (at address 7)**

#define set_tris_lcd(x) set_tris_c(x)

Figura 25. Fichero LCD.C a modificar

El programa se basa en producir una interrupción del *TIMER0* cada 0.5 segundos y leer el *TIMER0* que se utiliza como contador de pulsos externos. La lectura se debe multiplicar por 2 (*preescaler* mínimo del *TMR0*) y por 120 (puesto que al ser revoluciones por minuto, si se mide cada 0.5 segundos se necesitará multiplicar por este factor). La carga del *TMR1* con un *preescaler* de 8 es de 3036.

```
#include <16f876.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#USE DELAY (CLOCK=4000000)
#INCLUDE <kbd.C>
#include <lcd.c>
#USE STANDARD_IO (a)
int16 counter=0;

//*************** INT TIMER1********************
#int_TIMER1
void TIMER1_isr(void) {
counter=get_timer0();          // Lectura contador TMR0
counter=counter*2*120;         // Conversión a rpm
  printf(lcd_putc,"%6lu rpm",counter);
  lcd_gotoxy(1,1);
set_timer0(0);                 // Reinicia cuenta
set_timer1(3036);              // Recarga a 0.5 s
 }
//*******************************************
VOID MAIN()
{
   CHAR K,kant='0';
   char PWMH=0,PWML=0;
   lcd_init();
   kbd_init();
   PORT_B_PULLUPS(TRUE);

   setup_timer_0(rtcc_ext_l_to_h|RTCC_DIV_2);        // Configuración TMR0
   setup_timer_1(T1_internal|T1_DIV_BY_8);           // Configuración TMR1
   set_timer0(0);                                    // Borrado contador
   set_timer1(3036);                                 // Carga a 0.5 s
   enable_interrupts(int_timer1);
   enable_interrupts(global);            // Habilitación de interrupciones

WHILE (1) {
      k=kbd_getc
      if (k=='\0') k=kant;
```

```
        if ((k=='*') || (k=='#')) k='0';
        kant=k;
            k=k-48;
            PWMH=k*28;
            PWML=255-PWMH;
        for(PWMH;PWMH>0;PWMH--){
            OUTPUT_HIGH(PIN_A0);}
        for(PWML;PWML>0;PWML--){
            OUTPUT_LOW(PIN_A0);}
    }
}
```

Figura 26. Programa del ejemplo 6

Figura 27. Señal modula y salida del encoder

Ejemplo 7: *Según la duración de pulsación de un botón obtener tres tipos de funciones: una pulsación corta da lugar a una función (por ejemplo encender un led en el puerto A), una primera pulsación mayor de tres segundos da lugar a otra función (por ejemplo encender un led en el puerto C) y una segunda pulsación mayor de tres segundos de lugar a otra función (por ejemplo apagar el led del puerto C). Cuando se trabaje con la segunda o tercera función no se atenderán las pulsaciones cortas* (figura 28)*. Componentes ISIS: PIC18F877, BUTTON, RES, LED-RED, LED-BLUE.*

En este ejemplo se utiliza una técnica de *polling* (comprobación continua) del estado del pulsador pero empleando la interrupción del *TIMER0*; de esta forma no se tendrá al programa principal parado comprobando el estado del pulsador.

Figura 28. Ejemplo 7

Se programa el *TIMER0* para que provoque una interrupción cada determinado tiempo, en este caso de 20 ms. Cada vez que se produzca la interrupción se comprueba si el botón ha sido pulsado. En el caso de que haya sido pulsado se comprueba si es una pulsación corta o mayor de 3 segundos. Para este último caso se comprueba si el botón esta pulsado durante 150 interrupciones del *TIMER0* (3s/20ms = 150).

Si el botón no ha sido pulsado se designa como *Función_D6 = 0*, si ha sido pulsado una vez durante más de 3 segundos se designa como *Función_D6 = 1*, si ha sido pulsado momentáneamente se designa como *Función_D6 = 2* y si ha sido pulsado durante más de 3 segundos por segunda vez se designa como *Función_D6 = 3*.

Para distinguir si se ha pulsado durante más de 3 segundos una o dos veces se utiliza una variable de control (que se llamará *CON_D6*) que puede valer cero o uno, dependiendo de si es la primera vez o la segunda vez que se pulsa.

Para controlar el tiempo que está el botón pulsado se utiliza una variable (D6) que se irá incrementando si el botón está pulsado y se produce una interrupción del *TMR0*.

```
#INCLUDE <16F877.h>
#device adc=10
#use delay(clock=4000000)
#fuses XT,NOWDT,NOPUT,NOPROTECT,NOBROWNOUT,NOLVP,NOCPD,NOWRT,NODEBUG
#USE FAST_IO (B)

#BYTE TRISA = 0x85
#BYTE PORTA = 0x05
```

```
#BYTE TRISC = 0x87
#BYTE PORTC = 0x07
#BYTE TRISD = 0x88
#BYTE PORTD = 0x08
#BYTE TIMER0 = 0x01
#BIT RA5 = 0x05.5
#BIT RC2 = 0x07.2

CHAR D6,FUNCION_D6,CON_D6;
INT1 CNT;

//*************INTERRUPCION TIMER0**********************
#int_TIMER0
void TIMER0_isr(void) {
   IF  (INPUT(PIN_D6)==0) {                    // Si esta pulsado
      IF (D6 >= 150) {                         //Detecta si 3 s (20ms x 150)
          IF (CON_D6==0) {                     // Pulsado 1ª vez 3 s
             D6=0;
             FUNCION_D6=1;
             CON_D6=1; }
          ELSE {                               // Pulsado 2ª vez 3 s
             D6=0;
             FUNCION_D6=3;
             CON_D6=0;}
          }
       ELSE D6++;                  //Si no llega a los 3 s aumenta contador
   }
      ELSE {
      if (D6>0 && funcion_D6 ==1) D6=0;        // Si pulsado antes pero menos
                                               // 3 seg borra contado:
      IF (D6 > 0 && FUNCION_D6 !=1 && FUNCION_D6 !=3 ) // Si pulsado antes…
            {FUNCION_D6=2;                         // y NO F=1 y NO F=3…
             D6=0;}                                // entonces F=2
   }
SET_TIMER0 (100);                          // Reinicializa el contador
}
//*************************PRINCIPAL*********************************
void main() {
   disable_interrupts (GLOBAL);
   setup_timer_0(RTCC_INTERNAL|RTCC_DIV_128);
   D6=0; FUNCION_D6=0; CON_D6=0;           // Limpia contadores tecla
   TRISA = 0X00; //00000000
   TRISC = 0X00; //00000000
```

```
    TRISD = 0X40; //01000000
    BIT_CLEAR(PORTA,5);
    BIT_CLEAR(PORTC,2);
    enable interrupts (GLOBAL);
    SET_TIMER0 (100);     // TIMER0=20ms de polling; 20ms=(256-100)*1 us*128
    enable_interrupts (INT_TIMER0);

WHILE (1)
     IF (FUNCION_D6==1) RC2=1;          // Si funcion 1 enciendo RC2
     IF (FUNCION_D6==3) {               // Si función 3 apaga RC2.....
          RC2=0;          // Espera un tiempo para no provocar una función 2
          DELAY_MS(400);                // no deseada.
          FUNCION_D6=0;}                // Vuelve a funcion 0
     IF (FUNCION_D6==2) {               // Si función 2, una vez enciende led…
       CNT++;                     // segunda vez lo apaga, asi continuamente
       IF (CNT==0) RA5=1;
       ELSE RA5=0;
       FUNCION_D6=0;              // Vuelve a función 0
      }
    }
}
```

Figura 29. Programa del ejemplo 7

Capítulo 5

Convertidor Analógico – Digital

5.1 Introducción

Los microcontroladores PIC pueden incorporar un módulo de conversión de señal analógica a señal digital. Los módulos AD que utiliza Microchip hacen un muestreo y retención (*sample & hold*) con un condensador y después utiliza el módulo de conversión (figura 1). El módulo de conversión A/D es del tipo de aproximaciones sucesivas.

Figura 1. Fases de la conversión analógica/digital

El convertidor de aproximaciones sucesivas se utiliza en aplicaciones donde se necesitan altas velocidades de conversión. Se basa en realizar sucesivas comparaciones de forma ascendente o descendente hasta encontrar un valor digital que iguale la tensión entregada por el conversor D/A y la tensión de entrada.

Durante la fase de muestreo el interruptor se cierra y el condensador se carga a la tensión de entrada (el tiempo que el interruptor permanece cerrado es fundamental para la correcta carga del condensador). Una vez abierto el interruptor, el condensador mantendrá (teóricamente) la tensión de entrada mientras el modulo A/D realiza la conversión.

El módulo de conversión se caracteriza por parámetros como los siguientes:

- Rango de entrada.

- Número de bits.
- Resolución.
- Tensión de fondo de escala.
- Tiempo de conversión.
- Error de conversión.

El módulo que utilizan los PIC de gama media tiene un número de bits de 10, por lo que su resolución es:

$$resolución = \frac{V_{IN}}{2^N - 1}$$

siendo V_{IN} la tensión de entrada y N el número de bits del convertidor. Es decir, para la tensión máxima de entrada (5V) la resolución es de 0,0048 V (4,8 mV) por LSB.

La resolución sí cambia si se modifica la tensión de fondo de escala, es decir, la tensión de referencia. Los PICs permiten cambiar la tensión de referencia en un valor absoluto (de 0 a $+V_{ref}$) o en un margen (de $-V_{ref}$ a $+V_{ref}$).

Las tensiones a convertir siempre son **positivas**.

5.2 Módulo Convertidor (gama media)

El módulo convertidor A/D en la gama media posee hasta 8 entradas analógicas. Los 16F876/873 tienen 5 canales (en el puerto A) y los 16F877/874 tienen 8 canales (5 en el puerto A y 3 en el puerto E). El convertidor (figura 2) es de 10 bits y, tal como se ha comentado, es de aproximaciones sucesivas. Permite variar la tensión de referencia a la máxima V_{DD} o a una tensión positiva menor a través de AN3/V_{REF-} y a la mínima V_{SS} o a una tensión positiva mayor a través de AN2/V_{REF-}.

Figura 2. Bloques básicos de un convertidor A/D de aproximaciones sucesivas

Puede seguir funcionando cuando el PIC está en modo *SLEEP* ya que dispone de un oscilador RC interno propio.

Figura 3. Arquitectura del módulo convertidor A/D

La función de transferencia del convertidor A/D es el resultado de que la primera transición ocurra cuando la tensión analógica de entrada es igual a $V_{REF}/1024$.

La resolución vendrá dada por la siguiente ecuación:

$$1LSB = V_{REF-} + \frac{(V_{REF+} - V_{REF-})}{1024}$$

En el caso de que la $V_{REF+} = V_{DD}$ y $V_{REF-} = V_{SS}$ entonces la resolución es:

$$1LSB = \frac{5}{1024} = 4{,}8mV$$

de esta forma si la lectura es de 512 LSB, la tensión analógica leida es:

$$V_{IN} = 512 \cdot \frac{5}{1024} = 512 \cdot 4{,}8mV = 2{,}4576\ V$$

5.2.1 Registros FSR

Hay 11 registros asociados a este periférico:

- Definición de pines de entrada y señales aplicadas:

  TRISA – PORTA – TRISE – PORTE.

- Manejo de interrupciones:

  INTCON – PIE1 – PIR1.

- Control del conversor A/D:

  ADCON0 – ADCON1 – ADRESH – ADRESL.

Registro de control ADCON0 (dirección RAM: 1Fh) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0
ADCS1	ADCS0	CHS2	CHS1	CHS0	GO/$\overline{\text{DONE}}$		ADON
Bit7							Bit0

Figura 4. Registro ADCON0

bits 7:6 **DCS1:ADCS0:** Bits de selección del reloj para la conversión A/D.

00 = $F_{OSC}/2$.

01 = $F_{OSC}/8$.

10 = $F_{OSC}/32$.

11 = F_{RC} (Reloj del oscilador interno RC).

bits 5:3 **CHS2:CHS0:** Bits de selección del canal.

000 = Canal 0, (RA0/AN0).

001 = Canal 1, (RA1/AN1).

010 = Canal 2, (RA2/AN2).

011 = Canal 3, (RA3/AN3).

100 = Canal 4, (RA4/AN4).

101 = Canal 5, (RA5/AN5).

110 = Canal 6, (RA6/AN6).

111 = Canal 7, (RA7/AN7).

bit 2 **GO/ DONE:** Bits de estado de la conversión.

Si ADCON = 1

1 = Conversión en progreso (a 1 inicia una conversión).

0 = La conversión ha finalizado (este bit es borrado por hardware al terminar la conversión)

bit 1 No usado: valor 0.

bit 0 **ADCON:** Activación del conversor A/D.

1 = convertidor activo.

0 = convertidor no activo.

Registro de control ADCON1 (dirección RAM: 9Fh) [PIC16F87x]

R/W-0	U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
ADFM				PCFG3	PCFG2	PCFG1	PCFG0
Bit7							Bit0

Figura 5. Registro ADCON1

bit 7: **ADFM**: Selección de formato del resultado (figura 6):

1 = Justificación a derechas. Los 6 bits más significativos de *ADRESH* son leídos como '0'.

0 = Justificación a izquierdas. Los 6 bits menos significativos de *ADRESL* son leídos como '0'.

bit 6-4: No usado: valor 0

bit 3-0: **PCFG3:PCFG0:** Configuración de las entradas al módulo A/D (figura 7).

Figura 6. Justificación mediante bit ADFM

PCFG3 PCFG0	AN7 RE2	AN6 RE1	AN5 RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	V_{REF+}	V_{REF-}	CH_{AN}/ REFS
0000	A	A	A	A	A	A	A	A	VDD	VSS	8/0
0001	A	A	A	A	V_{REF+}	A	A	A	RA3	VSS	7/1
0010	D	D	D	A	A	A	A	A	VDD	VSS	5/0
0011	D	D	D	A	V_{REF+}	A	A	A	RA3	VSS	4/1
0100	D	D	D	D	A	D	A	A	VDD	VSS	3/0
0101	D	D	D	D	V_{REF+}	D	A	A	VA3	VSS	2/1
011X	D	D	D	D	D	D	D	D	VDD	VSS	0/0
1000	A	A	A	A	V_{REF+}	V_{REF-}	A	A	VA3	RA2	6/2
1001	D	D	A	A	A	A	A	A	VDD	VSS	6/0
1010	D	D	A	A	V_{REF+}	A	A	A	VA3	VSS	5/1
1011	D	D	A	A	V_{REF+}	V_{REF-}	A	A	VA3	RA2	4/2
1100	D	D	D	A	V_{REF+}	V_{REF-}	A	A	VA3	RA2	3/2
1101	D	D	D	D	V_{REF+}	V_{REF-}	A	A	VA2	RA2	2/2
1110	D	D	D	D	D	D	D	A	VDD	VSS	1/0
1111	D	D	D	D	V_{REF+}	V_{REF-}	D	A	VA3	RA2	1/2

Figura 7. Tabla de configuración de los canales

En las versiones PIC17F87xA existen unas pequeñas variaciones en estos dos registros.

En el **registro ADCON0 (dirección RAM:1Fh) [PIC16F87xA]** los bits:

bit 7-6: **ADCS1:ADCS0:** Selección del reloj para la conversión A/D junto con *ADCS2* que está en *ADCON1*.

ADCS2=0	00 = fOSC/2	01 = fOSC/8	10 = fOSC/32	11 = fRC
ADCS2=1	00 = fOSC/4	01 = fOSC/16	10 = fOSC/64	11 = fRC

En el **registro ADCON1 (dirección RAM: 9Fh) [PIC16F87xA]** el bit:

bit 6: **ADCS2**: Selección de reloj para conversión A/D junto con *ADCS1* y *ADCS0*.

Otros registros que afectan al módulo convertidor son los referentes a la interrupción: **INTCON, PIE1** y **PIR1**.

5.2.2 Proceso de conversión

Para realizar la conversión, el fabricante recomienda seguir los siguientes pasos:

1. Configurar el módulo A/D:
   a. Configuración de pines analógicos/tensión de referencia/E/S digitales (**ADCON1**).
   b. Selección de la entrada A/D (**ADCON0**).
   c. Selección de reloj para la conversión A/D (**ADCON0**).
   d. Habilitar módulo A/D (**ADON (ADCON0<0>)**
2. Configurar las interrupciones (si se desea):
   a. ADIF =0;
   b. *GIE = PEIE = ADIE = 1.*
3. Esperar el tiempo de adquisición.
4. Comenzar la conversión poniendo a '1' el bit *GO/DONE(ADCON0<2>).*
5. Esperar a que termine la conversión. Puede ser de dos formas:
   a. Mediante lectura continua del bit *GO/DONE* hasta que sea '0', indicando el fin de la conversión.
   b. Esperando a la interrupción.
6. Leer rel egistro de conversión **ADRES** y borrar en *flag* **ADIF** si es necesario.

7. Para la siguiente conversión se salta a los puntos 1, 2 ó 3 en función de lo que se necesite. El tiempo de conversión por bit se define como T_{AD}. Un mínimo de $2 \cdot T_{AD}$ son necesarios antes de la conversión (esto no es necesario para los PIC16F87xA debido a que el interruptor de muestreo se cierra en cuanto se obtiene el resultado).

Existen dos tiempos básicos de trabajo: el tiempo de adquisición (T_{ACQ}) y el tiempo de conversión T_{AD}.

- **Tiempo de adquisición** (T_{ACQ}): tiempo necesario para que se cargue el condensador de retención (C_{HOLD}) con la tensión de entrada. Este proceso de carga del condensador depende de distintos factores, entre otros, la impedancia de la fuente de tensión de entrada (el fabricante recomienda que se sitúe por debajo de los 10 kohm).

  El tiempo de adquisición dentro de los márgenes típicos es de, aproximadamente, 20 µs.

  La adquisición no comienza hasta que no acabe la conversión. Así que se debe espera un T_{ACQ} tras una conversión, tras seleccionar un nuevo canal o tras encender el módulo AD.

- **Tiempo de conversión** (T_{AD}): tiempo necesario para obtener el valor digital de la tensión analógica de entrada. Este tiempo depende de la fuente de reloj que se seleccione para la conversión. Para una correcta conversión A/D, el reloj debe seleccionarse para asegurar un tiempo mínimo T_{AD} de 1,6 µs. En la figura 8 se muestra la tabla de selección de fuentes de reloj con su T_{AD} asociado, las celdas sombreadas son las que no se recomienda su uso.

Fuente de reloj (T_{AD})		Frecuencia del dispositivo			
Operación	**ADCS1:ADCS0**	**20 MHz**	**5 MHz**	**1.25 MHz**	**333.33 kHz**
$2T_{osc}$	00	100 ns(2)	400 ns(2)	1.6 µs	6 µs
$8T_{osc}$	01	400 ns(2)	1.6 µs	6.4 µs	24 µs(3)
$32T_{osc}$	10	1.6 µs	6.4 µs	25.6 µs(3)	98 µs(3)
RC	11	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1,4)	2-6 µs(1)

Figura 8. Tabla de selección de fuentes de reloj

Las notas de la figura 8 indican:

(1): La fuente RC tiene un T_{AD} típico de 4 µs.

(2): Estos valores violan el mínimo tiempo requerido de T_{AD}.

(3): Para conversiones más rápidas se recomienda utilizar otra fuente de reloj.

(4): En PICs con frecuencias superiores a 1 MHz, el modo RC sólo es recomendable en modo *SLEEP*.

El T_{AD} se configura en ADCON0 (reloj de la conversión).

$T_{AD}=2 \cdot TOSC$	$T_{AD}=8 \cdot TOSC$	$T_{AD}=32 \cdot TOSC$	$T_{AD}=2\mu s \div 6\mu s$ (tip. 4µs)

También en el PIC16F87xA.

T_{AD}=4·TOSC	T_{AD}=16·TOSC	T_{AD}=64·TOSC

Para convertir 10 bits se requiere un tiempo de $12 \cdot T_{AD}$ (figura 9).

Figura 9. Ciclos de conversión

Considerando los dos tiempos (de adquisición y de conversión) la secuencia completa de muestreo/retención y adquisición en los PICs de gama media se muestra en la figura 10. Existe una diferencia entre los PIC16F87X y los PIC16F87XA; en los primeros es necesario esperar un tiempo $2 \cdot T_{AD}$ antes de iniciar una nueva adquisición, cosa que no ocurre en los segundos.

Figura 10. Tiempo de muestro y conversión

5.2.3 Efecto del modo SLEEP y RESET en el módulo AD

En el modo dormido (*sleep*), el módulo AD puede funcionar si se selecciona como reloj para la conversión el RC interno (**ADCS1= 1** y **ADCS0 = 1**). En este caso, el módulo espera 1 ciclo de instrucción antes de iniciar la conversión, permitiendo la ejecución de la siguiente instrucción *SLEEP*, eliminando así todo posible ruido de

conmutación durante la conversión. Al finalizar la conversión, el bit **GO/DONE** es puesto a "0" y el resultado se carga en los registros *ADRESH* y *ADRESL*. En el caso de que la interrupción del módulo AD esté habilitada (**ADIE = 1** y **PEIE = 1**) el dispositivo se despierta, pero en el caso de no estar habilitada, el módulo se apaga aunque el bit *ADON* siga a "1".

En el caso de que la fuente de reloj no es la RC interna, la ejecución de una instrucción *SLEEP* hace que la conversión que se esté realizando se pare y que el módulo se apague aunque el bit *ADON* siga a "1".

En el caso de producirse un *RESET*, los registros del módulo AD se inicializan a los valores indicados por el fabricante. El efecto del *RESET* es el apagado del módulo y la parada instantánea de la conversión actual, los terminales se reinicializan a entradas analógicas (*ADCON1* parte baja a 0); el valor de *ADRESX* no se modifica en el caso de un *Reset* pero en el caso de un *Power-on Reset* es desconocido.

5.3 Módulo AD en C

En el compilador C las funciones para manejar el convertidor AD son las siguientes:

setup_adc (modo);

modo: para la configuración del módulo conversor A/D correspondientes a los bits 7:6 del *ADCON0*.

Setup_adc(modo);	ADCON0(1Fh)
ADC_OFF	00000000 00h
ADC_CLOCK_INTERNAL	11000000 C0h
ADC_CLOCK_DIV_2	000000000 00h
ADC_CLOCK_DIV_8	01000000 40h
ADC_CLOCK_DIV_32	10000000 80h

setup_adc_ports (valor);

Valor: definición de las entradas analógicas correspondiente a los bits 3-0 del *ADCON1* (figura 11).

set_adc_channel (canal);

canal: selección del canal analógico correspondiente a los bits 5:3 de *ADCON0*.

0 (AN0)	1 (AN1)	2 (AN2)	3 (AN3)
4 (AN4)	5 (AN5)	6 (AN6)	7 (AN7)

PCFG3 PCFG0	AN7 RE2	AN6 RE1	AN5 RE0	AN4 RA5	AN3 RA3	AN2 RA2	AN1 RA1	AN0 RA0	**setup_adc_ports (valor);**
0000	A	A	A	A	A	A	A	A	ALL_ANALOG
0001	A	A	A	A	V_{REF-}	A	A	A	AN0_AN1_AN2_AN4_AN5_AN6_AN7_VSS_VREF
0010	D	D	D	A	A	A	A	A	AN0_AN1_AN2_AN3_AN4
0011	D	D	D	A	V_{REF-}	A	A	A	AN0_AN1_AN2_AN4_VSS_VREF
0100	D	D	D	D	A	D	A	A	AN0_AN1_AN3
0101	D	D	D	D	V_{REF-}	D	A	A	AN0_AN1_VSS_VREF
011X	D	D	D	D	D	D	D	D	NO_ANALOGS
1000	A	A	A	A	V_{REF-}	V_{REF-}	A	A	AN0_AN1_AN4_AN5_AN6_AN7_VREF_VREF
1001	D	D	A	A	A	A	A	A	AN0_AN1_AN2_AN3_AN4_AN5
1010	D	D	A	A	V_{REF-}	A	A	A	AN0_AN1_AN2_AN4_AN5_VSS_VREF
1011	D	D	A	A	V_{REF-}	V_{REF-}	A	A	AN0_AN1_AN4_AN5_VREF_VREF
1100	D	D	D	A	V_{REF-}	V_{REF-}	A	A	A_ANALOG_RA3_RA2_REF
1101	D	D	D	D	V_{REF-}	V_{REF-}	A	A	AN0_AN1_VREF_VREF
1110	D	D	D	D	D	D	D	A	AN0
1111	D	D	D	D	V_{REF-}	V_{REF-}	D	A	AN0_VREF_VREF

Figura 11. Posible valores de setup_adc_port(valor)

valor = read_adc ();

Lectura del resultado donde valor es un entero de 16 bits según la directiva *#DEVICE ADC=* empleada. Dicha directiva trabaja según la tabla:

DEVICE	8 bit	10 bit	11 bit	16 bit
ADC=8	00-FF	00-FF	00-FF	00-FF
ADC=10	x	0-3FF	x	x
ADC=11	x	x	0-7FF	x
ADC=16	0-FF00	0-FFC0	0-FFE0	0-FFFF

Por ejemplo, el fichero 16f876.h incluye como primera directiva *#device* PIC16F876. Para incluir la información del tipo de conversor A/D se debe añadir **#device adc = 10**.

READ_ADC() admite tres modos de funcionamiento:

ADC_START_AND_READ	Si no se indica nada es la opción por defecto. Permite iniciar y leer el Convertidor.
ADC_START_ONLY	Sólo inicia la conversión.
ADC_READ_ONLY	Sólo lee los registros del convertidor.

