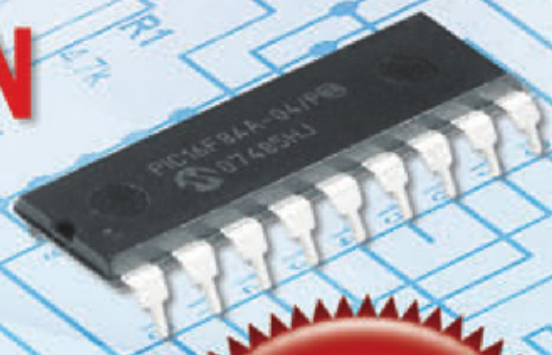
USERS

NO REQUIERE CONOCIMIENTOS PREVIOS

ELECTRÓNICA & MICROCONTROLADORES PIC

GUÍA PRÁCTICA DE PROGRAMACIÓN



ESCRITURA DE PROGRAMAS EN ASSEMBLER

ARQUITECTURA DE UN MICROCONTROLADOR PIC

PROGRAMACIÓN Y SIMULACIÓN POR SOFTWARE

GRABACIÓN DE PROGRAMAS EN EL MICROCONTROLADOR

CONTROL DE LEDS, DISPLAYS, TECLADOS Y SENSORES

por Víctor Rossano

★★★

RELOJ DIGITAL

FRECUENCÍMETRO

CONTADOR DE TURNOS

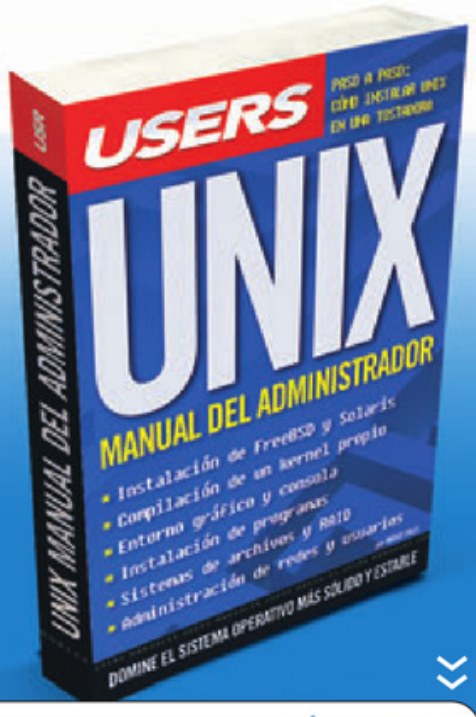
TERMÓMETRO DIGITAL

CERRADURA ELECTRÓNICA

INCLUYE PROYECTOS REALES

CONÉCTESE CON LOS MEJORES LIBROS DE COMPUTACIÓN

usershop.redusers.com

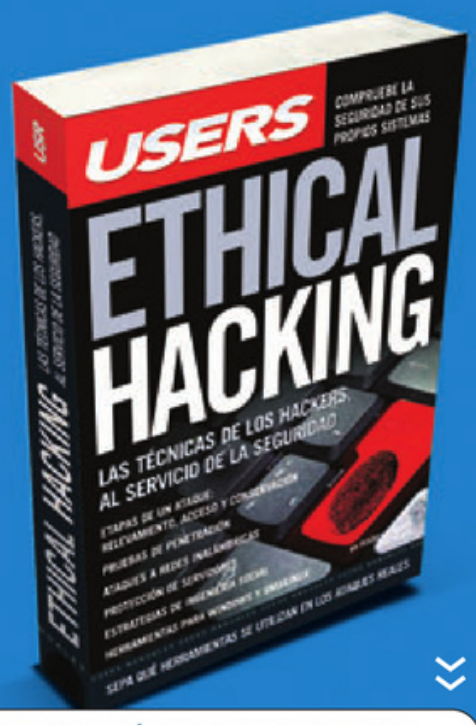


DOMINE EL SISTEMA OPERATIVO MÁS ESTABLE

MANUALES USERS | 320 páginas | ISBN 978-987-1347-94-0

UNA ALTERNATIVA COMPETITIVA A LOS MÉTODOS TRADICIONALES

DESARROLLADORES | 336 páginas | ISBN 978-987-1347-97-1



CONOZCA LAS TÉCNICAS DE LOS HACKERS

MANUALES USERS | 320 páginas | ISBN 978-987-1347-93-3

DESARROLLE DESDE LA PERSPECTIVA DE LOS MODELOS

DESARROLLADORES | 320 páginas | ISBN 978-987-1347-95-7

Léalo antes Gratis!

En nuestro sitio puede obtener, en forma gratuita, un capítulo de cada uno de los libros: **redusers.com**

RedUSERS
COMUNIDAD DE TECNOLOGÍA

Nuestros libros incluyen guías visuales, explicaciones paso a paso, recuadros complementarios, ejercicios, glosarios, atajos de teclado y todos los elementos necesarios para asegurar un aprendizaje exitoso y estar conectado con el mundo de la tecnología.

Conéctese con nosotros

> **ARGENTINA** ✆ (011) 4110.8700 | **CHILE** ✆ (2) 810.7400 | **ESPAÑA** ✆ (93) 635.4120

✉ **usershop@redusers.com**

redusers.com

TÍTULO: ELECTRÓNICA Y MICROCONTROLADORES PIC

AUTOR: Víctor Rossano

COLECCIÓN: Manuales USERS

FORMATO: 17 x 24 cm

PÁGINAS: 368

ISBN 978-987-663-002-3

Rossano, Víctor
Electrónica y microcontroladores PIC. - 1a ed. - Banfield - Lomas de Zamora : Gradi, 2009.
v. 169, 368 p. ; 24x17 cm. - (Manual users)

ISBN 978-987-663-002-3

1. Informática. I. Título

CDD 005.3

ELECTRÓNICA & MICROCONTROLADORES

PIC

GUÍA PRÁCTICA DE PROGRAMACIÓN

VÍCTOR ROSSANO

Es Ingeniero en Electrónica con especialidad en Sistemas Digitales. Cursó la carrera en la ESIME (Escuela Superior de Ingeniería Mecánica y Eléctrica), perteneciente al Instituto Politécnico Nacional, en la ciudad de México. Ha desarrollado su experiencia laboral en empresas dedicadas al ramo de la televisión, las telecomunicaciones y, de forma independiente, al de la computación. Desde siempre ha sido un apasionado por los temas relacionados con la ciencia y la tecnología.
Es autor del libro **Electrónica Digital**, de esta misma editorial.

Dedicatoria:

A mis padres, que siempre han estado ahí para apoyarme.

PRÓLOGO

Durante los últimos años se ha incrementado en forma notable el uso de microcontroladores para diseñar todo tipo de proyectos, desde los más sencillos hasta los más complejos. Antes de eso, se debían diseñar los circuitos digitales con componentes discretos (compuertas, flip-flops, contadores), pero hacer el diseño y luego construirlo era una tarea difícil y lenta, ya que para una aplicación medianamente compleja se podían necesitar muchos circuitos integrados, además de otros componentes. Estos circuitos eran difíciles de ensamblar y propensos a muchos errores. Un diseño con componentes discretos es complejo, grande, caro y poco confiable. Por eso, en la actualidad, el diseño con componentes discretos se limita sólo a los proyectos sencillos o con fines didácticos, ya que se ha reemplazado mayoritariamente por circuitos con microcontroladores.

Este libro pretende ser una introducción práctica al uso de microcontroladores PIC, y está dirigido a todos aquellos que deseen incursionar en el mundo del diseño de proyectos con microcontroladores. Para eso, hemos intentado hacer una descripción sencilla, fácil de entender y de seguir, aun para quienes no tengan ningún conocimiento o experiencia previa en el tema de los microprocesadores o microcontroladores. De todos modos, es bueno que quienes lean este libro cuenten con conocimientos de electrónica, y en especial de electrónica digital, al menos sobre los temas más básicos, lo que les facilitará en gran medida la comprensión de los conceptos que aquí se manejan.

En mi experiencia con el aprendizaje y el desarrollo de proyectos con microcontroladores, me he dado cuenta de que el verdadero conocimiento llega cuando uno mismo hace las cosas. No basta sólo con leer o entender los temas, es necesario entrar en acción y practicar mucho. Mi recomendación para lograr realmente un buen nivel de aprendizaje es construir los circuitos y ver cómo funcionan en la realidad. Además, y tal vez lo más importante, recomiendo escribir los propios programas. Si bien los ejemplos y librerías proporcionadas son funcionales, sólo son ejemplos, por lo que es aconsejable analizar y entender el funcionamiento de todos ellos. Además, siempre hay posibilidad de modificarlos, ampliarlos, mejorarlos, o adaptarlos a nuestras propias necesidades. La programación será más fácil a medida que practiquemos y escribamos varios programas por nuestra cuenta. Después de ensayar lo suficiente, podremos escribir programas eficientes de forma muy fácil y rápida. Espero que este libro sea de utilidad a cada uno de los lectores para iniciarse en este maravilloso mundo de los microcontroladores.

Víctor Rossano

EL LIBRO DE UN VISTAZO

En este libro nos adentraremos en el estudio de la arquitectura, la programación y las aplicaciones prácticas con microcontroladores PIC de la firma Microchip. En particular, hablaremos del PIC16F84A, uno de los microcontroladores más populares y empleados en todo el mundo, que nos servirá de plataforma para introducirnos en el mundo de los microcontroladores PIC, y para introducirnos en la arquitectura de estos componentes, que son cada vez más utilizados para el diseño de todo tipo de proyectos.

Capítulo 1

INTRODUCCIÓN A LOS MICROCONTROLADORES

Antes de comenzar el estudio en detalle del PIC16F84A, debemos conocer qué es un microprocesador y un microcontrolador, y cuál es la diferencia entre estos dos componentes. Por eso, en este capítulo hablaremos un poco de ello y comenzaremos a conocer el PIC16F84A.

Capítulo 2

ARQUITECTURA DEL PIC16F84A

Para comenzar el uso práctico de cualquier microcontrolador, en primer lugar debemos conocer su arquitectura y sus características particulares. En este capítulo desarrollaremos a fondo la estructura interna y externa del microcontrolador PIC16F84A: su memoria, sus puertos, sus osciladores, etcétera.

Capítulo 3

LENGUAJE ENSAMBLADOR

Los microcontroladores son sistemas programados, es decir, necesitan de un programa que gobierne sus acciones, por lo que debemos conocer el lenguaje de programación que utilizan. En este capítulo explicaremos el repertorio de las 35 instrucciones del PIC16F84A y conoceremos el funcionamiento de cada una de ellas.

Capítulo 4

EL ENTORNO DE DESARROLLO MPLAB IDE

MPLAB IDE es un software gratuito distribuido por el fabricante de ciruitos Microchip, que nos servirá para escribir nuestros programas para el microcontrolador, ensamblarlos y hasta simularlos. En esta sección aprenderemos a utilizar esta poderosa herramienta para desarrollar nuestros programas.

Capítulo 5

GRABADORES DE PIC

Después de escribir los programas para nuestro microcontrolador, necesitamos grabarlos en su memoria para que puedan funcionar en él. En este capítulo hablaremos de la grabación de microcontroladores PIC. Aprenderemos a construir un sencillo grabador y a utilizarlo.

Capítulo 6

TÉCNICAS DE PROGRAMACIÓN EN ENSAMBLADOR

Para escribir programas eficientes, no sólo debemos conocer el repertorio de instrucciones del PIC, sino que también debemos tener presentes algunas técnicas de programación que nos permitirán escribir programas poderosos. En este capítulo estudiaremos técnicas que nos serán de suma utilidad.

Capítulo 7

DISPLAYS LED Y LCD

La comunicación entre el microcontrolador y el usuario es muy importante en los proyectos, para poder observar qué es lo que ocurre. En este apartado estudiaremos el manejo y las aplicaciones de displays de siete segmentos y pantallas LCD con el PIC16F84A.

Capítulo 8

EL TIMER 0

El PIC16F84A tiene un temporizador/contador de 8 bits, llamado Timer 0. Este capítulo lo dedicaremos al estudio del funcionamiento y las aplicaciones de este Timer, que nos serán de mucha utilidad para contar eventos o tiempo en nuestros proyectos.

Capítulo 9

OTRAS FUNCIONES DEL PIC16F84A

El PIC16F84A cuenta con varias funciones que nos pueden resultar muy útiles y que estudiaremos en este capítulo: el direccionamiento indirecto, el Watch dog timer, el modo sleep, los resistores de pull-up del puerto B, la memoria EEPROM de datos, y el uso de macros en los programas.

Capítulo 10

INTERRUPCIONES

Cuando se requiere que el microcontrolador atienda procesos de manera inmediata, podemos interrumpir el flujo del programa para que así sea, generando una interrupción. En este capítulo estudiaremos los mecanismos de interrupción del PIC16F84A.

Capítulo 11

COMUNICACIÓN EN BUS I2C

La comunicación entre el PIC y otros dispositivos tiene una gran importancia. En este capítulo final estudiaremos el uso del protocolo de comunicación serial llamado I2C.

Servicios al lector

En este último apartado encontraremos un índice temático de palabras utilizadas en este libro, para encontrar rápidamente lo que necesitamos.

Contenido exclusivo online

En el sitio web de la editorial (**www.redusers.com**) incluimos los archivos de código fuente, códigos máquina y librerías que mencionamos en este libro, listos para su descarga y utilización.

! INFORMACIÓN COMPLEMENTARIA

A lo largo de este manual encontrará una serie de recuadros que le brindarán información complementaria: curiosidades, trucos, ideas y consejos sobre los temas tratados.
Cada recuadro está identificado con uno de los siguientes iconos:

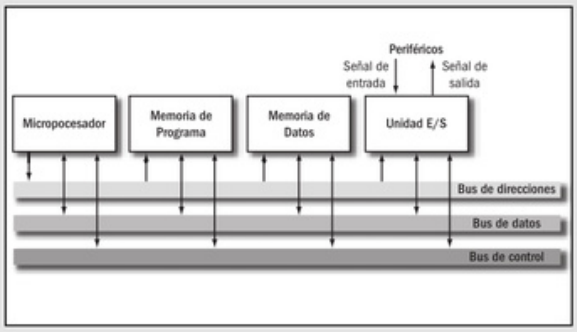

CURIOSIDADES E IDEAS


ATENCIÓN

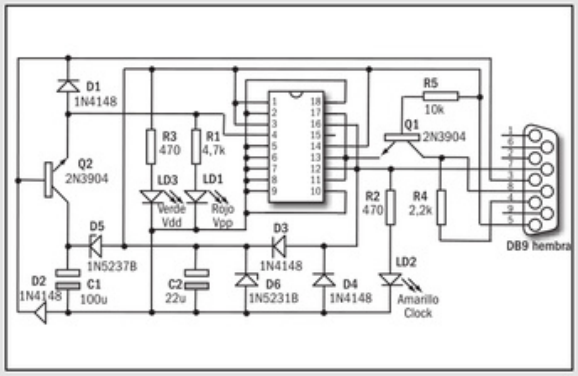

DATOS ÚTILES Y NOVEDADES


SITIOS WEB

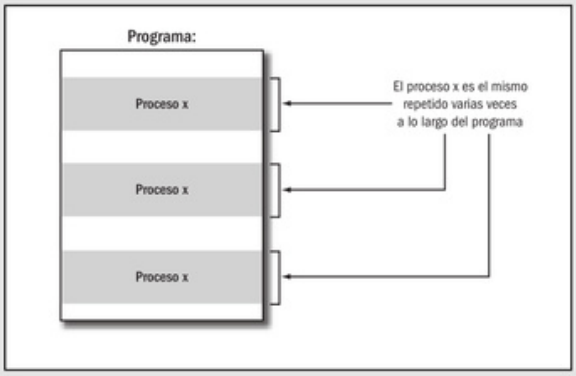
CONTENIDO



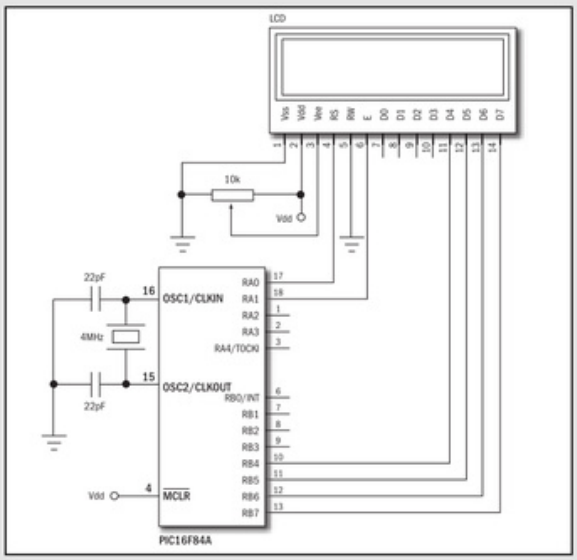
Capítulo 5



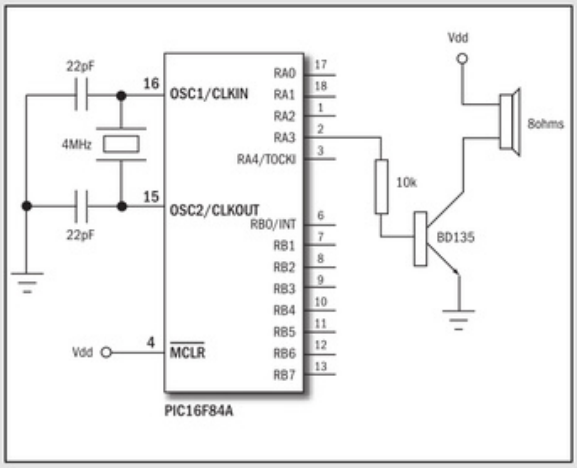
Capítulo 6



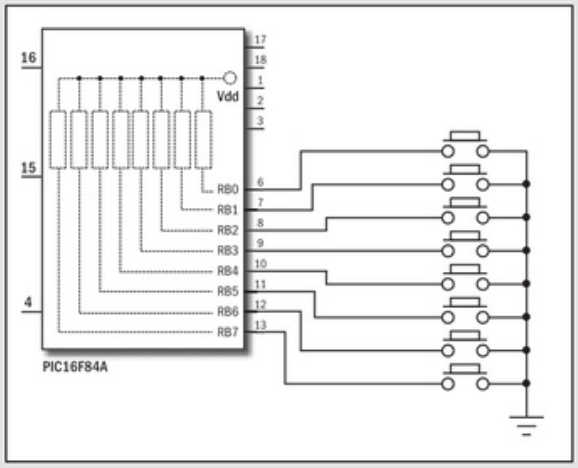
Capítulo 7



Capítulo 8

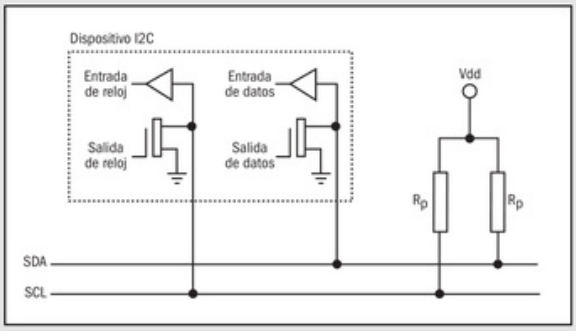


Capítulo 9



Capítulo 10

Capítulo 11



Servicios al lector

INTRODUCCIÓN

El objetivo de este libro es brindar a los lectores una introducción al manejo y a la programación de microcontroladores PIC, para el desarrollo de proyectos basados en ellos. En la actualidad, los microcontroladores tienen infinidad de aplicaciones tanto a nivel aficionado, como profesional o industrial. Por ejemplo, en un horno de microondas, quien controla todas sus funciones es precisamente un microcontrolador. En un automóvil, hay varios microcontroladores que atienden diferentes procesos, como el sistema de alarma, el sistema de inyección electrónica de combustible, e incluso procesos tan importantes como el sistema antibloqueo de frenos (ABS).

A lo largo de los capítulos aprenderemos la estructura, las funciones y la programación de microcontroladores PIC, en particular del PIC16F84A, que es uno de los microcontroladores más conocidos y utilizados. Es por eso que estudiaremos a fondo este pequeño pero poderoso dispositivo. En los primeros capítulos veremos su estructura interna y externa, desarrollaremos también la programación de microcontroladores PIC en lenguaje ensamblador y las herramientas necesarias para escribir, ensamblar y, finalmente, grabar nuestros programas en el microcontrolador, lo cual nos permitirá diseñar poderosos circuitos invirtiendo realmente poco dinero y poco tiempo. Poco a poco descubriremos cómo podemos programar y aplicar el PIC16F84A a muchos proyectos y veremos ejemplos y proyectos prácticos de aplicación de cada una de las funciones del PIC16F84A. Aprenderemos a utilizar diferentes dispositivos o periféricos junto con nuestro microcontrolador, como leds, displays, teclados, sensores, etcétera, para poder construir todo tipo de circuitos: juegos, cerraduras electrónicas, relojes, termómetros, y muchos más.

Descubriremos, también, cómo los microcontroladores, al ser dispositivos programados, permiten una gran variedad de aplicaciones y, a su vez, un gran potencial, ya que con sólo cambiar o modificar el programa que se ejecutará se pueden lograr aplicaciones diferentes, aun sin cambiar el circuito electrónico que el PIC gobernará. En resumen, a lo largo de este manual conoceremos a fondo el PIC16F84A, todas sus funciones y muchas aplicaciones prácticas diferentes que podemos darle.

Capítulo 1

Introducción a los micro-controladores

Los microcontroladores son muy utilizados hoy en día para múltiples aplicaciones, desde las más sencillas hasta las más complejas y poderosas, tanto a nivel aficionado como profesional e industrial. En este capítulo estudiaremos qué es un microcontrolador y daremos los primeros pasos para introducirnos en este interesante tema.

MICROPROCESADORES

Un **microprocesador** es un circuito integrado que contiene un circuito digital complejo, que se encarga de realizar diferentes tareas. Está diseñado para ejecutar una serie de **instrucciones** que nosotros le daremos en una lista, de acuerdo con lo que necesitemos. Esta lista se denomina **programa** y las instrucciones serán ejecutadas una a una por el microprocesador. De esta forma, al ser un sistema programado, podemos lograr que el circuito realice tareas distintas con tan sólo cambiar el programa que ejecutará.