Ejemplo 1: *Lectura de una tensión analógica por el canal AN0 (figura 12). Componentes ISIS: PIC16F876, POT-LIN, CELL y LM016L.*

Figura 12. Ejemplo 1

```
#include <16F876.h>
#device adc=10

#FUSES XT,NOWDT
#FUSES
#use delay(clock=4000000)
#include <LCD.C>

void main() {
   int16 q;
   float p;

   setup_adc_ports(AN0);              //Canal 0 analógico
   setup_adc(ADC_CLOCK_INTERNAL);     //Fuente de reloj RC

   lcd_init();

   for (;;) {
      set_adc_channel(0);             //Habilitación canal 0
      delay_us(20);
```

```
        q = read_adc();                    //Lectura canal 0
        p = 5.0 * q / 1024.0;              //Conversión a tension

        printf(lcd_putc, "\fADC = %4ld", q);
        printf(lcd_putc, "\nVoltage = %01.2fV", p);

        delay_ms(100);
        }
}
```

Figura 13. Programa del Ejemplo 1

Ejemplo 2: *Termómetro con una NTC NTSA0WB203* (figura 16). *Componentes ISIS: PIC16F876, NTSA0WB203, CELL y LM016L.*

Se utiliza una NTC NTSA0WB203 con una beta de 4050 y una resistencia a 25 °C de 20 kohm (figura 14); estas características se pueden ajustar así como la temperatura a medir en el menú de edición del componente (botón derecho, figura 15).

Part Number	Resistance (25 °C) (k ohm)	B-Constant (25-60 °C) (K)	Permissive Operating Current (25 °C) (mA)	Rated Electric Power (25 °C) (mW)	Typical Dissipation Constant (25 °C) (mW/ °C)	Thermal Time Constant (25 °C)(s)	Operating Temperature Range (°C)
NTSA0XM202□E1B0	2.0	3500 ±1%	1.05	21	2.1	7	-40 to 125
NTSA0XR502□E1B0	5.0	3700 ±1%	0.68	21	2.1	7	-40 to 125
NTSA0XH103□E1B0	10	3880 ±1%	0.48	15	1.5	7	-40 to 125
NTSA0XV103□E1B0	10	3900 ±1%	0.46	21	2.1	7	-40 to 125
NTSA0WB203□E1B0	20	4050 ±1%	0.33	21	2.1	7	-40 to 125
NTSA0WC303□E1B0	30	4170 ±1%	0.26	21	2.1	7	-40 to 125

Figura 14. Caracteristicas de la NTC NTSA0WB203

Figura 15. Edición del componente

La NTC se linealiza mediante una resistencia R1 de 10 kohm, el valor de tensión de la NTC (V_T) se puede obtener según las siguientes ecuaciones (temperaturas en grados kelvin):

$$R_T = \frac{V_T \cdot R_1}{V_{BAT} - V_T} = \frac{10000 \cdot V_T}{5 - V_T}$$

$$T_T = \frac{1}{\frac{1}{\beta}\ln(\frac{R_T}{R_{T25}}) + \frac{1}{T_{25}}} - 273.15 = \frac{1}{\frac{1}{4050}\ln(\frac{R_T}{20000}) + \frac{1}{298.15}} - 273.15$$

Figura 16. Ejemplo 2

```
#include "16f876.h"
#device adc=10
#fuses XT,NOWDT
#use delay(clock=4000000)
#include <math.h>            //Necesaria para los calculos matemáticos
#include <lcd.c>

void main() {
   float tv,tr,temp,y;       //Variables de ecuación
   int16 value;
   lcd_init();
   setup_port_a( ALL_ANALOG );
   setup_adc( ADC_CLOCK_INTERNAL );
```

```
    set_adc_channel( 0 );
    delay_us(10);
    do {
        value = Read_ADC();
        tv = 5.0 * value / 1024.0;
        tr = tv * 10000.0 / (5.0 - tv);
        y = log(tr/20000.0);
        y = (1.0/298.15) + (y *(1.0/4050.0));
        temp=1.0/y;
        temp = temp -273.15;
        printf(lcd_putc,"\nT = %04.2fC", temp);
    } while (TRUE);
}
```

Figura 17.- Programa de ejemplo 2

Ejemplo 3: *Realizar un barómetro/altímetro que mida en Kpa, Psi, Atm o mts mediante su selección por un pulsador* (figura 20). *Componentes ISIS: PIC16F876, MPX4115, NTSA0WB203, BUTTO, CAP y LM016L.*

El sensor de presión MPX4115 de 15 a 115 kPa (2.2 to 16.7 psi) con una tensión de salida de 0.2 a 4.8 V. La función de transferencia se muestra en la figura 18, la ecuación de la tensión de salida es:

$$V_{OUT} = V_S * (0.009 * P - 0.095) \pm (\text{ErrorPresion} * \text{FactorTmp} * 0.009\ V_S)$$

Figura 18. MPX4115 (cortesía de Motorola)

El error de Presión y el factor de temperatura vienen definidos por el fabricante (figura 19). El error de Presión se sitúa en ±1.5 y el factor de temperatura varia entre

1 y 3 según la temperatura. Este factor hace que sea necesario medir la temperatura de trabajo y, por ello, se utilizará una NTC.

Figura 19. Error de Presión y Factor de Temperatura (cortesía de Motorola)

La ecuación para calcular la presión con una V_S de 5V y un error de presión de ±1.5 es de:

$$P = \frac{0.475 + V_{OUT}}{0.045} \pm 1.5 \cdot FactorTmp$$

Una vez calculada la presión en Kpa se relaciona con Psi y Atm:

1 Kpa = 0.0098692 Atm.

1 Kpa = 0.1450377 Psi.

La altitud en metros se puede calcular según la ecuación:

$$H = -7990.6527 \cdot \ln\left(\frac{P_{Kpa}}{101.304}\right)$$

Figura 20. Ejemplo 3

```
#include <16F876.h>
#device adc=10
#FUSES XT,NOWDT
#FUSES
#use delay(clock=4000000)
#include <math.h>
#include <LCD.C>
#BYTE TRISA = 0x85
#BYTE PORTA = 0x05

void main() {
   int16 q;
   float tv,tr,temp,y,tf,error;                //Variables ecuaciones
   float p,presion,pres_atm,pres_psi,alt;      //Variables calculo presion
                                               //y altura
   int cnt=0;
   bit_set (TRISA , 2);
   setup_adc_ports(RA0_RA1_RA3_ANALOG);  //RA0 y RA1 analógica, RA2 digital
```

```
setup_adc(ADC_CLOCK_INTERNAL);
lcd_init();

for (;;) {
   set_adc_channel(0);              //Lectura presión en voltios
   delay_us(20);
   q = read_adc();
   p = 5.0 * q / 1024.0;
   presion= (0.475*p)/0.045;        //Lectura presión en Kpa

   set_adc_channel(1);              //Lectura temperatura
   delay_us(20);
   q = read_adc();
   tv = 5.0 * q / 1024.0;
   tr = tv * 10000.0 / (5.0 - tv);
   y = log(tr/20000.0);
   y = (1.0/298.15) + (y *(1.0/4050.0));
   temp=1.0/y;
   temp = temp -273.15;
      if (temp>=0 && temp<=85) TF=1.0;
      else TF=3.0;
   ERROR = TF * 1.5;  //Calculo del error de presión con la temperatura
                      //el error puede ser ± pero aquí usamos - o +
   presion=presion-ERROR;                  //Presión en Kpa
   pres_atm = presion * 0.0098692;         //Presión en Atm
   pres_psi = presion * 0.1450377;         //Presión en Psi
   alt = -7990.652789*log(presion/101.304);   //Altura
  if (BIT_TEST(PORTA,2)==0) cnt++;  //Calcular número veces pulsa botón
  if (cnt>=4) cnt=0;
   Switch (cnt) {          //Según número veces pulsa botón se elige menu
     case 0:
     lcd_gotoxy(1,1);
     printf(lcd_putc, "\P= %5.2f Kpa    ", PRESION);
     printf(lcd_putc,"\nT = %04.2f C", temp);
     break;
     case 1:
     lcd_gotoxy(1,1);
     printf(lcd_putc, "\P= %4.2f atm    ", PRES_atm);
     printf(lcd_putc,"\nT = %04.2f C", temp);
     break;
     case 2:
     lcd_gotoxy(1,1);
     printf(lcd_putc, "\P= %3.2f psi    ", PRES_psi);
```

```
        printf(lcd_putc,"\nT = %04.2f C", temp);
        break;
        case 3:
        lcd_gotoxy(1,1);
        printf(lcd_putc, "\Alt= %7.2f m   ", alt);
        printf(lcd_putc,"\nT = %04.2f C", temp);
        break;
        }
    delay_ms(100);
    }
}
```

Figura 21. Programa del ejemplo 3

Ejemplo 4: *Simulación de adquisición de tensiones negativas (figura 22). Componentes ISIS: PIC16F876, CELL y LM016L.*

Como se ha comentado al inicio del capítulo, el convertidos AD del PIC sólo puede adquirir tensiones *POSITIVAS*. Tal como está configurada la entrada del conversor, el PIC se "***QUEMARIA***" en el caso de introducir una señal de tensión negativa por los canales AD. Pero la simulación es "muy sufrida" y sí que permite adquirir tensiones negativas sin que al PIC le ocurra nada, pero la realidad es mucho "más cruel"; ¡***OJO*** con está diferencia entre la simulación y la realidad!

Figura 22. Ejemplo 4

```
#include <16F876.h>
#device adc=10
#FUSES XT,NOWDT
#FUSES
#use delay(clock=4000000)
#include <LCD.C>
void main() {
   int16 q;
   float p;
   setup_adc_ports(AN0_VREF_VREF);
   setup_adc(ADC_CLOCK_INTERNAL);
   lcd_init();

   for (;;) {
      set_adc_channel(0);
      delay_us(20);
      q = read_adc();
      p = (-2.0) + (4*q / 1024.0);
      printf(lcd_putc, "\fADC = %4ld", q);
      printf(lcd_putc, "\nVoltage = %01.2fV", p);
      delay_ms(100);
      }
}
```

Figura 23. Programa del ejemplo 4

La solución práctica para medir tensiones negativas es desplazar la tensión hasta valores positivos y después restar este desplazamiento por software.

Capítulo 6

Módulo CCP – Comparador, Captura y PWM

6.1 Introducción

Los módulos *CCP* permiten realizar tres funciones básicas basadas en el manejo de los temporizadores (*Timer*):

- Comparador: compara el valor del temporizador con el valor de un registro y provoca una acción en el PIC.
- Captura: obtiene el valor del temporizador en un momento dado, fijado por la acción de un terminal del PIC.
- *PWM*: genera una señal modulada en amplitud de pulso.

Los PIC de la gama media pueden tener hasta 2 módulos *CCP*. Los dos módulos *CCP* se comportan prácticamente igual (menos en un caso especial que se estudiará posteriormente). Tras producirse un *reset*, el módulo *CCP* se encuentra deshabilitado.

Cada módulo *CCP* posee un registro de 16 bits que puede utilizarse de las tres siguientes formas:

- Registro de 16 bits para capturar el valor del temporizador al producirse un evento (CAPTURA).
- Registro de 16 bits para comparar su valor con el valor del temporizador *TMR1*, pudiendo provocar un evento cuando se alcanza el valor contenido en este registro (COMPARADOR).
- Registro de 10 bits para el ciclo de trabajo de una señal *PWM* (*PWM*).

Cada uno de los registros *CCP* tiene asociados tres registros (la x indica *CCP1* o *CCP2*):

- **CCPxCON**: Registro de control del *CCP*.

- **CCPRxH**: Byte alto del registro de 16 bits del *CCP*.
- **CCPRxL**: Byte bajo del registro de 16 bits del *CCP*.
- **CCPx**: pin del *CCP*.

Registro de control CCPxCON (dirección RAM: 17h/D1h) [PIC16F87x]

Figura 1. Registro de control CCPxCON

U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
		CCPxX	CCPxY	CCPxM3	CCPxM2	CCPxM1	CCPxM0
Bit7							Bit0

bit 7:6 **No usados:** valor 0.

bit 5:4 **CCPxX:CCPxY**: bit1 y bit0 del *Duty Cycle* del *PWM*.

Modo captura: No se usa.

Modo comparación: No se usa.

Modo PWM: Son los dos bits menos significativos de los 10 bits utilizados para el *Duty Cycle* del *PWM*. Los ocho bits de mayor peso del *Duty Cycle* se encuentran en el registro **CCPRxL**.

bit 3:0 **CCPxM3:CCPxM0**: bits de selección del modo *CCPx*.

0000	*CCP* inhabilitado (reset del módulo *CCPx*).
0100	Modo de captura, cada flanco de bajada.
0101	Modo de captura, cada flanco de subida.
0110	Modo de captura, cada 4 flancos de subida.
0111	Modo de captura, cada 16 flancos de subida.
1000	Modo de comparación, pin *CCP* a 1 al igualarse (CCPxIF = 1).
1001	Modo de comparación, pin *CCP* a 0 al igualarse (CCPxIF = 1).
1010	Modo de comparación, genera interrupción al igualarse (CCPxIF = 1, *CCPx* no es afectado).
1011	Modo de comparación, lanza acción especial (CCPxIF = 1, *CCP1* resetea *TMR1*, *CCP2* resetea *TMR1* y lanza una conversión A/D (si está habilitada).
11xx	Modo *PWM*.

Debido a que los dos módulos *CCP* utilizan los temporizadores, cuando están activos los dos módulos se puede dar alguna interacción entre ellos. La siguiente tabla muestra dichas interacciones.

Modo CCPx	Modo CCPy	Interacción
Captura	Captura	Misma base de tiempos en *TMR1*.
Captura	Comparación	La comparación debe configurarse para la acción especial de disparo que borra el *TMR1*.
Comparación	Comparación	La comparación debe configurarse para la acción especial de disparo que borra el *TMR1*.
PWM	PWM	Los *PWM* deberán tener la misma frecuencia y tasa de actualización (interrupción *TMR2*).
PWM	Captura	Ninguna.
PWM	Comparación	Ninguna.

6.2 Modo Captura

En el modo de captura, **CCPRxH:CCPRxL** capturan el valor de los 16 bits del registro *TMR1* cuando ocurre un evento en el pin *CCPx*. Los posibles eventos son:

- Flanco de bajada.
- Flanco de subida.
- 4 flancos de subida.
- 16 flancos de subida.

Estos eventos se seleccionan con los bits **CCPxM3:CCPxM0** (**CCPxCON<3:0>**). En el momento de la captura, el bit **CCP1IF** (**PIR1<2>**) [y/o el **CCP2IF** (**PIR2<0>**)] se pone a 1, produciendo una interrupción en el caso de que esté habilitada. El *flag* ha de ser borrado por software. Si sucede otra captura antes de haber leído el registro **CCPRx**, el valor de la captura previa se habrá perdido.

En modo captura, el pin **RC2/CCP1** (y/o el pin **RC1/T1OSI/CCP2**) debe configurarse como entrada poniendo a uno el bit **TRISC<2>** [y/o el **TRISC<1>**]. Si el bit **RC2/CCP1** [y/o **RC1/CCP2**] se configura como salida, una escritura en este pin podría originar una captura.

Este modo trabaja con el *TMR1*. Además, es necesario que éste funcione como temporizador o como contador en modo síncrono. En modo asíncrono no funcionaría.

Si se produce un cambio en el modo captura, por ejemplo de flanco de bajada a cada 4 flancos, se puede dar una falsa interrupción. Por software se debe borrar el

bit **CCPxIE** para deshabilitar las interrupciones, limpiar el *flag* **CCPxIF** y cambiar el modo de captura.

Mediante el *preescaler* se puede alcanzar una resolución más precisa sobre las señales de entrada. Hay cuatro configuraciones de *preescaler* especificadas mediante los bits **CCPxM3:CCPxM0**. Cuando el módulo *CCPx* está inhabilitado no está en modo captura, el contador del predivisor es puesto a cero. Cualquier tipo de *reset* borra el *preescaler*.

Para cambiar el tipo de *preescaler* se debe apagar antes el módulo *CCPx* (borrar el *preescaler*) y posteriormente modificar dicho valor, de lo contrario se puede producir una interrupción.

Si se utiliza el modo *sleep* (dormido), el *TMR1* (configurado en modo síncrono) no se incrementa, pero el *preescaler* del *CCPx* si que continua incrementando el contador de eventos y, por lo tanto, cuando alcanza el valor prefijado, el bit **CCPxF** se pone a 1, lo que provoca un despertar del PIC; así el contenido del *TMR1* se guarda en los **CCPRx** pero su valor no es significativo dado que el *TMR1* estaba parado.

6.3 Modo Comparación

En el modo comparación el valor de 16 bits del registro **CCPRx** se compara continuamente con el valor del temporizador *TMR1*. Cuando ambos valores se igualan, en el pin **CCPx** se puede producir, o no, un evento de los siguientes:

Se pone a 1.

Se pone a 0.

No cambia.

Estos eventos se seleccionan mediante la configuración de los bits de control **CCPxM3:CCPxM0 (CCPxCON<3:0>)**:

1000	Modo de comparación, pin *CCP* a 1 al igualarse (**CCPxIF = 1**).
1001	Modo de comparación, pin *CCP* a 0 al igualarse (**CCPxIF = 1**).
1010	Modo de comparación, genera interrupción al igualarse (**CCPxIF = 1**, *CCPx* no es afectado).
1011	Modo de comparación, lanza acción especial (**CCPxIF = 1**, *CCP1* resetea *TMR1*, *CCP2* resetea *TMR1* y lanza una conversión A/D (si está habilitada).

Por otra parte, al producirse un evento en el pin, se producirá la interrupción en caso de que esté habilitada, ya que el *flag* **CCPxIF** (de **PIR1** o **PIR2**) se pone a 1.

El *TIMER1* se debe configurar en modo temporizador o modo contador síncrono para que el módulo *CCPx* funcione correctamente en el modo comparación.

Para trabajar en este modo, el pin *CCPx* debe configurarse como salida, poniendo a 0 el bit del registro **TRISC** correspondiente. Cuando se selecciona uno de los modos de comparación, el pin *CCPx* toma el nivel lógico contrario al que tiene que tomar cuando se produzca la igualdad (es decir si se tiene que poner a 1 en la igualdad, se pone a 0 en estado normal)

Hay un modo de trabajo (**CCPxM3:CCPxM0: 1010**) en el que se produce una interrupción al producirse la igualdad, se activa el *flag* **CCPxIF** y se genera la interrupción, si está habilitada, pero el pin *CCPx* no se ve afectado.

Por último, puede trabajar en modo de disparo de acción especial (**CCPxM3:CCPxM0: 1011**). En este caso cuando se produce la igualdad, el temporizador *TIMER1* se resetea, por lo que se utiliza como marcador de la acción. En el módulo *CCP2*, además de producirse el *reset* del *TIMER1*, se inicia una nueva conversión AD si dicho módulo está habilitado; esto permite realizar conversiones A/D periódicas.

En el modo *sleep* (dormido), el *TIMER1* no funciona y, por lo tanto, la comparación tampoco. El pin *CCPx* tendrá el mismo valor que antes de trabajar en modo *sleep*. Después de cualquier *reset*, el modulo *CCP* está deshabilitado.

6.4 Modo PWM

El modo *PWM* (*Pulse Width Modulation*) o *MODULACIÓN DE ANCHO DE PULSO*, permite obtener en los pines *CCPx* una señal periódica en la que se puede modificar su ciclo de trabajo (*Duty Cicle*). Es decir, puede variarse el tiempo en el cual la señal está a nivel alto (T_{ON}) frente al tiempo que está a nivel bajo (T_{OFF}); ver la figura 2. De esta forma, la tensión media aplicada a la carga es proporcional al T_{ON}, controlando, por ejemplo, la velocidad de motores, luminosidad de lámparas, etc.

Figura 2. Señal PWM

La resolución de salida es de hasta 10 bits. Para que este módulo funcione correctamente, el pin debe estar configurado como salida, mediante la configuración del *TRIS* correspondiente.

El periodo de la señal *PWM* se obtiene de configurar el *TIMER2* y el contenido del registro **PR2** (dirección 0x92); ver la figura 3. Este registro de 8 bits contiene los 8 bits más significativos de una cuenta de 10 bits. Para calcular el periodo de la señal PWM se utiliza la siguiente ecuación:

$$PWMT = (PR2 + 1)\cdot 4\cdot T_{OSC}\cdot(\text{Valor del Preescaler del TMR2})$$

Cuando el valor del *TMR2* se iguala al valor de *PR2*, pueden ocurrir los siguientes eventos:

- *TMR2* se borra.
- El pin *CCPx* se pone a 1 (excepción: si el *Duty Cycle* es 0%, el *CCPx* no se pone a 1).
- El valor de **CCPRxL** se carga en el **CCPRxH**, el cual es el que se compara con el *TIMER2* para fijar el *duty cycle*.

Figura 3. Señal de salida PWM

El ciclo de trabajo (*duty cycle*) se define por el valor del registro **CCPRxL** y con los bits **CCP1CON<5:4>** antes de comenzar un nuevo periodo. El registro **CCPRxL** contiene los 8 bit de mayor peso y el **CCP1CON<5:4>** contiene los 2 bits de menor peso. Por lo tanto, se consigue una resolución de 10 bits (**DCxB9:DCxB0**). El temporizador base con el que se compara el contenido de estos registros es el *TMR2*. La siguiente ecuación permite calcular el valor del *Duty Cycle*:

$$\text{PWM } duty\ cicle = (\text{CCPRxL:CCPxCON<5:4>})\cdot T_{OSC}\cdot TMR2_{PRESCALER}$$

Los registros **CCPR1L** y los bits **CCP1CON<5:4>** pueden escribirse en cualquier momento, pero no se cargan en **CCPR1H** hasta que finalice el periodo de la onda *PWM* actual (es decir que se produce la igualdad entre *PR2* y *TMR2*). En este modo el **CCPR1H** es de sólo lectura.

El *CCPx* se pone a 0, terminando el ciclo de trabajo, cuando el *TMR2* más 2 bits se iguala a **CCPxH** más 2 bits

Si el ciclo de trabajo de la onda *PWM* es mayor que el periodo, la señal que sale por la patilla *CCPx* está siempre a 1.

La resolución máxima en bits viene dada por la expresión:

$$resolucion = \frac{\lg\left(\frac{F_{OSC}}{F_{PWM}}\right)}{\lg 2} bits$$

Por ejemplo, para una frecuencia de *PWM* de 1.22 kHz con un *preescaler* de *TMR2* de 16 y una frecuencia de reloj de 20 MHz:

$$\frac{1}{1220} = (PR2 + 1) \cdot 4 \cdot \frac{1}{20 \cdot 10^6} \cdot 16$$

se obtiene un valor de **PR2 = 255 (0xFF)**, y una resolución de:

$$resolucion = \frac{\lg\left(\frac{20 \cdot 10^6}{1200}\right)}{\lg 2} > 10\ bits$$

Para poner en marcha el modo *PWM* se deben dar los siguientes pasos:

1. Configurar el periodo *PWM* mediante escritura del registro *PR2*.
2. Configurar el *Duty Cycle* escribiendo en el registro **CCPR1L** y los bits **CCP1CON<5:4>**.
3. Configurar *CCPx* como salida mediante el *TRISC*.
4. Configurar el *preescaler* del *TMR2* y habilitarlo mediante escritura en **T2CON**.
5. Configurar el módulo *CCP* para la operación en modo *PWM*.

6.5 Módulo CCP en C

El compilador C suministra una serie de funciones para el manejo del módulo *CCP*.

Configuración del módulo *CCPx*:

setup_ccpx (modo);

modo hace referencia a los bits **CCPxM3:CCPxM0** del registro **CCPxCON** (figura 4).

SETUP_CCPx(MODO);	MODO	Registro CCPxCON (17h/D1h)
CCP_OFF	Deshabilitación	00000000 (00h)
CCP_CAPTURE_FE	Captura por flanco de bajada	00000100 (04h)
CCP_CAPTURE_RE	Captura por flanco de subida	00000101 (05h)
CCP_CAPTURE_DIV_4	Captura tras 4 pulsos	00000110 (06h)
CCP_CAPTURE_DIV_16	Captura tras 16 pulsos	00000111 (07h)
CCP_COMPARE_SET_ON_MATCH	Salida a 1 en comparación	00001000 (08h)
CCP_COMPARE_CLR_ON_MATCH	Salida a 0 en comparación	00001001 (0A9)
CCP_COMPARE_INT	Interrupción en comparación	00001010 (0Ah)
CCP_COMPARE_RESET_TIMER	Reset TMR en comparación	00001011 (0Bh)
CCP_PWM	Habilitación PWM	00001100 (0Ch)

Figura 4. Modos de SETUP_CCPx(MODO)

Los valores para comparar se fijan en los registros **CCPRx**. En el compilador C, estos registros están definidos en el fichero *include*, por ejemplo en el 16F87x.h:

long CCP_1;	long CCP_2;
#byte CCP_1 = 0x15	#byte CCP_2 = 0x1B
#byte CCP_1_LOW = 0x15	#byte CCP_2_LOW = 0x1B
#byte CCP_1_HIGH = 0x16	#byte CCP_2_HIGH = 0x1C

Definición del ciclo de trabajo para PWM:

set_pwmx_ duty (valor);

valor: dato de 8 o 16 bits que determina el ciclo de trabajo. Este valor, junto con el valor del *preescaler* del *TMR2*, determina el valor del ciclo de trabajo. En la configuración del *TIMER2*, el *postcaler* debe valer 1.

Figura 5. Ejemplo 1

Ejemplo 1: *Medir el ancho de un pulso mediante el módulo CCP* (figura 5). *Componentes ISIS: PIC16F876 y LM016L. Instrumentos: OSCILLOSCOPE y Generadores: PULSE.*

Se utiliza el modo captura del *CCP*, configurándolo para que detecte el flanco de subida o de bajada del pulso a medir. Este ejemplo es similar al ejemplo 4 del tema de interrupciones, pero en ese la detección era por interrupción de *RB0*.