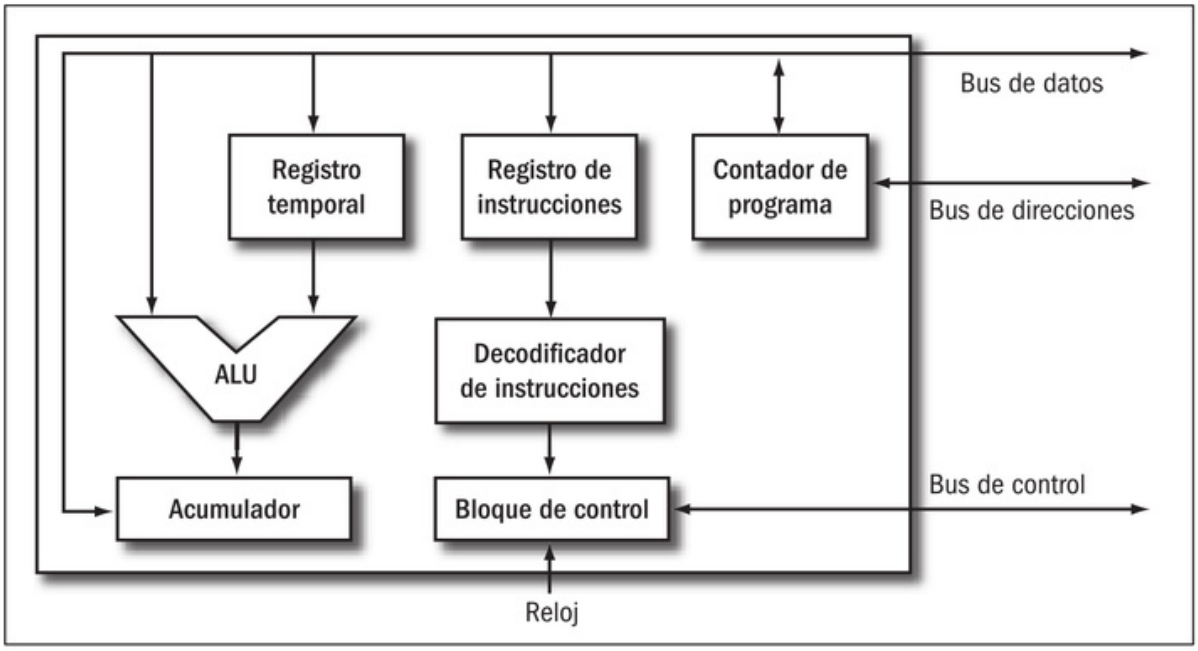


***Figura 1.** Un microprocesador es un circuito integrado digital que es capaz de realizar múltiples funciones.*

En la **Figura 1** vemos un diagrama simplificado de un microprocesador, el cual consta de un **contador de programa** (PC), que no es más que un contador binario que inicia desde cero al arrancar el sistema, y se va incrementando automáticamente. Su propósito es generar el acceso a las instrucciones que el microprocesador ejecutará. El contenido de este contador apunta a la dirección de memoria en donde están almacenadas las instrucciones del microprocesador a través del **bus de direcciones**. De esta manera, podemos ver que necesitaremos una memoria donde grabaremos las instrucciones que forman nuestro programa (llamada, precisamente, **memoria de programa**) y una memoria donde guardaremos los datos a procesar (llamada **memoria de datos**).

***Figura 2.** El microprocesador Core2quad Q9650 es uno de los más poderosos.*

El contador de programa contiene la dirección 0 (cero) al inicio. En esa dirección se almacena la primera instrucción, luego se incrementa para acceder a la dirección 1 y ejecutar la instrucción almacenada ahí, y así sucesivamente. Podemos tener dos tipos de arquitecturas dependiendo de la separación o no de la memoria de datos y de programa: la arquitectura **Harvard** y la **Von Neumman**.

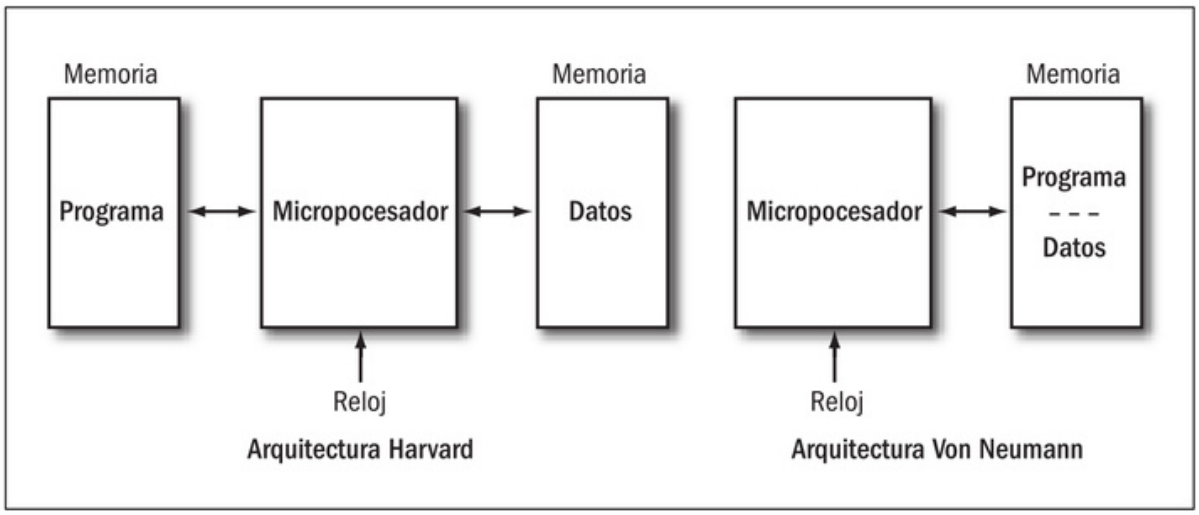


***Figura 3.** Comparación entre la arquitectura Harvard y la Von Neumman.*

El contador de programa va accediendo secuencialmente a las direcciones de la memoria de programa de donde se leerán las instrucciones almacenadas allí y pasarán al microprocesador para ser decodificadas y ejecutadas una a una. La **unidad aritmético-lógica** (**ALU**) es la encargada de llevar a cabo las operaciones necesarias, ya sean lógicas o aritméticas, tal como lo indica su nombre, con los datos.

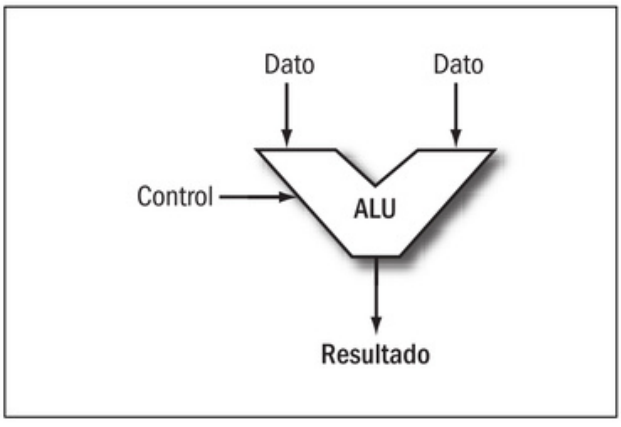


***Figura 4.** La unidad aritmético-lógica es una parte fundamental de todo microprocesador.*

A la salida de la ALU tenemos un registro especial llamado **acumulador**, que es en donde se guardarán los resultados de las operaciones. Este registro es muy importante ya que prácticamente todos los datos que maneja el microprocesador pasan por él. El bloque de control se encarga de llevar la correcta sincronía entre todos los

demás bloques, y de los elementos externos, por ejemplo, indicando a la memoria de datos si se va a leer o escribir en ella.

Como podemos darnos cuenta, el microprocesador necesita de algún **medio externo** a él para almacenar tanto las instrucciones como los datos que se están procesando, por lo que debemos agregar las memorias adecuadas para lograr que el sistema funcione. Además de las memorias, el microprocesador también puede comunicarse con otros dispositivos a través de sus buses.

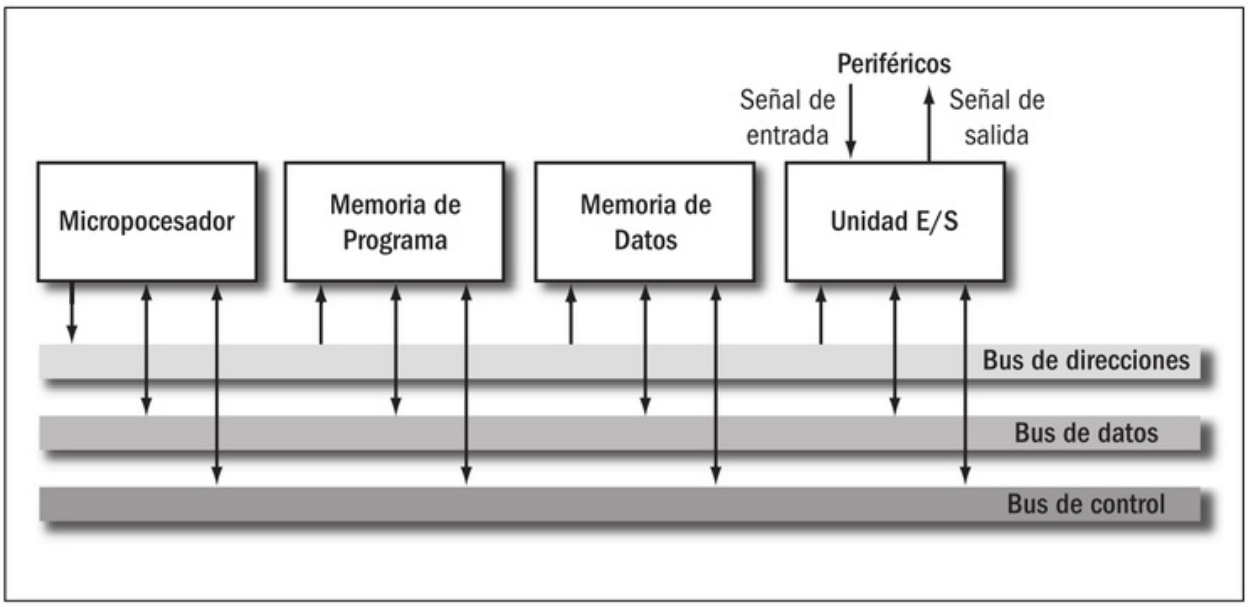


***Figura 5.** Comunicación de un microprocesador con otros dispositivos. Podemos tener más de una unidad de entrada/salida.*

De esta forma, mediante las **unidades de entrada/salida** (E/S) el microprocesador puede comunicar datos hacia el exterior o recibir datos. Por ejemplo, puede enviar datos hacia un display o una impresora, o recibir datos de un teclado o un convertidor analógico a digital. Estos dispositivos con los que el microprocesador se puede comunicar a través de la unidad E/S son comúnmente llamados **periféricos**. A un sistema completo de microprocesador, memorias y unidades de E/S se lo llama usualmente **microcomputadora**.

MILLONES DE TRANSISTORES

Los microprocesadores son circuitos integrados digitales complejos. Desde su nacimiento, allá por el año 1971, se han tornado cada vez más y más complejos y poderosos. Por ejemplo, los microprocesadores modernos que se usan en las computadoras pueden contener más de 300 millones de transistores.

Figura 6. *Una computadora es una de las aplicaciones clásicas de los microprocesadores.*

Para que el microprocesador realice las tareas que necesitamos, debemos darle la lista de instrucciones o programa, que lo grabaremos o guardaremos en la memoria de programa para que el sistema lo ejecute desde ahí. Las instrucciones son valores **binarios** que harán que se ejecute algún proceso específico en el microprocesador.

De esta forma, tenemos un sistema muy flexible y poderoso, ya que podemos elegir la cantidad de memoria de datos y de programa que necesitemos, además de cuáles y cuántos periféricos necesitamos integrar a nuestro sistema. Pero a su vez, es un sistema complejo, caro, y difícil de implementar, dado que tendremos varios **circuitos integrados** independientes que debemos interconectar (el microprocesador, las memorias y las unidades de E/S necesarias), haciéndolo difícil y lento de construir, por lo que podemos fácilmente cometer errores. El uso de varios circuitos integrados eleva el costo final y el tamaño del sistema. Es por eso que un sistema de microprocesador no es lo más conveniente siempre, sobre todo en aplicaciones sencillas o de propósito específico.

NACE UN GIGANTE

Los microprocesadores nacen en 1971 con el lanzamiento del modelo 4004 de Intel, un microprocesador de tan sólo 4 bits, que fue utilizado en calculadoras. Los microprocesadores son uno de los mayores desarrollos de las últimas décadas en la electrónica moderna.

LOS MICROCONTROLADORES

Para resolver el problema de la complejidad y el alto costo de los sistemas basados en microprocesadores, se crean los **microcontroladores**, que no es otra cosa que un sistema de microcomputadora completo. Es decir, un microcontrolador contiene en **un solo circuito integrado** el microprocesador, la memoria de datos, la memoria de programa y las unidades de entrada/salida, lo cual lo hace muy pequeño, barato y fácil de manejar, por lo que es ideal para muchas aplicaciones de propósito específico.

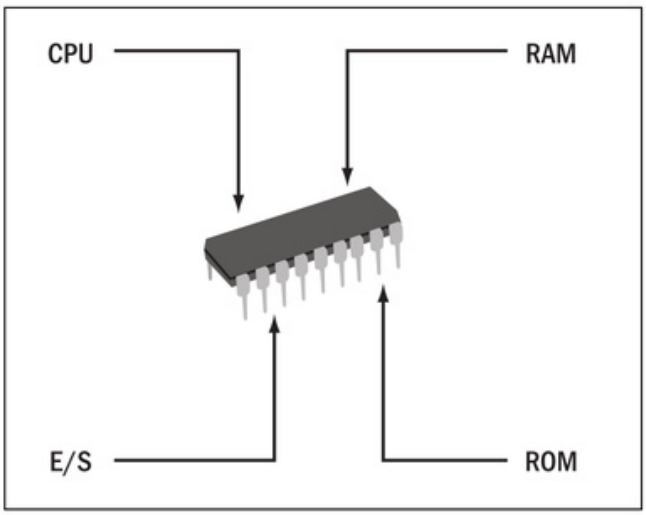


***Figura 7.** Un microcontrolador encierra todos los elementos de una microcomputadora en un solo circuito integrado.*

Los microcontroladores son cada vez más utilizados en muchos campos debido a sus ventajas y a la facilidad de diseñar circuitos con ellos, desde los más sencillos, hasta las aplicaciones más complejas. Desde aficionados hasta profesionales, incluso a nivel industrial, cada vez son más los sistemas que son gobernados por uno o varios microcontroladores. Por ejemplo, un horno de microondas, una lavadora, un juguete, los sistemas computarizados de los automóviles, los sistemas de alarma, etcétera, utilizan microcontroladores en sus circuitos. Actualmente, puede resultar más fácil, rápido y hasta más barato implementar un circuito electrónico con un microcontrolador que hacerlo con componentes discretos (compuertas,

III SISTEMAS ABIERTOS Y CERRADOS

Las computadoras son sistemas basados en microprocesadores, y son llamados **sistemas abiertos**, debido a la flexibilidad y posibilidad de elegir o cambiar sus componentes (el propio procesador, la memoria, los periféricos, etcétera), mientras que un microcontrolador es un **sistema cerrado**.

multivibradores, contadores, registros, y demás). Es por eso que el aprendizaje del manejo de microcontroladores cada vez toma mayor importancia para los aficionados y profesionales de la electrónica.

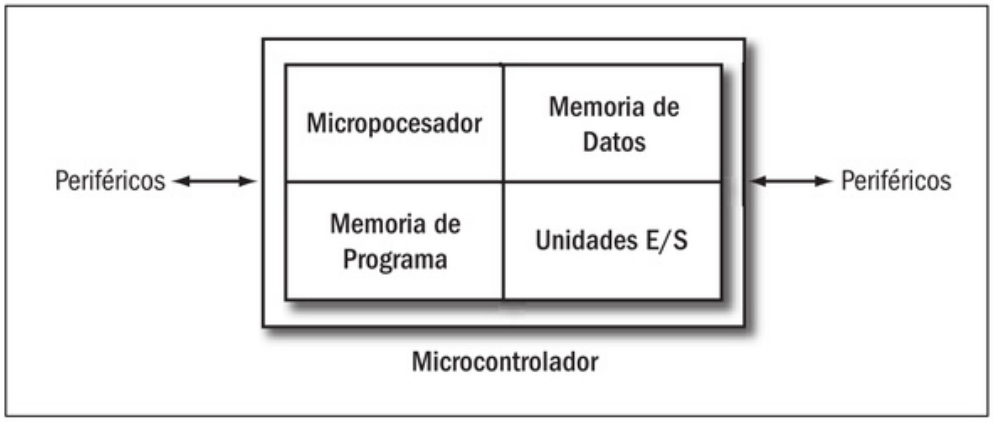


***Figura 8.** Un sistema basado en microcontrolador es más pequeño, sencillo y económico.*

Existe una gran gama de microcontroladores de muchos fabricantes con características y prestaciones muy variadas. En la actualidad, los fabricantes más importantes son: **Microchip**, **Freescale** (Motorola), **Intel**, **Atmel**. Nosotros elegiremos en esta ocasión los **microcontroladores PIC** de la firma **Microchip** para estudiar su estructura, programación y utilización práctica, ya que son ampliamente usados actualmente debido a su facilidad de uso, bajo costo, su gran oferta con una amplia gama de dispositivos, y a la facilidad de encontrar herramientas, tanto de software como de hardware, para el desarrollo de proyectos con estos versátiles microcontroladores.

Los microcontroladores PIC

Los microcontroladores PIC ofrecen una amplia gama de dispositivos desde 6 hasta 100 pines, desde los más sencillos de **8 bits** hasta los más poderosos de **32 bits**. La gama de microcontroladores de 8 bits consta de más de 400 modelos diferentes para elegir. Microchip es actualmente el mayor fabricante de microcontroladores de 8 bits en todo el mundo. Sus familias de 8 bits más importantes son: **PIC12**, **PIC16** y **PIC18**. En particular, para nuestro estudio de introducción a los microcontroladores PIC, elegiremos el **PIC16F84A**, que es de gama media, muy versátil, económico y fácil de usar. Este PIC es ideal para aprender las bases tanto del uso como de la programación de microcontroladores PIC. Además, una vez que conozcamos bien a fondo este dispositivo, la migración hacia cualquier otro

***Figura 9.** El PIC16F84A es un simple circuito integrado de 18 pines, pero encierra un dispositivo muy poderoso.*

microcontrolador PIC será bastante fácil. Incluso la compatibilidad entre los microcontroladores PIC hace que los programas puedan migrar de uno hacia otro con muy pocos cambios en ellos. Es por eso que el PIC16F84A nos servirá de plataforma para entrar en el mundo de los microcontroladores PIC.

HERRAMIENTAS NECESARIAS

Para el desarrollo de nuestros proyectos con microcontroladores PIC, necesitaremos algunas herramientas tanto de software como de hardware. Trataremos de que dichas herramientas sean fáciles de encontrar y de que sean gratuitas o de bajo costo, al menos la mayor parte de ellas. A lo largo de este libro veremos que realmente no necesitamos invertir una fortuna para poder construir poderosos circuitos electrónicos con microcontroladores PIC.

Componentes

Por supuesto que para construir nuestros proyectos necesitaremos los propios **componentes**. En primer lugar, el microcontrolador, por lo que es recomendable comprar al menos uno de ellos y, si podemos, uno más ya que nunca se sabe cuándo cometeremos un error. Además, debemos adquirir otros componentes que estudiaremos a lo largo del libro, como por ejemplo, **displays**, **teclados**, **leds**, algunos **circuitos integrados**, etcétera, los cuales iremos detallando en el momento en que sea necesario.

Editores y ensambladores

Sabemos que el microcontrolador es un dispositivo que funciona a través de un programa que escribiremos nosotros, por lo que debemos tener herramientas de software para poder escribir nuestros programas y **ensamblarlos** (más adelante estudiaremos qué es esto de ensamblar un programa), para luego poder grabarlos en la memoria de nuestro PIC. En este caso utilizaremos el entorno de desarrollo

III OTROS PIC C

Generalmente, la **C** en los microcontroladores PIC indicaba una memoria **EPROM**, con excepción del PIC16C84 que contaba con memoria **EEPROM**. Los modelos C han sido reemplazados casi por completo por modelos **F** con memoria Flash. La mayoría de los dispositivos C sólo podían ser grabados una sola vez.

llamado **MPLAB IDE**, que es distribuido gratuitamente por el propio fabricante de los microcontroladores PIC, Microchip. Este entorno de desarrollo contiene todas las herramientas de software necesarias para poder escribir nuestros programas. En el **Capítulo 4** estudiaremos con detalle el uso de este software.

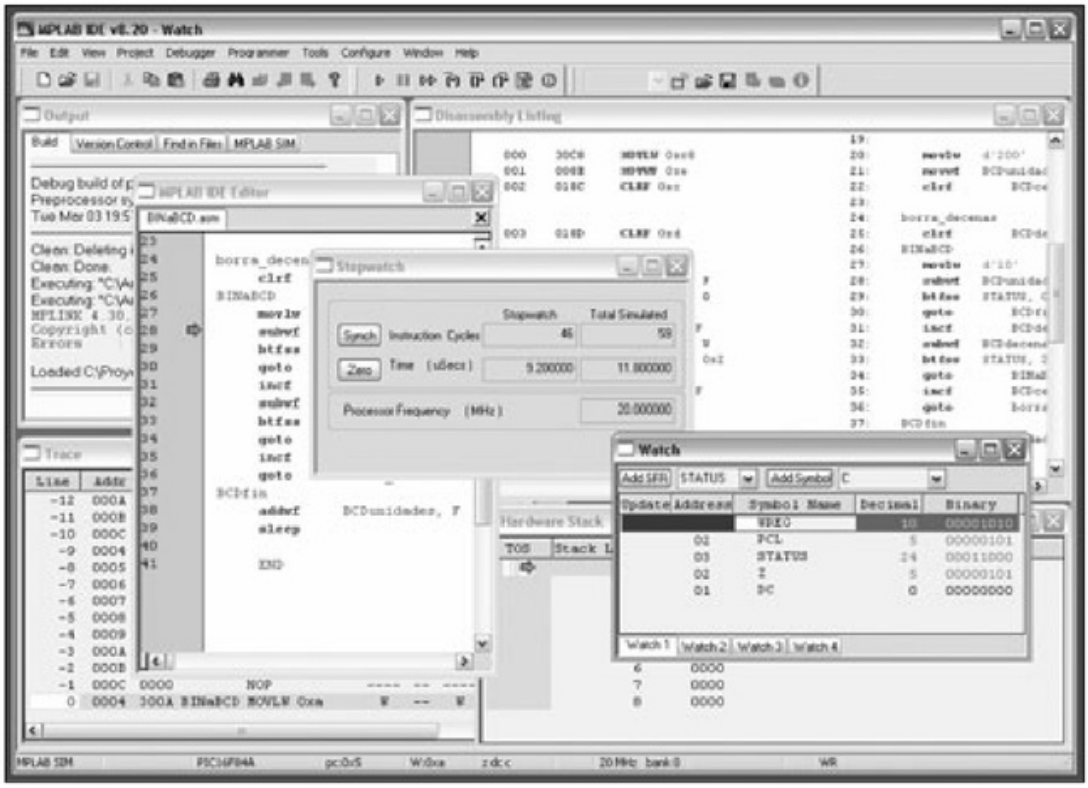


***Figura 10.** El programa MPLAB de la empresa Microchip contiene las herramientas de desarrollo de programas de forma gratuita.*

Simuladores

Existe software para la PC que permite **simular** el funcionamiento de los programas que estamos escribiendo para nuestro microcontrolador. Éstos pueden ser muy útiles para el desarrollo, la depuración y la corrección de errores, así evitamos tener que grabar varias veces nuestro microcontrolador para probar si los programas funcionan como deben, lo cual nos ahorrará tiempo y esfuerzo. El programa MPLAB contiene un simulador integrado llamado **MPLAB SIM**, que nos será de mucha utilidad, y que también estudiaremos en el **Capítulo 4**. Además de MPLAB SIM, existen otros simuladores, como **PIC Simulator IDE** o el simulador de circuitos **Proteus VSM**, que es uno de los mejores simuladores de circuitos electrónicos en la actualidad, ya que permite la simulación de cualquier tipo de circuito electrónico con animación y análisis de señales, incluyendo, por supuesto, circuitos con microcontroladores PIC. Los dos últimos simuladores que mencionamos tienen licencias pagas, por lo que quedará en nosotros la decisión de comprarlos para usarlos como herramientas de desarrollo. Por ahora nos bastará con estudiar MPLAB SIM, que es gratuito.

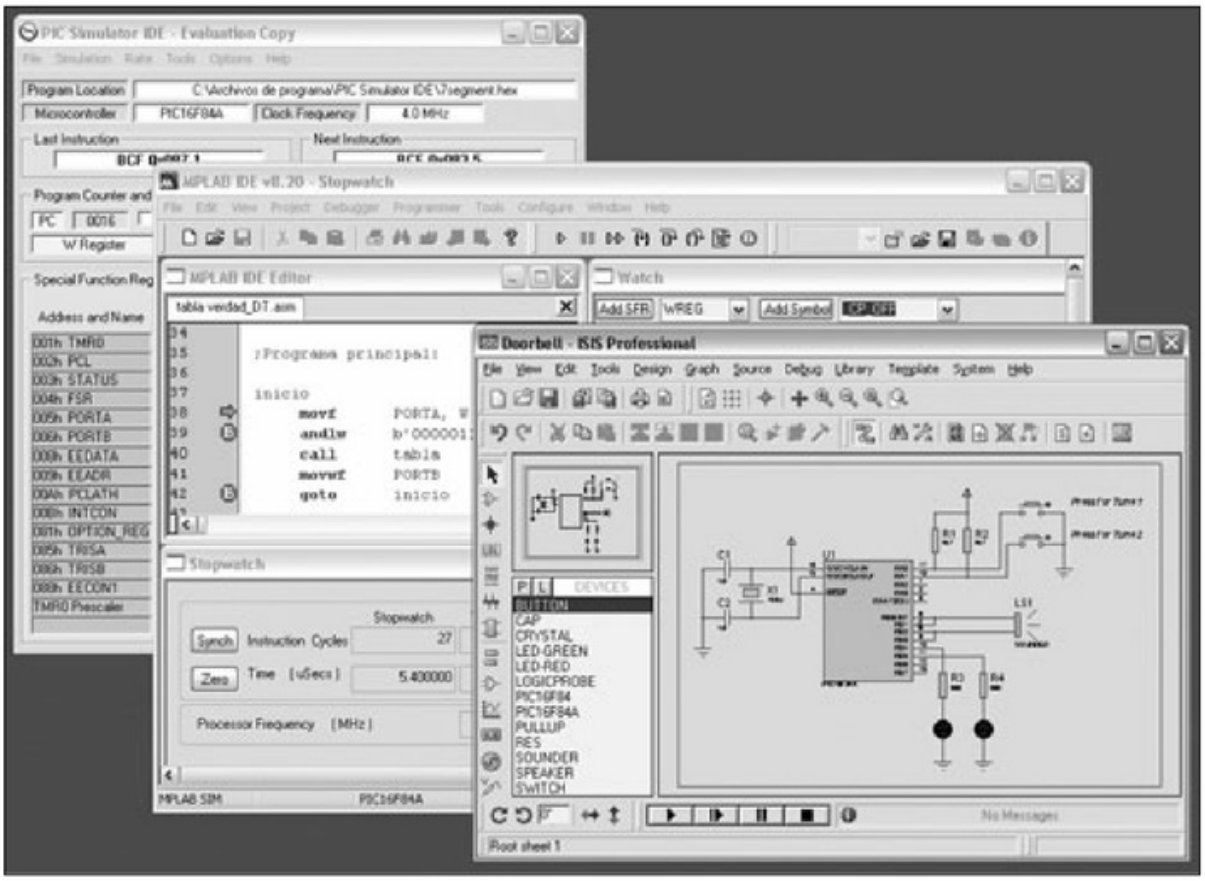


Figura 11. *Existe una gran gama de software de simuladores para verificar el funcionamiento de los programas.*

Grabadores

Después de desarrollar los programas debemos grabarlos de alguna forma en la memoria de nuestro PIC para que el sistema quede completo. Para esto tenemos que contar con un **grabador**, que no es más que un circuito que se conecta a la PC por medio de algún puerto, ya sea el puerto paralelo, serial, o USB, para poder enviar el programa desde la PC hacia la memoria de nuestro PIC. Existen multitud de grabadores para microcontroladores PIC; Microchip ofrece algunos que aseguran un perfecto funcionamiento y están listos para usar. La desventaja es que algunos pueden ser caros y difíciles de encontrar en las tiendas locales.

III EVOLUCIÓN DEL PIC16X84

El microcontrolador que estamos estudiando es ya un veterano en el mercado: nació originalmente como PIC16C84. La C indica una memoria de programa de tipo **EEPROM**, mientras que el PIC16F84, que fue su predecesor, tiene memoria de programa tipo **Flash**. Después viene el PIC16F84A, donde la A indica que es una versión mejorada.

Figura 12. *Mediante un grabador se puede leer o escribir en la memoria de programa del PIC.*

Por otra parte, están los grabadores que podemos adquirir en algunas tiendas del ramo, con diferentes tipos de conexión a la PC, y algunos pueden ser baratos y también están armados y listos para usar. La alternativa más económica, aunque puede que resulte la más difícil, es armar nuestro propio grabador de PICs. Existe suficiente información para poder hacerlo, sólo necesitamos unos cuantos componentes, un cable para la conexión a la PC y un poco de paciencia. En el **Capítulo 5** veremos cuáles son los tipos de grabadores y propondremos un circuito para la construcción de un versátil y económico grabador, además de estudiar su uso por completo.

El grabador que propondremos para su construcción se conectará a nuestra computadora a través del puerto serial, y no será necesario construirlo si ya contamos con algún grabador de microcontroladores PIC. Sólo lo proponemos para quienes todavía no tengan uno.

Éstas son las herramientas básicas que necesitaremos para desarrollar nuestros proyectos con microcontroladores PIC. Existen otras, como por ejemplo los **compiladores**, que permiten escribir programas en un lenguaje de alto nivel, como Basic o C, y los sistemas de desarrollo o **emuladores**, pero estas herramientas pueden ser complejas de usar para los principiantes, además de ser caras. Al menos, para usos sencillos, no las necesitaremos.

{ } PIC16F84 Y PIC16F84A

La **A** al final en el PIC16F84A indica que es una versión modernizada con respecto al PIC16F84, ya que tiene una memoria Flash mejorada, además de contar con la posibilidad de trabajar a una frecuencia mayor: 20 MHz para el PIC16F84A-20 contra 10 MHz máximos que puede funcionar el PIC16F84.

EL PIC16F84A

Comencemos a conocer la estructura de nuestro microcontrolador, que no es más que un **circuito integrado de 18 pines**. En la **Figura 13** vemos un diagrama completo con la disposición de los pines y el nombre que toma cada uno. Las flechas indican si el pin es de salida, de entrada o de entrada/salida.

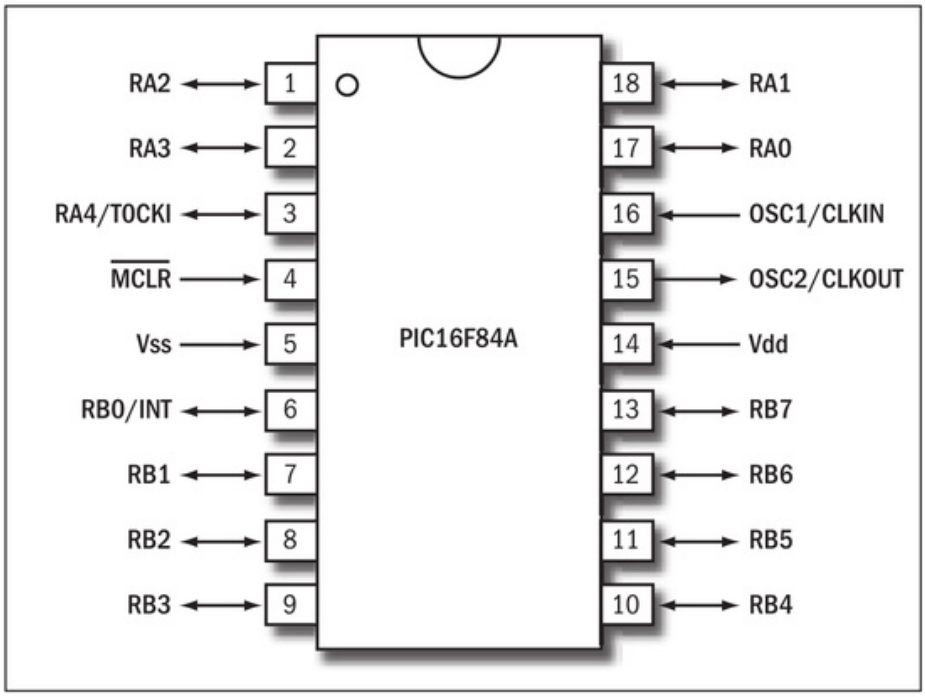


***Figura 13.** Nombre que reciben los pines del PIC16F84A. Algunos tienen más de una función.*

Veamos a grandes rasgos cuál es la función de cada uno de los pines. Por ahora no es necesario que los entendamos, ya que a lo largo de nuestro estudio hablaremos con detalle de cada función:

- **RA0:RA4**: pertenecen al **puerto A** y cada uno de estos pines puede utilizarse como entrada o salida de datos, y pueden configurarse independientemente como entrada o como salida, según necesitemos. El pin RA4 está marcado también

III HOJAS DE DATOS

Como todo componente electrónico, el PIC16F84A tiene una **hoja de datos** (**datasheet**), que proporciona la empresa Microchip, con la descripción completa de su arquitectura y su funcionamiento. Si vamos a la página oficial del fabricante (**www.microchip.com**) podremos descargarla, aunque se encuentra solamente en inglés.

como **T0CKI**, lo que significa que además cumple otra función que es la de servir de entrada a una señal para el **Timer 0**. En las secciones donde hablaremos del timer detallaremos esta función.

- $\overline{\text{MCLR}}$ (*Master clear*): este pin es el de **reset**. Es decir que si ponemos un nivel bajo en él, el microcontrolador irá al estado de reset y si hay un nivel alto, el microcontrolador funcionará normalmente.
- **Vss**, **Vdd**: estos pines pertenecen al voltaje de alimentación de nuestro PIC. En Vdd debemos poner un voltaje de 5 V y si es un voltaje regulado, mejor. En Vss debemos poner tierra (masa).
- **OSC1/CLKIN**, **OSC2/CLKOUT**: estos pines son la entrada y salida de la **señal de reloj** necesaria para que el sistema funcione correctamente. Más adelante hablaremos en detalle de este tema.
- **RB0:RB7**: pertenecen al **puerto B** y la función es la misma que la de los pines del puerto A. El pin RB0 además cumple con la función de **entrada de interrupción externa** (**INT**), que también detallaremos en su momento.

Alimentación del PIC16F84A

Como acabamos de mencionar, este PIC requiere de un voltaje de alimentación de **5 V** entre las terminales Vdd y Vss, así que debemos tener una fuente de 5 V para poder trabajar con los circuitos que armemos.

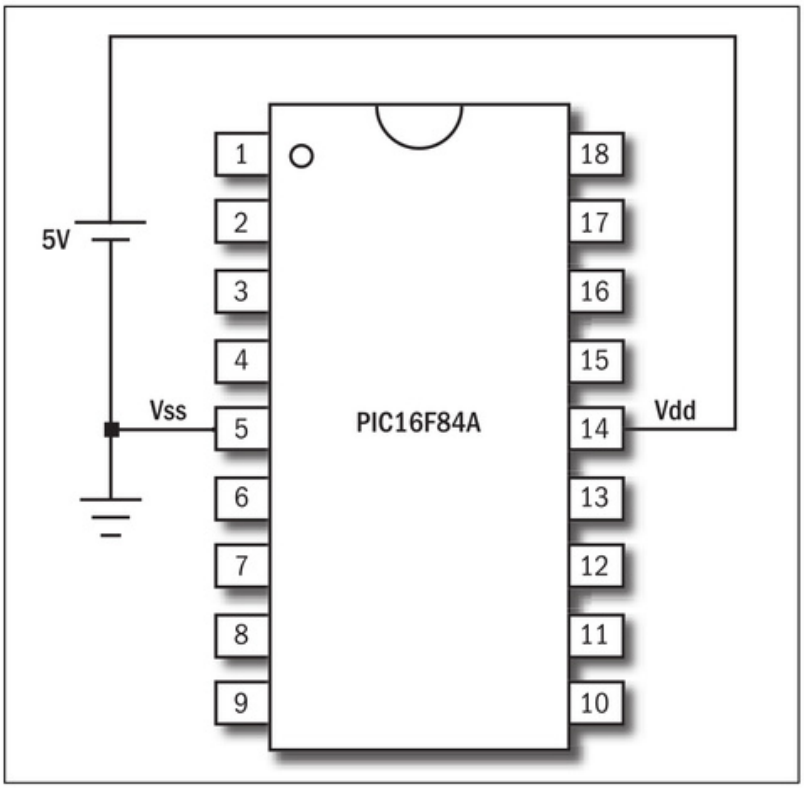


***Figura 14.** El PIC16F8A requiere de un voltaje de 5 V entre las terminales Vdd (positivo) y Vss (tierra).*

Es recomendable tener un voltaje regulado, entonces tenemos varias opciones:

- Si ya contamos con alguna fuente de 5 V regulados y que pueda entregar al menos 1A de corriente, podemos utilizarla.
- Si contamos con alguna fuente no regulada que entregue entre 7 y 12 V de potencia y al menos 1A a su salida, podemos agregar el regulador para obtener el voltaje de 5 V, como vemos en el diagrama de la **Figura 15**. No obstante ello, debemos tener en cuenta que tal vez sea necesario agregar un **disipador de calor** al regulador para evitar que se dañe por sobrecalentamiento (pero esto sólo si vamos a utilizar más de 500 mA de corriente).

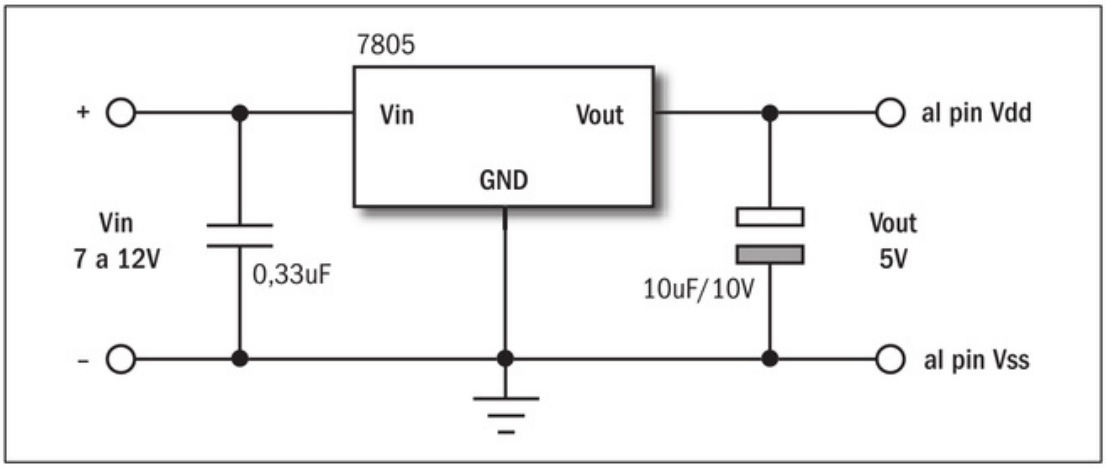


***Figura 15.** El LM7805 nos proporcionará una salida de 5 V regulados al colocarlo en una fuente no regulada.*

- Si no contamos con ninguna fuente de alimentación, podemos construirla siguiendo el diagrama de la **Figura 16**. De todos modos, más allá de que podamos armar nuestra propia fuente de alimentación, siempre es recomendable colocar un disipador de calor al regulador.

Es recomendable también, en cualquier caso, colocar un **capacitor de desacoplo** de 100 nf (0.1 uf) entre los pines Vss y Vdd del microcontrolador, lo más cercano posible a ellos para asegurar un correcto funcionamiento.

III SISTEMAS DE DESARROLLO

Un sistema de desarrollo es un conjunto de herramientas de software y hardware que permiten el rápido desarrollo de proyectos con PICs. Consiste en una placa con múltiples periféricos listos para funcionar, como teclados, displays, sensores, etcétera, e incluso permiten grabar el PIC en ellos. La desventaja es que el precio es generalmente alto.

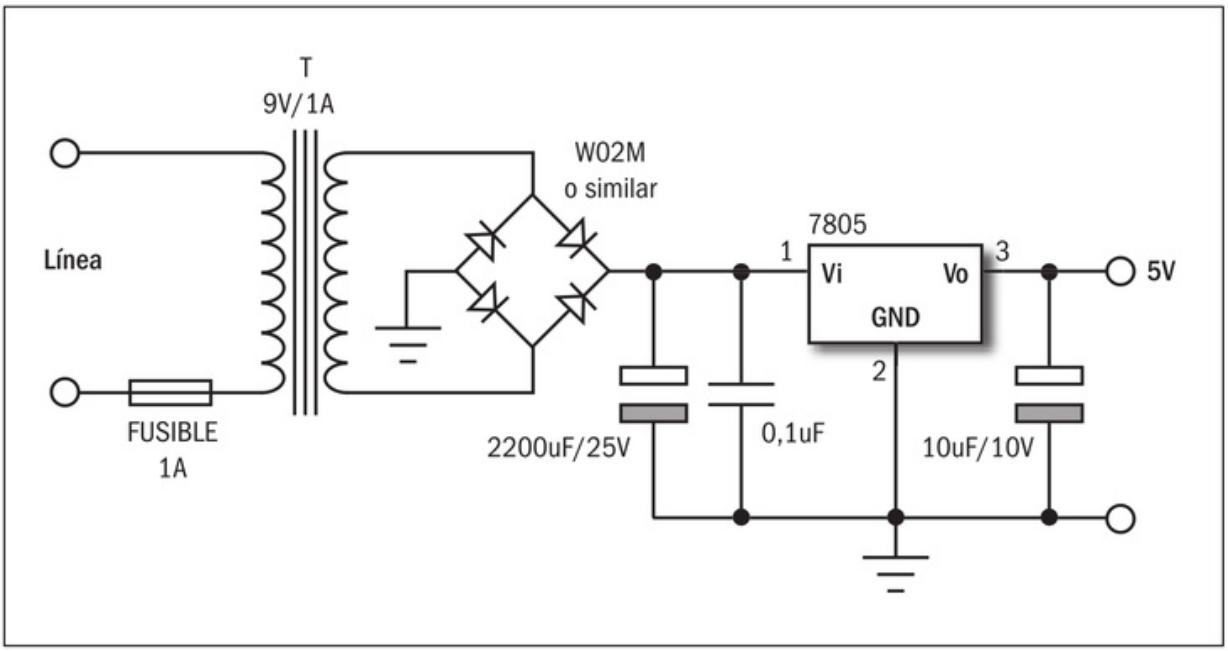


Figura 16. *Podemos construir esta fuente de alimentación para nuestros circuitos con el PIC16F84A.*

En el capítulo siguiente continuaremos estudiando la estructura interna del PIC16F84A para comprender su funcionamiento y posteriormente poder aprender a escribir programas en lenguaje ensamblador, y así tener la posibilidad de comenzar a diseñar y construir circuitos electrónicos con microcontroladores PIC.

RESUMEN

En este primer capítulo comprendimos qué es un microprocesador, cómo funciona y para qué sirve. También vimos qué es un microcontrolador y la diferencia con un microprocesador. Además, conocimos cuáles son las herramientas que necesitaremos para el desarrollo de proyectos con microcontroladores PIC, y también comenzamos a estudiar el PIC16F84A, del cual hablaremos a lo largo de los capítulos siguientes.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un microprocesador?

2 ¿Qué es el contador de programa?

3 ¿Para qué sirve una ALU?

4 ¿Cómo se llama la serie de instrucciones que ejecuta un microprocesador?

5 ¿Cuál es la diferencia entre la arquitectura Harvard y la Von Neumann?

6 ¿Qué es una microcomputadora?

7 ¿Qué es un microcontrolador?

8 ¿Cuántos pines tiene el PIC16F84A?

9 ¿Para qué sirven el Puerto A y el Puerto B del PIC16F84A?

10 ¿Cuál es el voltaje con el que se debe alimentar el PIC16F84A?

Capítulo 2

Arquitectura del PIC16F84A

Para poder programar un PIC16F84A, en primer lugar debemos conocer su estructura interna. Por eso, en este capítulo estudiaremos la estructura de nuestro microcontrolador y algunas funciones básicas, para luego enfocarnos en la programación en lenguaje ensamblador en el siguiente capítulo.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

DIAGRAMA INTERNO

En la **Figura 1** tenemos el diagrama interno del PIC16F84A, donde se muestran los diferentes bloques que lo forman. Es posible que a primera vista nos parezca confuso de interpretar ahora, pero poco a poco veremos en detalle qué es y cómo funciona cada bloque, conforme avancemos con la explicación.