Cada vez que se produzca una detección de flanco, el valor del *TMR1* pasará al registro del módulo *CCP*.

```
#include <16f876.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#include <lcd.c>
#byte PIR1=0x0C

int1 nuevopulso=0;              //Entra otro pulso
int16 TFB=0,TFS=0,TF=0;         //Tiempo flancos
float AP=0.0;                   //Valor final del ancho de pulso
int1 cambio=0;                  //Cambio de flanco de disparo

#int_ccp1
void ccp1_int(){        //Función interrupción
  if(cambio==0){        //Flanco de subida
    TFS=CCP_1;          //Carga del valor del registro CCPR1 en flanco subida
    setup_ccp1(CCP_CAPTURE_FE); //Configuración modo Captura en flanco bajada
    cambio=1;                   //Control de cambio de flanco
  } else {                      //Flanco de Bajada
  TFB=CCP_1;        //Carga del valor del registro CCPR1 en flanco bajada
    setup_ccp1(CCP_CAPTURE_RE); //Configuración modo Captura en flanco subida
    cambio=0;                   //Control de cambio de flanco

  if(nuevopulso==0){            //Fin de pulso...
   nuevopulso=1;                //pulso a medir
  }
  }
}

void main() {

lcd_init();
setup_timer_1(T1_INTERNAL);     //Configuración TMR1
setup_ccp1(CCP_CAPTURE_RE);     //Configuración modo Captura en flanco subida
cambio = 0;                     //Control de cambio a 0
```

```
enable_interrupts(int_ccp1);    //Habilitación interrupción modulo CCP
enable_interrupts(global);      //Habilitación interrupción global
  do {
    if(nuevopulso==1){          //¿Pulso nuevo?
       TF=(TFB-TFS);            //Ancho de pulso.
       AP = TF*1.0;           //Ancho de pulso en microsegundos (a 4 MHz:1 µs)
     printf(lcd_putc,"\nPulso = %6.1fuS ", AP);
    nuevopulso=0;               //Pulso ya medido, espera nuevo
   }
  } while (TRUE);
}
```

Figura 6. Programa del ejemplo 1

Ejemplo 2: *Generar una señal cuadrada de 2 kHz mediante el módulo CCP* (figura 7). *Componentes ISIS: PIC16F876 y LM016L. Instrumentos: OSCILLOSCOPE y COUNTER TIME.*

Figura 7. Ejemplo 2

Se utiliza el modo comparación del *CCP*, configurándolo en modo conmutación del pin *CCP* por comparación. Este ejemplo es similar a los ejemplos 1 y 5 del tema de interrupciones, pero en esos la detección era por interrupción del *TRM0* y *TMR2*, respectivamente.

El módulo *CCP* compara continuamente el valor de *TMR1* con el valor prefijado; cuando se produce la igualdad se produce el cambio de estado del pin *CCP* y la activación de la interrupción del módulo *CCP*.

En cada conmutación se debe fijar el valor a comparar para obtener una onda cuadrada (un ciclo de trabajo del 50%), es decir un semiperiodo de 250 µs (a 4 MHz el ciclo máquina es de 1 µs), por lo tanto, el *CCP2* se debe cargar con este valor (ajustando el valor final es de 199).

```
#include <16f876.h>
#fuses XT,NOWDT
int1 cambio=0;              //Variable de control de cambio

#int_ccp2
void ccp2_int(){            //Función de interrupción
  if(++cambio==1){
    setup_ccp2(CCP_COMPARE_CLR_ON_MATCH); //Modo Comparación, cambio a 0
  } else{
    setup_ccp2(CCP_COMPARE_SET_ON_MATCH); //Modo Comparación, cambio a 1
  }
  set_timer1(0);                          //Borrado de TMR1
  CCP_2 = 199;                          //Inicialización del registro CCPR2
para un Duty del 50%
}
void main() {
disable_interrupts(global);
setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);     //Configuración TMR1
setup_ccp2(CCP_COMPARE_SET_ON_MATCH); //Configuración inicial modulo CCP
CCP_2 = 199;                          //Inicialización del registro CCPR2
                                      //para un Duty del 50%
enable_interrupts(int_ccp2);      //Habilitación interrupción modulo CCP2
enable_interrupts(global);         //Habilitación interrupción general
do {
  } while (TRUE);
 }
```

Figura 8. Programa del ejemplo 2

Ejemplo 3: *Mediante la configuración del módulo CCP lanzar una conversión AD, automática cada 1 ms. Con el valor obtenido se realizará una conversión DA utilizando el PWM y un filtro paso-bajo* (figura 10). *Componentes ISIS: PIC16F876, CAP, CELL, LED-BARGRAPH-GRN, LM3914, POT-LIN y RES. Instrumentos: OSCILLOSCOPE.*

Se configura el módulo *CCP2* en modo comparación con acción especial de disparo, cuando se produce la igualdad se resetea el *TMR1* se y se lanza una conversión AD, si está habilitada. Cargando el registro de *CCP2* con el valor adecuado se consigue

que, cuando coincida con el valor de *TMR1*, se produzca una conversión AD, en este caso cada 1 ms. Dado que la frecuencia de trabajo es 4 MHz (1 ms/1 μs=1000), el registro *CCP2* se carga a 1000.

Figura 9. Señal de 2 KHz (ejemplo 2)

Con el valor obtenido de la conversión se puede obtener una señal *PWM* proporcional a este valor. Para ello se utiliza el módulo *CCP1*, dado que el valor de la conversión es de 10 bits y el registro del módulo *PWM* también es de 10 bits, el valor puede ser transferido directamente.

El periodo de la señal *PWM* viene fijado por el *TMR2*, con un *prescaler* y un *postcaler* de valor 1 y un valor de registro *PR2* de 224, lo que implica un periodo de 225 μs (4444 Hz).

Si dicha señal es filtrada con un filtro de paso bajo se obtiene una corriente continua proporcional al valor de la conversión. Como aplicación se puede aplicar esta señal a una barra de *leds* y obtener una señal luminosa proporcional a la señal adquirida (*voltmeter*). Para ello se utiliza el circuito integrado LM3914 (figura 11).

En la figura 10 se observa el funcionamiento del circuito; con una pila y un potenciómetro se varía la tensión de entrada, la cual es muestreada cada 1 ms. Con este valor se genera una señal *PWM* (ver canal A del osciloscopio) y mediante el filtro de paso bajo se obtiene la tensión media proporcional (ver canal B del osciloscopio). Dicha tensión se inyecta a la entrada del LM3914 que proporciona la señal a los *leds*.

Figura 10. Ejemplo 3

Figura 11. Aplicación típica del LM3914 (cortesía National SMC)

```
#include <16f876.h>
#device adc=10
#fuses XT,NOWDT
int16 valor;
#int_ad
```

```
void ad_int(){                          //Función interrupción AD
valor=read_adc();                       //Valor de fuente analógica...
set_pwm1_duty(valor);                   //a Duty de PWM
}
void main() {
  disable_interrupts(global);

  setup_adc_ports(AN0);                  //Habilitación RA0 analógico
  setup_adc(ADC_CLOCK_INTERNAL);        //Reloj interno RC
  set_adc_channel(0);                   //Canal 0

  setup_timer_2(T2_DIV_BY_1,224,1);          //PR2=224, Tpwm=225 us
  setup_ccp1(CCP_PWM);                       //CCP1 en modo PWM
  setup_ccp2(CCP_COMPARE_RESET_TIMER);       //CCP2 modo COMPARACION...
                                             // y disparo especial
  setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);  //Configuración TMR1
  set_timer1(0);                             //Puesta a 0
  ccp_2=1000;                                //Muestreo cada 1 ms a 4 MHz
  enable_interrupts(INT_AD);        //Habilitación Interrupción AD
  enable_interrupts(global);        //Habilitación Interrupción global
do {
  } while (TRUE);
}
```

Figura 12. Programa del ejemplo 3

Figura 13. Señal modulada y señal filtrada

NOTA

Puede detectarse un error en el módulo *CCP* en modo *PWM* con valores altos (ver el último ejemplo de este capítulo).

Ejemplo 4: *Mediante la configuración del módulo CCP lanzar una conversión AD automática. Con el valor obtenido representar la tensión de entrada en un display gráfico (figura 14). Componentes ISIS: PIC16F877, CELL, VSIN, LGM12641BS1R. Instrumentos: OSCILLOSCOPE.*

En este ejercicio hay que tener en cuenta dos factores. Por un lado que la frecuencia de muestreo debe ser, por lo menos, dos veces mayor que la frecuencia a muestrear; en este caso si la frecuencia de muestreo es de 5 kHz (200 μs), la frecuencia de la señal a muestrear debe ser inferior a 2 KHz.

Por otro lado, hay que ajustar el rango de tensión de entrada a valores positivos. Se utiliza una fuente de continua de 2.5 V para elevar una fuente senoidal de 5 V de pico a pico con una frecuencia de 10 Hz (también se puede desplazar la tensión con el *offset* de la fuente VSINE). Para representar esta tensión se ajusta el tamaño de la pantalla: en el eje de las X el tiempo (de 1 a 128 pixeles), en el eje de las Y la tensión (de 1 a 64 pixeles).

Figura 14. Ejemplo 4

```
#include <16f877.h>
#device adc=10
```

```
#use delay(clock=20000000)
#fuses HS,NOWDT
#include <HDM64GS12.C>
#include <graphics.c>
int16 valor;
float tension;
#int_ad
void ad_int(){                          //Función interrupción AD
valor=read_adc(ADC_READ_ONLY);          //Valor de fuente analógica...
ccp_2=1000;                             //reinicia cuenta
}
void main() {
  int8 xa=128,ya=0;
  glcd_init(ON);
  disable_interrupts(global);
  setup_adc_ports(AN0);                 //Habilitación RA0 analógico
  setup_adc(ADC_CLOCK_INTERNAL);        //Reloj interno RC
  set_adc_channel(0);                   //Canal 0
  setup_ccp2(CCP_COMPARE_RESET_TIMER);        //CCP2 modo COMPARACION...
                                              // y disparo especial
  setup_timer_1(T1_INTERNAL | T1_DIV_BY_1);          //Configuración TMR1
  set_timer1(0);                                     //Puesta a 0
  ccp_2=1000;                                 //Muestreo cada 200 us a 20 MHz
  enable_interrupts(INT_AD);            //Habilitación Interrupción AD
  enable_interrupts(global);            //Habilitación Interrupción global
  glcd_fillscreen(0);                   //Borrado de pantalla
  glcd_line(1,64,1,1,1);                //lineas de x e y
  glcd_line(1,31,128,31,1);
do {
    tension=(valor*5.0)/1024.0;         //Vin en voltios (de 0 a 5V)
    ya=1.0+12.6*tension;        //Escalado de Vin, y de 1 a 64 (y=1+64/5*Vin)
    glcd_pixel(xa,ya,1);
    xa--;
        if (xa<2)
       {xa=128;                         //Eje de tiempos (x de 1 a 128)
          glcd_fillscreen(0);
          glcd_line(1,64,1,1,1);        //lineas de x e y
          glcd_line(1,31,128,31,1);
       }
  } while (TRUE);
}
```

Figura 15. Programa del ejemplo 4

Ejemplo 5: *Realizar un control PID para regular la temperatura de un horno* (figura 16). *Componentes ISIS: PIC16F877, IRL1004, OVEN, RES y CELL. Instrumentos: OSCILLOSCOPE.*

Figura 16. Ejemplo 5

Uno de los controladores más comunes que se utilizan en el control de temperatura es el *PID* (Proporcional-Integral-Derivada). Sin entrar en teoría de control, se puede indicar que un controlador *PID* responde a la siguiente ecuación:

$$u(t) = K_p e(t) + \frac{K_p}{T_i}\int_0^t e(t)dt + K_p T_d \frac{de(t)}{dt}$$

donde e(t) es el error de la señal y u(t) es la entrada de control del proceso. K_p es la ganancia proporcional, T_i es la constante de tiempo integral y T_d es la constante de tiempo derivativa.

En el dominio s, el controlador *PID* se puede escribir como:

$$U(s) = K_p\left[1 + \frac{1}{T_i s} + T_d s\right]E(s)$$

Un controlador *PID* tiene tres parámetros (K_p, T_i, T_d) los cuales interactúan uno con el otro y su ajuste para obtener el mejor control puede ser muy complicado.

Ziegler/Nichols sugirieron valores para los parámetros del control *PID* basados en análisis de lazo abierto y lazo cerrado del proceso a controlar. En lazo abierto, muchos procesos pueden definirse según la siguiente función de transferencia:

$$G(s) = \frac{\kappa_0 \cdot e^{-s\tau_0}}{(1 + s\gamma_0)}$$

donde los coeficientes κ_0, τ_0 y γ_0 se obtienen de la respuesta del sistema en lazo abierto a una entrada escalón. Se parte del sistema estabilizado en $y(t) = y_0$ para $u(t) = u_0$; se aplica una entrada escalón de u_0 a u_1 (el salto debe estar entre un 10 y un 20% del valor nominal) y se registra la respuesta de la salida hasta que se estabilice en el nuevo punto de operación.

Figura 17. Respuesta de salida a una entrada escalón

Los parámetros se pueden obtener de esta respuesta:

$$\tau_0 = t_1 - t_0$$

$$\gamma_0 = t_2 - t_1$$

$$\kappa_0 = \frac{y_1 - y_0}{u_1 - u_0}$$

Según Ziegler/Nochols, las relaciones de estos coeficientes con los parámetros del controlador son:

$$K_p = \frac{1.2 \cdot \gamma_0}{\kappa_0 \cdot \tau_0} \qquad T_i = 2 \cdot \tau_0 \qquad T_d = 0.5 \cdot \tau_0$$

La realización de un controlador *PID* discreto viene dado por la transformada z:

$$U(z) = E(z)K_p\left[1 + \frac{T}{T_i(1-z^{-1})} + T_d\frac{(1-z^{-1})}{T}\right]$$

también:

$$\frac{U(z)}{E(z)} = a + \frac{b}{1-z^{-1}} + c(1-z^{-1})$$

donde:

$$a = K_p \qquad b = \frac{K_p \cdot T}{T_i} \qquad c = \frac{K_p \cdot T_d}{T}$$

Existen distintas posibilidades de la realización práctica de un controlador *PID*; una de las más habituales es la realización en paralelo (figura 18).

Figura 18. Diseño paralelo del controlador PID

El algoritmo utilizado para programar el PIC se muestra en la figura 19 de la página siguiente. El muestreo debe ser mucho menor que el tiempo de establecimiento del sistema en lazo abierto. En el modelo Ziegler/Nichols se toma un valor $T < \tau_0/4$ (también puede utilizarse $T < \gamma_0/10$).

Un problema asociado a este tipo de diseño es el llamado *"integral windup"*, el cual puede provocar largos periodos de sobreimpulsos (*overshoot*), motivados por los valores excesivos que alcanza la señal de control debido a la acumulación en el integrador. Para evitar este problema se suele limitar la señal de control entre un valor máximo y otro mínimo, impidiendo que el integrador actúe cuando se superan esos límites.

Figura 19. Algoritmo de programación del PID

En el ejemplo, el control se realiza sobre un horno con una resistencia calefactora. Se actúa sobre dicha resistencia mediante una señal *PWM* generada en función del control *PID*. Para facilitar la simulación se alimenta la resistencia con una fuente

de corriente continua de 12 V y se modifica su potencia de calentamiento (editar el componente con el botón derecho) a 120 W (figura 20).

Figura 20. Caracteristicas del OVEN

Dado que la corriente máxima es de 10 A se ha utilizado un *MOSFET* de potencia, el IRL1004.

Como sensor de temperatura se puede utilizar cualquier dispositivo NTC, RTD, termopares, etc.; pero el *OVEN* tiene un terminal que indica la temperatura del horno en grados Celsius. Será este terminal el que se utilice para introducir al PIC la temperatura del horno; dado que la temperatura está medida directamente en grados Celsius es conveniente utilizar (para este ejemplo) un divisor por 100 que permita obtener el valor en milivoltios de la temperatura (25 °C serán 250 mV).

Figura 21. Caracteristicas del OVEN

La temperatura a alcanzar (o *set point*) se podría introducir por teclado (y *LCD*) pero, para simplificar, se establece directamente en el programa (se deja al lector modificar dicho programa para poder variar la temperatura de *set point*). La temperatura inicial se puede modificar en el menú de edición del componente *OVEN*.

El resto de características a modificar en el componente *OVEN* son (figura 22):

- *Temperature coefficient (V/°C):* da una idea de la temperatura que alcanzará el horno según la tensión aplicada. Valor: 1 V/°C.
- *Oven time Constant (sec):* es la constante de tiempo del horno. Para evitar una larga simulación se ajusta a 10 segundos.
- *Heater time Constant (sec):* es la constante de tiempo del calefactor. Para evitar una larga simulación se ajusta a 1 segundo.
- *Thermal resistance to ambient (°C/W):* resistencia térmica horno-ambiente. Se ajusta al mismo valor que el dado por defecto, 0.7.

Figura 22. Características del OVEN

Para determinar los parámetros (κ_0, τ_0 y γ_0) del control *PID* basados en el análisis de lazo abierto según Ziegler/Nichols, se realiza un análisis transitorio del horno con una entrada escalón de 0 a 2 V (10% a 20% del valor nominal); ver figura 23.

Para trazar la gráfica se utiliza la *Simulation graphs* en el ISIS (figura 24); el tipo *interactive*. En el comando *GRAPH/ADD TRACE* (figura 25) se añade la traza de la sonda de tensión colocada en la salida de temperatura del horno. Al realizar la simulación temporal de la forma habitual (o con la barra espaciadora) se lanza también la simulación gráfica y durante el tiempo fijado en la gráfica (editando sus características -figura 26-), el valor de la sonda se reflejará en dicha gráfica (se deja abierto el interruptor un tiempo y después se cierra para provocar el escalón

de entrada). Realizando una pulsación en la barra de título de la gráfica se puede ampliar y utilizar un cursor para realizar las medidas; también se puede exportar la gráfica a un fichero.

Figura 23. Respuesta a una entrada escalón

Figura 24. Simulation Graphs

Figura 25. Menú Graph

Figura 26. Características del gráfico

De la recta de máxima pendiente se deducen los parámetros κ_0, τ_0 y γ_0 definidos para el análisis en lazo abierto de Ziegler/Nichols (figura 27).

Figura 27. Determinación de los parámetros κ_0, τ_0 y γ_0

Los parámetros K_p, T_i y T_d se calculan en base a éstos:

$$K_p = \frac{1.2 \cdot \gamma_0}{\kappa_0 \cdot \tau_0} = \frac{1.2 \cdot 13}{125.5 \cdot 1} = 0.1243$$

$$T_i = 2 \cdot \tau_0 = 2 \cdot 1 = 2$$

$$T_d = 0.5 \cdot \tau_0 = 0.5 \cdot 1 = 0.5$$

Los parámetros del controlador discreto se calculan en base al tiempo T que según Ziegler/Nichols es de 0.1 s por ser menor de $T < \tau_0/4$.

$$a = K_p = 0.1243$$

$$b = \frac{K_p \cdot T}{T_i} = \frac{0.1243 \cdot 0.1}{2} = 0.0062$$

$$c = \frac{K_p \cdot T_d}{T} = \frac{0.1243 \cdot 0.5}{0.1} = 0.6215$$

```
#INCLUDE <16F877.h>
#device adc=10
#use delay(clock=4000000)
#fuses XT,NOWDT
#BYTE TRISC = 0x87

void main(){
int16 valor;                                    //lectura de temperatura
int16 control;                                  //valor del PWM
float a,b,c;                                    //constantes del PID
float temp_limit;                               //temperatura a alcanzar
float rT,eT,iT,dT,yT,uT,iT0,eT0,iT_1,eT_1;      //variables de ecuaciones
float max,min;                          //limites maximo y minimo de control
min=0.0;                                //inicialización variables
max=1000.0;
iT_1=0.0;
eT_1=0.0;
a=0.1243;                               //constantes del PID
b=0.0062;
c=0.6215;
temp_limit=500.0;                       //Temperatura a alcanzar
TRISC=0;
setup_timer_2(t2_div_by_4,249,1);       //periodo de la señal PWM a 1ms
setup_ccp1(ccp_pwm);                    //Módulo CCP a modo PWM
setup_adc_ports(all_analog);            //Puerto A analógico
setup_adc(ADC_CLOCK_INTERNAL);         //reloj convertidor AD interno
set_adc_channel(0);                     //Lectura por el canal 0
while(1) {
  valor=read_adc();                     //Lectura de la temperatura
  yT=valor*5000.0/1024.0;              //conversión a mV (0.25V a 250mV)
  rT=temp_limit;
  eT=rT-yT;                            //Cálculo error
  iT=b*eT+iT0;                          //Cálculo del término integral
```

```
  dT=c*(eT-eT0);                    //Cálculo del término derivativo
  uT=iT+a*eT+dT;                    //Cálculo de la salida PID
  if (uT>max) {                     //Salida PID si es mayor que el MAX
     uT=max;}
  else {
     if (uT<min){                   //Salida PID si es menor que el MIN
        uT=min;}
        }
  control=uT;                 //Transferencia de salida PID a señal PWM
  set_pwm1_duty(control);
  iT0=iT;                           //Guardar variables
  eT0=eT;
  delay_ms(100);                    //Tiempo de muestreo
  }
}
```

Figura 28. Programa del ejemplo 5

La respuesta del sistema se muestra en la figura 29; si se reduce el tiempo de muestreo a 1 ms se reduce el sobreimpulso (figura 30).

Figura 29. Respuesta con T = 100 ms

Figura 30. Respuesta con T = 1 ms

Ejemplo 6: *Problemas en la simulación del módulo CCP en modo PWM con el ISIS (figura 16). Componentes ISIS: PIC16F876. Instrumentos: OSCILLOSCOPE.*

Se ha detectado un problema, que pronto será resuelto por LabCenter, en el módulo *CCP* trabajando en modo *PWM*. Los valores del *dutty* pueden ir de 0 a 1023 dando una señal modulada desde 0 V con el valor 0 a 5 V con el valor 1023; pero en el *ISIS*, a partir del valor 900 la señal se convierte en una señal cuadrada con un ciclo del 50% y un periodo doble del que tenía. En el ejemplo se observa este defecto.

Figura 31. Ejemplo 6

Figura 32. *PWM con valor menor de 900*

Figura 33. PWM con valor mayor de 900

```
#include <16f876.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
int16 i;
void main() {
  setup_timer_2(T2_DIV_BY_1,224,1);   //PR2=224, Tpwm=225µs
  setup_ccp1(CCP_PWM);                //CCP1 en modo PWM
for (i=0;i<1024;i++){
  set_pwm1_duty(i);                   //s Duty de PWM
  delay_ms(25);
  }
}
```

Figura 34. Programa del ejemplo 6

Capítulo 7

Transmisión serie

7.1 Introducción

Los PIC utilizan, entre otros, dos modos de transmisión en serie:

- El puerto serie síncrono (*SSP*).
- La interfaz de comunicación serie (*SCI*) o receptor transmisor serie síncrono-asíncrono universal (*USART*).

El *SSP* se suele utilizar en la comunicación con otros microcontroladores o con periféricos. Las dos interfaces de trabajo son:

- Interfaz serie de periféricos (*SPI*): desarrollada por Motorota para la comunicación entre microcontroladores de la misma, o diferente, familia en modo maestro-esclavo; *Full-duplex*.
- Interfaz Inter-Circuitos (I^2C): Interfaz desarrollado por Philips, con una gran capacidad para comunicar microcontroladores y periféricos; *Half-duplex*.

La configuración *USART* (transmisor-receptor serie síncrono-asíncrono universal), también conocido como *SCI* (interfaz de comunicación serie), permite la comunicación con un ordenador trabajando en modo *full-duplex* asíncrono o con periféricos trabajando en modo *half–duplex*. En general, puede trabajar de dos formas:

- Asíncrono (*full-duplex*).
- Síncrono (*half-duplex*).

Otros tipos de comunicación soportados por los PIC son: *1-Wire bus*, *LIN* (Local *Interconnect Network*), *USB* (*Universal Serial Bus*), el *CAN* (*Controller Area Network*) y *Ethernet*.

7.2 El módulo USART/SCI

7.2.1 Introducción

Algunos PIC disponen del módulo de comunicación serie *USART/SCI*, tal vez el más utilizado entre los módulos de interfaz serie.

La principal función del *USART* es la de transmitir o recibir datos en serie. Esta operación puede dividirse en dos categorías: síncrona o asíncrona. La transmisión síncrona utiliza una señal de reloj y una línea de datos, mientras que en la transmisión asíncrona no se envía la señal de reloj, por lo que el emisor y el receptor deben tener relojes con la misma frecuencia y fase. Cuando la distancia entre el emisor y el receptor es pequeña se suele utilizar la transmisión síncrona, mientras que para distancias mayores se utiliza la transmisión asíncrona.

El *USART* puede transmitir o recibir datos serie. Puede transferir tramas de datos de 8 o 9 bits por transmisión y detectar errores de transmisión. También puede generar interrupciones cuando se produce una recepción de datos o cuando la transmisión ha sido completada.

Algunos PIC tienen un *USART* direccionable o *AUSART* (*Addresable USART*) que utiliza el noveno bit de datos para distinguir entre la recepción de datos o de dirección. En algunos PIC se ha mejorado el módulo *USART* dando lugar al *EUSART* o *USART* mejorado, el cual permite la detección automática de baudios, el despertar automático al recibir la señal de sincronismo o la transmisión del carácter *Break* de 12 bits, permitiendo su utilización en sistemas de redes de interconexión local (bus *LIN*).

Básicamente, la transmisión serie consiste en enviar los datos bit a bit a través de una línea común en periodos de tiempo fijos, dando lugar a la llamada velocidad de transmisión o número de bits enviados por segundo (baudios). Tanto el emisor como el receptor poseen registros de desplazamiento para realizar la comunicación. Los bits están codificados en *NRZ* (nivel alto: 1, nivel bajo: 0), *NRZI* (cambio de nivel: 1, sin cambio de nivel: 0), etc.

En el modo síncrono se permite la transmisión continua de datos y no existe un límite de tamaño, es un modo *semi-duplex* (la comunicación serie se establece a través de una única línea, en ambos sentidos, pero no se puede transferir información en ambos sentidos de forma simultánea). En este modo de transmisión se puede trabajar de dos formas:

- En modo Maestro, donde el microcontrolador maestro genera la señal de reloj e inicia o finaliza la comunicación.
- En modo Esclavo, donde el microcontrolador esclavo recibe la señal de reloj y depende del microcontrolador maestro para recibir o enviar información.

Figura 1. Transmisión sincrona

En el modo asíncrono se emplean relojes tanto en el emisor como en el receptor. Ambos relojes deben ser de igual frecuencia y deben estar en fase o sincronizados. La frecuencia de reloj se acuerda antes de la transmisión configurando la velocidad mientras que la sincronización se realiza durante la transmisión. Cada trama de datos tiene un tamaño fijo y poseen un bit inicial o de arranque (*start*) y un bit final o de parada (*stop*) que permiten realizar dicha sincronización. La transmisión es en modo *full-duplex* (se utilizan dos líneas, una transmisora -TX- y otra receptora -RX-, transfiriendo información en ambos sentidos; se puede transmitir y recibir información de forma simultánea).

Figura 2. Transmisión asíncrona

El modo más habitual de transmisión por el *USART* es el modo asíncrono, puesto que permite comunicaciones en largas distancias. Existen distintas normas de transmisión serie asíncrona, como la *RS232*, la *RS485*, etc. Los niveles de tensión empleados en estas normas son diferentes al empleado por el PIC. Por ello, suele ser necesaria la utilización de circuitos externos de adaptación

Los terminales utilizados en el modulo *USART* son el **RC6/TX/CK** y el **RC7/RX/DT**:

- En el modo síncrono maestro, el pin **RC6/TX/CK** se utiliza como señal de reloj (de salida) y el **RC7/RX/DT** como línea de datos a enviar o recibir.
- En el modo síncrono esclavo, el pin **RC6/TX/CK** se utiliza como señal de reloj (de entrada) y el **RC7/RX/DT** como línea de datos a enviar o recibir.
- En el modo asíncrono, el pin **RC6/TX/CK** se utiliza como terminal de transmisión de datos y el **RC7/RX/DT** como terminal de recepción de datos.

Los registros asociados al módulo *USART/SCI* son:

- **SPBRG**: Generador del ratio de baudios.
- **TXSTA**: Estado de transmisión y control.

- **RCSTA**: Estado de recepción y control.
- **TXREG**: Registro de datos de transmisión.
- **RCREG**: Registro de datos de recepción.
- **PIR1**: *Flag* de interrupción.
- **PIE1**: Habilitación de la interrupción.

Registro TXSTA (dirección RAM: 98h) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
CSRC	TX9	TXEN	SYNC		BRGH	TRMT	TX9D
Bit7							Bit0

Figura 3. El registro TXSTA

bit 7: **CSRC**: Bit de selección de la fuente de reloj.

En modo asíncrono no interviene.

En modo síncrono:

1 = Modo maestro (genera señal de reloj mediante **BRG**).

0 = Modo esclavo (fuente de reloj exterior).

bit 6: **TX9**: Bit de habilitación de la transmisión de 9 bits:

1 = Transmisión de 9 bits.

0 = Transmisión de 8 bits.

bit 5: **TXEN**: Bit de habilitación de la transmisión:

1 = Transmisión habilitada.

0 = Transmisión deshabilitada.

bit 4: **SYNC**: Bit de selección del modo del *USART*:

1 = Transmisión sincrona.

0 = Transmisión asincrona.

bit 3: No implementado. Se lee como 0.

bit 2: **BRGH**: Bit de selección del valor de baudios.

Modo asíncrono:

1 = Alta velocidad.

0 = Baja velocidad.

No se utiliza en el modo sincrono.

bit 1: **TRMT**: Bit de estado del registro *TSR*:

1 = *TSR* vacío.

0 = *TSR* lleno.

bit 0: **TX9D**: 9 bit de datos transmitidos. Puede ser el bit de paridad.

Figura 4. Esquema del proceso de transmisión

El *USART* puede configurarse para transmitir 8 o 9 bit de datos configurando el bit **TX9** del registro **TXSTA**. Si se utiliza el formato de 9 bits, el noveno bit debe colocarse en el bit **TX9D** del registro **TXSTA** antes de escribir los 8 bit en el registro **TXREG**. Una vez están todos los bits en dicho registro, son transferidos al registro de desplazamiento de transmisión (**TSR**). Desde allí son transmitidos al ciclo de reloj por el pin **TX** comenzando por el bit de *start* y terminando por el bit de *stop*.

Figura 5. El envío de las tramas

Registro RCSTA (dirección RAM: 18h) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	U-0	R-0	R-0	R-0
SPEN	RX9	SREN	CREN		FERR	OERR	RX9D
Bit7							Bit0

Figura 6. El registro RCSTA

bit 7: **SPEN**: Bit de habilitación del puerto serie.

1 = Habilitado (**RX/DT** y **TX/CK** como puerto serie).

0 = Deshabilitado.

bit 6: **RX9**: Bit de habilitación de la recepción de 9 bits.

1 = Recepción de 9 bits.

0 = Recepción de 8 bits.

bit 5: **SREN**: Bit de habilitación de recepción sincrona.

No utilizado en modo asíncrono.

Modo síncrono:

1 = Recepción única habilitada.

0 = Deshabilitada. (Se pone a 0 después de una recepción completa).

bit 4: **CREN**: Bit de habilitación de recepción continua.

Modo asincrono:

1 = Habilitada.

0 = Deshabilitada.

Modo sincrono:

1 = Habilitada hasta que el bit **CREN** es puesto a 0.

0 = Deshabilitada.

bit 3: No implementado. Se lee como 0.

bit 2: **FERR**: Bit de error de trama.

1 = Error (Se actualiza al leer **RCREG**).

0 = No error.

bit 1: **OERR**: Bit de error de *overrrun*.

1 = Error (Se pone a 0 si **CREN** es 0).

0 = No error.

bit 0: **RX9D**: 9 bit de datos transmitido

Figura 7. El esquema del proceso de la recepción de datos

El *USART* puede configurarse para recibir 8 o 9 bit configurando el bit **RX9** del registro **RCSTA**. Después de la detección del bit de *start*, los 8 o 9 bits entrantes por el pin **RX** son desplazados por el registro de desplazamiento de entrada (**RSR**) uno a uno. Después de que el último bit ha sido desplazado dentro, el bit de *stop* es testeado y el dato (el paquete de bits) es transferido a un *buffer* el cual, a su vez, lo transfiere al registro **RCREG** si está vacío. El *buffer* y el registro **RCREG** forman una **FIFO** de dos elementos (el primer dato que entra es el primer dato que sale). En el caso de transmisión de 9 bit, el noveno bit pasa la bit **RX9D** del registro **RCSTA** del mismo modo que los otros 8 pasan al registro **RCEG**.

Algunos dispositivos tienen un *USART* modificada, llamado **AUSART** o *USART direccionable*, que permite filtrar automáticamente ciertas transmisiones. Los datos recibidos son separados en dos categorías, dirección y datos, que se indican por el noveno bit. Sólo los bytes de dirección son procesados por el *USART*, los datos son ignorados. Este echo se utiliza normalmente cuando hay varios dispositivos en un bus y las transmisiones se direccionan a uno en concreto. Los dispositivos que reciben la transmisión ignoran todos los bytes de datos con el noveno bit a 0 y sólo reciben los bytes de dirección con el noveno bit a 1. Cuando se recibe el byte de dirección y coincide, el dispositivo puede pasar a recepción normal y recibir el resto de los datos. En este tipo de dispositivos el bit 2 del **RCSTA** es:

bit 3: **ADDEN**: Bit de habilitación de detección de dirección

1=Habilitada (Sólo si RX9=1)

0=Deshabilitada

Figura 8. Esquema del proceso de recepción en los AUSART

Registro SPBRG (0x99) [PIC16F87x]

La velocidad de comunicación se controla por el valor cargado en este registro. Genera el reloj que permite la comunicación. La velocidad se expresa en baudios (bit/s).

En modo asíncrono:

BRGH=0 (baja velocidad) $$BAUDIOS = \frac{f_{OSC}}{64 \cdot (SPBRG + 1)}$$

BRGH=1(alta velocidad) $$BAUDIOS = \frac{f_{OSC}}{16 \cdot (SPBRG + 1)}$$

En modo síncrono: $$BAUDIOS = \frac{f_{OSC}}{4 \cdot (SPBRG + 1)}$$

Siempre hay que considerar un margen de error.

7.2.2 El módulo USART en C

Configuración genérica del *USART*:

#USE RS232 (opciones)

Esta directiva permite configurar varios parámetros del *USART*: velocidad de transmisión, pins utilizados, etc. Se puede modificar en cualquier parte del programa pero siempre después de haber definido la directiva *#USE DELAY*. Esta directiva habilita el uso de funciones tales como *GETCH*, *PUTCHAR* y *PRINTF*. Permite su uso en dispositivos que no poseen módulo *USART* mediante software *USART*.

Cuando se utilizan dispositivos con *USART*, si no se puede alcanzar una tasa de baudios dentro del 3% del valor deseado utilizando la frecuencia de reloj actual, se generará un error.

BAUD=X	Velocidad en Baudios.
XMIT=pin	Pin de transmisión.
RCV=pin	Pin de recepción.
FORCE_SW	Usa un software *UART* software en lugar del hardware aun cuando se especifican los pines del hardware.
RESTART_WDT	Hace que la función *GETC()* ponga a cero el **WDT** mientras espera un carácter.
BRGH1OK	Permite velocidades de transmisión bajas en *chips* que tienen problemas de transmisión.

ENABLE=pin	El pin especificado estará a nivel alto durante la transmisión. Utilizado en transmisión 485.
DEBUGGER	Permite depuración a través del *ICD*. El pin por defecto es el **B3**; usar **XMIT** y **RCV** para cambiar el pin (debe ser el mismo en ambos).
RESTART_WDT	Provoca que la función *GETC()* borre el **WDT** si espera un carácter.
INVERT	Invierte la polaridad de los pines serie (normalmente no es necesario con el convertidor de nivel, como el **MAX232**). No puede usarse con el *USART* interno.
PARITY=X	Donde X es N, E, u O.
BITS =X	Donde X es 5-9 (no puede usarse 5-7 con el *USART* interno).
FLOAT_HIGH	Se utiliza para las salidas de colector abierto.
ERRORS	Indica al compilador que guarde los errores recibidos en la variable *RS232_ERRORS* para restablecerlos cuando se producen.
SAMPLE_EARLY	No se puede utilizar con *USART* interno. Provoca que el muestreo del dato a través de la función *GETC()* se realice al principio de un bit de tiempo.
RETURN=pin	Para *FLOAT_HIGH* y *MULTI_MASTER*, éste pin se usa para leer la señal de retorno. Por defecto, para *FLOAT_HIGH* es **XMIT** y para *MULTI_MASTER* es **RCV**.
MULTI_MASTER	Usa el pin de *RETURN* para determinar si otro master en el bus está transmitiendo al mismo tiempo. Si se detecta una colisión, el bit 6 se pone a 1 en *RS232_ERRORS* y todos los posibles *PUTC()* son ignorados hasta que el bit 6 esté a 0. La señal es comprobada al final y al principio de cada bit de tiempo. No se puede utilizar con *USART* interno.
LONG_DATA	Permite manejar **INT16** a las funciones *GETC()* y *PUTC()*. En formatos de datos de 9 bits.
DISABLE_INTS	Provoca la deshabilitación de interrupciones cuando se ejecuta *GETC()* y *PUTC()* evitando distorsiones en los datos.

STOP=x	Numero de bits de stop (por defecto 1)
TIMEOUT=x	Para establecer el tiempo que *GETC()* espera un carácter (en ms). Si no se recibe carácter en este tiempo, el *RS232_ERRORS* se pone a 0.
SYNC_SLAVE	Provoca una linea *RS232* en modo esclavo síncrono, haciendo la patilla de recepción como entrada de reloj y la de transmisión como entrada/salida de datos.
SYNC_MASTER	Provoca una linea *RS232* en modo maestro síncrono, haciendo la patilla de recepción como salida de reloj y la de transmisión como entrada/salida de datos.
SYNC_MATER_CONT	Provoca una línea *RS232* en modo maestro síncrono en modo continuo, haciendo la patilla de recepción como salida de reloj y la de transmisión como entrada/salida de datos.
UART1	Configurar el **XMIT** y **RCV** para el *USART1*.
UART2	Configurar el **XMIT** y **RCV** para el *USART2*.

Ejemplos:

```
#use delay (clock=2000000);
#use rs232 (BAUD=9600 , XMIT=PIN_C6 , RCV=PIN_C7 , BITS=8)
#use rs232 (BAUD=9600 , XMIT=PIN_A2 , RCV=PIN_A3)
```

setup_uart(baud)

baud es una constante que define la velocidad. Un 1 enciende el *USART* y un 0 lo apaga. Con cualquier valor de velocidad, el USART se enciende. En dispositivos que utilizan *AUSART* se admiten los siguientes parámetros:

UART_ADDRESS UART: sólo acepta datos con el noveno bit a 1.

UART_DATA UART: acepta todos los datos.

set_uart_speed (baud)

Idéntica a la función anterior.

```
// Se establece la velocidad mediante la combinación de las patillas B0 y B1
switch( input_b() & 3 ) {
   case 0 : set_uart_speed(2400);   break;
   case 1 : set_uart_speed(4800);   break;
   case 2 : set_uart_speed(9600);   break;
   case 3 : set_uart_speed(19200);  break;
}
```

Transmisión de datos:

putc (cdata)

putchar (cdata)

cdata es un carácter de 8 bits. Esta función envía un carácter mediante la patilla **XMIT**. La directiva *#USE RS232* debe situarse siempre antes de utilizar ésta función.

puts (string)

string: cadena de caracteres constante o matriz de caracteres terminada con un 0. La función *puts()* manda los caracteres de la cadena, uno a uno, a través del bus *RS-232* utilizando la función *PUTC();* detrás de la cadena envía un *RETURN* (13) y un retorno de carro (10).

printf (fname, cstring, values...)

cstring: es una cadena de caracteres (constante) o matriz de caracteres terminada con un 0.

fname: las funciones a utilizar para escribir la cadena indicada; por defecto se utiliza la función *PUTC(),* que permite escribir en el bus *RS-232*.

values: valores a incluir en la cadena separados por comas; se debe indicar **%nt**. El formato es **%nt**, donde **n** es opcional y puede ser:

1-9 para especificar cuantos caracteres deben ser especificados;

01-09 para indicar la cantidad de ceros a la izquierda;

1.1 - 9.9 para coma flotante.

t puede indicar:

c	Carácter.
s	Cadena o carácter.
u	Entero sin signo.
d	Entero con signo.
Lu	Entero largo sin signo.
Ld	Entero largo con signo.
x	Entero hexadecimal (minúsculas).
X	Entero hexadecimal (mayúsculas).

Lx	Entero largo hexadecimal (minúsculas).
LX	Entero largo hexadecimal (mayúsculas).
f	Flotante con truncado.
g	Flotante con redondeo.
e	Flotante en formato exponencial.
w	Entero sin signo con decimales insertados. La 1ª cifra indica el total, la 2ª el número de decimales.

Recepción de datos:

value=getc()

value=getch()

value=getchar()

value es un carácter de 8 bits. Espera recibir un carácter por la línea *RS-232* y devuelve su valor. En los dispositivos con *USART* interno, se pueden almacenar hasta tres caracteres; para evitar esperas se puede usarla función *KBHIT()*.

valor = kbhit()

valor es 0 (*FALSE*) si *GETC()* debe esperar a que llegue un carácter; 1 (*TRUE*) si ya hay un carácter listo para ser leído por la función *GETC()*.

Ejemplo 1: *Enviar los datos del 0 al 10, en modo asíncrono, entre dos PIC. Visualizar con un LCD los datos enviados y los datos recibidos; la recepción del dato deberá ser por interrupción del USART.* (ver figura 10). *Componentes ISIS: PIC16F876 y LM016L. Instrumentos: VIRTUAL TERMINAL.*

```
#include <16F876.h>
#FUSES XT,NOWDT
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7)
#include <LCD.C>

void main() {
   int valor;
   lcd_init();

while(1){
   for (valor=0;valor<=10;valor++) {
     PUTC(valor);
```

```
    printf(lcd_putc,"\fenviando=%1D",VALOR);
    delay_ms(500);
      }
    }
}
```

Figura 9. Programa PIC_1 del ejemplo 1

Figura 10. El ejemplo 1

```
#include <16F876.h>
#FUSES XT,NOWDT
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7, bits=8)
#include <LCD.C>
#BYTE TRISA=0X85
```

```
#BYTE PORTA=0X05
int valor;
#int_RDA
RDA_isr()
{
 valor=GETC();
}

void main() {
  bit_clear(TRISA,0);
  lcd_init();
   enable_interrupts(INT_RDA);
   enable_interrupts(GLOBAL);
for (;;) {
     lcd_gotoxy(1,1);
     printf(lcd_putc,"recibiendo=%lD",valor);
   }
}
```

Figura 11. Programa PIC_2 del ejemplo 1

Con el programa *VIRTUAL TERMINAL* (botón derecho: *Hex display mode*) se pueden visualizar los datos del bus serie (figura 12).

Figura 12. Virtual Terminal

7.2.3 La norma RS232

La norma *RS232* es la más habitual en la comunicación serie. Básicamente comunica un equipo terminal de datos (*DTE* o *Data Terminal Equipment*) y el equipo de comunicación de datos (*DCE* o *Data Communications Equipment*).

Las características eléctricas de la señal en esta norma establecen que la longitud máxima entre el *DTE* y el *DCE* no debe ser superior a los 15 metros y la velocidad máxima de transmisión es de 20.000 bps. Los niveles lógicos no son compatibles

TTL, deben situarse dentro de los siguientes rangos: 1 lógico entre -3V y -15V y 0 lógico entre +3V y +15V. Se utilizan conectores de 25 patillas (DB 25) o de 9 patillas (DB 9) siendo asignado el conector macho al *DTE* y el conector hembra al *DCE*.

Para una comunicación *full duplex* desde el *USART* del PIC, se debe conectar un mínimo número de señales, **TXD** y **RXD** así como la masa (**GND**). Los PIC utilizan señal *TTL* en el módulo *USART* por lo que se debe utilizar un conversor de nivel a *RS232*, como el *MAX232*.

Figura 13. Conexión básica full duplex entre PIC y PC

En la mayoría de los PC actuales, sobre todo el los portátiles, están desapareciendo los puertos serie. Como solución se pueden utilizar cables de conversión *SERIE-USB* que utilizan el *Universal Serial Port (USB)*, no se debe confundir con la utilización del módulo *USB* integrado en el PIC con gestión de comunicación *USB* (ver la figura 14).

Figura 14. Diferencia entre un convertidor serie-usb y un módulo USB integrado

Estos cables (ver la figura 15) se basan en integrados como el *FT232BM* de FTDI chip (figura 16) (*http://www.ftdichip.com/Products/FT232BM.htm*). En la propia *Web* del fabricante se pueden encontrar los *drivers* para la configuración de *Windows* (figura 17) y los diseños de un sistema de conversión *SERIE-USB*.

Figura 15. Cable SERIE-USB

Figura 16. FT232BM

Figura 17. Universal Serial Port

El *ISIS* del *PROTEUS* proporciona un potente componente que permite la simulación a través del puerto serie: *COMPIM* (figura 18).

Figura 18. El componente COMPIM

Utilizando este componente no es necesario añadir al esquema del PIC un *MAX232*, ya que el propio componente gestiona la comunicación con el puerto del PC. Utilizando este componente podemos comunicarnos con el propio PC (si tiene 2 puertos serie o 1 puerto serie y un puerto *USB* –utilizando el convertidor– como podemos ver en la figura 19) y manejar los datos que proceden del PIC con cualquier programa de aplicación (*Visual BASIC, Visual C*, etc.).

Figura 19. Las conexiones serie-serie o serie-usb

La configuración del puerto se realiza como en cualquier componente y se pueden cambiar prácticamente todas las propiedades de un puerto serie: número de puerto, velocidad, paridad, número de bits, etc. (ver la figura 20).

Figura 20. Configuración COMPIM

Ejemplo 2: *Enviar los datos de una conversión AD al puerto serie de un PC* (figura 21). *Componentes ISIS: PIC16F876, COMPIM, POTLIN y LM016L. Instrumentos: DC VOLTMETER y VIRTUAL TERMINAL.*

Para probar este ejemplo se pueden utilizar dos PC o un PC con dos puertos serie o un PC con un puerto serie y un puerto *USB* (utilizando un cable *SERIE-USB*).

Para observar los datos que envía el PIC se utiliza un *COMPIM* y también se puede utilizar un *COMPIM* para leer los datos que entran por el PC o, en este caso, utilizar el *HyperTerminal* de *Windows* o cualquier otro programa emulador del puerto serie.

Por el *Virtual Terminal* se pueden comprobar los datos de salida y entrada. En este caso se han conectado los dos puertos serie del PC (*COM1* y *COM2* con un cable).

Se puede utilizar el *HyperTerminal* de *Windows* (*Accesorios/Comunicaciones/HyperTerminal*), configurándolo según las siguientes figuras: figura 23, figura 24, figura 25, figura 26 y la figura 27. En este caso se han conectado el puerto *COM1* del PC y un *USB* configurado en *COM6*.

Figura 21. El ejemplo 2

Figura 22. Virtual Terminal de salida (COM1) y entrada (COM2): *las pantallas son idénticas*

Figura 23. El nombre de la conexión

Figura 24. La selección del puerto (COM1 en el PROTEUS y COM6 en el HyperTerminal)

Figura 25. La configuración del puerto

Figura 26. El comando Archivo/Propiedades para modificar la visualización

Figura 27. Activar "Agregar avance de línea..."

Figura 28. La simulación entre PROTEUS y el PC utilizando el HyperTerminal

```
#include <16F877.h>
#device adc=10
#FUSES XT,NOWDT
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7, bits=8, parity=N)
#include <LCD.C>

void main() {
   int16 q;
   float p;
   setup_adc_ports(AN0);
   setup_adc(ADC_CLOCK_INTERNAL);
   lcd_init();
   for (;;) {
      set_adc_channel(0);
      delay_us(10);
      q = read_adc();
      p = 5.0 * q / 1024.0;
      printf(lcd_putc, "\fADC = %4ld", q);
      printf(lcd_putc, "\nVoltage = %01.2fV", p);
      printf("ADC = %4ld ", q);
      printf("Voltage = %01.2fV\r", p);  // \r permite cambiar de linea.
      delay_ms(100);
      }
}
```

Figura 29. El programa del ejemplo 2

Ejemplo 3: *Enviar un dato desde el PC al PIC por el puerto serie. Cuando lo reciba debe visualizarlo en un LCD y enviar la palabra "recibido" al PC. Emplear interrupciones* (figura 31). *Componentes ISIS: PIC16F876, COMPIM y LM016L.*

```
#include <16F876.h>
#FUSES XT,NOWDT
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7, bits=8, parity=N)
#include <LCD.C>

char ch;

#int_rda
void serial_isr() {
    ch=getchar();
    puts("Recibido");
}
```

```
void main()
{
lcd_init();
   enable_interrupts(global);
   enable_interrupts(int_rda);
while(1)  {
   printf(lcd_putc, "\n\r Valor %c",ch);
    }
}
```

Figura 30. El programa del ejemplo 3

Figura 31. El ejemplo 3

Figura 32. Configuración del HyperTerminal para Enviar/Recibir

Figura 33. Comunicación full duplex con el PC

7.3 Puerto serie síncrono (SSP)

Los dos modos de trabajo son:

- Interfaz serie de periféricos (*SPI*): Desarrollada por Motorola para la comunicación entre microcontroladores de la misma o diferente familia en modo maestro-esclavo. *Full-duplex*.
- Interfaz Inter-Circuitos (I^2C): Interfaz desarrollado por Philips, con gran capacidad para comunicar microcontroladores y periféricos. *Half-duplex*.

7.3.1 Interfaz Inter-Circuitos (I^2C)

El bus I^2C se basa en la comunicación a través de 2 hilos. Cada dispositivo conectado al bus tiene una dirección. Puede configurarse como comunicación de un maestro y varios esclavos o una configuración *Multimaestro*. En ambas configuraciones, el dispositivo maestro es el que tiene la iniciativa en la transferencia, decide con quién se realiza, el sentido de la misma (envío o recepción desde el punto de vista del maestro) y cuándo finaliza. Cuando el maestro inicia una comunicación, primero transmite la dirección del dispositivo con el cual se quiere comunicar y los esclavos comprueban si la dirección concuerda con la suya. La transmisión puede

ser de lectura o escritura, el último bit de la dirección lo indica; así el maestro estará en transmisión y el esclavo en recepción o viceversa. En cualquier caso la señal de reloj la genera el maestro.

Los dos hilos del bus I^2C son dos líneas de colector abierto: la señal de reloj *SCL* o pin **RC3** y la línea de datos *SDA* o pin **RC4**. Se deben utilizar unas resistencias externas o de *pull-up* para asegurar un nivel alto cuando no hay dispositivos conectados al bus.

El número de dispositivos conectados y la longitud de conexión están limitados por la capacidad de direccionamiento (de 7 a 10 bits) y por la máxima carga del bus (400 pF). La velocidad máxima estándar es de hasta 100 Kbps, la rápida hasta 400 Kbps y la Alta hasta los 3.4 Mbps

Figura 34. Cálculo del valor de R_L en función de la capacidad y velocidad del bus (cortesía de ST)

La transmisión se inicia con un bit de inicio o *START* y termina con el bit de finalización o *STOP*. *START* se establece con una transición de alto a bajo en la línea *SDA* (normalmente a nivel alto) cuando la línea *SCL* está a nivel alto. *STOP* se establece cuando se produce una transición de bajo a alto en la línea *SDA* cuando *SCL* está a nivel alto; de esta forma los datos en la línea *SDA* sólo cambian en el estado bajo de la línea *SCL* (figura 35).

Figura 35. Condiciones de la transmisión

Al iniciar la transmisión, el master envía la dirección del esclavo con el que desea establecer la comunicación. La dirección puede ser de 7 o 10 bits con formato de byte (uno o dos bytes respectivamente). Tras la dirección se adjunta un bit de lectura/escritura (figura 36).