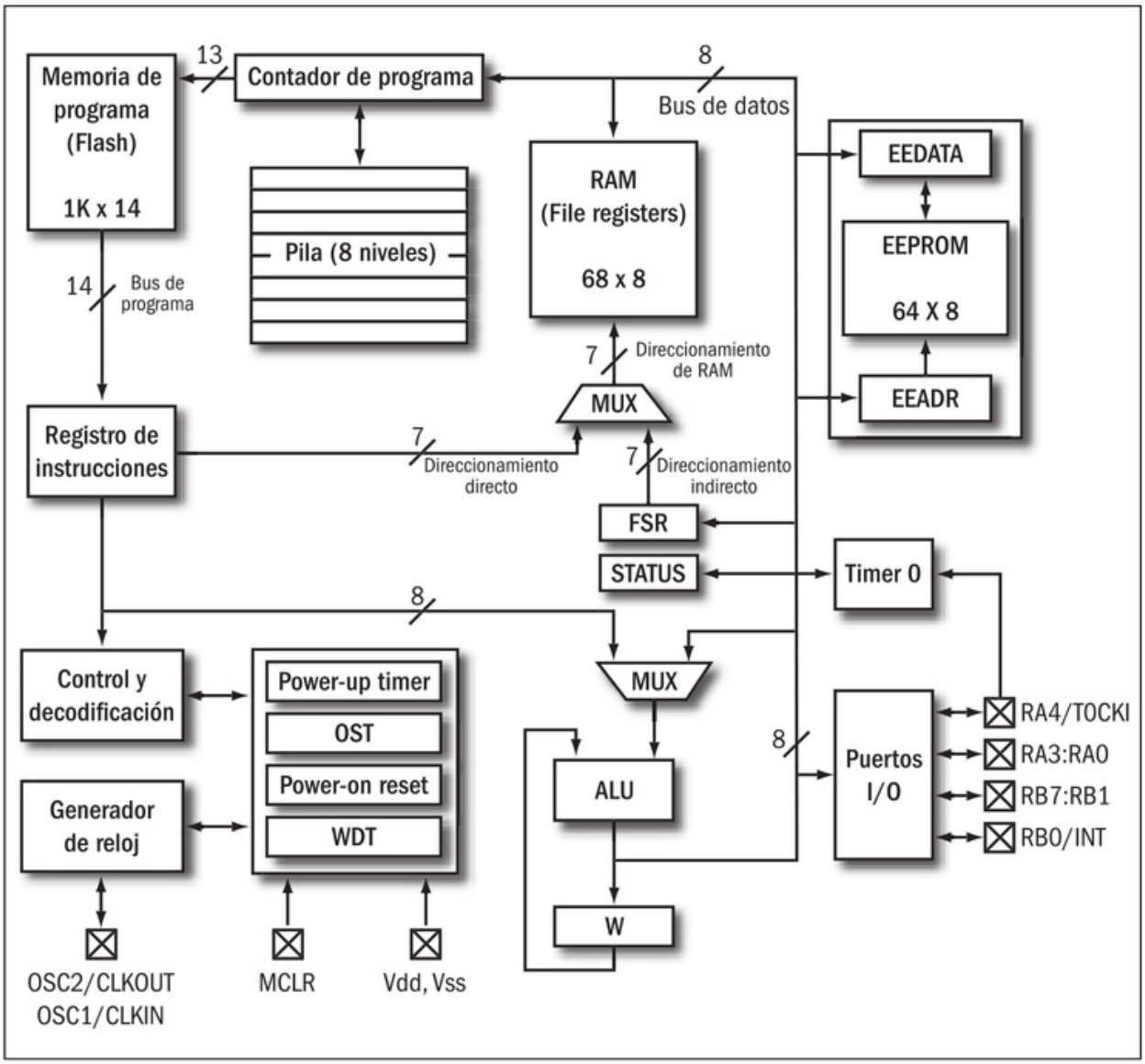


***Figura 1.** Es importante conocer el PIC16F84A internamente para poder entender su funcionamiento.*

Lo primero que podemos observar en el diagrama es que el PIC16F84A tiene una arquitectura Harvard, ya que la memoria de datos y la de programa están separadas. La memoria de programa es de tipo **Flash**, es una memoria no volátil de **1k x 14**, lo que significa un total de 1024 espacios de memoria de 14 bits cada uno. Por otra parte, la memoria de datos es una memoria **RAM estática** formada por registros de 8 bits cada uno con un total de 68 de ellos.

EL OSCILADOR

Como todo circuito digital, el PIC16F84A necesita una **señal de reloj**, la cual sincroniza y determina la velocidad de los procesos. Esta es una parte muy importante. Tenemos cinco formas de obtener o generar dicha señal de reloj, que resumimos en la **Tabla 1**. Los pines **OSC1/CLKIN** y **OSC/CLKOUT** son los encargados de generar o recibir en cada caso la señal.

MODO	DESCRIPCIÓN
XT	Cristal de cuarzo/resonador cerámico
HS	Cristal/resonador de alta velocidad
LP	Cristal de baja potencia
RC	Red RC (resistor y capacitor)
Externa	Señal de reloj externa

***Tabla 1.** Distintos modos de señal de reloj.*

En los primeros cuatro modos, la señal de reloj (**clock**) es generada por el propio microcontrolador mediante la conexión de algunos componentes externos. El más común es el modo **XT**, en el cual se conecta un cristal de cuarzo de la frecuencia deseada para que el microcontrolador genere la señal de reloj en los pines 15 y 16 (OSC/CLKOUT y OSC/CLKIN).

***Figura 2.** Lo típico para un PIC16F84A es un cristal de cuarzo de 4 MHz para generar la señal de reloj.*

Existen dos versiones del microcontrolador según la frecuencia máxima a la que puede trabajar: el **PIC16F84A-04**, con una frecuencia máxima de 4 MHz, y el **PIC16F84A-20**, con una frecuencia de trabajo máxima de 20 MHz. Lo más común es usar el microcontrolador PIC16F84A-04, ya que es más fácil de encontrar, más económico, y con 4 MHz es suficiente para la gran mayoría de los proyectos, por lo tanto, de ahora en adelante usaremos la configuración del oscilador de 4 MHz.

III PALABRAS EN INGLÉS

Algunos nombres de elementos o funciones que estamos estudiando, como **Reset** o **Timer**, se utilizan en inglés, ya que no tienen una traducción exacta al español, o porque se acostumbra usarlos así, tal como están en la hoja de datos. Es sólo cuestión de acostumbrarnos a estos términos.

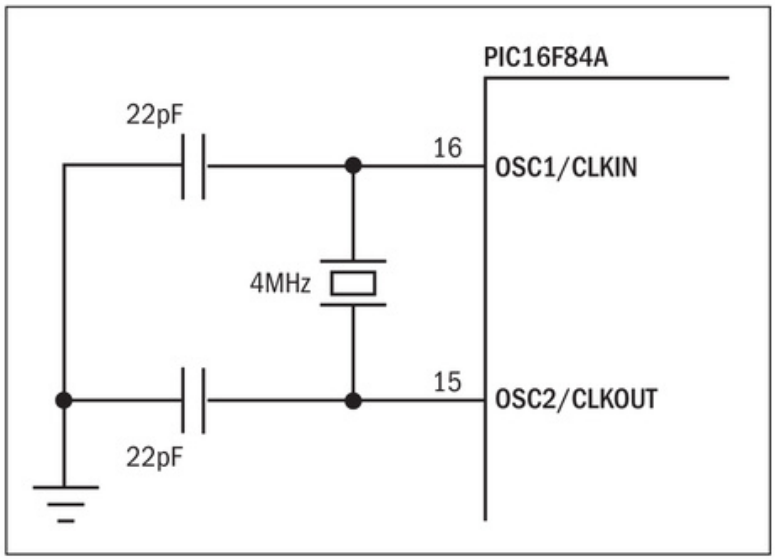


Figura 3. *Configuración típica para un oscilador con cristal de cuarzo de 4 MHz.*

La configuración **RC** puede usarse para frecuencias bajas y que no requieran de mucha exactitud en la frecuencia de los pulsos de reloj, siendo más económico al tener que conectar sólo un resistor y un capacitor. La frecuencia en este modo es inestable, y depende de los valores de R, C, y del voltaje de alimentación. R debe estar entre 5 KOhms y 100 KOhms, y C debe ser mayor de 20 pf. Para determinar la frecuencia de oscilación según los valores de R, C y Vdd, debemos consultar los gráficos de la hoja de datos del microcontrolador en caso de que necesitemos usar este modo.

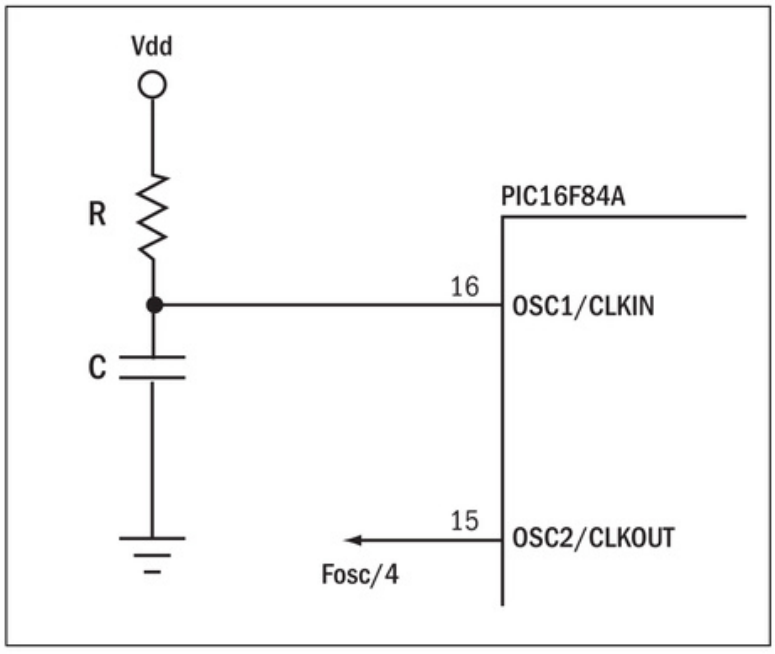


Figura 4. *Configuración para un oscilador RC.*

Como el oscilador RC se conecta a un solo pin (OSC/CLKIN), el pin OCS/CLKOUT queda libre y en él tendremos una señal de Fosc/4, es decir, la frecuencia del oscilador dividida entre cuatro.

También podemos obtener una señal de reloj independiente generada por algún otro circuito y llevarla al **pin 16** de nuestro PIC. En este modo tendremos una **señal de reloj externa** que puede controlar más de un circuito o más de un microcontrolador, aunque no es muy común utilizar este modo.

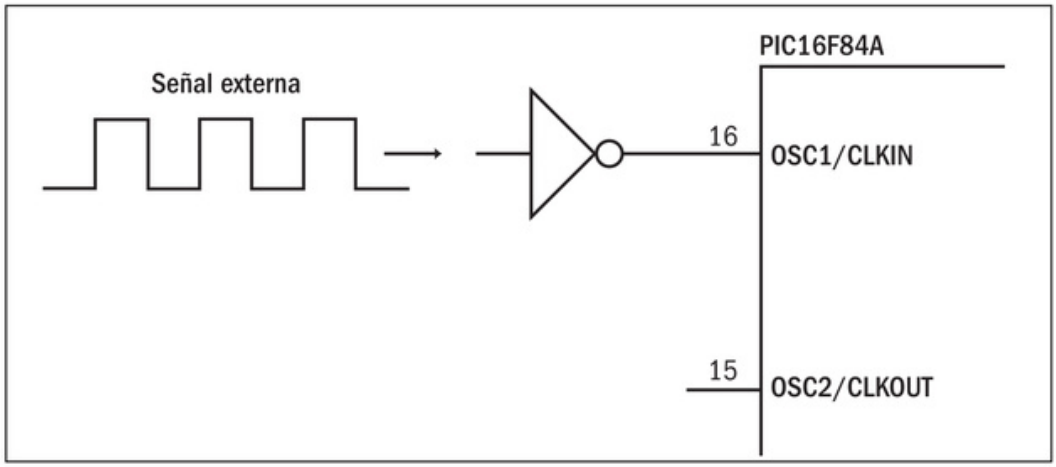


Figura 5. *Se puede aplicar una señal de reloj externa al PIC16F84A, pero no es muy usual hacerlo así.*

EL RESET

Un **Reset** es cuando el microcontrolador es detenido y forzado a iniciar su funcionamiento desde el principio. Es decir, cuando el funcionamiento del programa comienza desde su inicio. Existen varios métodos de provocar un Reset a nuestro PIC:

- Power-on reset (**POR**).
- Reset externo a través del pin Master clear ($\overline{\text{MCLR}}$)
- Por el temporizador de perro guardián (**WDT**).

El primer método que señalamos es cuando encendemos la fuente de alimentación del circuito. El llamado **Power-on reset** mantiene al microcontrolador en

OSCILADOR RC Y LA EXACTITUD

El oscilador RC no debe usarse en aplicaciones donde se necesiten tiempos exactos, o donde la sincronización de los procesos sea importante, ya que al estar formado por un capacitor y un resistor, está expuesto a variaciones de la frecuencia de oscilación por diversos factores como la temperatura, las variaciones en el voltaje de alimentación, entre otras.

estado de Reset (detenido) mientras el voltaje de alimentación alcanza un nivel adecuado para que el sistema comience a funcionar (**Figura 6**).

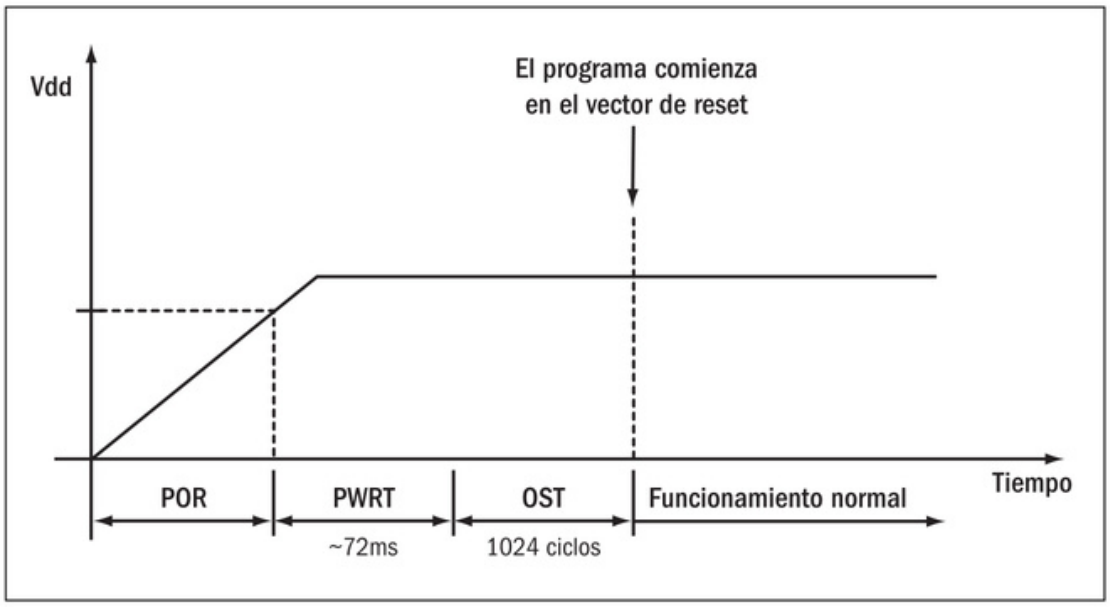


***Figura 6.** Funcionamiento de inicialización del microcontrolador al encender la fuente de alimentación.*

Después del Power-on reset podemos tener un retardo extra llamado **Power-up timer**, que es un retardo de aproximadamente 72 milisegundos con el fin de dar un poco más de tiempo a que el voltaje se estabilice y así asegurar el correcto inicio del sistema. Este temporizador está controlado por un oscilador RC interno y su tiempo es fijo. El **PWRT** podemos activarlo o no, más adelante veremos cómo hacerlo.

Después del PWRT viene el **OST** (*Oscillator Start-up timer*), el cual retarda el inicio durante 1024 ciclos de reloj. Esto es para asegurar que el oscilador arranque de forma correcta y se estabilice antes de iniciar la ejecución del programa. El OST sólo funciona cuando estamos usando un cristal, es decir, sólo en los modos XT, HS o LP. Estos retardos aseguran una correcta inicialización del sistema en el encendido. Una vez que ha pasado este tiempo, el programa inicia en el **vector de reset**, que es la dirección cero de la memoria de programa, es decir, en la primera dirección.

III MCLR = VPP

El pin Master clear tiene una función especial extra, y es la de servir de entada para el voltaje llamado **Vpp** o **voltaje de programación**, el cual se utiliza para la grabación de los datos de programa en la memoria Flash del microcontrolador. Este voltaje debe ser de entre 12 y 14 V. Cuando es aplicado este voltaje, el PIC entra en el modo de programación.

Circuito externo de reset

Existe una forma externa de reset que podemos utilizar en caso de que así lo necesitemos: a través del pin **Master clear** ($\overline{\text{MCLR}}$), cuya función es precisamente esa, la de poder provocar un reset externo. Si tenemos un nivel lógico alto en este pin, el sistema funcionará normalmente, y si llevamos este pin a un nivel lógico bajo, entonces provocará un reset del sistema. Si no vamos a necesitar un reset externo, entonces debemos conectar este pin directamente a Vdd para que siempre haya un nivel alto en él y el sistema funcione normalmente todo el tiempo.

Si queremos disponer de un reset externo podemos colocar un pulsador para llevar al estado bajo al pin Master clear, y así provocar un reset en nuestro microcontrolador cuando lo necesitemos. En la **Figura 7** tenemos la configuración recomendada para este circuito de reset.

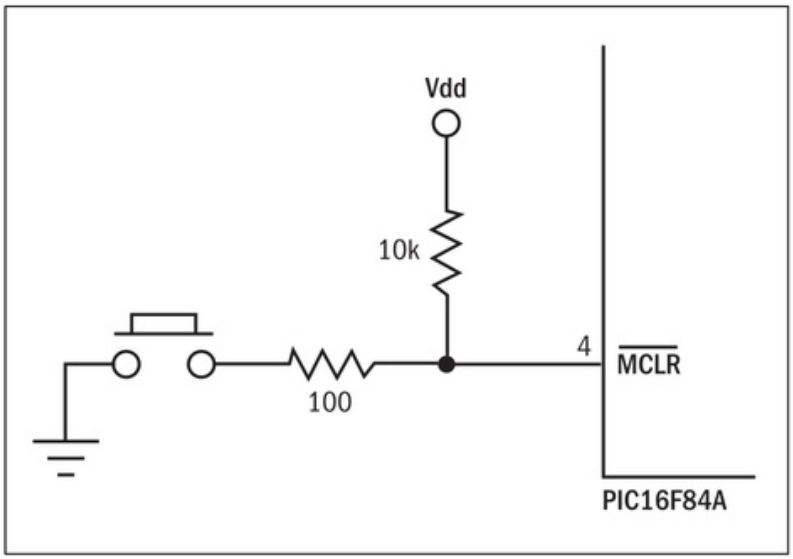


***Figura 7.** Mediante un pulsador en el pin 4 podemos provocar un reset externo en cualquier momento.*

Como podemos apreciar en el diagrama, el resistor de 10 k obliga a que en el pin 4 tengamos un estado alto, hasta que presionemos el pulsador, lo cual llevará un estado lógico bajo al estar conectado a tierra. Al soltar el pulsador habrá nuevamente

{} HS Y AMPLITUD DE OSCILACIÓN

En el modo **HS** la amplitud de la oscilación es mayor que en los otros debido a la alta frecuencia. Debemos evitar usar este modo si no estamos empleando realmente un cristal de alta frecuencia, ya que si la amplitud de las oscilaciones es muy grande podemos dañar el microcontrolador.

un estado lógico alto y el microcontrolador iniciará de nuevo su funcionamiento en el vector de reset, como podemos apreciar en la **Figura 8**. El oscilador seguirá funcionando mientras mantengamos el pulsador presionado.

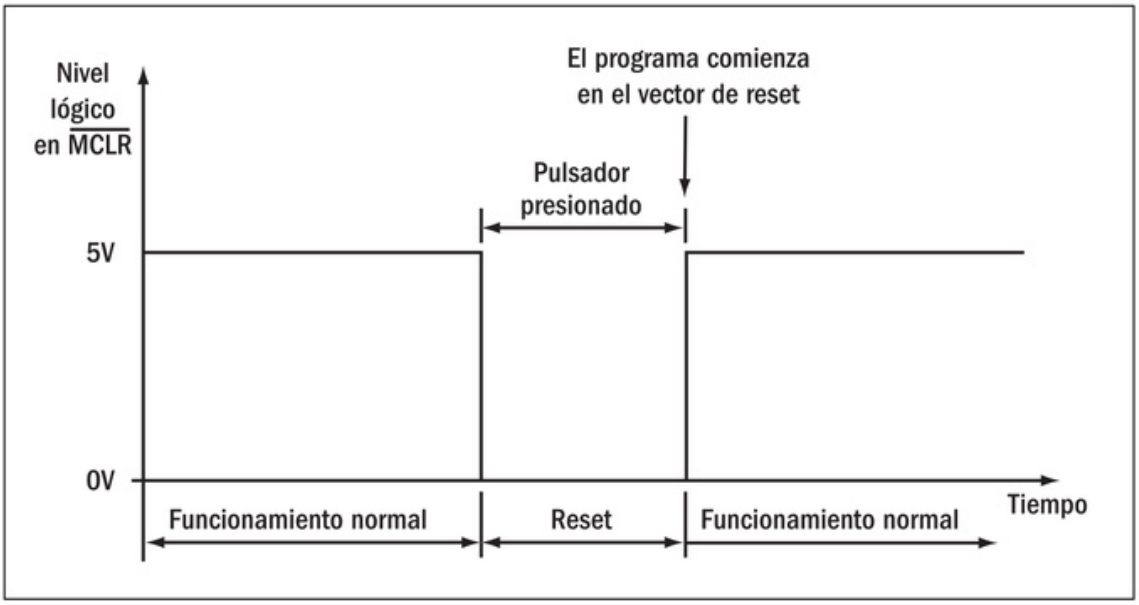


***Figura 8.** Funcionamiento del reset externo mediante un pulsador conectado al pin Master clear.*

También podemos conectar el pin Master clear a algún otro circuito que controlará o provocará el reset cuando sea necesario. El reset provocado por el temporizador de perro guardián (WDT) lo estudiaremos en la sección dedicada a ese tema.

PUERTOS DE ENTRADA/SALIDA

Como ya mencionamos, este microcontrolador cuenta con dos puertos de entrada/salida conocidos como el **Puerto A** y el **Puerto B**. Estos puertos los utilizaremos para enviar datos hacia el exterior de nuestro microcontrolador, o para recibir datos desde otros dispositivos. El Puerto A consta de 5 líneas llamadas **RA0** a **RA4**, y el Puerto B

III PULL-UP Y PULL-DOWN

Los resistores de **Pull-up** o **Pull-down** son encargados de mantener algún nivel lógico alto o bajo, respectivamente, en alguna entrada, ya que si no hay ninguna conexión se dice que la entrada está "flotando" y no hay ningún estado lógico definido en ella. Los términos se podrían traducir cómo "llevar hacia arriba" o "llevar hacia abajo", aunque se utiliza la denominación en inglés.

de 8 líneas llamadas **RB0** a **RB7**. Cada línea de cada puerto puede configurarse en forma independiente, ya sea como entrada o como salida según necesitemos.