Figura 36. Los formatos de dirección

Una vez el master envía la dirección (o datos), el esclavo genera un bit de reconocimiento (*ACK*), si el master no recibe este bit la comunicación se interrumpe, generando la señal de *STOP*. El maestro también puede recibir datos, en este caso es él quién genera la señal de reconocimiento para cada byte recibido menos para el último, en este caso el esclavo libera la línea *SDA* y el master genera un *STOP*.

Existe la posibilidad de que el master, tras una transmisión/recepción, no abandone el bus y siga en comunicación con el esclavo; en esta ocasión genera una nueva condición de *START*, llamada *START REPETIDA* (Sr), idéntica a la anterior pero después de un pulso de reconocimiento.

En los PIC de la gama media existen dos módulos que permiten realizar una comunicación *I²C*, el *BSSP* (*Basic Synchronous Serial Port*) y el *MMSP* (*Master Synchronous Serial Port*), y se diferencian en modo de trabajo maestro. El módulo *MSSP* permite detectar condiciones de *START* y *STOP* por interrupción. Este módulo puede trabajar en tres modos:

- Maestro.
- Esclavo con dirección de 7 bits.
- Esclavo con dirección de 10 bits.

Los registros asociados a este módulo son seis: **SSPCON**, **SSPCON2**, **SSPADD**, **SSPBUF**, **SSPSTAT** y el **SSPSR**.

Registro SSPSTAT (dirección RAM: 94h) [PIC16F87x]

R/W-0	R/W-0	R-0	R-0	R-0	R-0	R-0	R-0
SMP	CKE	D/$\overline{A}$	P	S	R/$\overline{W}$	UA	BF
Bit7							Bit0

Figura 37. El registro SSPSTAT

bit 7: **SMP**: Bit de muestreo.

SPI en modo maestro:

1 = El dato se muestrea al final de ciclo.

0 = El dato se muestrea en el medio del ciclo.

SPI en modo esclavo:

SMP debe ponerse a '0' cuando se trabaje en modo esclavo.

I²C en modo master o esclavo:

1 = Deshabilitación del control *Slew rate* para una velocidad estándar (100 kHz i 1 MHz)

0 = Habilitación del control *Slew rate* para alta velocidad (400 kHz)

bit 6: **CKE**: Selección de flanco de reloj en modo SPI.

CPK = 0

1 = El dato se transmite en el flanco de subida de **CKS**.

0 = El dato se transmite en el flanco de bajada de **CKS**.

CKP = 1

1 = El dato se transmite en el flanco de bajada de **CKS**.

0 = El dato se transmite en el flanco de subida de **CKS**.

I²C en modo master o esclavo:

1 = Niveles de entrada conforme especificaciones **SMBUS**.

0 = Niveles de entrada conforme especificaciones *I²C*.

bit 5: **D/A** : Bit de datos/ dirección (sólo en el modo *I²C*).

1 = Indica que el último byte recibido o transmitido era un dato.

0 = Indica que el último byte recibido o transmitido era una dirección.

bit 4: **P**: Bit de *Stop* (sólo en el modo *I²C*).

1 = Indica que ha sido detectada una condición de *Stop*.

0 = No se ha detectado la condición de *Stop*.

bit 3: **S**: Bit de *Start* (sólo en el modo *I²C*).

1 = Indica que ha sido detectada una condición de *Start*.

0 = No se ha detectado la condición de *Start*.

bit 2: **R/W**: Bit de Lectura/ Escritura (sólo en el modo *I²C*). Este bit retiene la información de lectura o escritura después de la última detección de dirección correcta. Sólo es válido desde la confirmación de dirección hasta el siguiente bit de *Start*, *Stop* o no **ACK**.

En *I²C* modo esclavo:

1 = Lectura.

0 = Escritura.

En *I²C* modo master:

1 = Transmisión en progreso.

0 = Transmisión en no progreso.

bit 1: **UA**: Actualización de dirección (sólo en el modo *I²C* de 10 bits).

1 = Se necesita una actualización de dirección en el registro **SSPADD**.

0 = La dirección no necesita una actualización.

bit 0: **BF**: Bit de *buffer* lleno.

Recepción (modos *SPI* e *I²C*):

1 = Recepción completada, **SSPBUF** está lleno.

0 = La recepción no ha finalizado, **SSPBUF** está vacío.

Transmisión:

1 = Transmisión en proceso, **SSPBUF** lleno.

0 = Transmisión completa, **SSPBUF** vacío.

Registro SSPCON (dirección RAM: 14h) [PIC16F87x]

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WCOL	SSPOV	SSPEN	CKP	SSPM3	SSPM3	SSPM1	SPM0
Bit7							Bit0

Figura 38. El registro SSPCON

bit 7: **WCOL**: Bit de detección de colisión.

Modo master:

1 = Se ha producido una escritura en **SSPBUF** sin que las condiciones del *I²C* sean válidas.

0 = No hay colisión.

Modo Esclavo:

1 = El registro **SSPBUF** ha sido escrito mientras se realizaba una transmisión previa.

0 = No hay colisión.

bit 6: **SSPOV**: Bit de *overflow* (debe ser borrado por software).

En modo *SPI*:

1 = Un nuevo dato se ha recibido cuando aún no se ha leído el dato anterior almacenado en **SSPBUF**. El dato del registro **SSPSR** se pierde y se mantiene el anterior. Sólo se produce *overflow* en modo esclavo.

0 = No hay *overflow*.

En modo *I²C*:

1 = Un nuevo byte es recibido cuando aún no se ha leído el registro **SSPBUF** donde se encuentra el byte anteriormente recibido.

0 = No hay overflow.

bit 5: **SSPEN**: Bit de habilitación del puerto serie síncrono.

En ambos modos, los pines han de ser correctamente configurados como entradas o salidas.

En modo *SPI*:

1 = Habilitación del puerto serie y configuración de los pines **SCK**, **SDO**, **SDI** y **SS** como fuente del puerto.

0 = Puerto serie deshabilitado y pines configurados como E/S.

En modo *I²C*:

1 = Habilitación del puerto serie y configuración de los pines **SDA** y **SCL** como fuente del puerto.

0 = Puerto serie deshabilitado y pines configurados como E/S.

bit 4: **CKP**: Bit de selección de la polaridad del reloj.

En modo *SPI*:

1 = El estado de reposo para el reloj es el nivel alto.

0 = El estado de reposo para el reloj es el nivel bajo.

En modo *I²C* esclavo: (control de liberación de **SCK**).

1 = Habilitación del reloj.

0 = Mantiene el reloj en estado bajo.

bits 3:0 **SSPM3:SSPM0**: Selección del modo del módulo *SSP*.

0000 = *SPI*, modo maestro, reloj = $F_{osc}/4$.

0001 = *SPI*, modo maestro, reloj = $F_{osc}/16$.

0010 = *SPI*, modo maestro, reloj = $F_{osc}/64$.

0011 = *SPI*, modo maestro, reloj = Salida del TMR2/2.

0100 = *SPI*, modo esclavo, reloj = pin **SCK**, pin **SS** habilitado.

0101 = *SPI*, modo esclavo, reloj = pin **SCK**, pin **SS** deshabilitado. Puede usarse como pin de E/S.

0110 = *I²C*, modo esclavo, dirección de 7 bits.

0111 = *I²C*, modo esclavo, dirección de 10 bits.

1000 = *I²C* modo master, reloj=$F_{osc}/4$*(SSPADD+1)).

1011 = *I²C* en modo maestro controlado por *firmware* (esclavo inactivo).

1110 = *I²C* en modo maestro controlado por *firmware* (dirección 7 bit con interrupción de bit *START* y *STOP*).

1111 = *I²C* en modo maestro controlado por *firmware* (dirección 10 bit con interrupción de bit *START* y *STOP*).

1001, 1010, 1100, 1101 = Reservado.

Registro SSPCON2 (dirección RAM: 91h) [PIC16F87x].

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
GCEN	ACKSTAT	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
Bit7							Bit0

Figura 39. El registro SSPCON2

bit 7: **GCEN**: Bit de habilitación llamada general (sólo en modo *I²C* esclavo).

1 = Habilita la interrupción cuando se recibe una llamada general (dirección 0000h) en el **SSPSR**.

0 = Deshabilitado.

bit 6: **ACKSTAT**: Bit de estado de reconocimiento (sólo en modo *I²C* master).

En modo master transmisión:

1 = No recibido **ACK** del esclavo.

0 = **ACK** recibido del esclavo.

bit 5: **ACKDT**: Bit de dato de reconocimiento (sólo en modo *I²C* master).

En modo master recepción: Valor que será transmitido cuando el usuario inicie una secuencia de reconocimiento al final de una recepción:

1 = No **ACK**.

0 = **ACK**.

bit 4: **ACKEN**: Habilitación secuencia de **ACK** (sólo en modo *I²C* master).

En modo master recepción:

1 = Inicia secuencia de reconocimiento de **SDA** y **SCL**, y transmite un *ACKDT*. Borrado por hardware.

0 = Desactivado.

bit 3: **RCEN**: Bit de habilitación de recepción (sólo en modo *I²C* master).

1 = Recepción habilitada.

0 = Deshabilitada.

bit 2: **PEN**: Habilitación de la secuencia de *STOP* (sólo en modo *I²C* master).

1 = Inicia condición de *Stop* en **SDA** y **SCL**. Borrado por hardware.

0 = Deshabilitada.

bit 1: **RSEN**: Habilitación del *START* repetido (sólo en modo *I²C* master).

1 = Inicia la condición de *SR* en **SDA** y **SCL**. Borrado por hardware.

0 = Deshabilitada.

bit 0: **SEN**: Habilitación del *START* (sólo en modo *I²C* master).

1 = Inicia la condición de *START* en **SDA** y **SCL**. Borrado por hardware.

0 = Deshabilitada.

El resto de registros son:

- El registro **SSPBUF** (dirección Ram: 13h) es un *buffer* de transmisión/recepción serie: es el registro desde el cual se leen o escriben los datos a transmitir.
- El registro **SSPSR** es un registro de desplazamiento *SSP* (no accesible directamente). Desplaza el dato para transmitirlo o recibirlo. En una transmisión, el dato se escribe desde el registro **SSPBUF**, mientras que en una recepción, se carga el dato de **SSPSR** a **SSPBUF**.
- El registro **SSPADD** (dirección Ram: 93h) define la dirección del esclavo o los baudios de la comunicación del master. En este registro se almacena la dirección del esclavo; en el modo de 10 bits primero se debe cargar el byte alto (1111 0 A9 A8 0) y después el byte bajo (A7:A0).

Otros registros relacionados con el módulo *MSSP* son el **TRISC** (dirección Ram: 87h) para definir **RC3** y **RC4** como entradas. El **PIR1/PIE1** (direcciones Ram:

0Ch/8Ch) para la gestión de interrupciones (**SSPIF/SSPIE**). El **PIR2/PIE2** (direcciones Ram: 0Dh/8Dh) para la gestión de la interrupción por colisión del bus (**BCLIF/BCLIE**) y el **INTCON** (direcciones Ram: 0Bh/8Bh/10Bh/18Bh) para la habilitación de las interrupciones de periféricos.

7.3.1.1 I²C en C

Configuración genérica del *I²C*:

#use I2C (opciones)

Opciones: separadas por comas, pueden ser las siguientes:

MULTI_MASTER	Establece modo Multimaestro.
MASTER	Establece modo maestro.
SLAVE	Establece modo esclavo.
SCL=pin	Especifica el pin **SCL**.
SDA=pin	Especifica el pin **SDA**.
ADDRESS=nn	Especifica la dirección en modo esclavo.
FAST	Utiliza velocidad alta.
SLOW	Utiliza velocidad baja.
RESTART_WDT	Borra el **WDT** mientras espera una lectura.
FORCE_HW	Utiliza las funciones *I²C* hardware.
NOFLOAT_HIGH	No permite señales flotantes.
SMBUS	Utiliza el bus en formato *SMBUS*.
STREAM=id	Asocia un identificar *stream*.

Esta directiva (*#USE I2C*) tiene efecto sobre las funciones *I2C_START*, *I2C_STOP*, *I2C_READ*, *I2C_WRITE* e *I2C_POLL*. Se utilizan funciones software a menos que se especifique *FORCE_HW*. El modo esclavo sólo puede ser usado con el módulo físico *SSP*.

```
#use I2C(master, sda=PIN_B0, scl=PIN_B1)
#use I2C(slave, sda=PIN_C4, scl=PIN_C3, address=0xa0, FORCE_HW)
#use I2C(master, scl=PIN_B0, sda=PIN_B1, fast=450000)
```

Las funciones asociadas son

- **I2C_WRITE()**.
- **I2C_START()**.

- **I2C_READ().**
- **I2C_STOP().**
- **I2C_POLL().**
- **I2C_ISR_STATE().**
- **I2C_SlaveAddr().**

I2C_START();

En modo master, esta función inicializa la transmisión. Después de la condición de *Start*, el reloj es puesto a nivel bajo hasta que se escribe con la función *I2C_WRITE()*. Si se llama a otra función *I2C_START* antes de un *I2C_STOP* se está utilizando un *START* repetido (Sr). Esta función dependerá de la respuesta del esclavo.

```
i2c_start();            //Inicialización de la transmisión
i2c_write(0xa0);        //Dirección del esclavo
i2c_write(address);     //Datos a esclavo
i2c_start();            //Restart
i2c_write(0xa1);        //Cambio a lectura
data=i2c_read(0);       //Datos del esclavo al master.
i2c_stop();             //Finalización de la transmisión
```

I2C_STOP ();

Finaliza la transmisión.

I2C_WRITE(dato);

Dato es un entero de 8 bits que envía por el bus. En modo master, esta función genera la señal de reloj que marca la velocidad de transmisión del dato; en modo esclavo espera la señal de reloj que genere el maestro.

Devuelve el bit de reconocimiento **ACK** que envía el receptor cuando la transmisión ha terminado: 0 indica **ACK**, 1 indica un **NO** ACK y un 2 indica una colisión en modo multimaster.

El bit de menor pesor (*lsb*) del primer dato transmitido tras un *START* indica el sentido de la comunicación (si el bit es "0", la información se transmitirá de maestro a esclavo).

dato = I2C_READ([ACK]);

Dato es un entero de 8 bits leído del bus. En modo master, esta función genera la señal de reloj; en modo esclavo espera la señal de reloj. No hay *timeout* por lo que se utiliza junto con *I2C_POLL* para prevenir bloqueos.

Opcionalmente se puede incluir un **ACK** donde 1 indica un **ACK** y un 0 indica un **NO ACK**. Se puede borrar el *Watchdog* mientras se espera a leer el dato, para ello se debe incluir la opción *RESTART_WDT* en la directiva *#use i2c()*.

valor = I2C_POLL();

Se utiliza sólo si el PIC tiene módulo *SSP*. Devuelve un *TRUE* (1) si se ha recibido el dato en el *buffer* y un *FALSE* (0) si no se ha recibido. Cuando devuelve un *TRUE*, la función *I2C_READ()* guarda el dato leido.

I2C_SlaveAddr(int8 adr);

Se especifica la dirección del dispositivo en modo esclavo.

estado = I2C_ISR_STATE();

Se utiliza sólo si el PIC tiene módulo *SSP*. Devuelve el estado del bus en modo esclavo después de una interrupción.

Estado es un entero de 8 bits:

0 = Indica dirección coincidente con un R/W a cero.

1-0x7F = El master ha escrito un dato, se debe utilizar *I2C_READ()*.

0x80 = Indica dirección coincidente con un R/W a uno, responder con *I2C_WRITE()*.

0x81-0xFF = Transmisión terminada y reconocida, se responde con *I2C_WRITE()*.

Ejemplo 4: *Guardar y leer datos en las 10 primeras posiciones de memoria de una EEPROM I²C. Representar los valores escritos y leidos en un display LCD* (figura 40). *Componentes ISIS: PIC16F877, M24512, RES y LM016L. Instrumentos: I2C DEBUGGER.*

Figura 40. El ejemplo 4

La *EEPROM I²C* 24512 tiene un byte de control (figura 41) donde la parte alta tiene un valor fijo (Ah) y la parte baja consta de la dirección impuesta en sus patillas (**E2:E0**) y el bit de lectura/escritura, de tal forma que si se fijan las entradas de dirección a masa, el byte de control puede tener los valores **A0h** para escritura y **A1h** para lectura.

	Device Type Identifier				Chip Enable Address			$\overline{RW}$
	b7	b6	b5	b4	b3	b2	b1	b0
Device Select Code	1	0	1	0	E2	E1	E0	$\overline{RW}$

Figura 41. Byte de control del 24512 (cortesía de ST)

El formato de escritura es el mostrado en la figura 42, donde tras un *START* se escribe la palabra de control para seleccionar el dispositivo y el modo de trabajo, dos bytes para la dirección de escritura en el dispositivo y el dato a escribir.

Figura 42. El formato de escritura (cortesía de ST)

A continuación se describe una función para la escritura en la *EEPROM*, que debe ser llamada desde el programa principal donde se le pasa la dirección a escribir y el dato.

```
void write_ext_eeprom(long int address, BYTE data)
{
   short int status;
   i2c_start();                  //Inicializa la transmisión
   i2c_write(0xA0);              //Escribe la palabra de control (dirección
                                 //0h + 0 para escritura)
   i2c_write(address>>8);        //Parte alta de la dirección a escribir en la
                                 //EEPROM
   i2c_write(address);           //Parte baja de la dirección a escribir en la
                                 //EEPROM
   i2c_write(data);                 //Dato a escribir
   i2c_stop();                      //Finalización de la transmisión.
   i2c_start();                     //Reinicio
   status=i2c_write(0xa0);       //Lectura del bit ACK, para evitar escrituras
                                 //incorrectas
```

```
    while(status==1)          //Si es 1 esperar a que responda el esclavo
    {
        i2c_start();
        status=i2c_write(0xa0);
    }
}
```

Figura 43. Función de escritura en la EEPROM

El formato de lectura puede ser de cuatro formas: lectura de la dirección actual en el bus, lectura de una dirección cualquiera, lectura secuencial a partir de la dirección actual y lectura secuencial a partir de una dirección cualquiera. La forma más normal es la de leer una dirección cualquiera (figura 44), donde el proceso es muy similar al de escritura y tras una reinicialización hay dos ciclos donde se indica el modo de lectura y se envía el dato. En este caso, el master debe devolver un **NO ACK**.

Figura 44. Lectura de una dirección cualquiera (cortesía de ST)

A continuación se describe una función para la lectura de la *EEPROM*, que debe ser llamada desde el programa principal donde se le pasa la dirección a leer.

```
BYTE read_ext_eeprom(long int address) {
    BYTE data;
    i2c_start();                //Inicializa la transmisión
    i2c_write(0xA0);            //Escribe la palabra de control (dirección 0h
                                //+ 0 para escritura)
    i2c_write(address>>8);      //Parte alta de la dirección a escribir en la
                                //EEPROM
    i2c_write(address);         //Parte baja de la dirección a escribir en la
                                //EEPROM
    i2c_start();                //Reinicio
    i2c_write(0xa1);            //Escribe la palabra de control (dirección 0h
                                //+ 1 para lectura)
    data=i2c_read(0);           //lectura del dato
    i2c_stop();                 //Finalización de la transmisión.
    return(data);
}
```

Figura 45. Función de lectura de la EEPROM

Para la aplicación del ejemplo se utilizan estas funciones en el programa principal.

```
#include <16F877.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#use delay(clock=4000000)
#use i2c(Master,sda=PIN_C4,scl=PIN_C3)          //Configuracion I2C
#include <lcd.c>

#define EEPROM_ADDRESS long int

void write_ext_eeprom(long int address, BYTE data)
{//incluir la función explicada anteriormente} ←

BYTE read_ext_eeprom(long int address)
{//incluir la función explicada anteriormente} ←

void main() {

   int8 valor=0, dato;
   EEPROM_ADDRESS address;;
   lcd_init();

    for (address=0;address<=9;address++) {
      WRITE_EXT_EEPROM(address, valor);
      lcd_gotoxy(1,1);
      printf(lcd_putc,"enviando=%1D",valor);
      delay_ms(500);
      valor++ ;
   }
   for (address=0;address<=9;address++) {
      dato=READ_EXT_EEPROM( address);
      lcd_gotoxy(1,2);
      printf(lcd_putc,"recibiendo=%1D",dato);
      delay_ms(500);
   }
 }
```

Figura 46. El programa del ejemplo 4

En el *I2C debugger* se pueden seguir las transiciones del bus. En la figura 47 se muestra una operación de lectura tal como se ha explicado anteriormente. La **S** significa *START*, la **A** es reconocimiento (**ACK**), la **Sr** es *START repetido*, la **N** es *NO ACK* y la **P** indica **STOP** (compararla con la figura 44, en este caso la dirección es 01h y el dato es 01h).

Figura 47. La operación de lectura con el I2C debugger

Ejemplo 5: *Leer la temperatura y hora de lectura y guardar los datos en una EEPROM; utilizar un sensor de temperatura I2C (DS1621), un reloj en tiempo real I2C (DS1307) y una EEPROM serie I2C (M24512). La lectura se realizará en función de una orden dada por el puerto serie; con otra orden se visualizarán los primeros datos de la EEPROM en un monitor del puerto serie* (figura 48). *Componentes ISIS: PIC16F877, M24512, RES, COMPIM, DS1621, DS1307 y LM016L. Instrumentos: I2C DEBUGGER.*

Figura 48. El ejemplo 5

Para facilitar el desarrollo del programa se van ha crear 3 ficheros *driver* para cada uno de los periféricos. Para la *EEPROM* serie I2C M24512 se utilizarán los algorit-

mos descritos en el ejemplo anterior (figura 49). La dirección asignada en el esquema es la 0x00.

```
void write_ext_eeprom(long int address, BYTE data)
{
   short int status;
   i2c_start();
   i2c_write(0xa0);
   i2c_write(address>>8);
   i2c_write(address);
   i2c_write(data);
   i2c_stop();
   i2c_start();
   status=i2c_write(0xa0);
   while(status==1)
   {
      i2c_start();
      status=i2c_write(0xa0);
   }
}
BYTE read_ext_eeprom(long int address) {
   BYTE data;
   i2c_start();
   i2c_write(0xa0);
   i2c_write(address>>8);
   i2c_write(address);
   i2c_start();
   i2c_write(0xa1);
   data=i2c_read(0);
   i2c_stop();
   return(data);
}
```

Figura 49. Fichero EEPROM_24512.C

Para el reloj en tiempo real DS1307 se necesitan conocer sus características y escribir el *driver*. El DS1307 suministra segundos, minutos, horas, día, mes y año en tiempo real (mediante una batería y un cristal de cuarzo exterior permite un funcionamiento independiente del sistema). Posee una serie de registros donde aparecen los datos necesarios (figura 50), los cuales se suministran en código BCD con el formato indicado en la parte derecha de la figura. En este ejemplo sólo se leerán los segundos, minutos y horas.

Para la escritura y lectura del integrado, el fabricante recomienda los ciclos indicados en la figura 51. Podemos observar que el ciclo de escritura se inicia con la

palabra 0xD0 y la de lectura con 0xD1. En el ciclo de escritura, el segundo byte es un puntero que debe indicar la dirección de inicio (en este caso 0). En el ciclo de lectura se realizara la lectura de los tres primeras direcciones de la memoria del DS1307 (los segundos, minutos y horas), el último byte debe indicar un *NACK* al master. Tan sólo queda convertir los bytes en BCD a binario. El fichero se muestra en la figura 52.

Figura 50. Registros con direcciones y formatos (cortesía de Dallas Smc.)

Figura 51. Ciclo de escritura y lectura (cortesía de Dallas Smc.)

```
int BCDaBIN(int bcd){    //Conversión de BCD a Binario
  int varia;
  varia = bcd;
  varia >>= 1;
  varia &= 0x78;
  return(varia + (varia >> 2) + (bcd & 0x0f));
}
```

```
void  tiempo(byte &hor, byte &min, byte &sec){
  i2c_start();          //Escritura
  i2c_write(0xD0);      //Código de escritura
  i2c_write(0x00);      //Puntero a la primera dirección
  i2c_start();          //Lectura
  i2c_write(0xD1);      //Código de lectura
  sec = BCDaBIN(i2c_read()&0x7f);     //Lectura de los 7 bit de los segundos
  min = BCDaBIN(i2c_read()&0x7f);     //Lectura de los 7 bit de los minutos
  hor  = BCDaBIN(i2c_read(0)&0x3f);  //Lectura de los 6 bit de las horas
  i2c_stop();
}
```

Figura 52. Fichero RTC_DS1307.C

El termómetro digital y termostato I2C DS1621 permite medir temperaturas entre -55 °C y 125 °C. El valor de temperatura se suministra en dos bytes, el byte alto es el valor entero con resolución de 1 °C y el segundo byte es el valor decimal con resolución de 0.5 °C (figura 53).

TEMPERATURE	DIGITAL OUTPUT (Binary)	DIGITAL OUTPUT (Hex)
+125°C	01111101 00000000	7D00h
+25°C	00011001 00000000	1900h
+½°C	00000000 10000000	0080h
+0°C	00000000 00000000	0000h
-½°C	11111111 10000000	FF80h
-25°C	11100111 00000000	E700h
-55°C	11001001 00000000	C900h

Figura 53. Formato de la temperatura (cortesía de Dallas Smc.)

La dirección asignada en el esquema es la 0x01. Tiene un registro de control para el funcionamiento como termostato que en esta aplicación no se utiliza. La palabra de control para la lectura o escritura es 1001A3A2A1-R/W (figura 55). Los comandos de control pueden ser, entres otros, 0xAA para lectura de la temperatura, 0xEE para el inicio de la conversión. Con estos datos se puede escribir el fichero para el control del DS1621 (figura 54).

```
void init_temp(int address) {
   i2c_start();
   i2c_write(0x90|(address<<1)); //Genera primer byte (1001A2A1A0W)
   i2c_write(0xee); //Inicia conversión
   i2c_stop();
}
float read_full_temp(int address) {
   signed int datah;
   int datal;
   float tura;
```

```
    i2c_start();
    i2c_write(0x90 |(address<<1)); //Genera primer byte (1001A2A1A0-W)
    i2c_write(0xaa);               //Leer temperatura
    i2c_start();
    i2c_write(0x91 |(address<<1)); //Genera primer byte (1001A2A1A0-R)
    datah=i2c_read();             //Lectura parte alta
    datal=i2c_read(0);            //Lectura parte baja y NACK
    i2c_stop();
       tura=datah;               //Conversión a flotante
       if (datal==128) tura=tura+0.5;
       return(tura);
}
```

Figura 54. Fichero TEMP_DS1621.C

Figura 55. Protocolo de lectura/escritura (cortesia de Dallas Smc.)

Puesto que la temperatura es un *FLOAT* no se puede guardar este dato directamente en la *EEPROM* según las funciones dadas en el fichero EEPROM_24512.C; por lo tanto, se

puede utilizar el fichero suministrado por CCS C llamado FLOATEE.C para guardar y leer datos tipo *FLOAT* en una *EEPROM*. El fichero se muestra en la figura 56.

```
WRITE_FLOAT_EXT_EEPROM(long int n, float data) {
   int i;
   for (i = 0; i < 4; i++)
     write_ext_eeprom(i + n, *((int8 *)(&data) + i));
}
float READ_FLOAT_EXT_EEPROM(long int n) {
   int i;
   float data;
   for (i = 0; i < 4; i++)
     *((int8 *)(&data) + i) = read_ext_eeprom(i + n);
   return(data);
}
```

Figura 56. Fichero FLOATEE.C

Una vez definidos los ficheros para el manejo de los periféricos se puede escribir el programa principal. El programa se comunica con un terminal del puerto serie de tal forma que mediante un menú se pueden elegir dos opciones: Con 1 se inicia la lectura de temperatura y tiempo para almacenarlo en la *EEPROM* y con 2 se visualiza, a través del puerto serie, los primeros datos de la *EEPROM* (los 4 bytes del *FLOAT* de la temperatura y los 3 bytes del tiempo –seg., min. y horas–). La comunicación serie se realiza por interrupción.

```
#include <16F876.h>
#fuses XT,NOWDT,NOPROTECT,NOLVP
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=pin_c6, rcv=pin_c7, bits=8,parity=N)
#use i2c(Master,sda=PIN_C4,scl=PIN_C3)

#include <lcd.c>
#include <EEPROM_24512.c>      //Ficheros driver de los periféricos
#include <RTC_ds1307.c>
#include <TEMP_ds1621.c>
#include <floatee.c>

int dat_in, cnt, hr,min,sec;
int16 address=0;
int dat_serie[7];
float dato;

#int_rda                //Interrupción para el puerto serie
```

```
rda_isr()
{
dat_in=getc();   //Lee el puerto serie
printf("\r");
if (dat_in=='2') //Si es "2" se visualizan los primeros datos de la EEPROM
  {
    for(cnt=0;cnt<=6;cnt++)  //Lectura de los 7 primeros bytes de la EEPROM
     {
      dat_serie[cnt]=read_ext_eeprom(cnt);
     }
   for(cnt=0;cnt<=6;cnt++)  //Visualiza los 7 primeros bytes de la EEPROM
     {
      printf("Byte %u=%3u \r",cnt,dat_serie[cnt]);
     }
  }
}
void main() {
   lcd_init();
   enable_interrupts(int_rda);
   enable_interrupts(global);
   address=0;

  printf("Pulsar 1 para leer datos\r");    //Menú para el terminal serie
  printf("Pulsar 2 para visualizar datos\r");

while(1){

 if(dat_in=='1')   //Si es "1" se inicia la lectura y grabado en la EEPROM
  {
   init_temp(0x01);       //Inicializa el DS1621
   delay_ms(100);
   tiempo(hr,min,sec);   //Lee tiempo del DS1307
   dato = read_full_temp(0x01);  //Lee temperatura del DS1621
   WRITE_FLOAT_EXT_EEPROM(address,dato);  //Guarda 4 bytes del FLOAT
   address=address+4;

   WRITE_EXT_EEPROM(address++,hr);     //Guarda byte de hora
   WRITE_EXT_EEPROM(address++,min);    //Guarda byte de minuto
   WRITE_EXT_EEPROM(address++,sec);    //Guarda byte de segundo
   lcd_gotoxy(1,1);
   printf(lcd_putc,"Temp=%4.1f C\n",dato);  //Visualiza la temperatura
   printf(lcd_putc,"%2u:%2u:%2u",hr,min,sec);  //Visualiza la hora
```

```
    if (address==0xffff) address=0;     //Cuando se termina la EEPROM vuelve
                                        //al principio.
  }
 }
}
```

Figura 57. Programa principal del ejemplo 5

Se puede utiliza el *HyperTerminal* para comunicarse con el PIC. La configuración se muestra en la figura 58; el resultado se muestra en la figura 59.

Figura 58. Configuración del HyperTerminal

Figura 59. Pantalla de comunicación

En el comando *DEBUG* del PROTEUS se puede visualizar la *EEPROM* serie mediante el comando *I2C MEMORY INTERNAL MEMORY* (figura 60).

Figura 60. Memoria interna de la EEPROM I2C

Capítulo 8

Gama Alta – PIC18

8.1 Introducción

En los últimos años, Microchip ha lanzado varias gamas de PIC con elevadas prestaciones, los PIC18, los PIC24 y los dsPIC. Con la gama alta (PIC18), Microchip mantiene la arquitectura básica que tan buenos resultados ha obtenido con la gama baja y media y, además, reduce las limitaciones de estas dos últimas. Los PIC18 tienen una arquitectura *RISC* avanzada *Harvard* con 16 bits de bus de programa y 8 bits de bus de datos (figura 1).

Figura 1. Arquitectura PIC18

La memoria de programa aumenta hasta 1 MWord (en realidad se manejan hasta 64 Kbytes pero llegan hasta los 2 Mbytes con memoria externa) y desaparece la paginación. La memoria de datos *RAM* puede llegar hasta 16 x 256 (4 KBytes) y hasta los 1 Kbytes de *EEPROM*.

La pila aumenta hasta 31 niveles. Incluyen tres punteros *FSR* que permiten direccionar la memoria de datos de forma indirecta y sin bancos. El juego de instrucciones aumenta hasta las 75 instrucciones. Introduce un multiplicador hardware 8x8. La frecuencia máxima de reloj es de 40 MHZ y la velocidad de procesador llega a los 10 MIPS con oscilador de 10 MHz. Incluye periféricos de comunicación avanzados (*CAN* y *USB*). Con la filosofía tradicional de Microchip, los PIC18 son compatibles con los PIC16CXX y PIC17CXX. Además ha desarrollado un compilador C específico para esta gama alta, el C18.

Características Gama Alta	PIC18F2420	PIC18F2520	PIC18F4420	PIC18F4520
Frecuencia de trabajo	DC-40MHz	DC-40MHz	DC-40MHz	DC-40MHz
Reset (y delays)	POR, BOR, Instrucción RESET, Pila llena, Desbordamiento de Pila (PWRT, OST), MCLR (opcional), WDT	POR, BOR, Instrucción RESET, Pila llena, Desbordamiento de Pila (PWRT, OST), MCLR (opcional), WDT	POR, BOR, Instrucción RESET, Pila llena, Desbordamiento de Pila (PWRT, OST), MCLR (opcional), WDT	POR, BOR, Instrucción RESET, Pila llena, Desbordamiento de Pila (PWRT, OST), MCLR (opcional), WDT
Memoria de Programa (bytes)	16384	32768	16384	32768
Memoria de Programa (instrucciones)	8192	16384	8192	16384
Memoria Datos (bytes)	768	1536	768	1536
Memoria Datos EEPROM	256	256	256	256
Interrupciones	19	19	20	20
Puertos E/S	Ports A,B,C,(E)	Ports A,B,C,(E)	Ports A,B,C,E	Ports A,B,C,D,E
Temporizadores	4	4	4	4
Módulos CCP	2	2	1	1
Módulos CCP mejorados	0	0	1	1
Comunicaciones serie	MSSP, EUSART	MSSP,EUSART	MSSP,EUSART	MSSP,EUSART
Comunicaciones Paralelo			PSP	PSP
Modulo AD de 10 bits	10 CANALES	10 CANALES	13 CANALES	13 CANALES
Número de Instrucciones	75 (83 Ext.)	75 (83 Ext.)	75 (83 Ext.)	75 (83 Ext.)
Programación con bajo voltaje	Si	Si	Si	Si
Detección de baja tensión	Si	Si	Si	Si
Encapsulados	28-pin PDIP 28-pin SOIC 28-pin QFN	28-pin PDIP 28-pin SOIC 28-pin QFN	40-pin PDIP 44-pin QFN 44-pin TQFP	40-pin PDIP 44-pin QFN 44-pin TQFP

Figura 2. Características de PIC18F2420, PIC18F2520, PIC18F4420 y PIC18F4520

8.2 Organización de la memoria

El PIC18F4520 dispone de las siguientes memorias:

- **Memoria de programa:** memoria *FLASH* interna de 32.768 bytes:
  - Almacena instrucciones y constantes/datos.
  - Puede ser escrita/leída mediante un programador externo o durante la ejecución del programa mediante unos punteros.
- **Memoria RAM de datos:** memoria *SRAM* interna de 1.536 bytes en la que están incluidos los registros de función especial:
  - Almacena datos de forma temporal durante la ejecución del programa.
  - Puede ser escrita/leída en tiempo de ejecución mediante diversas instrucciones.

- **Memoria EEPROM de datos:** memoria no volátil de 256 bytes.
  - Almacena datos que se deben conservar aun en ausencia de tensión de alimentación.
  - Puede ser escrita/leida en tiempo de ejecución a través de registros.
- **Pila:** bloque de 31 palabras de 21 bits.
  - Almacena la dirección de la instrucción que debe ser ejecutada después de una interrupción o subrutina.
- **Memoria de configuración:** memoria en la que se incluyen los bits de configuración (12 bytes de memoria *flash*) y los registros de identificación (2 bytes de memoria de sólo lectura).

8.2.1 Arquitectura HARDVARD

El PIC18F4520 dispone de buses diferentes para el acceso a la memoria de programa y a la memoria de datos (arquitectura *Harvard*):

- Bus de la memoria de programa:
  - 21 líneas de dirección.
  - 16/8 líneas de datos (16 líneas para instrucciones/8 líneas para datos).
- Bus de la memoria de datos:
  - 12 líneas de dirección.
  - 8 líneas de datos.

Esto permite acceder simultáneamente a la memoria de programa y a la memoria de datos. Es decir, se puede ejecutar una instrucción (lo que por lo general requiere acceso a la memoria de datos) mientras se lee de la memoria de programa la siguiente instrucción (proceso *pipeline*).

8.2.2 Memoria de Programa

El PIC18F4520 dispone de una memoria de programa de 32.768 bytes (0000H-7-FFFH) (figura 3). Las instrucciones ocupan 2 bytes (excepto las instrucciones *CALL, MOVFF, GOTO* y *LSFR* que ocupan 4). Por lo tanto, la memoria de programa puede almacenar hasta 16.384 instrucciones.

Primero se almacena la parte baja de la instrucción y luego la parte alta (para las instrucciones de 4 bytes primero los bytes menos significativos y luego los más significativos). Las instrucciones siempre empiezan en direcciones pares.

La operación de lectura en la posición de memoria por encima de 7FFFH da '0' como resultado (equivalente a la instrucción *NOP*).

Direcciones especiales de la memoria de programa:

- Vectorización del *Reset* es 0000H.
- Vectorización de las interrupciones de alta prioridad es la 0008H.
- Vectorización de las interrupciones de baja prioridad es la 0018H.

Figura 3. Memoria de Programa

La memoria de programa puede ser leída, borrada y escrita durante la ejecución del programa. La operación que se utiliza normalmente en tiempo de ejecución es la de lectura de tablas o datos almacenados en memoria de programa.

8.2.3 Contador de Programa

El PC (contador de programa) tiene 21 bits (*PCU, PCH* y *PCL*). El bit menos significativo del PC apunta a BYTEs, no a WORDs, por lo que es "0". El PC se incrementa de dos en dos. Se dispone de los correspondientes registros auxiliares *PCLATU* y *PCLATH* para actuar de forma combinada con el PC cuando éste se escribe o se lee.

- **PCU:** parte superior del PC, registro no directamente accesible; las operaciones de lectura/escritura sobre este registro se hacen a través del registro *PCLATU*.
- **PCH:** parte alta del PC, registro no directamente accesible; las operaciones de lectura/escritura sobre este registro se hacen a través del registro *PCLATH*.
- **PCL:** parte baja del PC, registro directamente accesible. Una operación de lectura sobre *PCL* provoca que los valores de *PCU* y *PCH* pasen a los registros *PCLATU* y *PCLATH*, respectivamente. Y una operación de escritura sobre *PCL* provoca que los valores de *PCLATU* y *PCLATH* pasen a *PCU* y *PCH*, respectivamente. El *PCL* siempre tiene el bit menos significativo a '0' debido a que las instrucciones siempre empiezan en direcciones pares.

8.2.4 Memoria de Configuración

Se trata de un bloque de memoria situado a partir de la posición 30000H de la memoria de programa (más allá de la zona de memoria de programa de usuario). En esta memoria de configuración se incluyen:

- **Bits de configuración:** contenidos en 12 bytes de memoria *flash* permiten la configuración de algunas opciones del PIC como:
  - Opciones del oscilador.
  - Opciones de *reset*.
  - Opciones del *watchdog*.
  - Opciones de la circuitería de depuración y programación.
  - Opciones de protección contra escritura de la memoria de programa y de la memoria *EEPROM* de datos.

Estos bits se configuran generalmente durante la programación C, aunque también pueden ser leidos y modificados durante la ejecución del programa.

- **Registros de identificación:** se trata de dos registros situados en las direcciones 3FFFFEH y 3FFFFFH que contienen información del modelo y revisión del dispositivo. Son registros de sólo lectura y no pueden ser modificados por el usuario.

8.2.5 Pila

La Pila es un bloque de memoria *RAM* independiente, de 31 palabras de 21 bits y un puntero de 5 bits, que sirve para almacenar temporalmente el valor del PC cuando se produce una llamada a una subrutina o interrupción. El *"Top Of Stack"* es accesible, se puede leer y escribir (será conveniente quitar previamente las interrupciones).

El puntero de pila (contenido en el registro *STKPTR*) es un contador de 5 bits que indica la posición actual del final de pila. El contenido del final de pila es accesible mediante los registros *TOSU, TOSH, TOSL*.

Cuando se procesa una interrupción o se ejecutan las instrucciones *CALL* o *RCALL* (el PC está apuntando a la siguiente instrucción) se incrementa el *STKPTR* y se almacena el valor del PC en el final de pila. Cuando se ejecutan las instrucciones *RETURN, RETLW* o *RETFIE* se copia en el PC el valor almacenado en la cima de pila y se decrementa el *STKPTR*.

8.2.6 Memoria de Datos

Los PIC18 tienen hasta un total de 4 KBytes agrupados en 16 bancos, con 256 bytes cada uno. Como en el resto de las gamas, existen los registros de propósito general *GPR* y los registros especiales *SFR*; éstos últimos se sitúan en la zona más alta (desde F00h hasta FFFh).

El PIC18F4520 dispone una memoria RAM de datos 1.536 bytes (6 bancos de 256 bytes). Además dispone de 126 bytes dedicados a los registros de función especial (*SFRs*) situados en la parte alta del banco 15 (figura 4).

Figura 4. Memoria RAM PIC18F4520

Para acceder a un byte de la memoria *RAM* de datos primero se debe seleccionar el banco al que pertenece el byte mediante el registro de selección de banco (*BSR*) y, a continuación, direccionar el byte dentro del banco. Además existe una modalidad de acceso rápido a las 126 posiciones de la parte baja del banco 0 y a los 126 bytes de *SFR* (banco de acceso rápido).

La memoria *RAM* de datos se compone de registros de propósito general (*GPRs*) y de registros de función especial (*SFRs*). Los *SFRs* son los registros mediante los cuales se puede monitorizar/controlar el funcionamiento de la *CPU* y de las unidades funcionales del PIC. En el PIC18F4520 se sitúa en el bloque de memoria de 0xF80 a 0xFFF (figura 5). Se distinguen dos conjuntos de *SFRs*:

SFRs asociados con el núcleo del PIC:

CPU: WREG, STATUS, BSR, etc.

Interrupciones: INTCON, PIE1, PIR1, IPR1, etc.

Reset: RCON.

SFRs asociados con las unidades funcionales:

Timers: T0CON, TMR1H, TMR1L, T1CON, etc.

Convertidor A/D: ADRESH, ADRESL, ADCON0, ADCON1, etc.

EUSART: TXREG, TXSTA, RCSTA, etc.

CCP: CCPR1H, CCPR1L, CCP1CON, etc.

MSSP: SSPSTAT, SSPDATA, SSPCFG, etc.

Puertos de E/S: TRISA, PORTA, TRISB, PORTB, etc.

PORT A	0xF80	SPBRG	0xFAF
PORT B	0xF81	-----	
PORT C	0xF82	TIMER1L	0xFCE
PORT D	0xF83	TIMER1H	0xFCF
PORT E	0xF84	-----	
-----		TIME0L	0xFD6
TRIS A	0xF92	TIMER0H	0xFD7
TRIS B	0xF93	-----	
TRIS C	0xF94	WERG	0xFE8
TRIS D	0xF95	-----	
TRIS E	0xF96	STKPTR	0xFFC

Figura 5. Registros SFR

8.2.7 Memoria EEPROM

El PIC18F4520 dispone una memoria *EEPROM* de datos de 256 bytes. Al ser una memoria no volátil, los datos almacenados en ella se mantienen en ausencia de

tensión de alimentación. El acceso a esta memoria se realiza mediante los *SFRs*: *EECON1, EECON2, EEDATA* y *EEADR*. Esta memoria permite hasta 1.000.000 de ciclos de borrado/escritura. Se puede leer/escribir de forma individual en cada una de las 256 posiciones de memoria.

Cuando se realiza una operación de escritura, la circuitería interna del PIC se encarga de borrar previamente la posición en la que se desea escribir. La duración de un ciclo completo de borrado/escritura de un byte en la memoria *EEPROM* suele ser de unos 4 ms.

8.2.8 Modos de Direccionamiento

El modo de direccionamiento es la forma en la que se obtienen los datos que van a ser utilizados en la instrucción. Existen 4 modos de direccionamiento: **INHERENTE, LITERAL, DIRECTO** e **INDIRECTO**.

- **Modo de direccionamiento inherente:** en este modo, o bien la instrucción no tiene operando o bien el operando viene especificado en el propio código de operación de la instrucción.
- **Modo de direccionamiento literal:** en este modo, el valor del operando viene indicado de forma explícita en la instrucción.
- **Modo de direccionamiento directo:** en este modo, la dirección en la que se encuentra el valor del operando viene indicada de forma explícita en la instrucción.
- **Modo de direccionamiento indirecto:** en este modo, la dirección de memoria en la que se encuentra el dato viene especificado en uno de los registros *FSR0, FSR1* y *FSR2*.

8.2.9 Interrupciones

Se dispone de dos niveles de prioridad:

- Nivel alto vectorizado en la dirección 0008H.
- Nivel bajo, vectorizado en la dirección 0018H.

Todas las interrupciones pueden ser programadas con cualquiera de las dos prioridades, salvo la interrupción externa 0 (que siempre tiene alta prioridad). Se puede forzar el modo compatible "sólo alta prioridad", mediante el bit **IPEN = 0**.

GIE/GIEH & PEIE/GIEL controlan los respectivos permisos globales. Cuando se sirve una interrupción, automáticamente se quita su correspondiente permiso global. El servicio de interrupción de alta prioridad impide el servicio de baja prioridad. Cuando se ejecuta *RETFIE* se pone el permiso correspondiente al nivel que se está sirviendo.

Todas las interrupciones disponen de 3 bits de configuración (excepto la interrupción externa 0 que tiene dos):

- **Bit de habilitación de interrupción:** permite habilitar a nivel individual la interrupción.
- **Flag de interrupción:** se pone a '1' cuando se produce la condición de interrupción independientemente de si la interrupción está habilitada o no. Este *flag* debe ponerse '0' por software cuando se procesa la interrupción.
- **Bit de prioridad de interrupción:** establece si la interrupción es de alta o de baja prioridad (este bit no está disponible para la interrupción externa 0).

El PIC18F4520 dispone de 20 fuentes de interrupciones. Se distinguen dos grupos de interrupciones:

- Grupo general de interrupciones:

Interrupción de Temporizador 0
Interrupción por cambio en PORTB
Interrupción externa 0
Interrupción externa 1
Interrupción externa 2

- Grupo de interrupciones de periféricos:

Interrupción del SSP	**Interrupción del fallo del oscilador**
Interrupción del A/D	**Interrupción del comparador**
Interrupción de recepción de la EUSART	**Interrupción del CCP2**
Interrupción de transmisión de la EUSART	**Interrupción de escritura en Flash/EEPROM**
Interrupción del MMSP	**Interrupción de colisión de bus (MSSP)**
Interrupción del CCP1	**Interrupción de detección de anomalías en V_{pp}**
Interrupción del Temporizador 1	**Interrupción del Temporizador 2**
Interrupción del Temporizador 3	

En el compilador C se modifica la directiva **#INT_XXXX** de tal forma que se pueden incluir las palabras clave **HIGH** y **FAST**.

Una prioridad **HIGH** puede interrumpir a otra interrupción. Una prioridad **FAST** se realiza sin salvar o restaurar registros (ver el siguiente apartado).

Así, en los PIC18 se pueden dar las siguientes interrupciones en C:

#INT_xxxx	Prioridad normal (baja) de interrupción. El compilador guarda y restaura los registros clave. Esta interrupción no interrumpe a otras en progreso.
#INT_xxxx FAST	Interrupción de alta prioridad. En compilador NO guarda y restaura los registros clave. Esta interrupción puede interrumpir a otras en progreso. Sólo se permite una en el programa.
#INT_xxxx HIGH	Interrupción de alta prioridad. El compilador guarda y restaura los registros clave. Esta interrupción puede interrumpir a otras en progreso.

- **#INT_xxxx:** Prioridad normal (baja) de interrupción. El compilador guarda y restaura los registros clave. Esta interrupción no interrumpe a otras en progreso.
- **#INT_xxxx FAST:** Interrupción de alta prioridad. En compilador **NO** guarda y restaura los registros clave. Esta interrupción puede interrumpir a otras en progreso. Sólo se permite una en el programa.
- **#INT_xxxx HIGH:** Interrupción de alta prioridad. El compilador guarda y restaura los registros clave. Esta interrupción puede interrumpir a otras en progreso.

El PIC18F4550 tiene las fuentes de interrupción mostradas en la figura 6.

Figura 6. Fuentes de interrupción del PIC18F4520 desde el Wizard del CCS

8.2.9.1 Registros de salvaguarda

Las interrupciones se disparan durante la ejecución de código del programa principal o de otra interrupción. Esto hace que durante la ejecución de la rutina de tratamiento de la interrupción se pueda modificar el valor de los registros que están siendo utilizados por otras partes del código.

Para evitar que estas modificaciones alteren el correcto funcionamiento del sistema conviene almacenar los valores de estos registros al inicio de la interrupción para recuperarlos al final.

Se puede salvar y restaurar el contenido de las variables de entorno (*WREG*, *STATUS* y *BSR*) en sus respectivos registros sombra, lo que equivale a una pila de un sólo nivel.

8.2.10 Registro W

El registro *WREG* pasa a ser un registro direccionable (0xFE8), por lo que se puede utilizar de forma explícita.

8.2.11 Oscilador

El PIC18F4520 permite múltiples configuraciones:

LP	Cristal Baja-Potencia (max. 200KHz)
XT	Cristal-Resonador (max. 4MHz)
HS	Cristal-Resonador Alta-Velocidad (max. 40MHz)
HSPLL	Cristal-Resonador Alta-Velocidad con habilitación de PLL (max. 10MHz)
RC	Externa R-C con FOSC/4 salida en RA6 (max. 4MHz)
RCIO	Externa R-C con I/O en RA6 (max. 4MHz)
INTIO1	Oscilador Interno FOSC/4 salida en RA6 e I/O en RA7 (30/500 KHz, 1/4/8 MHz)
INTIO2	Oscilador Interno con I/O en RA6 y RA7
EC	Reloj Externo con FOSC/4 de salida (max. 40MHz).
ECIO	Reloj Externo con I/O en RA6 (max. 40MHz).

La disponibilidad de oscilador interno permite múltiples configuraciones (figura 7).

Figura 7. Configuración del oscilador interno

8.2.12 Unidades Funcionales

El PIC18F4520 dispone de una serie de *Unidades Funcionales* que le permiten:

- Realizar tareas específicas especializadas (conversión A/D, transmisión/recepción de datos, generación de señales digitales con temporizaciones programables, etc.).

- Optimizar el rendimiento del *PIC*, ya que estas unidades trabajan en paralelo a la *CPU* permitiendo que ésta se centre en otras tareas como el procesado de datos, cálculos, movimiento de datos, etc.

Las *Unidades Funcionales* más importantes del PIC18F4520 son:

Puerto de E/S	**Unidad de Comparación/Captura/PWM mejorada (ECCP)**
Temporizador 0	**Canal de comunicación serie EUSART**
Temporizador 1	**Canal de comunicación serie MSSP**
Temporizador 2	**Módulo analógico de comparación**
Temporizador 3	**Canal de transmisión de datos en paralelo (SPP)**
Convertidor A/D	**Acceso a memoria externa (EMA)**
	Unidad de Comparación/Captura/PWM (CCP)

8.2.12.1 Puertos de entrada/salida

El PIC18F4520 dispone de 5 puertos de E/S que incluyen un total de 36 líneas digitales de E/S:

PUERTO	LINEAS DE ENTRADA/SALIDA
PORTA	8 LINEAS DE ENTRADA/SALIDA
PORTB	8 LINEAS DE ENTRADA/SALIDA
PORTC	6 LINEAS DE ENTRADA/SALIDA + 2 LINEAS DE ENTRADA
PORTD	8 LINEAS DE ENTRADA/SALIDA
PORTE	3 LINEAS DE ENTRADA/SALIDA + 1 LINEA DE ENTRADA

Todas las líneas digitales de E/S disponen, como mínimo, de una función alternativa asociada a alguna circuitería específica del PIC. Cuando una línea trabaja en el modo alternativo no puede ser utilizada como línea digital de E/S estándar.