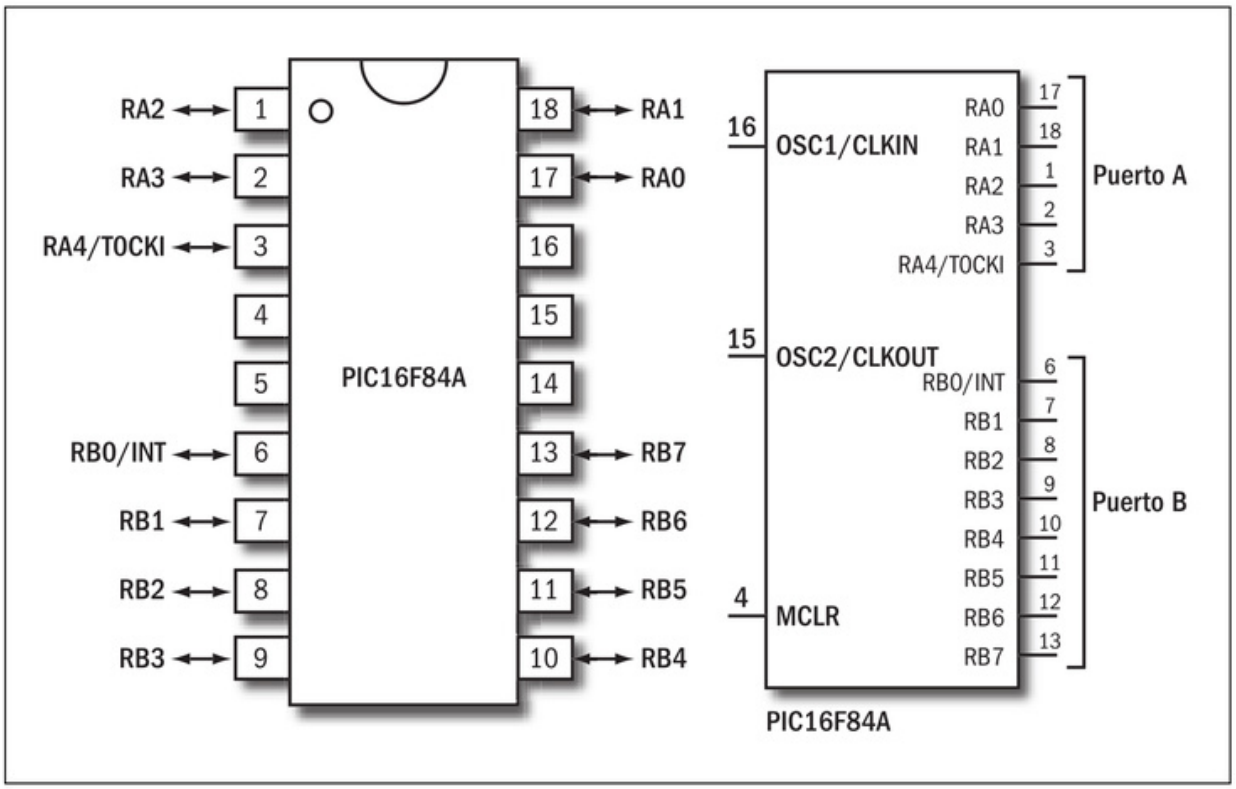


***Figura 9.** Localización de los puertos en el circuito físico y en el símbolo esquemático del PIC16F84A.*

Interruptores y pulsadores

A través de los puertos podemos enviar datos hacia nuestro microcontrolador desde diferentes dispositivos o circuitos. La forma más sencilla de hacerlo es mediante **interruptores** o **pulsadores**.

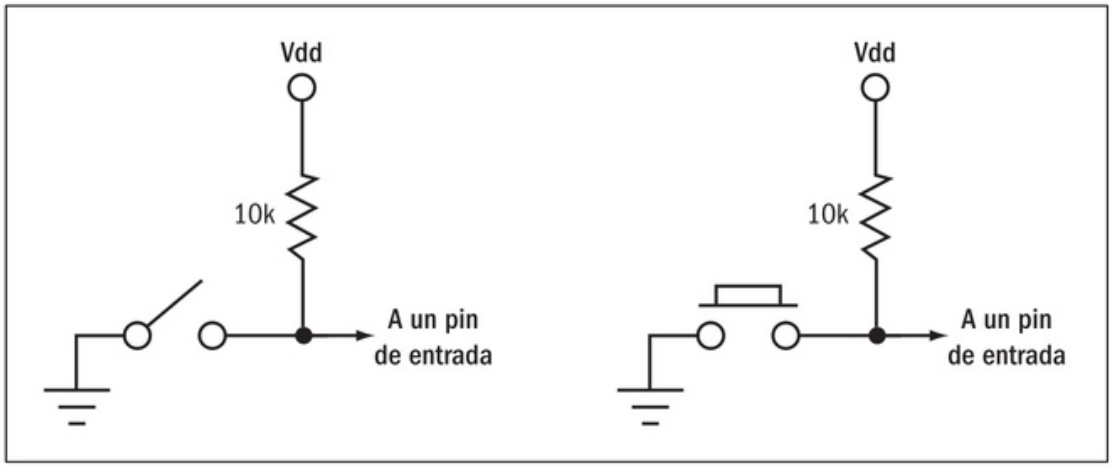


***Figura 10.** Los pulsadores o interruptores son la forma más simple de enviar datos hacia los puertos de nuestro PIC.*

De esta forma podemos enviar un 1 ó un 0 al cerrar o abrir el interruptor o el pulsador. Los resistores de **Pull-up** mantienen un nivel lógico alto en las entradas al estar conectados a Vdd, hasta que el interruptor o pulsador se cierra enviando un 0.

Leds

También podemos enviar datos desde el PIC hacia otros dispositivos o circuitos externos. Lo más sencillo es colocar leds en las líneas de salida para observar su estado, tal como graficamos en la **Figura 11**.

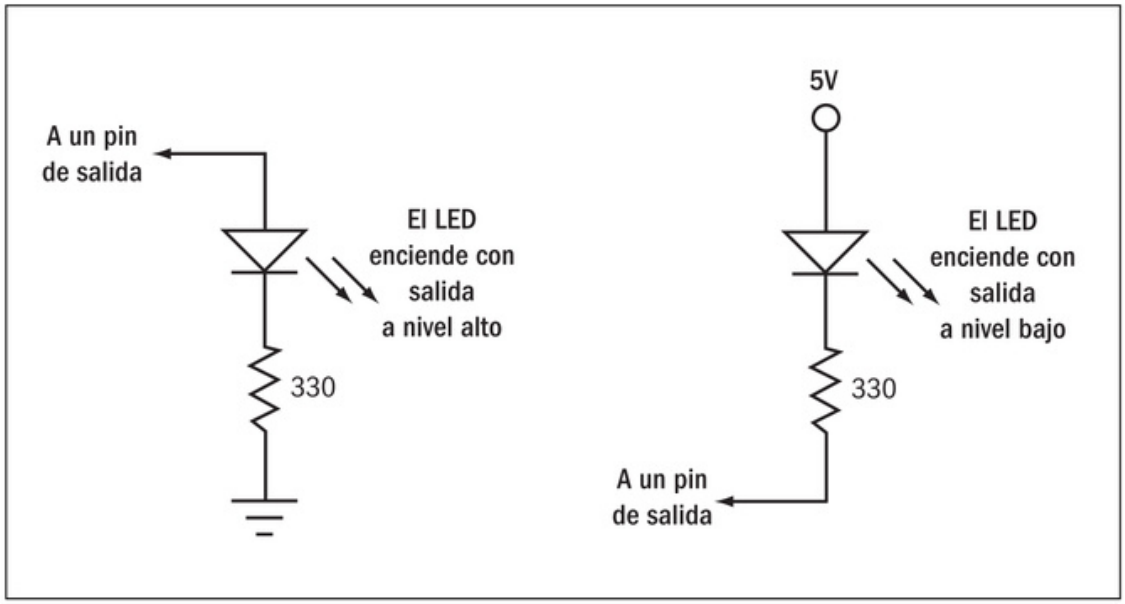


***Figura 11.** El más sencillo dispositivo de salida puede ser un simple led.*

El resistor limita la corriente que pasa por el led, y su valor depende del tipo y color del led a usar. Si deseamos más corriente y más luminosidad en el led podemos poner un resistor de 270 ó de 220 ohms para leds rojos comunes. También podemos enviar datos desde nuestro PIC hacia otros dispositivos o circuitos.

Señales máximas de los puertos

Para usar los puertos, ya sea como entrada o como salida, debemos configurarlos para que funcionen de esa manera. Más adelante hablaremos de cómo configurar las líneas de los puertos. Los puertos tienen una capacidad limitada para entregar corriente a la salida o para recibir corriente en ellos, y cada fabricante especifica los rangos máximos, que son los siguientes:

- Cada pin de los puertos puede entregar o recibir individualmente hasta 25 mA, es decir, no debemos exceder los 25mA de corriente a la salida o a la entrada de cualquiera de los pines de los puertos.

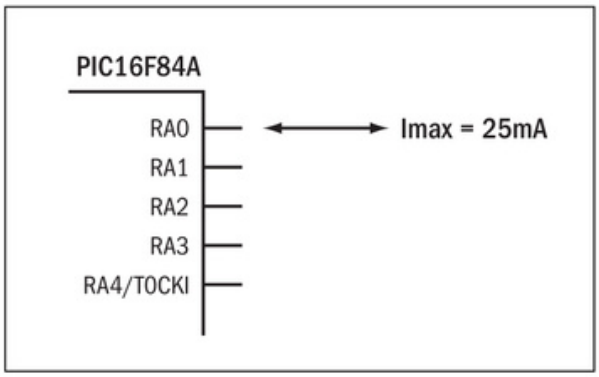


Figura 12. *Cada pin individual de los puertos puede recibir o entregar un máximo de 25 mili-amperes de corriente.*

- Sin embargo, la suma de todas las líneas del Puerto A no debe exceder los 50 mA cuando sale corriente de ellas, ni los 80mA cuando entra corriente a ellas.

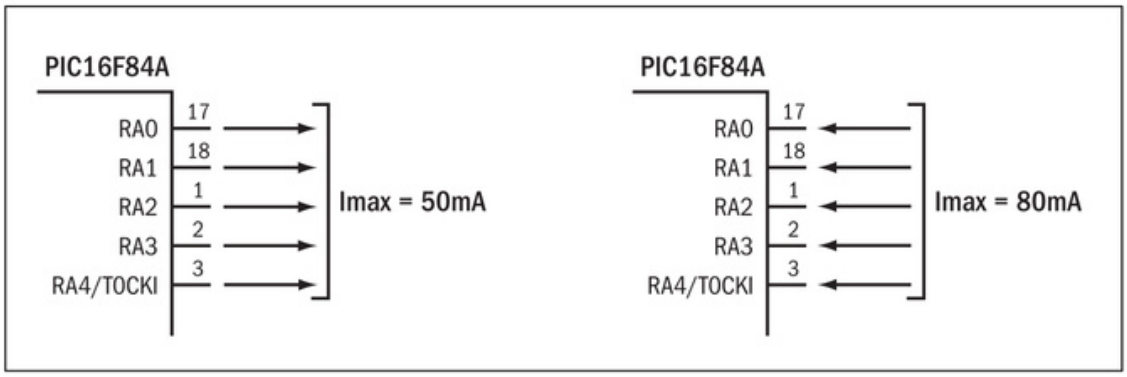


Figura 13. *La suma de las corrientes de entrada o salida de todas las líneas del Puerto A tiene un límite.*

- Para el Puerto B, la suma de las corrientes de salida no debe superar los 100 mA de salida, ni los 150 mA de entrada de corriente.

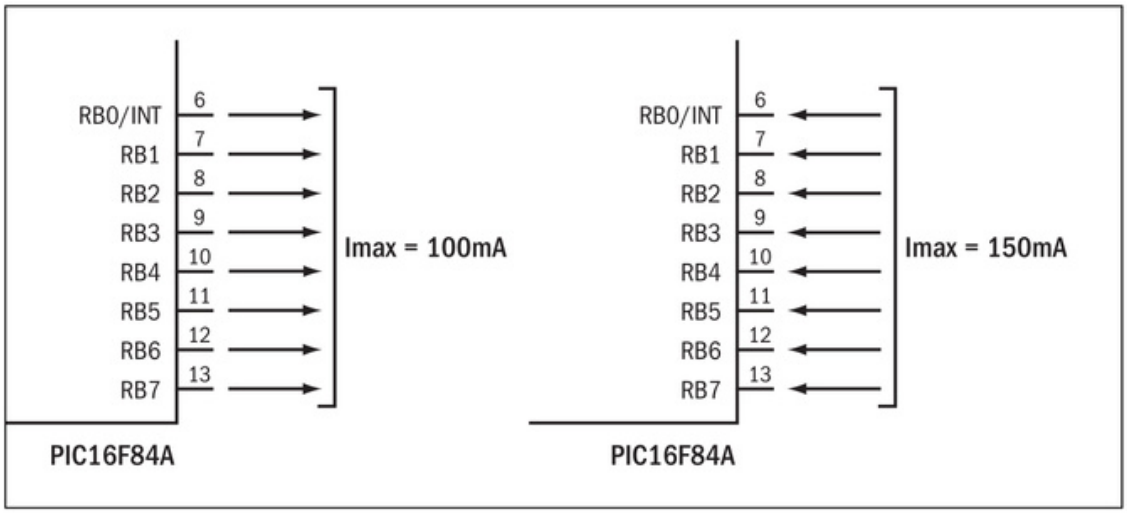


Figura 14. *La suma de las corrientes de entrada o salida de todas las líneas del Puerto B también tiene un límite.*

Es muy importante tomar en cuenta estos límites al momento de diseñar nuestros circuitos para evitar dañar nuestro microcontrolador. Si necesitamos manejar cargas mayores, podemos hacerlo mediante transistores, relevadores, optoacopladores u otros componentes para evitar sobrecargar los puertos.

El pin **RA4/T0CKI** tiene una configuración un poco diferente al tener una función alternativa como entrada del **Timer 0**, por lo que a la entrada tiene una configuración de **disparador Schmitt** que le proporciona cierta inmunidad al ruido. Es por eso que debemos usar este pin preferentemente como entrada sobre los demás, cuando sea posible. Cuando lo usamos como salida tiene una configuración de **drenador abierto**, por lo que debemos colocar un resistor de Pull-up externo para que funcione correctamente. Es muy importante que no nos olvidemos de esto.

ORGANIZACIÓN DE LA MEMORIA

Como ya sabemos, nuestro microcontrolador tiene una arquitectura Harvard, ya que la memoria de programa y la de datos están separadas. Cada una de las memorias tiene su propio bus, así que podemos acceder a ellas en forma independiente.

La memoria de programa

La **memoria de programa** nos servirá, como su nombre lo indica, para grabar en ella el programa que el microcontrolador ejecutará. Esta memoria es del tipo Flash y es no volátil, así que después de escribir las instrucciones de nuestro programa en ella, permanecerán ahí incluso si desconectamos la alimentación del circuito. Las instrucciones del programa son fijas y no se necesitará que cambien durante la operación, por lo tanto, esta memoria se escribirá o grabará una sola vez. Esto se hace mediante un grabador especial que, con la ayuda de una computadora, enviará el programa que diseñemos a esta memoria y entonces estará listo para ejecutarse. Más adelante, en el **Capítulo 5**, hablaremos con detenimiento de los grabadores.

III LÍNEAS DE PUERTOS SIN USAR

Es común en muchas aplicaciones no utilizar todas las líneas de los puertos. En este caso podemos tener dos opciones: configurar las líneas que no vamos a utilizar como salidas y conectarlas mediante un resistor a tierra, aunque la opción más fácil y también válida es configurarlas como salidas y dejarlas sin conexión.

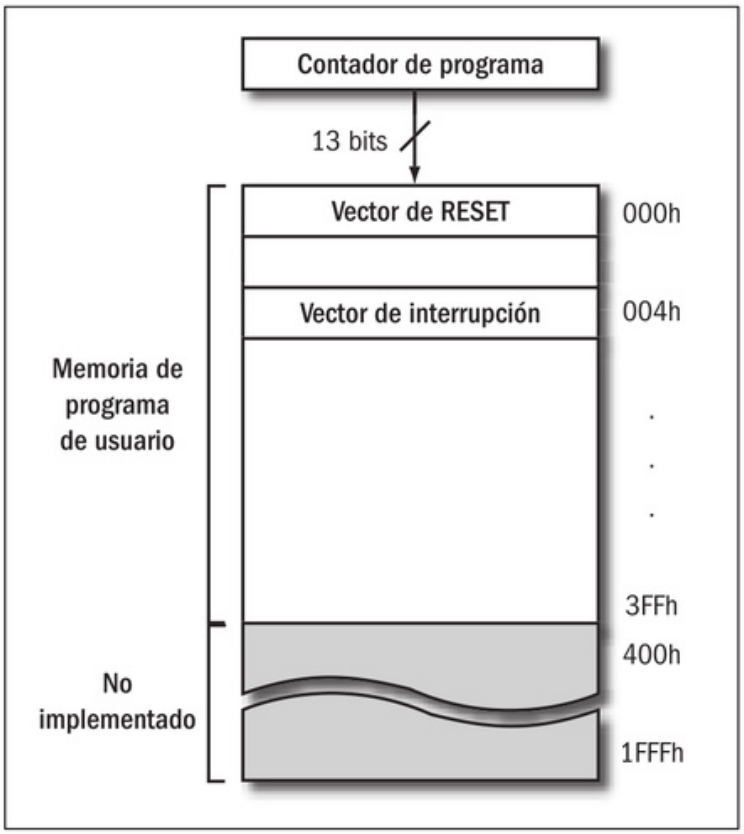


***Figura 15.** La memoria de programa almacenará las instrucciones del programa que gobernará el PIC.*

El contador de programa (PC)

El microcontrolador PIC16F84A cuenta con un **contador de programa**, que se abrevia **PC** (*Program Counter*), de 13 bits, con el que teóricamente se puede acceder a 8192 (8 k) direcciones de memoria, pero sólo están implementadas 1024 de ellas, es decir, el total de memoria de programa para este microcontrolador es de 1 k. La primera dirección de la memoria de programa es la 0000h y la última es la 03FFh, y cada palabra de la memoria es de 14 bits, ya que las instrucciones tienen esa longitud. Al iniciar o encender el sistema, el contador de programa inicia con un valor 0000h y se incrementa en forma secuencial para ir accediendo a las instrucciones.

La dirección 0000h es donde inicia el programa al encender el sistema, y al aplicar cualquier reset. Es por eso que es llamado **vector de reset**. La dirección 0004h es el **vector de interrupción**, es decir que cuando se genera una interrupción, el programa saltará a esta dirección. En el **Capítulo 10**, que dedicaremos al tema de las interrupciones, hablaremos en detalle de esto.

La memoria de datos

Durante la ejecución del programa necesitaremos hacer operaciones con valores o datos, los cuales cambiarán muchas veces, y estos necesitan un lugar donde ser

almacenados, ya que no podemos escribir en la memoria de programa desde el mismo programa. Para almacenar los datos generados por nuestros programas tenemos la **memoria de datos**, que es una memoria de tipo **SRAM** y es volátil, es decir, si desconectamos la alimentación del circuito, los datos almacenados en ella se perderán.

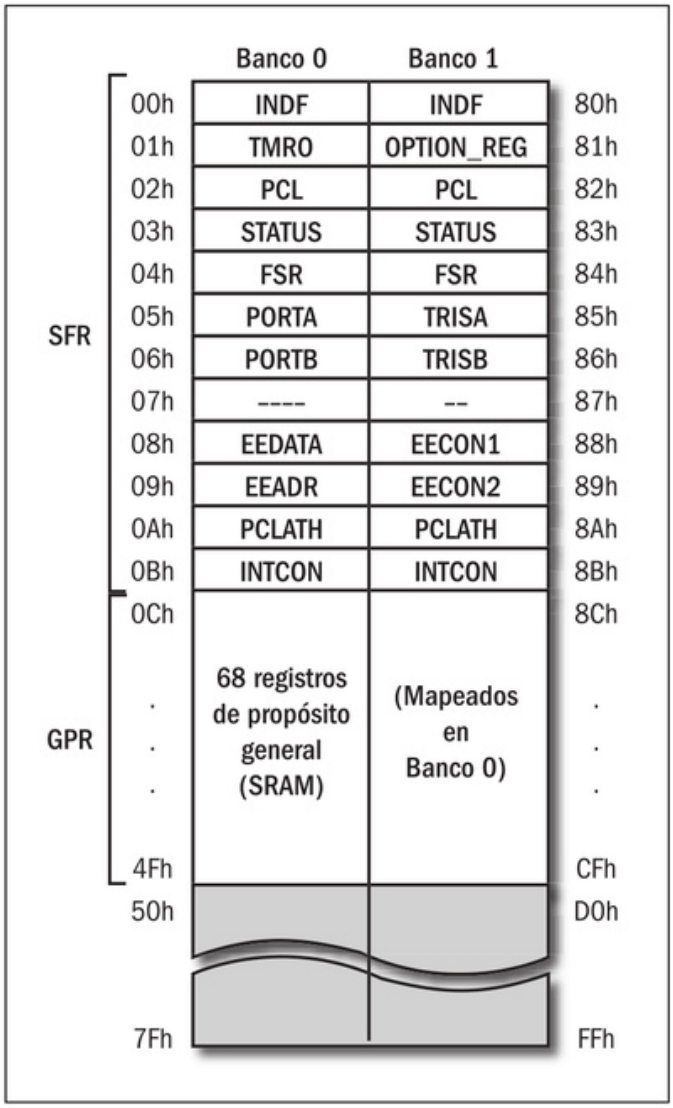


Figura 16. *Organización de la memoria de datos (SRAM) del PIC16F84A.*

La memoria de datos tiene una organización algo especial: está dividida en dos partes o **bancos**. Como podemos apreciar en la **Figura 16**, el banco 0 inicia en la dirección 00h, y el banco 1 inicia en la dirección 80h. Los primeros registros de cada banco son llamados **registros de función especial** o **SFR** (*Special Function Registers*) y sirven para propósitos específicos en el funcionamiento y la configuración del microcontrolador. Después de los SFR está el área de los **registros de propósito general** o **GPR** (*General purpose registers*), que podemos usar libremente para almacenar datos. Éstos comienzan en la dirección 0Ch para el banco 0, y en la dirección 8Ch para el banco 1, aunque realmente los registros de propósito general del banco 1 están mapeados, es decir, duplicados del banco 0, así

que si intentamos escribir o leer en algún registro de propósito general del banco 1, realmente lo estaremos haciendo en los registros del banco 0. En el área de los SFR algunos registros están duplicados en los dos bancos, pero otros no. Cada uno de los registros de la memoria de datos es de 8 bits.

De la dirección 50h en el banco 0 y la D0h en el banco 1 en adelante no están implementadas, y si intentamos leerlas devolverán ceros. Cada uno de los registros del área SFR tiene un propósito específico y no debemos usarlos para almacenar nuestros datos, ya que si lo hacemos alteraremos los valores que contienen y provocaremos un mal funcionamiento del programa. Es muy importante recordar que sólo debemos usar los registros del área GPR para almacenar nuestros datos.

REGISTROS DEL ÁREA SFR

A continuación, describiremos el uso, el funcionamiento y la estructura de algunos **registros de propósito específico**, y el resto los iremos estudiando a lo largo de los siguientes capítulos donde corresponda tratar el tema de cada uno de ellos. En la **Tabla 2** tenemos en detalle los SFR.