Figura 8. Esquema de un terminal

Cada puerto de E/S tiene asociado 3 registros:

- Registro **TRIS:** mediante este registro se configuran cada una de las líneas de E/S del puerto como *ENTRADA* (bit correspondiente a '1') o como *SALIDA* (bit correspondiente a '0').
- Registro **PORT:** mediante este registro se puede leer el nivel de pin de E/S y se puede establecer el valor del *latch* de salida.
- Registro **LAT:** mediante este registro se puede leer o establecer el valor del *latch* de salida.

8.2.12.2 Temporizadores

TEMPORIZADOR 0:

- Configurable como temporizador/contador de 8 bits/16 bits.
- Pre-escalar de 8 bits programable.
- Interrupción por desbordamiento.

TEMPORIZADOR 1:

- Configurable como temporizador/contador de 16 bits.
- Dispone de un oscilador propio que puede funcionar como:
    - Señal de reloj del temporizador 1.
    - Señal de reloj del PIC en modos de bajo consumo.
- Pre-escalar de 3 bits programable.
- Interrupción por desbordamiento.

TEMPORIZADOR 2:

- Temporizador de 8 bits (registro *TMR2*).
- Registro de periodo *PR2*.
- Pre-escalar de 2 bits programable (1:1, 1:4, 1:16).
- Post-escalar de 4 bits (1:1...1:16).
- Interrupción por igualdad entre *TMR2* y *PR2*.
- Se puede utilizar junto con los módulos *CCP* y *ECCP*.
- Se puede utilizar como señal de reloj del módulo *MSSP* en modo *SPI*.

TEMPORIZADOR 3:

- Configurable como temporizador/contador de 16 bits.
- Dispone de varias opciones de señal de reloj en el modo temporizador:
    - Oscilador principal con o sin pre-escalar.
    - Oscilador del temporizador 1 con o sin pre-escalar.

- Pre-escalar de 3 bits programable.
- Interrupción por desbordamiento.

Figura 9. WDT y TMR0, TMR1 y TMR2

Figura 10. TMR3

8.2.12.3 Convertidor Analógico-Digital

- 10 bits de resolución.
- 13 canales multiplexados.

- Señal de reloj de conversión configurable.
- Tiempo de adquisición programable (0 a 20 TAD).
- Posibilidad de establecer el rango de tensiones de conversión mediante tensiones de referencia externas.

Figura 11. Modulo AD

8.2.12.4 Canal de Comunicación Serie (EUSART)

Características fundamentales:

- Modos de trabajo:
  - Modo asíncrono de 8 bits.
  - Modo asíncrono de 9 bits.
  - Modo síncrono *Maestro*.
  - Modo síncrono *Esclavo*.
- Auto-activación por detección de dato recibido.
- Detección automática de velocidad de comunicación (*baudrate*).
- Transmisión y detección de carácter de *BREAK* (bus *LIN*).

Figura 12. **Comunicación Serie**

8.2.12.5 Modulo Master SSP (MSSP)

El módulo *MSSP* es un interfaz serie capaz de comunicarse con periféricos u otro microcontrolador. Puede operar en dos modos:

- *Serial Peripheral Interface (SPI).*
- *Inter-Integrated Circuit (I2C).*

La interfaz *I2C* admite los siguientes modos:

- Master mode.
- Multi-Master mode.
- Slave mode.

Figura 13. Modo SPI

Figura 14. Modo I2C

8.2.12.6 Modulo de Compración/Captura/PWM (CCP)

Dispone de tres modos de funcionamiento:

- **Modo de Captura:** se utiliza para medir eventos externos como la duración de pulsos digitales.
- **Modo de Comparación:** se utiliza para generar señales digitales con temporizaciones programables. Este tipo de señales son muy útiles para el control de etapas de potencia (convertidores DC/DC, DC/AC, AC/DC, AC/DC).
- **Modo PWM:** se utiliza para generar señales de modulación de ancho de pulso (*PWM*).

También existe un modulo de **comparacion/captura/PWM mejorado (ECCP).**

Dispone de cuatro modos de funcionamiento:

- **Modo de Captura:** se utiliza para medir eventos externos como la duración de pulsos digitales.
- **Modo de Comparación:** se utiliza para generar señales digitales con temporizaciones programables. Este tipo de señales son muy útiles para el control de etapas de potencia (convertidores DC/DC, DC/AC, AC/DC, AC/DC).
- **Modo PWM:** se utiliza para generar señales de modulación de ancho de pulso (*PWM*).
- **Modo PWM mejorado:** se utiliza para generar señales *PWM* complementarias para el control de semipuentes de transistores.

Figura 15. Módulos CCP y ECCP

NOTA
El módulo *ECCP* no funciona correctamente en la última versión del *Proteus*. Es de esperar que LabCenter lo solucione en breve.

8.2.12.7 Modulo Comparador

El módulo de comparadores analógicos contiene dos comparadores que pueden ser configurados de distintas formas. Las entradas pueden seleccionarse entre las entradas analógicas de los pins RA0 a RA5. Las salidas digitales (normal o invertida) son accesibles exteriormente y pueden ser leídas a través de un registro de control.

Figura 16. Módulo Comparador

8.2.12.8 Modulo de referencia

El módulo de referencia está formado por una red de resistencias y permite seleccionar una tensión de referencia.

Figura 17. Referencia

8.2.12.9 Modulo detector de Alto/Bajo Voltaje

Este módulo programable permite, al usuario, definir un punto umbral de tensión y la dirección de cambio. Si el dispositivo experimenta un cambio en la tensión y en la dirección indicada sobre el punto umbral se produce una interrupción.

Figura 18. HLVD y LVD

Ejemplo 1: *Diseñar un Reloj en Tiempo Real (RTC) utilizando la interrupción del TMR1 con oscilador exterior; utilizar para la visualización un display de 7 segmentos* (figura 19). *Componentes ISIS: PIC18F4520, RES, 7SEG-BCD y Generador ISIS: Pulse.*

Figura 19. Ejemplo 1

El *Proteus* tiene un componente *7SEG-BCD* que es un display 7 de segmentos con el decodificador *7 SEG-BCD* integrado, lo que facilita la simulación.

El empleo del *TIMER1* con oscilador exterior permite trabajar a dicho *TIMER1* incluso cuando el PIC está en modo *sleep*. El *Proteus* no simula el oscilador externo del *TMR1* con cristal de cuarzo (figura 20), por lo que hay que sustituirlo por un

generador de pulsos con la frecuencia adecuada e introducirlo por el pin *RC0/T1OSO* (figura 21). Al editar las características del generador de pulso hay que cambiar el *Pulse (High) voltaje* a 5 V y la *frequency* a 32.768 Hz.

Para conseguir una interrupción cada 1 seg con dicha frecuencia hace falta precargar el *TMR1* a 0x8000 según los cálculos de la siguiente ecuación.

Figura 20. TMR1 con oscilador de cuarzo externo

Figura 21. TMR1 con oscilador externo simulado con un generador

Figura 22. Características del generador Pulse

La interrupción del *TMR1* deberá ser de alta prioridad. Los datos en binario (segundos, minutos y horas) se deben pasar a dos dígitos en *BCD*.

NOTA

Se han utilizado los puertos A, B y D (ya que el C es necesario para el oscilador). En el puerto A de los PIC18, la patilla RA4 ya NO es un *Drenador Abierto* pero el *PROTEUS* sigue tratándola como tal, por lo que hace falta una resistencia de *pull-up* a la salida. En realidad esto no sería necesario. Por otra parte, la simulación no es en tiempo real por lo que 1 segundo de simulación puede tardar varios segundos de procesador del ordenador; se puede comprobar la relación en la barra inferior del *Proteus* donde nos indica el tiempo de procesado (figura 23).

Figura 23. Tiempo de procesado

```
#include <18F4520.h>
#device high_ints=true        //Para manejar interrupciones de alta prioridad
#use delay(clock=10000000)
int horas=0,minutos=0,segundos=0;
int LSdigito, MSdigito,BCD;
#INT_TIMER1 high              //Interrupción de alta prioridad del TMR1
VOID int_tmr1(void)
{
segundos++;                     //Cuenta los segundos
if (segundos==60)
   {minutos++;                  //Cuenta los minutos
```

```
   segundos=0;}
if (minutos==60)
   {horas++;                    //Cuenta las horas
    minutos=0;}
if (horas==24) horas=0;
set_timer1(0X8000);             //Precarga el TNMR1 antes de salir
}
void BINaBCD(int valor){        //Función de conversión de Binario a BCD
 MSdigito=0;
 if (valor>=10){
    do{
      valor=valor-10;
      MSdigito++;
    }while (valor>=10);
  }
  LSdigito=valor;                 //Unidades en BCD
  MSdigito=Msdigito<<4;         //Decenas en BCD (se desplazan a la parte
                                //alta del int)
  BCD=MSdigito | LSdigito;      //OR entre los dos números para obtener un
                                //entero que se pueda sacar directamente por
}                               //el puerto
void main()
{
   setup_adc_ports(NO_ANALOGS|VSS_VDD);
   setup_wdt(WDT_OFF);
   setup_timer_1(T1_EXTERNAL|T1_DIV_BY_1);   //TMR1 con oscilador externo
                                             //modo asincrono y prescaler=1
   setup_comparator(NC_NC_NC_NC);
   setup_vref(FALSE);
   setup_oscillator(False);
   enable_interrupts(INT_TIMER1);
   enable_interrupts(GLOBAL);
   set_timer1(0X8000); //TMR1=65536-(1/TOSC1)=65536-32768=32768=0x8000
   while(true)
   {
   BINaBCD(segundos);     //Convierte los segundos de binario a BCD…
   OUTPUT_D(BCD);   //los saca por el puerto D.
   BINaBCD(minutos);
   OUTPUT_B(BCD);
   BINaBCD(horas);
   OUTPUT_A(BCD);
   }
}
```

Figura 24. Programa del Ejemplo 1

Es necesario deshabilitar el *watchdog* en el PIC (figura 25).

Figura 25. Deshabilitación del Watchdog

Ejemplo 2: *Elevador de tensión (voltaje-boost converter) con realimentación de control (figura 26). Componentes ISIS: PIC18F4520, RES, IRF130, INDUCTOR, 10BQ015 y CAP.*

Figura 26. Elevador de tensión

NOTA

En *PROTEUS*, el módulo comparador en el modelo del PIC18 no funciona correctamente en la simulación, así también el módulo *ECCP* tiene algunos problemas. Esperemos que LabCenter lo solucione en breve.

Se configura el módulo *CCP* para trabajar en modo *PWM* y el comparador para que comparé la señal de salida (dividida por un coeficiente) y la tensión de referencia interna (en este caso se fija a 3.59 V aunque se puede modificar su valor por programación). En el programa se modifica el módulo de referencia con la función **setup_vref()**, pero el comparador se modifica directamente en su registro **CMCON** ya que con la función **setup_comparator()** existe un incorrecta asignación de valores en el fichero <18F4520.h>.

El módulo de referencia se configura para obtener un valor de 3.59 V mediante la programación del registro **CVRCON=0b11001111** o en compilador C de CCS: **setup_vref(VREF_HIGH|15|VREF_F5)**.

El módulo comparador se configura para introducir la señal externa por *RA0* y utilizar la referencia interna (figura 27) mediante la programación del registro **CMCON= 0b00000110**, en C debería ser **setup_comparator(A0_VR_A1_VR)** pero se ha detectado un error en la asignación **#define** del fichero de cabecera (se puede modificar o utilizar directamente el valor de **CMCON**). La frecuencia de la señal *PWM* se ha fijado en 4 KHz.

Figura 27. Modo del comparador

El sistema funciona en un equilibrio dinámico, mientras la tensión de salida (dividida) sea menor que la V_{REF} la señal *PWM* actúa, si es mayor la señal *PWM* es 0. De esta forma se consigue una tensión continua en la salida del convertidor. De esta forma se consigue una V_{out}=20 ± 0.27 V.

Es importante inicializar la salida (con un label **IC=5** sobre el cable) para evitar errores de convergencia en la simulación.

```
#include <18f4520.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#byte CMCON=0xFB4
#byte TRISA=0xF92
```

```
void main() {
int16  dutty;
TRISA= 0b11101011;
CMCON= 0b00000110;                    //setup_comparator(A0_VR_A1_VR);
setup_vref(VREF_HIGH|15|VREF_F5);     //CVRCON=0b11001111;
  setup_timer_2(T2_DIV_BY_4,62,1);    //4KHz
  setup_ccp1(CCP_PWM);                //CCP1 en modo PWM
  set_pwm1_duty(714);

while(1) {
  IF(C1OUT) dutty=714;                //Si Vout/Coef < VREF
  else dutty=0;                       //Si Vout/Coef > VREF
  set_pwm1_duty(dutty);
  }
}
```

Figura 28.Programa del ejemplo 2

Si se quiere utilizar una V_{REF} distinta a la interna para ajustar más finamente la tensión de salida, se puede utilizar el módulo comparador según la figura 29, introduciendo la señal por *RA0* y una referencia externa por *RA3*.

Figura 29. Modo del Comparador

Figura 30. Circuito con referencia externa

```
#include <18f4520.h>
#fuses XT,NOWDT
#use delay(clock=4000000)
#byte CMCON=0xFB4
#byte TRISA=0xF92
void main() {
int16  dutty;

TRISA= 0b11101011;
CMCON= 0b00000100;
  setup_timer_2(T2_DIV_BY_4,62,1);
  setup_ccp1(CCP_PWM);
  set_pwm1_duty(714);


while(1) {
  IF(C1OUT) dutty=714;
  else dutty=0;
  set_pwm1_duty(dutty);
  }
}
```

Figura 31. Programa para referencia externa

Capítulo 9

RTOS – Real Time Operating System

9.1 Introducción

El *Sistema Operativo en Tiempo Real* simplifica el desarrollo de una aplicación y, mediante el uso de tareas, reduce los errores de programación. En general, se puede definir un *RTOS* como un programa que trabaja en segundo plano, controla la ejecución de varias tareas y facilita la comunicación entre ellas. En el caso de que se esté ejecutando más de una tarea al mismo tiempo, el sistema se denomina multitarea; cada tarea tiene asignado un tiempo de procesador.

El *RTOS* no es exactamente un *SO* (sistema operativo) a pesar de que los dos se basan en un núcleo (*kernel*) que se encarga de controlar la de ejecución de las tareas. La diferencia estriba en la carga inicial, si es sólo el núcleo (*RTOS*) o si además se cargan otros procesos (*SO*). El *RTOS* está pensado para trabajar con los microcontroladores. Puede utilizarse en los PIC de gama media pero donde mayor rendimiento se obtiene es en los PIC de gama alta.

El *RTOS* que utiliza *CCS* permite el PIC ejecute regularmente las tareas programadas sin necesidad de interrupciones. Este se logra a través de la función **RTOS_RUN()** que actúa como planificador de tareas (*dispatcher*). La función del planificador consiste en dar el control del procesador a la tarea que debe ejecutarse en un momento dado.

Cuando la tarea ha terminado de ejecutarse o ya no necesita del procesador, el control de dicho procesador es devuelto al planificador el cual dará el control del procesador a la siguiente tarea que esté lista para ejecutarse en ese momento. Este proceso se conoce como cooperativo multitarea (*cooperative multi-tasking*)

Un buen ejemplo de la optimización del uso del PIC con el *RTOS* es el *PID* utilizado en el tema sobre el módulo *CCP* para regular la temperatura de un horno (ejemplo

5). Estudiando el programa realizado en esa ocasión se observa como el PIC está **totalmente ocupado en realizar el proceso PID**; si se necesitase utilizar dicho microcontrolador para realizar más funciones se deberia realizar una programación bastante compleja, intentando siempre respetar el tiempo de muestreo del PID.

Utilizando el *RTOS* en dicho ejemplo, no sólo se simplifica el programa sino que además se consigue mejorar la respuesta en el sistema ya que se optimizan los recursos del micro.

De los dos casos estudiados (tiempos de muestreo de 100 ms y 1 ms), en el primer caso se reduce el sobreimpulso y las oscilaciones y en el segundo caso se alcanza antes la temperatura fijada (Figura 1).

Pero además, la utilización de *RTOS* va ha permitir añadir fácilmente más tareas al PIC como veremos en los próximos ejemplos.

Figura 1. Respuesta del PID con y sin programación RTOS

9.2 RTOS en C

Las funciones que incorpora *CCS* para la gestión del *RTOS* son las que aparecen en la siguiente tabla:

rtos_run()	Inicia la operación del *RTOS*. Todas las tareas se ejecutan a través de esta función.
rtos_terminate()	Finaliza la operación del *RTOS* y devuelve el control al programa principal, a la línea siguiente de la función *rtos_run()*. Es como una función *RETURN*.
rtos_enable(task)	Habilita una de las tareas *RTOS*. Una vez la tarea es habilitada, la función *rtos_run()* llamará a esta tarea cuando le toque por tiempo. El parámetro de esta función es el nombre de la tarea.
rtos_disable(task)	Deshabilita una de las tareas *RTOS*. Una vez la tarea es deshabilitada, la función *rtos_run()* no llamará a esta tarea hasta que vuelva a ser habilitada mediante la función *rtos_enable()*.El parámetro de esta función es el nombre de la tarea.
rtos_msg_poll()	Devuelve un *TRUE* si hay un dato en la cola de mensajes de tareas.
rtos_msg_read()	Devuelve el siguiente byte del dato contenido en la cola de mensajes de tareas.
rtos_msg_send(task,byte)	Envía un byte de datos para una tarea concreta. El dato es situado en la cola de mensajes de tareas.
rtos_yield()	Se llama desde una de las tareas y devuelve el control del programa a la función *rtos_run()*. Todas las tareas deben tener una llamada a esta función al final de su código.
rtos_signal(sem)	Incrementa un SEMAFORO que se utiliza para difundir la disponibilidad de un recurso limitado.
rtos_wait(sem)	Espera a que el recurso asociado con el semáforo esté disponible y entonces decrementa el semáforo para reclamar el recurso.
rtos_await(expre)	Espera a que la EXPRESION sea *TRUE* antes de permitir que la tarea continúe.
rtos_overrun(task)	Retorna un *TRUE* si la tarea ha sobrepasado el tiempo permitido de ejecución.
rtos_stats(task,stat)	Retorna una ESTADISTICA de una tarea concreta. La estadística incluye el tiempo mínimo y máximo de ejecución de la tarea y el tiempo total que la tarea ha sobrepasado su ejecución.

Las directivas asociadas son dos:

#use rtos (options)

Las *options* pueden ser las siguientes (separadas por comas):

timer=X: donde X (0 a 4) indica el *TIMER* que se utilizará para la ejecución de las tareas. El *TIMER* seleccionado debe utilizarse para el *RTOS*.

minor_cycle=time: donde *TIME* es un número seguido de s, ms, µs o ns. Indica el tiempo que deberá tardar en ejecutarse una tarea. Los tiempos de ejecu-

ción de cada tarea deben ser múltiplos de esta cantidad. Si no se especifica, el compilador se encargará de calcularlo.

statistics: indica el tiempo mínimo, máximo y el total utilizado por cada tarea.

#task (options): Esta directiva indica al compilador que la función que le sigue es una tarea de *RTOS*.

Las *options* pueden ser las siguientes (separadas por comas):

rate=time: donde *TIME* es un numero seguido de s, ms, μs, o ns. Especifica la frecuencia con se ejecutará la tarea. Debe ser igual o múltiplo del valor *minor_cycle*.

max=time: donde *TIME* es un numero seguido de s, ms, μs, o ns. Especifica el tiempo de ejecución de la tarea. Este tiempo debe ser menor o igual que el valor *minor_cycle*. El compilador no puede hacer cumplir este tiempo por lo que el programador debe tener cuidado en asignar el tiempo de ejecución. Además, este tiempo activa la función *rtos_overrun(task)*.

queue=bytes: especifica cuantos bytes son colocados en la cola de mensajes. El valor por defecto es 0.

Como primer ejemplo se puede utilizar el controlador *PID* (ejemplo 5) del tema sobre el módulo *CCP*. Comparando el programa escrito en aquella ocasión y el que se presenta ahora con *RTOS* (figura 2), se observa que el bloque del *PID* que estaba en el programa principal es ahora una tarea. El tiempo de muestreo estaba especificado con un *delay_ms()* y ahora se utiliza el tiempo de ejecución de la tarea como tiempo de muestreo. Tal como se ha comentado en el punto anterior, se ha obtenido una mejora en el funcionamiento del sistema (figura 1).

```
#INCLUDE <16F877.h>
#device adc=10
#use delay(clock=4000000)
#fuses XT,NOWDT
#Byte TRISC = 0x87

#use rtos(timer=0,minor_cycle=1ms)        //Directiva del RTOS

int16 valor;                    //lectura de temperatura
int16 control;                  //valor del PWM
float a=0.1243;                 //constantes del PID para 100ms
float b=0.00006;
float c=62.1514;
float temp_limit=500.0;                 //temperatura a alcanzar
float rT,eT,pT,qT,yT,uT;                //variables de ecuaciones
float pT_1=0.0;
float eT_1=0.0;
```

```
float max=1000.0;                //limites máximo y minimo de control
float min=0.0;

#task(rate=1ms, max=1ms)     //Indica que la siguiente función es una Tarea
void pid ( );                //que se ejecutará cada 100ms.

void main(){
TRISC=0;
setup_timer_2(t2_div_by_4,249,1);      //Periodo de la señal PWM a 1ms
setup_ccp1(ccp_pwm);                   //Módulo CCP a modo PWM
setup_adc_ports(all_analog);           //Puerto A analógico
setup_adc(ADC_CLOCK_INTERNAL);         //reloj convertidor AD interno
set_adc_channel(0);                    //Lectura por el canal 0

rtos_run ( );                          //Inicia la operación de RTOS
}

  void pid ( )                         //Tarea PID
{
  output_bit( PIN_C0, 0);
   valor=read_adc();
  yT=valor*5000.0/1024.0;
  rT=temp_limit;
  eT=rT-yT;
  pT=b*eT+pT_1;
  qT=c*(eT-eT_1);
  uT=pT+a*eT+qT;
  if (uT>max) {
     uT=max;}
  else {
     if (uT<min){
        uT=min;}
        }
  control=uT;
  set_pwm1_duty(control);
  pT_1=pT;
  eT_1=eT;
}
```

Figura 2. Programa del PID con RTOS

En el ejemplo se puede añadir una tarea que permita visualizar en un *display LCD* la temperatura que tiene en horno (figura 3). Al manejar varias tareas es conveniente utilizar la función **rtos_yield()** para devolver el control al planificador de tareas

y coordinar las tareas mediante las funciones **rtos_wait()** y **rtos_signal()**; para utilizar estas dos funciones se debe utilizar un "semáforo", el cual permite a una tarea utilizar un recurso compartido o esperar en el caso de que esté en uso.

Figura 3. Dos tareas: PID y DISPLAY

En este ejemplo no se utiliza ningún recurso compartido, por lo que no se utilizan las funciones **rtos_wait()** y **rtos_signal()**.

```
#INCLUDE <16F877.h>
#device adc=10
#use delay(clock=4000000)
#fuses XT,NOWDT
#include <lcd.c>
#use rtos(timer=0,minor_cycle=1ms)
#Byte TRISC = 0x87
int16 valor, control;
float a=0.1243;
float b=0.00006;
```

```
float c=62.1514;
float temp_limit=500.0;
float rT,eT,pT,qT,yT,uT;
float pT_1=0.0;
float eT_1=0.0;
float max=1000.0;
float min=0.0;
float tempera;

#task(rate=1ms,max=1ms)                    //Tarea del PID
void pid ( );

#task(rate=10ms,max=1ms)                   //Tarea del DISPLAY
void display( );

void main(){
lcd_init();
TRISC=0;
setup_timer_2(t2_div_by_4,249,1);
setup_ccp1(ccp_pwm);
setup_adc_ports(all_analog);
setup_adc(ADC_CLOCK_INTERNAL);
set_adc_channel(0);

rtos_run ( );
}

void pid ( )                               //Tarea del PID
   {
  valor=read_adc();
  yT=valor*5000.0/1024.0;
  rT=temp_limit;
  eT=rT-yT;
  pT=b*eT+pT_1;
  qT=c*(eT-eT_1);
  uT=pT+a*eT+qT;
  if (uT>max) {
     uT=max;}
  else {
     if (uT<min){
        uT=min;}
        }
  control=uT;
```

```
  set_pwm1_duty(control);
  pT_1=pT;
  eT_1=eT;

   rtos_yield();                //Se devuelve el control al planificador
   }

void display()                  //Tarea del DISPLAY
{
lcd_gotoxy(1,1);
tempera=yT/10;
printf(lcd_putc, "Temp= %F",tempera);

   rtos_yield();               //Se devuelve el control al planificador
}
```

Figura 4. El programa

Las variables se pueden definir de forma global o particular como en cualquier función. En el caso de variables globales, todas las tareas pueden utilizarlas, pero en el caso de funciones particulares se pueden utilizar las funciones de correo: **rtos_msg_poll(), rtos_msg_read()** y **rtos_msg_send(task,byte)** para transferir información. En el ejemplo anterior la variable utilizada es global, por lo que puede ser utilizada por las dos tareas, pero en el caso de ser local se podrían utilizar las funciones de correo (aunque la variable sea global también se pueden utilizar). En el caso de variables globales puede ser interesante el uso de estas funciones en aquellos casos en los que una de las tareas esté esperando que la variable modifique su valor a través de otra tarea.