DIRECCIÓN	NOMBRE	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
Banco 0									
00h	INDF	Usado para direccionamiento indirecto (no es un registro físico)							
01h	TMR0	Timer / contador de 8 bits							
02h	PCL	Registro con los 8 bits más bajos del contador de programa (PC)							
03h	STATUS	IRP	RP1	RP0	TO'	PD'	Z	DC	C
04h	FSR	Puntero para el direccionamiento indirecto							
05h	PORTA	-	-	-	RA4/T0CKI	RA3	RA2	RA1	RA0
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT
07h	-	No implementado (se lee como ceros)							
08h	EEDATA	Registro de datos de la EEPROM							

III ARQUITECTURA DE REGISTROS

Los PIC están basados en la **arquitectura de registros** (**file registers**). Esto significa que tanto los registros de datos (RAM) como de puertos y demás funciones y configuraciones están organizados en un sólo banco de registros en la memoria de datos. De esta forma, las instrucciones pueden operar en cualquiera de ellos y hacer, así, más sencilla la programación.

DIRECCIÓN	NOMBRE	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
09h	EEADR	Registro de direcciones de la EEPROM							
0Ah	PCLATH	-	-	-	Buffer de los 5 bits más altos del PC				
0Bh	INTCON	GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF
Banco 1									
80h	INDF	Usado para direccionamiento indirecto (no es un registro físico)							
81h	OPTION_REG	RBPU	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0
82h	PCL	Registro con los 8 bits más bajos del contador de programa (PC)							
83h	STATUS	IRP	RP1	RP0	TO'	PD'	Z	DC	C
84h	FSR	Puntero para el direccionamiento indirecto							
85h	TRISA	-	-	-	Configuración de las líneas del Puerto A				
86h	TRISB	Configuración de las líneas del Puerto B							
87h	-	No implementado (se lee como ceros)							
88h	EECON1	-	-	-	EEIF	WRERR	WREN	WR	RD
89h	EECON2	Registro de control para grabación en EEPROM							
8Ah	PCLATH	-	-	-	Buffer de los 5 bits más altos del PC				
8Bh	INTCON	GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF

***Tabla 2.** Los registros de propósito específico del PIC16F84A.*

Los registros de los puertos

Anteriormente estudiamos cómo podemos recibir o enviar datos mediante los puertos. Ahora estudiaremos los registros que se usan para tal fin en el microcontrolador. Como podemos apreciar en la **Tabla 2**, los registros que están en la dirección 05h y 06h del banco 0 son llamados **PORTA** y **PORTB**, respectivamente. Éstos son los registros de los puertos, y en ellos pondremos los datos que saldrán del puerto en caso de que sea usado como salida, o se reflejarán los datos de entrada en su caso. Podemos ver cómo el nombre de cada bit de estos dos registros es el mismo que el de los pines del microcontrolador para cada puerto.

En el banco 1 podemos apreciar que los registros equivalentes a los anteriores, es decir, en las direcciones 85h y 86h, no tienen el mismo nombre. Esto es debido a que en este caso los registros no se repiten en los bancos de memoria. Los registros **TRISA** y **TRISB** se utilizan para configurar las líneas de los puertos, ya sea como entrada o salida. Si escribimos un 0 en alguno de los bits de estos registros estaremos configurando esa línea o pin como salida, y si escribimos un 1 lo configuraremos como entrada. Recordemos que podemos configurar cada línea de los puertos en forma independiente, por ejemplo, si enviamos un dato 00001111 al registro TRISB estaremos configurando las líneas más bajas (RB0 a RB3) como entradas y las líneas más altas del puerto (RB4 a RB7) como salidas.

El registro de estado (STATUS)

Uno de los registros más importantes, y con seguridad el que más utilizaremos, es el registro de estado o **STATUS**. Se encuentra en la dirección 03h del banco 0 y se repite en la dirección 83h del banco 1. Los bits de este registro son llamados, en ocasiones, **banderas** (*flags*), así que también podemos denominarlo el **registro de banderas**. Sus bits nos ayudan a saber el estado de la última operación lógica o aritmética que se ha realizado, también nos indica cuál es la causa de un reset y, lo más importante, nos permite cambiar de banco en la memoria de datos. En la **Tabla 3** tenemos al registro STATUS.

Bit 7	Bit 6	**Bit 5**	Bit 4	Bit 3	**Bit 2**	**Bit 1**	**Bit 0**
IRP	RP1	**RP0**	TO'	PD'	**Z**	**DC**	**C**

Tabla 3. *El registro de estado o STATUS.*

A continuación, veremos cuál es la función de los cuatro bits que hemos resaltado en la **Tabla 3**. No nos olvidamos del resto, sino que los estudiaremos más adelante, en el momento que los necesitemos.

Bit 0: C (Carry) accarreo

Este es el bit o bandera de acarreo. Sirve para indicar si ha habido un acarreo en el bit más significativo en la última operación. Para una suma aritmética, indica si se ha sobrepasado la capacidad del registro donde se lleva a cabo la suma. Es decir, cada registro es de 8 bits, por lo que la cantidad máxima que puede contener es $11111111_2 = 255_{10}$. Si el resultado de la suma es mayor a este valor, el bit de acarreo se activará (se pondrá a 1).

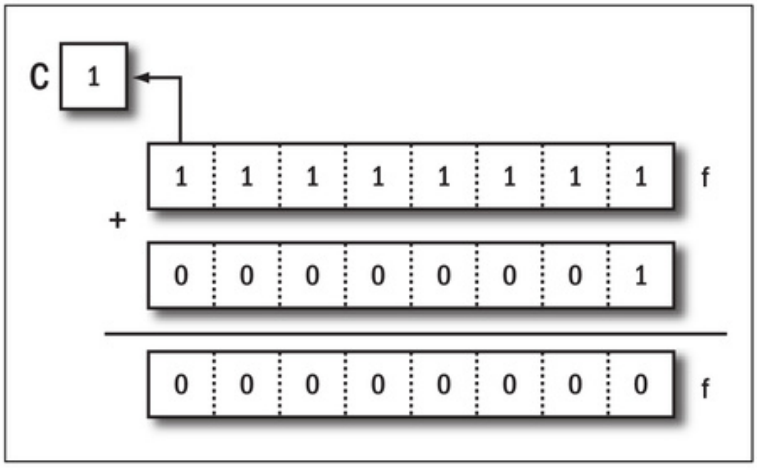


Figura 17. *Si en una suma se ha sobrepasado la capacidad de un registro se activará el bit de acarreo C.*

En el caso de la resta aritmética sucede lo contrario. Si este bit se pone a 0 significa que el resultado de la resta ha sido negativo. Por lo que en resumen:

C=0
Suma: indica que no ha habido acarreo en el bit más significativo.
Resta: indica que el resultado ha sido negativo.

C=1
Suma: indica que ha habido acarreo en el bit más significativo.
Resta: indica que el resultado ha sido positivo.

Bit 1: DC (Digit Carry) acarreo de dígito

Este bit es análogo al bit C, pero indica el acarreo en el cuarto bit menos significativo, es decir, se activa cuando hay un acarreo del bit 3 al bit 4. Por ejemplo, si tenemos en un registro el valor 00001111 y lo incrementamos en 1, el valor resultante será 00010000, por lo que hubo un acarreo del bit 3 al 4 y la bandera DC se activará (se pondrá a 1).

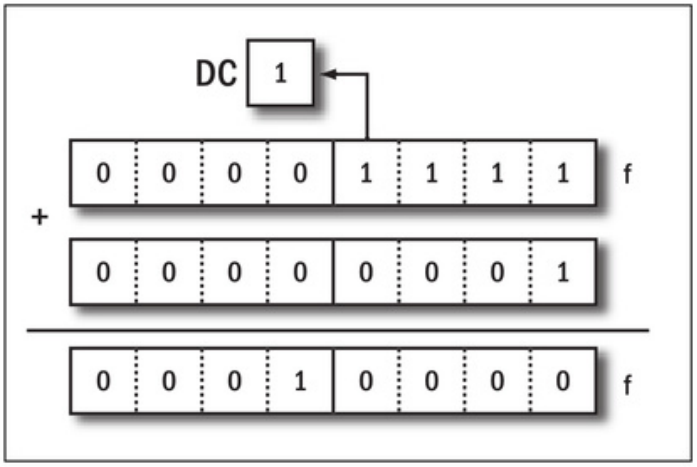


***Figura 18.** Si en una suma se ha sobrepasado la capacidad del nibble bajo de un registro, se activará el bit de acarreo de dígito DC.*

Bit 2: Z (Zero) Cero

Este bit o bandera de cero se activa (se pone a 1) cuando el resultado de la última operación aritmética o lógica realizada ha sido 0. Si el resultado ha sido diferente de 0, entonces no se activará (tendrá un 0).

NO DEJAR FLOTANDO EL MCLR

El pin Master clear es de entrada, tanto para el reset externo como para el voltaje de programación Vpp. Por eso, es importante no dejar este pin "flotando", sin conexión. Si no vamos a utilizar algún reset externo en nuestro microcontrolador, entonces lo debemos conectar a Vdd.

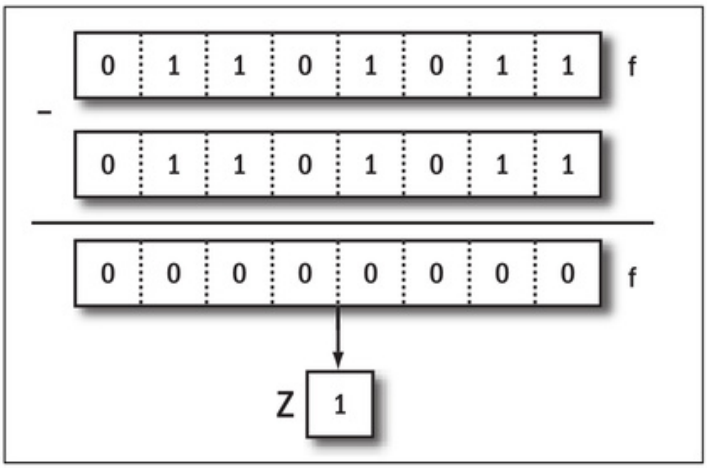


Figura 19. *Si el resultado de cualquier operación es 0, se activará el bit Z del registro STATUS.*

Bit 5: RP0 (Register bank select bit) Bit de selección de banco

Este bit es muy importante ya que nos permitirá seleccionar el banco de memoria en el cual deseamos acceder en la memoria de datos. Debemos poner el valor 1 ó 0 en él, según necesitemos:

Si **RP0=0**: se selecciona el banco 0 de la memoria de datos.
Si **RP0=1**: se selecciona el banco 1 de la memoria de datos.

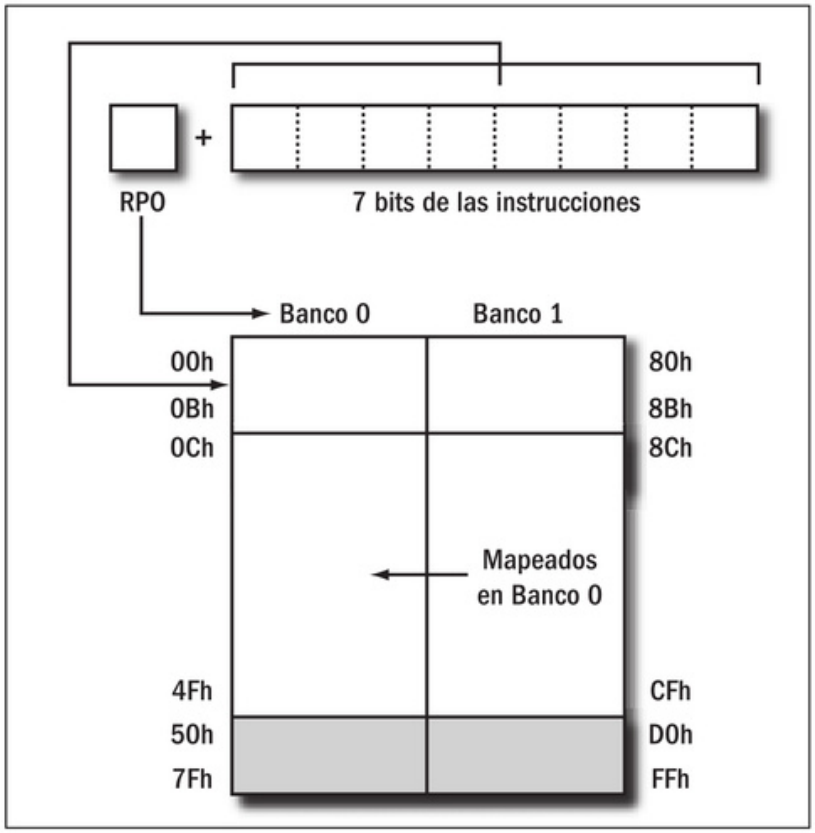


Figura 20. *El bit RP0 nos permite trabajar en un banco u otro de la memoria de datos en el direccionamiento directo.*

Así, si necesitamos configurar las líneas de alguno de los puertos como entradas o salidas, ya sabemos que debemos hacerlo con los registros TRISA o TRISB, pero para poder escribir en ellos necesitamos, primero, poner un 1 en el bit RP0 del registro STATUS para así acceder al banco 1 de la memoria donde se encuentran estos registros.

Los bits 6 y 7 no tienen ninguna utilidad en este microcontrolador y no están implementados. Los bits 3 y 4 los estudiaremos posteriormente.

Los registros PCL y PCLATH

El contador de programa del PIC16F84A es de 13 bits, de los cuales los 8 bits más bajos están mapeados en la memoria de datos, como el **registro PCL** (*Program Counter Low*) en la dirección 02h del banco 0 y en la dirección 82h del banco 1. Pero como sabemos, los registros de la memoria de datos sólo son de 8 bits, así que necesitaremos otro registro para los 5 bits más altos. Este es el **registro PCH**, que contiene estos 5 bits más altos del contador de programa. Pero este registro no es directamente accesible (no está mapeado en la memoria de datos).

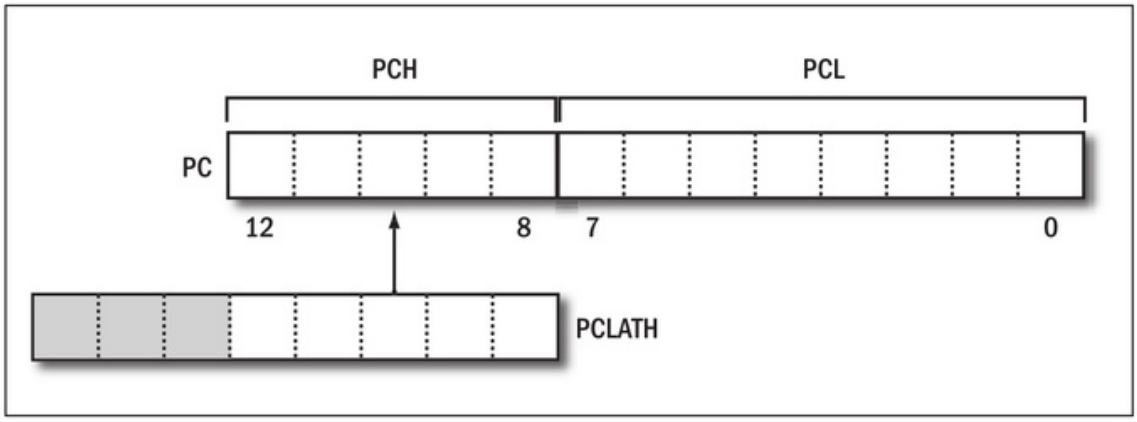


***Figura 21.** La estructura completa del contador de programa y los registros relacionados con él en la memoria de datos.*

El **registro PCLATH** (*PC Latch High*) que está en la dirección 0Ah del banco 0 y se repite en la dirección 8Ah del banco 1, es un registro que sirve para escribir en los 5 bits más altos del contador de programa. De modo que si queremos escribir en el registro PCH debemos hacerlo a través del registro PCLATH, dado que no podemos hacerlo directamente. Los bits del registro PCL reflejan en todo momento los bits más bajos del contador de programa.

El registro W

El **registro de trabajo** o **W** (*Working Register*) es muy importante. Es un registro de 8 bits y no está en la memoria de datos, sino que se encuentra dentro de

la CPU del microcontrolador, como vimos en la **Figura 1**. Casi todos los datos que manejemos pasarán por el registro W.

El PIC16F84A cuenta con una unidad aritmético-lógica (ALU) de 8 bits, que es la que lleva a cabo las operaciones aritméticas o lógicas. Si observamos la **Figura 1**, el registro W está a la salida de la ALU y, a su vez, a una entrada de ésta, por lo que las operaciones aritméticas o lógicas que lleve a cabo la ALU siempre serán entre algún registro y el registro W, y el resultado se puede almacenar ya sea en el registro W o en el otro registro involucrado en la operación. Se puede escribir en el registro W en cualquier momento, además de transferir su contenido a otro registro y viceversa.

RESUMEN

En este capítulo hemos estudiado la arquitectura interna del PIC16F84A para entender algunos de los bloques que lo forman, cómo funcionan y para qué sirven. También vimos cómo es la estructura de la memoria de datos y de programa, así como algunos registros especiales (SFR) que nos servirán para el uso y la configuración del microcontrolador. Con esto ya estamos listos para comenzar a estudiar las instrucciones del PIC16F84A.

TEST DE AUTOEVALUACIÓN

1 ¿Cuál es la arquitectura del PIC16F84A de acuerdo con su separación de memorias de programa y datos?

2 ¿Qué se usa en el modo de oscilador XT?

3 ¿Qué es un Reset?

4 ¿Cuántos tipos de Reset tiene el PIC16F84A?

5 ¿Qué es el PWRT?

6 ¿Para qué sirve la memoria de programa?

7 ¿Cuál es la capacidad de almacenamiento de la memoria de programa del PIC16F84A?

8 ¿Qué es la memoria de datos?

9 ¿Para qué sirven el Puerto A y el Puerto B del PIC16F84A?

10 ¿Para qué sirve el bit RP0 del registro STATUS?

Capítulo 3

Lenguaje ensamblador

Los microcontroladores son sistemas programados, es decir, funcionan a través de programas escritos por nosotros para realizar una tarea específica, por lo tanto, es importante conocer las instrucciones del microcontrolador que utilicemos, en este caso, el PIC16F84A. En este capítulo conoceremos el repertorio de instrucciones para luego poder aprender a escribir nuestros programas.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

EL LENGUAJE MÁQUINA

Los microcontroladores son circuitos digitales, solamente manejan niveles lógicos, es decir, unos y ceros, y por lo tanto las instrucciones que ejecutarán son grupos o palabras de unos y ceros que le indicarán al circuito digital qué tarea realizar. Cada familia de microprocesadores y microcontroladores tiene su propio repertorio de instrucciones particular y es diferente tanto en longitud como en los propios códigos de las instrucciones. A estas instrucciones en **binario** se las llama precisamente **lenguaje máquina** o **código máquina**, ya que es el único lenguaje que la máquina (el microprocesador o microcontrolador) puede entender y ejecutar.

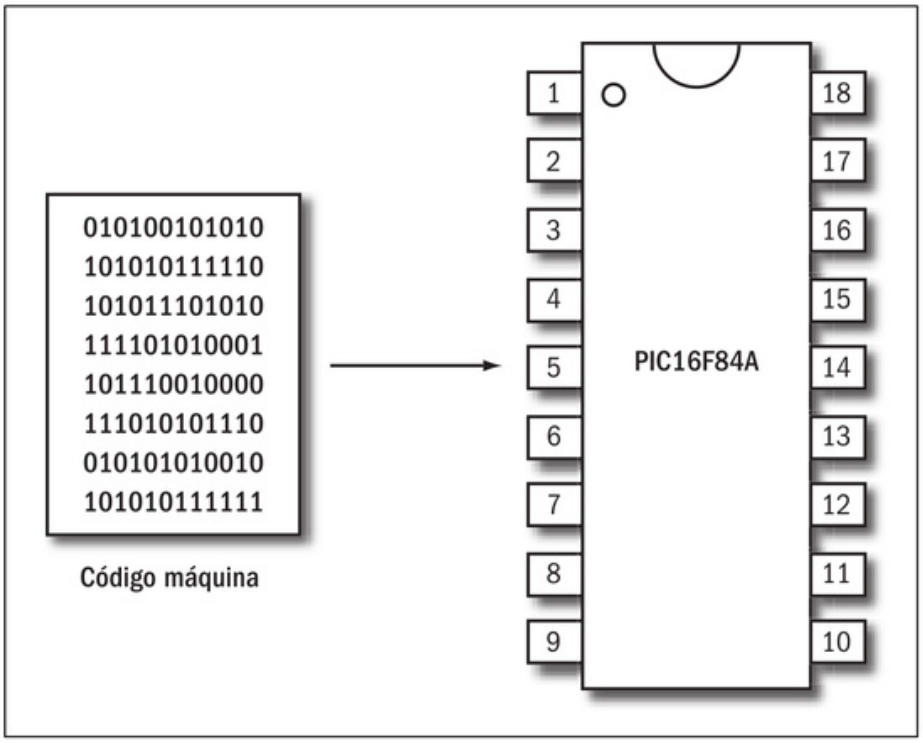


***Figura 1.** El microcontrolador sólo puede recibir las instrucciones en binario.*

Para el PIC16F84A, y muchos otros de esta familia, cada instrucción es un grupo de **14 bits**. De esta forma, nuestro programa será en esencia un conjunto de palabras de 14 bits cada una, las cuales grabaremos en la memoria de programa.

ARQUITECTURA RISC

El PIC16F84A tiene una arquitectura **RISC** (**Reduced Instruction Set Computer**) ya que contiene un repertorio reducido de sólo 35 instrucciones con las que podemos programar cualquier aplicación necesaria. No hay instrucciones complejas o especiales, y de esta manera la programación se hace más fácil y rápida.

Por ejemplo, la instrucción 11000000101111 le indicará a nuestro microcontrolador que guarde el número 47_d en el registro W.

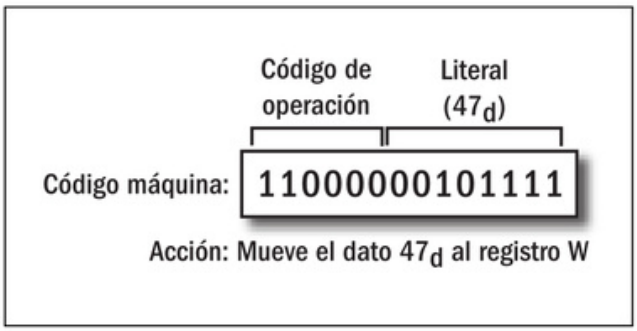


***Figura 2.** Las instrucciones son códigos binarios que llevarán a cabo un proceso definido.*

Para el microcontrolador, las instrucciones deben estar en código binario, pero para nosotros sería bastante difícil y confuso aprender y manejar las instrucciones de esa forma, sería tedioso, lento y cometeríamos muchos errores que serían muy difíciles de localizar y corregir. Debemos buscar entonces alguna forma de facilitar la representación de las instrucciones. Para comenzar, podemos utilizar código hexadecimal en lugar de binario, de esa manera, la instrucción que vimos antes (11000000101111) se puede expresar como $302F_h$, lo cual lo hace más simple, pero aun así sigue siendo difícil, ya que esto se aleja mucho de nuestro lenguaje natural.

EL LENGUAJE ENSAMBLADOR

Si observamos la **Figura 2** podemos darnos cuenta de que el código máquina es difícil de aprender e interpretar pero, en cambio, la acción nos es mucho más familiar, comprensible y fácil de recordar, ya que se parece mucho más a nuestra forma de lenguaje que al de la máquina. De esta manera, podemos escribir nuestros programas utilizando una forma más familiar para nosotros a través del llamado **lenguaje ensamblador**, que es una serie de **mnemónicos** (palabras o abreviaciones que representan la instrucción en binario).

Por ejemplo, la instrucción que hemos estado estudiando podemos escribirla en lenguaje ensamblador como **movlw d'47'**, que es la abreviación (en inglés) de *move literal to W,* y en español lo podemos traducir precisamente como **mueve la literal al registro W**, donde **d'47'** indica cuál es esa literal y en qué sistema está expresada, en este caso, en decimal. Así, cada una de las instrucciones para nuestro microcontrolador tiene un mnemónico que la representa, y de esta forma no debemos preocuparnos por los códigos máquina.

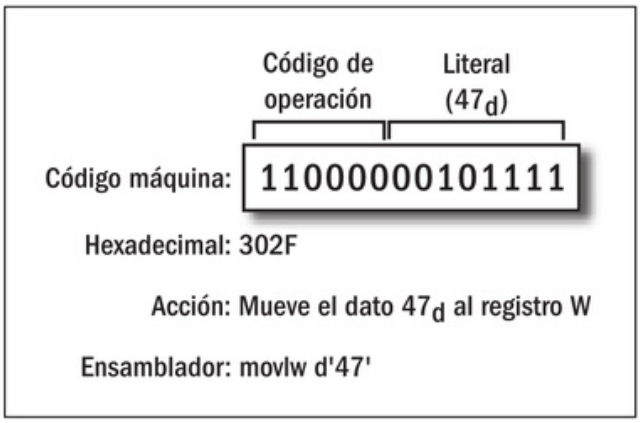


Figura 3. *El lenguaje ensamblador nos facilitará la escritura de los programas para nuestro PIC.*

El lenguaje ensamblador es llamado un lenguaje de **bajo nivel**, porque está muy cerca del lenguaje máquina, ya que cada mnemónico se traduce como una única instrucción en código máquina. Existen lenguajes de **alto nivel**, en los cuales se utilizan expresiones que son más parecidas al lenguaje humano. La ventaja de los lenguajes de alto nivel es que son más fáciles de aprender y de utilizar al estar más arriba, es decir, más cerca de nuestro lenguaje natural, pero tienen la desventaja de generar códigos de máquina mucho más grandes ya que una instrucción en un lenguaje de alto nivel equivale, generalmente, a varias instrucciones en ensamblador, así que el lenguaje ensamblador genera códigos mucho más pequeños y eficientes.

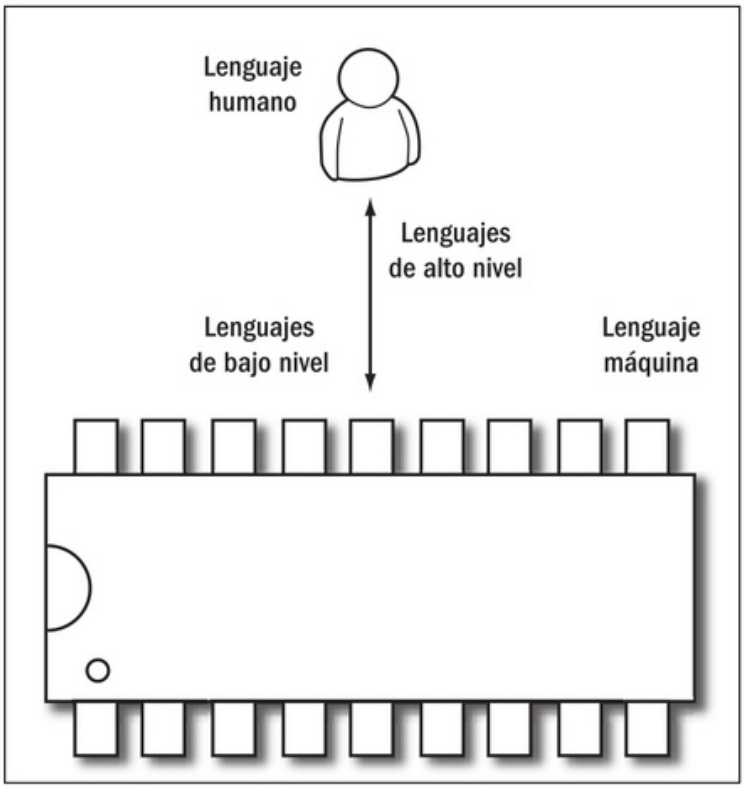


Figura 4. *El lenguaje de programación se acerca o aleja del lenguaje humano según su nivel.*

Se pueden utilizar lenguajes de alto nivel para escribir programas para los microcontroladores PIC como C o BASIC, pero antes de ellos es importante conocer el lenguaje ensamblador, que es más poderoso y eficiente, al menos para aplicaciones sencillas. Además, la ventaja de usar un lenguaje de alto nivel radica en sacar provecho de que ya se tiene el conocimiento de ese lenguaje y así se evita tener que aprenderlo (es por eso que no lo haremos ahora).

El programa ensamblador

Podemos entonces usar el lenguaje ensamblador para escribir nuestros programas de forma fácil y rápida, ya que de ese modo nos será sencillo memorizar y utilizar las instrucciones. A nuestro programa escrito en lenguaje ensamblador lo llamaremos **código fuente** o **programa fuente**. Para escribir nuestro programa fuente utilizaremos ya sea una simple hoja de papel, o mejor aun, algún software que nos permita hacerlo en nuestra computadora, como veremos en breve. Así, será más fácil escribirlo, modificarlo o ampliarlo.

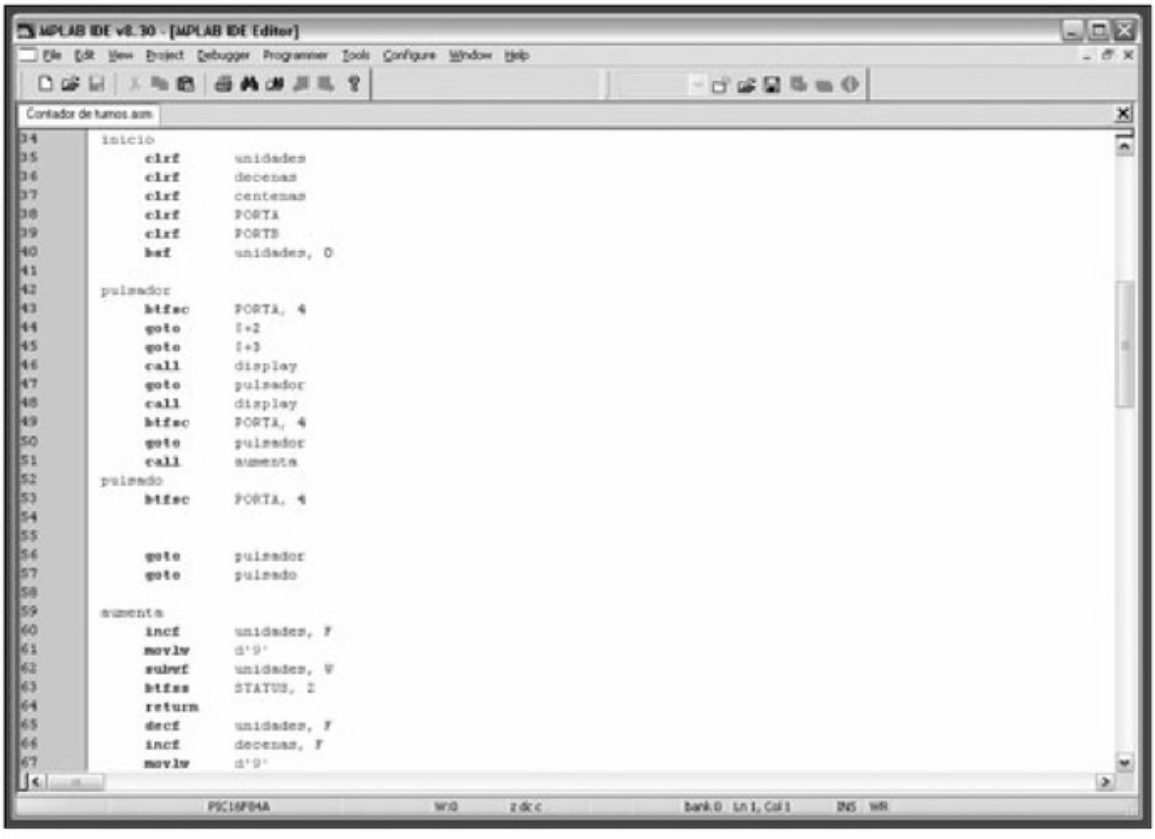


Figura 5. *Utilizaremos un editor para escribir nuestros código fuente.*

Una vez escrito nuestro programa en lenguaje ensamblador, no podemos llevarlo al microcontrolador de esa manera, sino que debemos traducirlo al lenguaje máquina para luego poder grabarlo en la memoria de programa del PIC. Por supuesto, esta laboriosa tarea de traducir los mnemónicos a lenguaje máquina no será

nuestra, para ello utilizaremos un software llamado **programa ensamblador**, que es un programa capaz de reconocer los mnemónicos del código fuente y traducirlos automáticamente a lenguaje máquina. Este proceso de traducir el código fuente a lenguaje máquina se llama **ensamblado**.

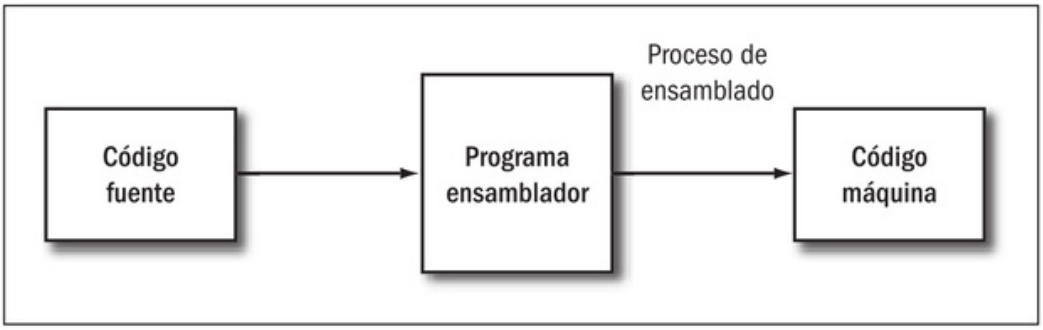


***Figura 6.** El programa ensamblador genera el código máquina en el proceso llamado ensamblado.*

Existen diversos programas ensambladores para microcontroladores PIC, pero lo más común es usar el entorno de desarrollo MPLAB, que es un programa distribuido gratuitamente por el propio fabricante de los microcontroladores PIC, y contiene todas las herramientas, incluido el ensamblador llamado **MPASM**, para escribir los programas y ensamblarlos, es decir, generar el código máquina a partir del código fuente que escribiremos. En el **Capítulo 4** hablaremos en detalle del uso de este programa.

El ciclo de máquina

El tiempo de ejecución de una instrucción se mide en **ciclos de máquina**, ya que se deben llevar a cabo varios pasos para ejecutar una sola instrucción. Por ejemplo, una instrucción, en general, requiere de estos pasos para ser ejecutada: en el primer ciclo de reloj se busca la instrucción en la memoria, en el segundo se decodifica dicha instrucción, en el tercero se la ejecuta y en el cuarto se almacena el resultado obtenido en algún lugar. Este tiempo es el que se conoce como ciclo de máquina o ciclo de instrucción. Para el PIC16F84A el ciclo de máquina dura cuatro ciclos de reloj.

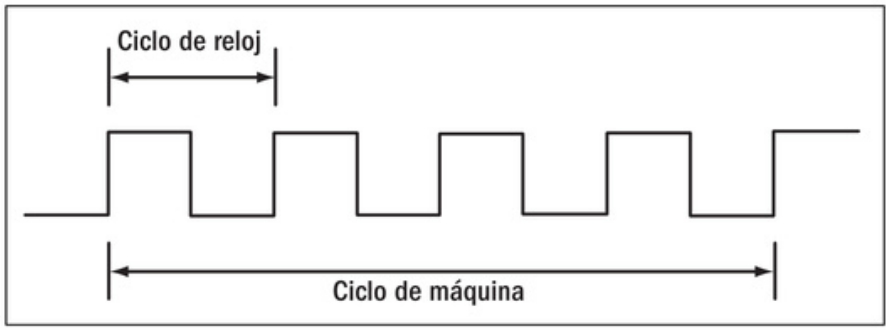


***Figura 7.** El ciclo de máquina consta de 4 ciclos de reloj. Casi todas las instrucciones se ejecutan en un ciclo de máquina.*

Este ciclo de máquina determina el tiempo de ejecución de las instrucciones. La mayoría de las instrucciones del PIC16F84A se ejecutan en un ciclo de máquina, excepto aquéllas en las que se modifica el contador de programa, es decir, las instrucciones de salto, las cuales requieren de dos ciclos de máquina. Si sabemos la frecuencia de la señal de reloj y cuántos ciclos de máquina requiere alguna instrucción, podemos calcular el tiempo que le tomará ejecutarse.

EL REPERTORIO DE INSTRUCCIONES

El PIC16F84A tiene una arquitectura **RISC** con tan sólo **35 instrucciones** que son fáciles de aprender y de usar. En la **Tabla 1** reunimos un resumen de las 35 instrucciones. Podemos apreciar que el repertorio está dividido en tres grandes grupos.

INSTRUCCIÓN	SIGNIFICADO	DESCRIPCIÓN	CM	BANDERAS
Operaciones orientadas a bytes (registros)				
addwf f,d	Add W and f	Suma W y f	1	C, DC, Z
andwf f,d	AND W with f	Operación AND entre W y f	1	Z
clrf f	Clear f	Borra el registro f	1	Z
clrw	Clear W	Borra el registro W	1	Z
comf f,d	Complement f	Complementa el registro f	1	Z
decf f,d	Decrement f	Decrementa el registro f	1	Z
decfsz f,d	Decrement f, skip if 0	Decrementa f, salta si es 0	1(2)	
incf f	Increment f	Incrementa el registro f	1	Z
incfsz f,d	Increment f, skip if 0	Incrementa f, salta si es 0	1(2)	
iorwf f,d	Inclusive OR W with f	Operación OR entre W y f	1	Z
movf f,d	Move f	Mueve f	1	Z
movwf f	Move W to f	Mueve el registro W al f	1	
nop	No opeartion	No operación	1	
rlf f,d	Rotate left f through carry	Rota f a la izquierda	1	C
rrf f,d	Rotate right f through carry	Rota f a la derecha	1	C

III PROCESADOR SEGMENTADO

El procesador del PIC16F84A que estamos estudiando es un **procesador segmentado** (Pipelined), lo cual significa que mientras se ejecuta una instrucción, el procesador es capaz de buscar la siguiente en la memoria para tenerla lista al término de la instrucción anterior, lo que permite mayor velocidad y eficiencia en los programas.

INSTRUCCIÓN	SIGNIFICADO	DESCRIPCIÓN	CM	BANDERAS
subwf f,d	Substract W from f	Resta W de f	1	C, DC, Z
swapf f,d	Swap nibbles in f	Intercambia nibbles en f	1	
xorwf f,d	Exclusive OR W with f	Operación XOR entre W y f	1	Z
Operaciones orientadas a bits				
bcf f,b	Bit clear f	Borra el bit b de f	1	
bsf f,b	Bit set f	Pon a 1 el bit b de f	1	
btfsc f,b	Bit test f, skip if clear	Prueba el bit b de f, salta si es 0	1(2)	
btss f,b	Bit test f, skip if set	Prueba el bit b de f, salta si es 1	1(2)	
Operaciones orientadas literales y de control				
addlw k	Add literal and W	Suma la literal k y W	1	C, DC, Z
andlw k	AND literal with W	Operación AND entre literal y W	1	Z
call k	Call subroutine	Llama a la subrutina	2	
clrwdt	Clear watch dog timer	Borra el WDT	1	TO', PD'
goto k	Go to adress	Salta a la dirección k	2	
iorlw k	Inclusive OR literal with W	Operación OR entre literal k y W	1	Z
movlw k	Move literal to W	Mueve la literal k a W	1	
retfie	Return from interrupt	Regresa de la interrupción	2	
retlw	Return with literal in W	Regresa con literal k en W	2	
return	Return from subroutine	Regresa de la subrutina	2	
sleep	Go into standby mode	Entra en modo de bajo consumo	1	TO', PD'
sublw k	Substract W from literal	Resta W de la literal k	1	C, DC, Z
xorlw k	Exclusive OR literal with W	Operación XOR entre literal k y W	1	Z

***Tabla 1.** El repertorio de instrucciones del PIC16F84A.*

Las instrucciones están formadas por el mnemónico seguido de diferentes operandos que dependen del tipo de instrucción que se trate, y si son necesarios para indicar literales, registros o destinos de almacenamiento. En las siguientes secciones veremos qué significa cada una de las instrucciones, y los operandos que las acompañan, si los hay. En la **Tabla 1** indicamos el significado en inglés, ya que los mnemónicos son abreviaciones de este significado. También indicamos cuántos ciclos de máquina requiere cada instrucción para ejecutarse. Algunas sólo requieren un ciclo de máquina, otras dos, y algunas en las que hay una condición de salto como en **btss**, están indicadas como **1(2)**, ya que mientras no se lleve a cabo el salto, sólo se requiere un ciclo de máquina, y cuando la condición se cumple y se realiza el salto, entonces se requieren dos ciclos para completar el salto. Además, indicamos qué bandera o banderas del registro STATUS son afectadas en cada caso. Por ejemplo, algunas instrucciones sólo afectan la bandera de cero (Z) y algunas no afectan ninguna bandera.

Operaciones orientadas a bytes (registros)

Este grupo de instrucciones trabaja con los registros de la memoria de datos, es decir, realiza una operación con un registro completo. En la **Figura 8** tenemos el formato general de este grupo de instrucciones.

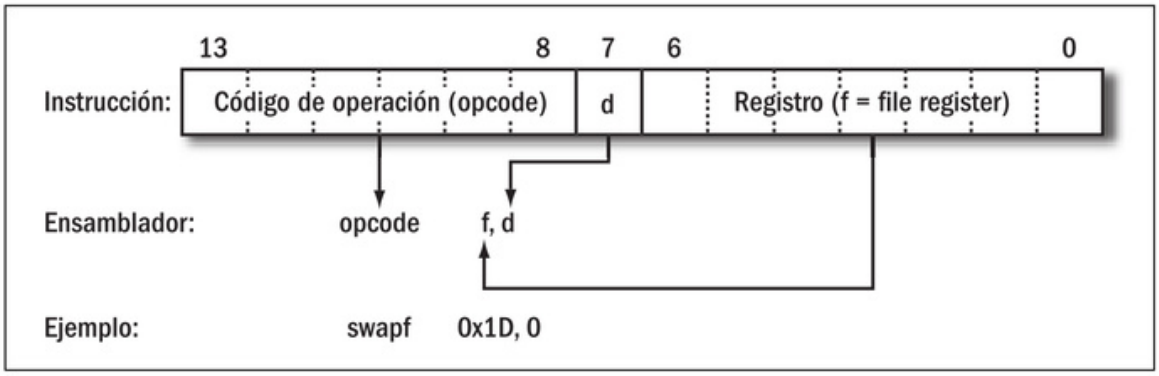


Figura 8. *Las instrucciones orientadas a registros tienen un formato general, como el que mostramos aquí.*

Los 6 bits más altos (8 al 13) en estas instrucciones son el **código de operación** (*opcode*), que es simplemente el código que le dirá al microcontrolador cuál es la tarea que debe realizar. Los 7 bits más bajos (0 a 6) indican la **dirección del registro** (*file register*) sobre el cual se va a trabajar, que puede ser cualquier registro de la memoria de datos. El bit 7 es el bit de **destino** (d) y se usa para indicar en dónde se almacenará el resultado de la instrucción. Si el bit de destino tiene el valor 0, el resultado se almacenará en el registro de trabajo (W), y si tiene el valor 1, entonces el resultado se almacenará en el registro con el que se está trabajando, es decir, en el mismo registro dado por los 7 bits más bajos de la instrucción, reemplazando el valor que contenía antes de ella. Estos son los operandos de este grupo de instrucciones, algunos puede que usen un solo operando o ninguno, pero si se usan los dos deben separarse con una coma. A continuación, estudiaremos en detalle cada una de las instrucciones de este grupo y daremos algunos ejemplos de su utilización, para que podamos comprender mejor su uso.

addwf f,d (add W and f)

Esta instrucción calcula la suma entre el contenido del registro W y el contenido del registro definido por f, y el resultado se almacena en el registro definido como destino.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

C: si el resultado de la suma ha sido mayor a FF entonces se activa el bit de acarreo.

DC: si ha habido un acarreo del nibble bajo al nibble alto, se activa la bandera de acarreo de dígito.

Z: si el resultado de la suma ha sido cero, se activa la bandera de cero.

Ciclos de máquina: 1

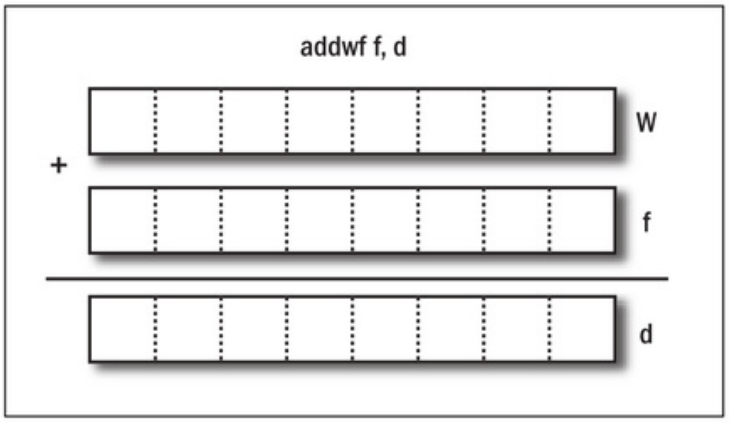


***Figura 9.** La suma de los contenidos de los registros W y f se realiza mediante la instrucción addwf.*

Ejemplo:

```
addwf 0x06, 1
```

En este ejemplo, la instrucción **addwf** realizará la suma entre el contenido del registro W y el contenido del registro que está en la dirección 0x06 de la memoria de datos, tal como lo indica este operando (recordemos que el registro que está en la dirección 0x06 pertenece al Puerto B del PIC16F84A). Como en este caso el bit de destino tiene el valor 1, el resultado de la suma se almacenará en el mismo registro 0x06, sobrescribiendo su contenido anterior. El registro W no es alterado en este ejemplo.

andwf f,d (AND W with f)

Realiza una operación lógica AND entre cada uno de los bits de los registros W y el registro definido por f y almacena el resultado en el destino definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado de la operación ha sido cero, se activa la bandera de cero.