Para utilizar estas funciones en el ejemplo se modifica la llamada a la tarea *DISPLAY* y se añade una variable para el correo (figura 5).

```
INT16 valor_1;

#task(rate=10ms,max=1ms,queue=2)                //Cola de 2 bytes.
void display( );
```

Figura 5. Modificaciones del programa (I)

Se realiza el envío de la variable valor desde la tarea *PID* y se recibe en la tarea *DISPLAY* para su posterior uso (figura 6). El resultado es similar al del ejemplo anterior.

```
void pid ( )
 {
 valor=read_adc();
 yT=valor*5000.0/1024.0;
 rT=temp_limit;
```

```
.................                          //El resto de código como siempre
rtos_msg_send(display,valor);              //Envia los 2 bytes de Valor a DISPLAY
rtos_yield();
 }

void display()
{
valor_i=rtos_msg_read();                   //Recibe los 2 bytes de Valor

tempera=valor_i*500.0/1024.0;
lcd_gotoxy(1,1);
printf(lcd_putc, "Temp= %F", tempera);

rtos_yield();
}
```

Figura 6. Modificaciones del programa (II)

Para terminar con este ejemplo se ha añadido una tercera tarea que permita modificar la temperatura límite en cualquier momento mediante un botón en la patilla *RB0*. Para esta tercera opción se ha cambiado a un PIC18F4520. Como ya se ha comentado, el *RTOS* adquiere toda su eficiencia en los PIC de gama alta (figura 7).

Figura 7. Tres tareas: PID, DISPLAY y TECLA

Dado que en este caso se modifica una variable en dos de las tareas es conveniente utilizar las funciones **rtos_wait()** y **rtos_signal()** (figura 8).

```
#INCLUDE <18F4520.h>
#device adc=10
#use delay(clock=4000000)
#fuses XT,NOWDT
#include <lcd.c>
#use rtos(timer=0,minor_cycle=1ms)
#use standard_io(B)
#use standard_io(c)
#define B0 PIN_B0

int16 valor;            //lectura de temperatura
int16 control;          //valor del PWM
float a=0.1243;
float b=0.00006;
float c=62.1514;                //constantes del PID
float temp_limit=500.0;         //temperatura a alcanzar
float rT,eT,pT,qT,yT,uT;
float pT_1=0.0;
float eT_1=0.0;                 //variables de ecuaciones
float max=1000.0;
float min=0.0;          //límites máximo y mínimo de control.
float tempera;          //Para visualizar la temperatura del horno.
int16 t_1;              //Para visualizar la temperatura límite.
int8 sem;               //Variable de semáforo.


#task(rate=1ms,max=1ms)
void pid ( );

#task(rate=10ms,max=1ms,queue=2)
void display( );

#task(rate=10ms,max=1ms)
void teclado ( );


void main(){

lcd_init();
```

```
setup_timer_2(t2_div_by_4,249,1);  //periodo de la señal PWM a 1ms
setup_ccp1(ccp_pwm);               //Módulo CCP a modo PWM

setup_adc_ports(all_analog);       //Puerto A analógico
setup_adc(ADC_CLOCK_INTERNAL);     //reloj convertidor AD interno
set_adc_channel(0);

sem=1;
rtos_run ( );
}

void pid ( )
   {
   rtos_wait(sem);

  valor=read_adc();                //Lectura de la temperatura
  yT=valor*5000.0/1024.0;          //conversión a mV (0.25V a 250mV)
  rT=temp_limit;
  eT=rT-yT;              //Cálculo error
  pT=b*eT+pT_1;          //Cálculo del término integral
  qT=c*(eT-eT_1);        //Cálculo del término derivativo
  uT=pT+a*eT+qT;         //Cálculo de la salida PID

  if (uT>max) {          //Salida PID si es mayor que el MAX
     uT=max;}
  else {
     if (uT<min){        //Salida PID si es menor que el MIN
        uT=min;}
        }
  control=uT;            //Transferencia de salida PID a señal PWM
  set_pwm1_duty(control);

  pT_1=pT;               //Guardar variables
  eT_1=eT;

   rtos_signal(sem);
   rtos_yield();

   }

void display()
{
rtos_wait(sem);
```

```
tempera=yT/10;
lcd_gotoxy(1,1);
printf(lcd_putc, «Temp= %F\n», tempera);
t_l=temp_limit/10;
printf(lcd_putc, "Limite= %Lu", t_l);

rtos_signal(sem);
rtos_yield();
}

void teclado()
{
rtos_wait(sem);
if (input(PIN_C4)==0) temp_limit=temp_limit+10.0;
if (temp_limit > 1000.0) temp_limit=500.0;

rtos_signal(sem);
rtos_yield();
}
```

Figura 8. Programa con PIC18F4520

Capítulo 10

USB – Universal Serial Bus

NOTA DEL AUTOR
Cuando este libro estaba prácticamente en imprenta, LabCenter realizó una actualización de la versión 7 con un nuevo (y esperado) modelo: el USB (*USB-CONN*). CCS C ya proponía ejemplos y suministraba librerías para utilizar USB que no se podían simular en ISIS. Con este nuevo modelo se puede abandonar ya la simulación por puerto serie RS-232 (la mayoría de los PC ya no disponen de este puerto) y afrontar el USB. Al estar el libro en imprenta, sólo he podido incluir algo de teoría básica de USB y algún ejemplo sencillo sobre emulación RS-232 con USB; espero que si hay más ediciones de este libro pueda completar y aumentar este capítulo.

10.1 Introducción

El *Bus Serie Universal* fue creado en los años 90 por una asociación de empresas con la idea, entre otras, de mejorar las técnicas *plug-and-play*, es decir, permitir a los dispositivos conectarse y desconectarse sin necesidad de reiniciación, configurándose automáticamente al ser conectados; además se le dotó de transmisión de energía eléctrica para los dispositivos conectados.

Este bus tiene una estructura de árbol y se pueden ir conectado dispositivos en cadena, pudiéndose conectar hasta 127 dispositivos permitiendo la transferencia síncrona y asíncrona.

Se puede clasificar según su velocidad de transferencia de datos (desde kilobits hasta megabits): Baja Velocidad (1.0) utilizado para los Dispositivos de Interfaz Humana (HID) como ratones, etc.; Velocidad Completa (1.1) y Alta Velocidad (2.0) para conexiones a Internet, etc.

Físicamente, los datos del *USB* se transmiten por un par trenzado (D+ y D-) además de la masa y alimentación (+5V). Los conectores están sujetos al estándar (tipo A, tipo B).

USB es un bus punto a punto, con inicio en el *HOST* y destino en un dispositivo o en un *HUB*; sólo puede existir un único *HOST* en la arquitectura *USB*. *HOST* se define como el dispositivo maestro que inicializa la comunicación y *HUB* es el dispositivo que contiene uno o más conectores o conexiones hacia otros dispositivos *USB*; cada conector es un puerto *USB*. El protocolo de comunicación se basa en el paso de testigo (*token*), donde el *HOST* proporciona el testigo al dispositivo seleccionado y éste le devuelve el testigo como respuesta.

10.1.1 Migración de RS232 a USB

La interfaz serie RS-232 está desapareciendo prácticamente de los ordenadores personales y esto supone un problema, ya que muchas de las aplicaciones con microcontroladores utilizan este bus para su comunicación con el PC. La solución ideal es migrar a una interfaz USB y existen distintas formas de hacerlo. El método más sencillo es emular RS-232 con el USB, con la ventaja de que el PC verá la conexión USB como una conexión COM RS-232 y no requerirá cambios en el software existente. Otra ventaja es que se utilizan *drivers* suministrados por *Windows*, por lo que no es necesario desarrollar uno *ad hoc*; estos *drivers* son el **usbser.sys** y el **ccport.sys**. Además, puesto que el protocolo USB maneja comunicaciones de bajo nivel, los conceptos *baud rate*, bit de paridad y control de flujo para el RS-232 ya no importan.

10.1.1.1 USB CDC (Communication Device Class)

Una clase USB es una agrupación de dispositivos de características comunes, es decir, utilizan una misma forma de comunicarse con el entorno. La clase de dispositivo permite conocer la forma en que la interfaz se comunica con el sistema, el cual puede localizar el *driver* que puede controlar la conectividad entre la interfaz y el sistema.

USB sólo permite al *driver* comunicarse con el periférico a través de las tuberías (*pipes*) establecidas entre el sistema *USB* y los *endpoints* del periférico. Los tipos de transferencia a través de las *pipes* dependen del *endpoint* y pueden ser: *Bulk*, *Control*, *Interrupt* e *Isochronous*. Una tubería es un enlace virtual entre el *HOST* y el dispositivo *USB*, donde se configura el ancho de banda, el tipo de transferencia, la dirección del flujo de datos y el tamaño del paquete de datos.

Estos enlaces se definen y crean durante la inicialización del *USB*. Un *endpoint* es un *buffer* dentro del dispositivo o periférico donde se almacenan paquetes de información; todos los dispositivos deben admitir el *endpoint 0*, el cual recibe el control y las peticiones de estado durante la enumeración del dispositivo. Cuando se conecta un dispositivo al *HOST* se produce la enumeración en la cual el *HOST* interroga al

dispositivo sobre sus características principales, asignándole una dirección y permitiendo la transferencia de datos.

La especificación *Clase de Dispositivo de Comunicación* (CDC) define algunos modelos de comunicación, incluyendo la comunicación serie. *Windows* suministra el *driver* **usber.sys** para esta especificación. Para la especificación *CDC* se necesitan dos interfaces *USB*, primero la interfaz *Communication Class* usando un IN *interrupt endpoint* de interrupción y el segundo es la intefaz *Data Class* usando un OUT *bulk endpoint* y un IN *bulk endpoint*. Esta interfaz es utilizada para transferir los datos que normalmente deberían ser transferidos a través de la interfaz RS-232.

Desde el punto de vista de sistema USB, el dispositivo puede tener distintas configuraciones, para cada una de las cuales puede funcionar de forma distinta. Los dispositivos suministran la información necesaria al sistema USB a través de los descriptores; éstos contienen unos campos que permiten al sistema clasificar al dispositivo y asignarle un *driver*. La primera información que necesita es la del fabricante y producto (*USB vendor ID – VIP* y el *Product ID – PID*). El VIP es un número de 16 bits asignado por el *USB implementers Forum (USB-IF)* y debe ser obtenido por el fabricante del dispositivo USB; cada *VID* puede contener 65.536 PID diferentes al ser también un número de 16 bits. Microchip suministra su VIP y los PID para cada familia de PIC con USB.

Microsoft Windows (2000 o XP) no tiene un fichero **.inf* estándar para el driver *CDC*, así que es necesario suministrar este fichero cuando se conecta un dispositivo USB por primera vez al sistema. Microchip suministra el fichero **mchpcdc.inf** necesario para sus dispositivos PIC.

10.2 USB con ISIS y CCS C

10.2.1 USB en ISIS

LabCenter ha incorporado, en su versión 7, dos herramientas para la simulación de circuitos con USB: el conector USB llamado **USBCONN** (figura 1), el cual permite conectar y desconectar el bus y el visualizador de USB llamado **Analizador de Transiciones USB** (figura 2); éste último se debe adquirir como un módulo aparte.

Figura 1. USBCONN

Además, el software de LABCENTER incorpora los *drivers* necesarios para la simulación con USB. Para instalarlos hay que ir a la opción *INICIO > PROGRAMAS >*

PROTEUS 7 PROFESSIONAL > VIRTUAL USB > INSTALL USB DRIVERS. Tras este proceso ya se puede trabajar con el **USBCONN**.

Figura 2. Analizador de Transiciones USB

10.2.2 USB en CCS C

CCS suministra librerías para comunicar PIC con el PC utilizando el bus USB, mediante periféricos internos (familia PIC18F4550 o el PIC16C765) o mediante dispositivos externos al PIC (del tipo USBN9603).

Las librerías suministradas son:

- **pic_usb.h:** *driver* de capa hardware de la familia PIC16C765.
- **pic_18usb.h:** *driver* de capa hardware de la familia PIC18F4550.
- **usbn960x.h:** *driver* de capa hardware para el dispositivo externo USBN9603/USBN9604. De esta forma, se puede utiliza el bus USB en cualquier PIC.
- **usb.h:** definiciones y prototipos utilizados en el *driver* USB.
- **usb.c:** El USB *stack*, que maneja las interrupciones USB y el USB *Setup Requests* en *Endpoint 0*.
- **usb_cdc.h:** *driver* que permite utilizar una clase de dispositivo *CDC USB*, emulando un dispositivo RS232 y lo muestra como un puerto *COM* en *Windows*.

Las funciones más importantes, entre otras muchas, son:

- **usb_init():** Inicializa el hardware USB. Espera en un bucle infinito hasta que el periférico USB es conectado al bus (aunque eso no significa que ha sido enumerado por el PC). Habilita y utiliza la interrupción USB.
- **usb_task():** Si se utiliza una detección de conexión para la inicialización, entonces se debe llamar periódicamente a esta función para controlar el pin de detección de conexión. Cuando el PIC es conectado o desconectado del bus, esta función inicializa el periférico USB o resetea el *USB stack* y el periférico.

- **usb_enumerated():** Devuelve un *TRUE* si el dispositivo ha sido enumerado por el PC y, en este caso, el dispositivo entra en modo de operación normal y puede enviar y recibir paquetes de datos.

Existen funciones específicas para *CDC*, entre ellas:

- **usb_cdc_putc(c):** Es idéntica a *put(c)* y envía un carácter. Coloca un carácter en el *buffer* de transmisión; en el caso de que esté lleno esperará hasta que pueda enviarlo.
- **usb_cdc_getc(c):** Es idéntica a *get(c)* y lee un carácter. Recibe un carácter del *buffer* de transmisión; en el caso de estar vacío esperará hasta que se reciba.

CCS aporta varios ejemplos de aplicación de USB según las clases de dispositivos, por ejemplo para el *CDC* encontramos el **EX_USB_SERIAL.C** y el **EX_USB_SERIAL2.C**.

Ejemplo 1: *Enviar los datos de una conversión AD al puerto USB como Virtual Comm. Componentes ISIS: PIC18F4550, USBCONN, POTLIN, CELL, RES y LED-BLUE.*

El ejemplo se basa en la aplicación para *CDC*, **EX_USB_SERIAL2.C**, donde el USB emula un puerto serie COM. De todas las posibles clases de dispositivos, la *CDC* es la más sencilla de aplicar y entender (dada su similitud con el funcionamiento de un puerto serie). El ejemplo **EX_USB_SERIAL2.C** permite la visualización de una parte de la memoria *EEPROM* del PIC. La estructura de librerías de CCS se muestra en la figura 3.

Figura 3. Librerías en EX_CDC_SERIAL2.C

Se va ha modificar este ejemplo para adaptarlo a los ejemplos del tema 7 sobre los puertos serie. Además de modificar el programa principal, hay que realizar una modificación importante en la librería de descriptores **USB_DESC_CDC.H**, donde se indica, al final de la librería, el fabricante y el producto (VIP/PID); en este caso aparece:

```
0x61,0x04, //vendor id (0x04D8 is Microchip, or is it 0x0461 ??) == 8,9
0x33,0x00, //product id == 10,11
```

Para trabajar con Microchip hay que indicar en el identificador de fabricante el *VENDOR ID 0x04D8* y en el identificador de productor el *PRODUCT ID 0x0A* para la familia de los PIC18. Es decir, esas dos líneas deben de quedar así (es aconsejable hacer una copia de la librería original antes de proceder al cambio):

```
0xD8,0x04, //vendor id == 8,9
0x0A,0X00, //product id == 10,11
```

Estos dos identificadores permiten la conexión con el *driver* de *Windows*; al iniciarse la conexión, *Windows* recibe los dos identificadores y localiza el *driver* necesario para la conexión. En el caso de no localizarlo, permite la instalación de los recursos necesarios para la conexión; en este caso a través del fichero **mchpcdc.inf** suministrado por Microchip.

Por otra parte, al final de la librería se encuentra la descripción textual del dispositivo que será detectado por *Windows*; para ello utiliza *USB_STRING_DESC*. Se puede modificar a gusto del usuario, teniendo cuidado con la definición de la posición de *strings* y sus tamaños.

```
char USB_STRING_DESC_OFFSET[]={0,4,12};
char const USB_STRING_DESC[]={
  //string 0
      4, //length of string index
      USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
      0x09,0x04, //Microsoft Defined for US-English
  //string 1
      8, //length of string index
      USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
      'C',0,
      'C',0,
      'S',0,
  //string 2
      30, //length of string index
      USB_DESC_STRING_TYPE, //descriptor type 0x03 (STRING)
      'C',0,
      'C',0,
```

```
        'S',0,
        ' ',0,
        'R',0,
        'S',0,
        '2',0,
        '3',0,
        '2',0,
        ' ',0,
        'D',0,
        'e',0,
        'm',0,
        'o',0
};
```

El ejemplo (figura 4) realiza la lectura de una señal analógica por el canal *AN0* y envía el dato por el *USB* emulando RS-232, de forma similar al ejemplo 2 del tema 7; el dato se enviará sólo en el caso de que varíe la tensión.

Figura 4. Ejemplo 1

Al igual que en el tema 7 se puede utilizar el *Hyperterminal* de *Windows* (o cualquier otro visor del puerto serie) para visualizar los datos. En este caso, hay que **esperar a conectar el USB** para que aparezca el puerto en las posibles conexiones del *Hyperterminal*.

Figura 5. Hyperterminal conectado al puerto Virtual

```
#include <18F4550.h>
#device adc=10
#fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL5,CPUDIV1,VREGEN
#use delay(clock=48000000)

 #define USB_CON_SENSE_PIN PIN_B2

 #include <usb_cdc.h>

void main() {
   BYTE i, j, address, value;

   int16 q,ql;
   float p;
   ql=0;
   setup_adc_ports(AN0|VSS_VDD);
   setup_adc(ADC_CLOCK_INTERNAL);
   set_adc_channel(0);

   usb_cdc_init();
   usb_init();

   do {
      usb_task();
      if (usb_enumerated()) {

      q = read_adc();
      if (q!=ql) {
      p = 5.0 * q / 1024.0;
      printf(usb_cdc_putc, "\n\r Voltage = %01.2fV", p);
      }
```

```
        q1=q;
        delay_ms(500);
    }
        } while (TRUE);
}
```

Figura 6. Programa del ejemplo 1

La configuración de PIC18F4550 es la habitual, tan sólo debemos tener en cuenta que la frecuencia debe ser de 48 MHz pero que utilizando el bit de configuración *PLL5* se puede emplear un cristal de cuarzo exterior de 20 MHz. Es importante configurar el pin de detección del sentido de transmisión (*USB_CON_SENSE_PIN*) para poder controlar la conexión o desconexión del PIC al USB; la conexión física se muestra en la figura 4. Antes de utilizar el puerto en el programa hay que inicializarlo (*usb_cdc_init()* y *usb_init()*) y, "muy importante", comprobar que ha sido enumerado por el *HOST* (*usb_enumerated()*). Con el fin de inicializar o resetear la conexión con el USB se debe realizar una llamada periódica a la función *usb_task()*.

Al iniciar la simulación con el *PLAY* del ISIS, el sistema permanecerá inactivo hasta que se conecte el **USBCONN** (con las flechas rojas correspondientes). Tras conectar, por primera vez el USB, *Windows* solicita la instalación del *driver* (se debe indicar el directorio donde se guarde el fichero **mchpcdc.inf**) (figura 7 y figura 8).

Figura 7. Pasos en la instalación del driver

Tras esta instalación y cada vez que se conecte el **USBCONN** aparecerá el puerto COM virtual en el administrador de dispositivos de *Windows* (figura 9). También desaparecerá el COM virtual cada vez que se desconecte.

Figura 8. Pasos en la instalación (continuación)

Figura 9. Puerto COM virtual en el administrador de dispositivos

En este instante el PIC queda conectado al PC y se puede abrir un visor del puerto serie para comprobar la transmisión (figura 5).

En los ejemplos de CCS también se pueden encontrar aplicaciones para *Mass Storage Device Class* (MSD) y *Human Interface Device Class* (HID).

Ejemplo 2: *Enviar los datos de una conversión AD al puerto USB como Virtual Comm para determinar la respuesta en lazo abierto de un horno* (figura 10). *Componentes ISIS: PIC18F4550, USBCONN, CELL, RES, LED-BLUE, SW-SPDT y OVEN.*

El ejemplo es similar al anterior, en este caso se utiliza *EXCEL* con *Visual Basic*, de tal forma que se pueden guardar los datos de tiempo y temperatura en una hoja de cálculo y representarlos gráficamente.

Figura 10. Ejemplo 2

Para utilizar el fichero en *Excel*, **HORNO.XLS**, se debe configurar el puerto serie virtual: en el fichero está definido como el puerto 9 y además habilitar las macros en el caso de tener Excel protegido (*Herramientas > Macro > Seguridad > Media/Bajo*). Para cambiarlo a la medida del usuario hay que abrir el editor de *Visual Basic* (*Herramientas > Macro > Editor de Visual Basic*), buscar *Commport = 9* y sustituirlo por el número adecuado de puerto serie virtual (figura 11).

```
Private Sub CommandButton1_Click()
     CommandButton2.BackColor = RGB(0, 255, 0)
     CommandButton1.BackColor = RGB(200, 200, 200)
     Worksheets("Hoja1").Columns("F").NumberFormat = "0.000"
     Worksheets("Hoja1").Columns("E").NumberFormat = "0.0"
    If puerto = 0 Then puerto = 1
    With MSComm1
    .Settings = "9600,N,8,1"
    .CommPort = 9
    .RThreshold = 1
    .InputMode = comInputModeBinary
    .InputLen = 0
    .PortOpen = True
    End With
    i = 0
End Sub
Private Sub CommandButton2_Click()
     CommandButton1.BackColor = RGB(0, 255, 0)
     CommandButton2.BackColor = RGB(200, 200, 200)
```

Figura 11. Modificación del puerto serie

```
#include <18F4550.h>
#device adc=10
#fuses HSPLL,NOWDT,NOPROTECT,NOLVP,NODEBUG,USBDIV,PLL5,CPUDIV1,VREGEN
#use delay(clock=48000000)
 #define USB_CON_SENSE_PIN PIN_B2
```

```
 #include <usb_cdc.h>
void main() {
   int16 q,n;
   float p;
   int qL;
   int qH;
   int nL,nH;
   setup_adc_ports(AN0);
   setup_adc(ADC_CLOCK_INTERNAL);

   n=0;
   usb_cdc_init();
   usb_init();
delay_ms(3000); //Retardo para poder abrir el puerto con el Excel
 do {
      usb_task();
      if (usb_enumerated()) {

      set_adc_channel(0);
      delay_us(10);
      q = read_adc();
      p = 5.0 * q / 1024.0;

      nL=make8(n,0);
      nH=make8(n,1);
      if (usb_cdc_putready()) usb_cdc_putc(nL);
      if (usb_cdc_putready()) usb_cdc_putc(nH);

      qL=make8(q,0);
      qH=make8(q,1);
      if (usb_cdc_putready()) usb_cdc_putc(qL);
      if (usb_cdc_putready()) usb_cdc_putc(qH);
      delay_ms(500);   //Se envian datos cada 0.5 s
      n++;
      }
            } while (TRUE);
}
```

Figura 12. Programa del ejemplo 2

Para la simulación se debe abrir el fichero de *Excel* (figura 13); en este fichero existen dos botones para abrir el puerto y cerrarlo (también hay un botón para borrar los datos adquiridos). Antes de abrir el puerto se debe arrancar el ISIS e iniciar la simulación del programa, a continuación se puede conectar el **USBCONN** y, una

vez conectado, se puede abrir el puerto con el botón de *Excel* para iniciar la adquisición.

Figura 13. Fichero de Excel

Para ver la curva de calentamiento se puede cerrar el interruptor del horno de tal forma que comience a calentarse (figura 14).

Figura 14. Datos adquiridos y su representación

Esta edición se terminó de imprimir en junio de 2008. Publicada
por ALFAOMEGA GRUPO EDITOR, S.A. de C.V. Apartado Postal
73-267, 03311, México, D.F. La impresión y encuadernación se realizaron
en FUENTES IMPRESORES, S.A, Centeno No. 109, Col. Granjas Esmeralda,
Iztapalapa, 09810, México, D.F.

Compilador C CCS y simulador PROTEUS para Microcontroladores PIC

Los microcontroladores *PICmicro* de Microchip han experimentado un importante aumento de presencia en el sector industrial, esto se debe, entre otros muchos factores, a la política de apertura que tiene Microchip, ya que facilita y potencia el desarrollo de herramientas por parte de otras compañías.

En lenguajes de programación destacan los compiladores C para PIC de compañías como CCS Inc. El desarrollo de un lenguaje C específico para un microcontrolador permite obtener el máximo rendimiento del micro.

Los programas de simulación permiten depurar hasta casi la perfección el diseño antes de ser montado en una placa. No hace falta explicar el ahorro de tiempo y coste que ello supone. Tal vez uno de los mejores simuladores para microcontroladores es el ISIS de PROTEUS.

En el capítulo 1 de este libro se hace una breve, pero intensa, descripción del ISIS de PROTEUS, de forma que el lector pueda afrontar la simulación de diseños sin ningún problema. En el capítulo 2 también se realiza un repaso del compilador C para PIC de CCS; obviamente no puede explicarse este lenguaje en un solo capítulo, pero tras su lectura cualquier lector podrá afrontar los pequeños programas de diseño que se exponen en los siguientes capítulos.

Del capítulo 3 al 7 se desarrollan los distintos módulos que integran un PIC (*ADC, USART,CCP*, etc.) a nivel hardware, enlazándolos con las correspondientes directivas y funciones del C. En cada capítulo se plantean y desarrollan sencillos ejemplos de aplicación que el lector podrá estudiar y, como no, modificar para completar sus conocimientos.

En el capítulo 8 se expone la gama alta (PIC18) y en el capítulo 9 una aplicación más compleja, el *RTOS (Real Time Operating System)*. También en estos capítulos se incorporan distintos ejemplos de aplicación. Por último, en el capítulo 10 se desarrolla el *USB* de reciente incorporación al ISIS.

Este libro está enfocado a todos aquellos lectores movidos por el interés acerca de los microcontroladores PIC sin necesidad de tener conocimientos muy profundos en la materia. Los ejemplos desarrollados no tienen una excesiva complejidad, son breves y permiten ir afianzando los conocimientos capítulo a capítulo.

www.alfaomega.com.mx

ISBN 978-970-15-1397-2

9 789701 513972

Alfaomega Grupo Editor