Ciclos de máquina: 1

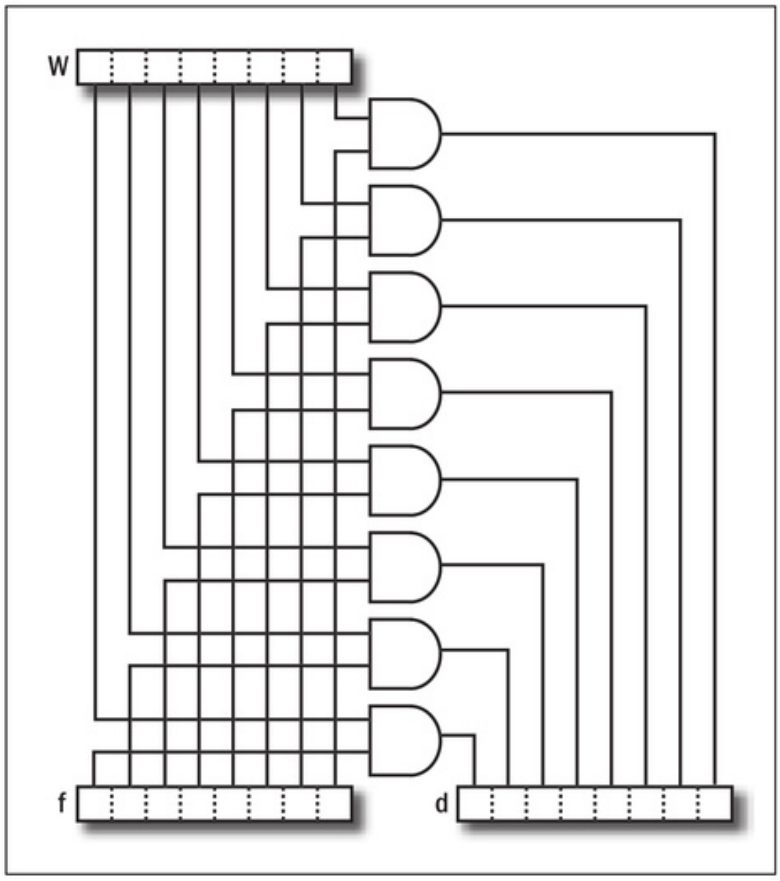


Figura 10. *La instrucción andwf hace una operación lógica AND entre los bits de los registros W y f.*

Ejemplo:

```
andwf temp, 0
```

Se hace una operación AND bit a bit entre los registros W y el registro llamado **temp** (más adelante estudiaremos cómo podemos asignar nombres a los registros, de tal forma que podamos utilizar su nombre en lugar de su dirección) y el resultado se almacena en el registro W sobrescribiendo su contenido anterior.

clrf f (Clear f)

Esta instrucción borra o limpia (pone a 0) todos los bits del registro dado por f y el resultado se almacena en el mismo registro f.

Banderas afectadas:

Z: se activa la bandera de cero.

Ciclos de máquina: 1

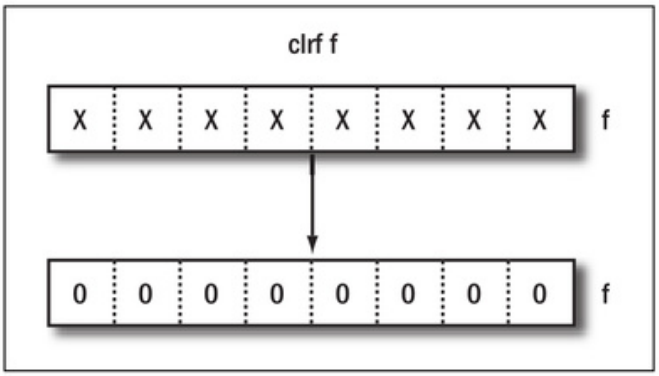


Figura 11. *La instrucción clrf borra o pone a cero todos los bits de cualquier registro.*

Ejemplo:

```
clrf PORTB
```

Mediante esta instrucción se pone a 0 todos los bits del registro PORTB.

clrw (Clear W)

Esta instrucción borra (pone a 0) todos los bits del registro W. No tiene operandos.

Banderas afectadas:

Z: se activa la bandera de cero.

Ciclos de máquina: 1

comf f,d (Complement f)

Esta instrucción complementa o invierte los bits del registro definido por f. El resultado se almacena en el registro definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado ha sido cero se activa la bandera de cero.

Ciclos de máquina: 1

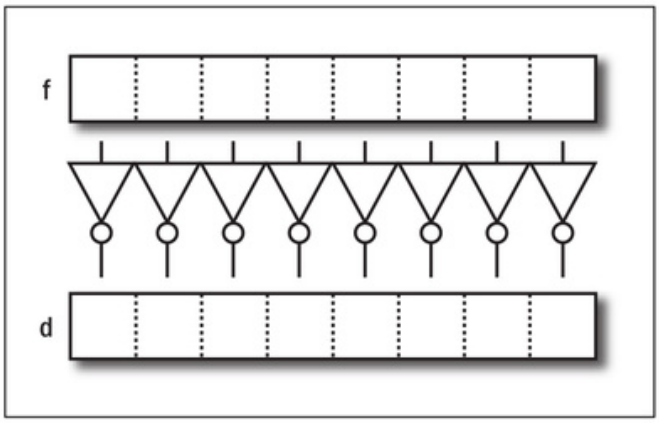


Figura 12. *La instrucción comf invierte los bits del registro f.*

Ejemplo:

```
comf 0x0C, 0
```

Complementa o invierte los bits del registro 0x0C y guarda el resultado en el registro W, ya que en este ejemplo el destino es 0.

decf f,d (Decrement f)

Esta instrucción decrementa en una unidad (le resta 1) al registro f y almacena el resultado en el registro definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado ha sido cero se activa la bandera de cero.

Ciclos de máquina: 1

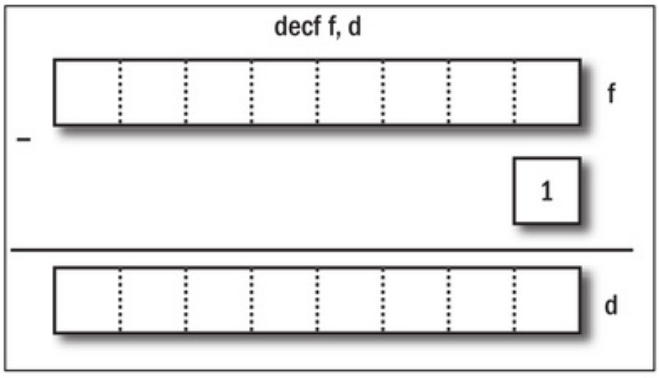


Figura 13. *La instrucción decf le resta 1 a cualquier registro.*

Ejemplo:

```
decf contador, 1
```

Decrementa en 1 el registro contador y almacena el resultado en el mismo registro.

decfsz f,d (Decrement f, skip if 0)

Esta instrucción decrementa en una unidad (es decir, le resta 1) al registro f y almacena el resultado en el registro definido por d. Además, hace una verificación: si el resultado del decremento ha sido 0, entonces salta a la instrucción siguiente, pero si no lo ha sido, la ejecuta.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas: ninguna.

Ciclos de máquina: 1 si no salta, 2 cuando la condición es cumplida y realiza el salto.

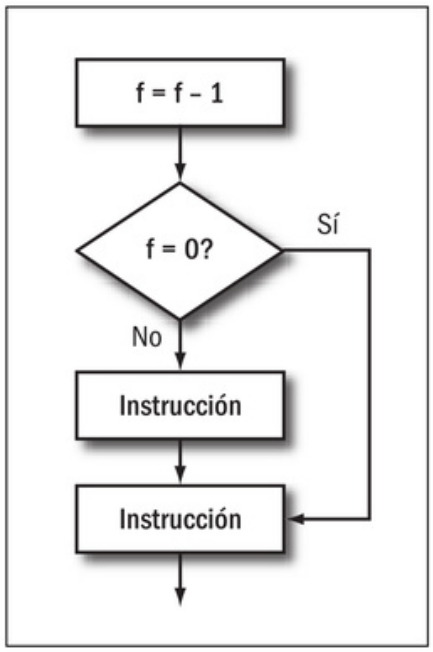


Figura 14. *Funcionamiento de la instrucción decfsz, la cual realiza un salto condicional.*

Ejemplo:

```
decfsz contador, 1
```

```
[instrucción 1]
[instrucción 2]
```

Decrementa en 1 el registro llamado contador y almacena el resultado en él. Al decrementar el registro contador también hace la verificación, de modo que si el decremento ha dado como resultado un número diferente de 0, entonces se ejecuta la instrucción 1. Si el decremento ha dado como resultado 0, entonces la instrucción 1 es saltada y se ejecuta la instrucción 2.

incf f,d (Increment f)

Esta instrucción incrementa en una unidad (le suma 1) al registro f y almacena el resultado en el registro definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado ha sido cero se activa la bandera de cero.

Ciclos de máquina: 1

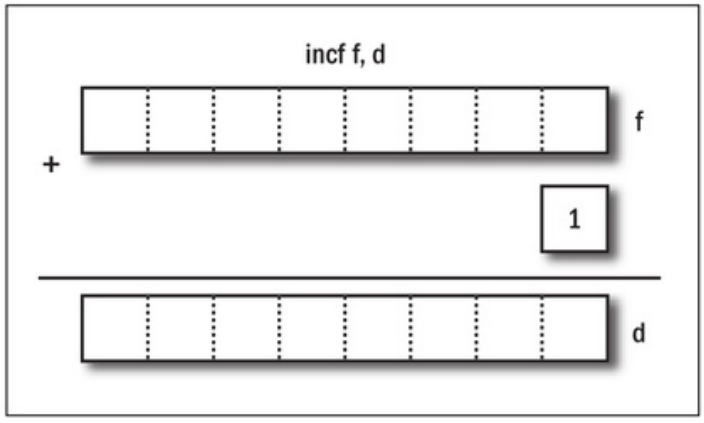


***Figura 15.** La instrucción incf le suma 1 a cualquier registro.*

Ejemplo:

```
incf contador, 1
```

Como vemos en el ejemplo, mediante la instrucción **incf** se incrementa en 1 el registro llamado contador y almacena el resultado en el mismo registro.

incfsz f,d (Increment f, skip if 0)

Esta instrucción incrementa en una unidad (le suma 1) el registro f y almacena el resultado en el registro definido por d. Además, hace una verificación: si el resultado del incremento ha sido 0, entonces salta la instrucción siguiente, si no ha sido 0, ejecuta la instrucción siguiente.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas: ninguna.

Ciclos de máquina: 1 si no salta, 2 cuando la condición es cumplida y realiza el salto.

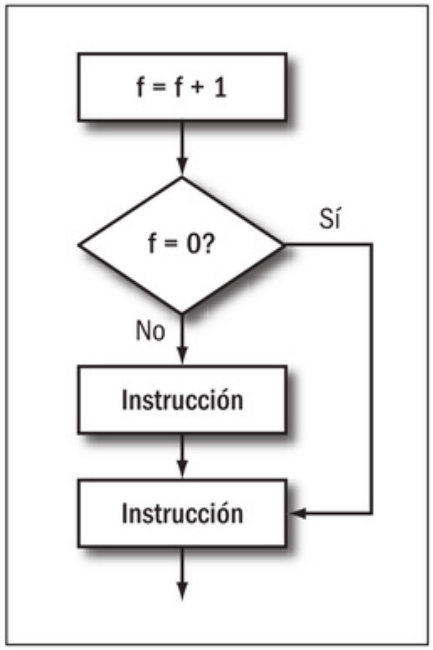


Figura 16. *Funcionamiento de la instrucción incfsz, la cual realiza un salto condicional.*

Ejemplo:

```
incfsz contador, 1
[instrucción 1]
[instrucción 2]
```

Incrementa en 1 el registro llamado contador y almacena el resultado en el destino definido por d. Si el decremento ha dado como resultado un número diferente de 0, entonces se ejecuta la instrucción 1. Si el decremento ha dado como resultado 0, entonces la instrucción 1 es saltada y se ejecuta la instrucción 2.

iorwf f,d (Inclusive OR W with f)

Esta instrucción realiza una operación lógica OR entre los bits del registro W y el registro f. El resultado se almacena en el destino definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1.

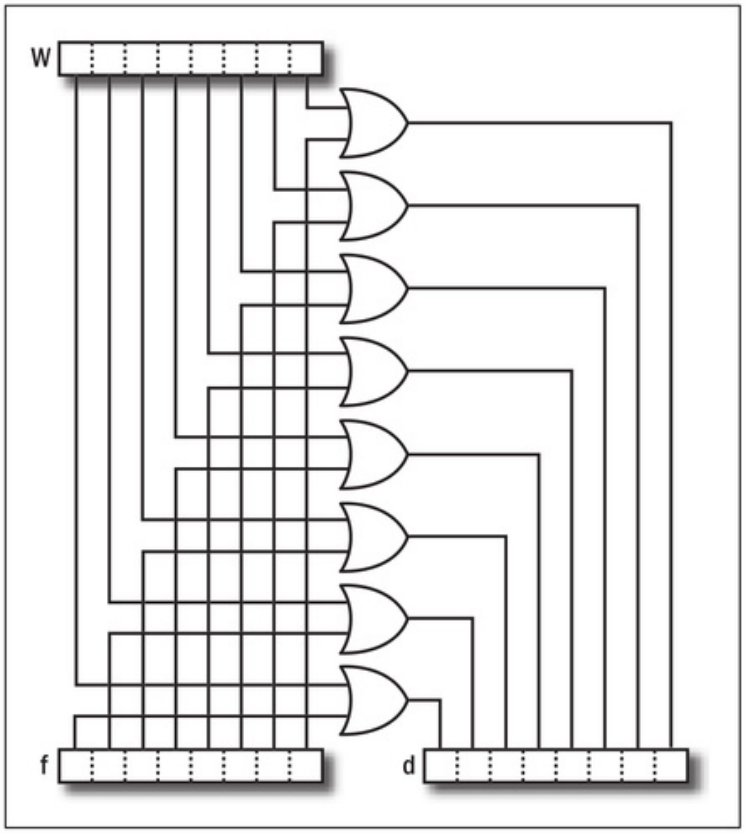


Figura 17. *Se puede hacer una operación OR entre los bits de W y cualquier registro con la instrucción iorwf.*

Ejemplo:

```
iorwf PORTA, 0
```

Se realiza la operación OR entre los bits del registro W y el registro PORTA y el resultado se almacena en el registro W.

movf f,d (Move f)

Como su nombre lo indica, la instrucción **movf** mueve el contenido de un registro a otro. El resultado se almacena en el destino d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado ha sido cero se activa la bandera de cero.

Ciclos de máquina: 1.

Ejemplo:

```
movf PORTB, 0
```

El contenido del registro PORTB se mueve al registro W, dado que el destino es 0.

movwf f (Move W to f)

Esta instrucción mueve el contenido del registro W al registro f.

Banderas afectadas: ninguna

Ciclos de máquina: 1.

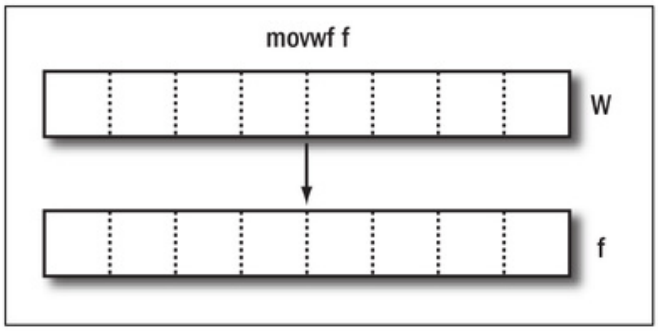


***Figura 18.** La instrucción movwf nos permite mover el contenido de W a cualquier otro registro.*

Ejemplo:

```
movwf PORTB
```

El contenido del registro W es transferido al registro PORTB.

nop (No operation)

Esta instrucción esencialmente no hace nada, sólo gasta un ciclo de máquina.

Banderas afectadas: ninguna

Ciclos de máquina: 1.

Ejemplo:

```
[instrucción 1]
nop
[instrucción 2]
```

No hay ninguna operación. En este ejemplo, la instrucción **nop** retrasa la ejecución de la instrucción 2 durante un ciclo de máquina. Precisamente, la instrucción **nop** se usa para agregar pequeños retardos en nuestros programas si lo necesitamos.

rlf f,d (Rotate left f through carry)

Esta instrucción rota el registro f a la izquierda una vez, pasando por el bit de acarreo (C) del registro STATUS. Es decir, el bit de acarreo pasa al bit menos significativo y el bit más significativo pasa al acarreo. El resultado se almacena en el registro definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

C: el bit de acarreo es rotado en la instrucción.

{} ¿MOVF F, 1?

Si ejecutáramos la instrucción **movf f, 1** parecería inútil, ya que el contenido del registro f se almacena en el mismo registro f. Esta instrucción puede ser útil para comprobar si el valor de un registro es 0 o no. Por ejemplo, si el contenido de f es 0, al ejecutar la instrucción se activará el bit Z del registro STATUS y así sabremos que f contiene 0.

Ciclos de máquina: 1.

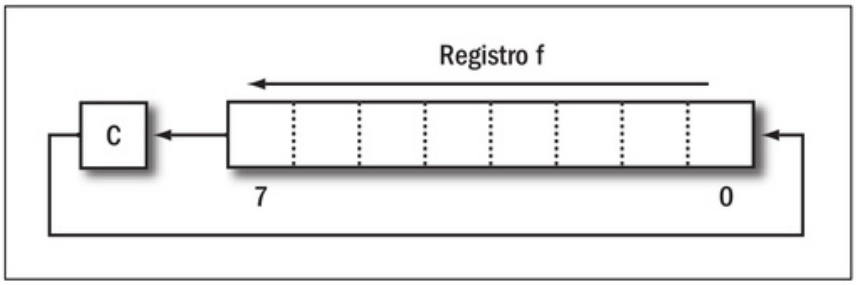


Figura 19. *La instrucción rlf rota los bits del registro f e incluye el bit de acarreo C en la rotación.*

Ejemplo:

```
rlf PORTB, 1
```

Se rota el registro PORTB una vez a la izquierda y el resultado se almacena en el mismo registro PORTB.

rrf f,d (Rotate right f through carry)

Esta instrucción rota el registro f a la derecha (*right*) una vez, pasando por el bit de acarreo (C) del registro STATUS. Es decir, el bit de acarreo pasa al bit más significativo y el bit menos significativo pasa al acarreo. El resultado se almacena en el registro definido por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

C: el bit de acarreo es rotado en la instrucción.

Ciclos de máquina: 1.

III LA ROTACIÓN Y EL BIT DE ACARREO

Cuando necesitamos hacer una rotación de algún registro no debemos olvidarnos de que el bit de acarreo (C) del registro STATUS interviene en la rotación, por lo que debemos poner atención en ello y poner este bit a 1 ó 0, según necesitemos antes de hacer la rotación de algún registro.

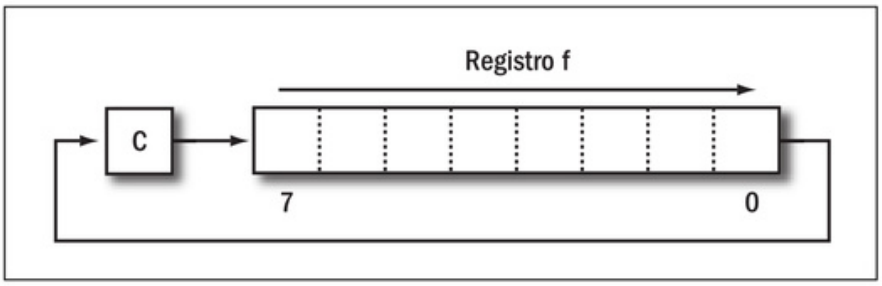


***Figura 20.** La instrucción rrf rota los bits del registro f e incluye el bit de acarreo C en la rotación.*

Ejemplo:

```
rrf PORTB, 1
```

Como vemos en el ejemplo, se rota el registro PORTB una vez a la derecha, y el resultado se almacena en el mismo registro PORTB.

subwf f,d (Substract W from f)

Esta instrucción lleva a cabo una resta entre el contenido del registro f y el del registro W (**f – W**), y el resultado se almacena en el registro dado por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

C: si el resultado de la resta es positivo, este bit se pone a 1. Si ha sido negativo, a 0.

DC: si ha habido un acarreo del nibble bajo al nibble alto, se activa la bandera de acarreo de dígito.

Z: si el resultado de la resta ha sido 0, se activa la bandera de 0.

III ¿CERO EN INCREMENTO?

Recordemos que los registros de la memoria de datos y el registro W son de 8 bits, así que el valor máximo que pueden almacenar es $11111111_b = 255_d$, por lo que cuando se le hace un incremento a un registro que contiene el valor máximo (255_d), su contenido pasará de nuevo a cero: 00000000_b, y es cuando la instrucción **incfsz** cumple la condición del cero.

Ciclos de máquina: 1.

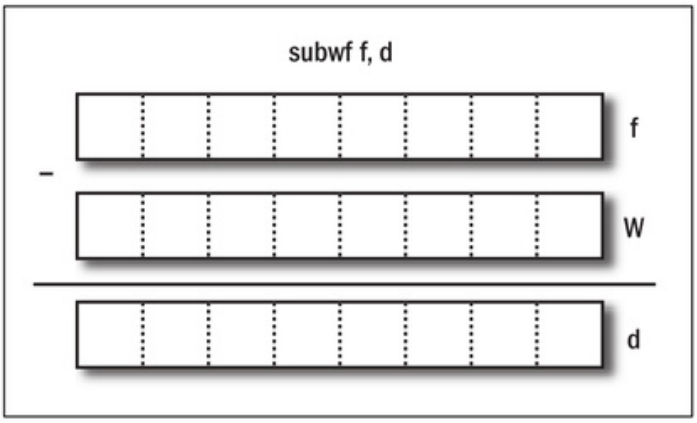


Figura 21. *La instrucción subwf resta el registro W del registro f.*

Ejemplo:

```
subwf cuenta, 1
```

Realiza la resta del contenido del registro llamado **cuenta**, menos el contenido del registro W. En este caso, el resultado se almacena en el propio registro cuenta.

swapf f,d (Swap nibbles in f)

Esta instrucción intercambia los nibbles alto y bajo del registro f. Es decir, cambia los cuatro bits más altos por los cuatro más bajos y viceversa. Almacena el resultado en el registro dado por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas: ninguna

Ciclos de máquina: 1.

{ } COMPATIBILIDAD DE LOS SETS

Los PICs de la gama baja sólo tienen 33 instrucciones. Cuando migramos hacia arriba, los demás sets de instrucciones son compatibles con los anteriores. Por ejemplo, el set de 35 instrucciones de la gama media contiene las 33 de la gama baja, y el set de 58 instrucciones de la gama alta, a su vez, contiene las 35 instrucciones de la gama media.

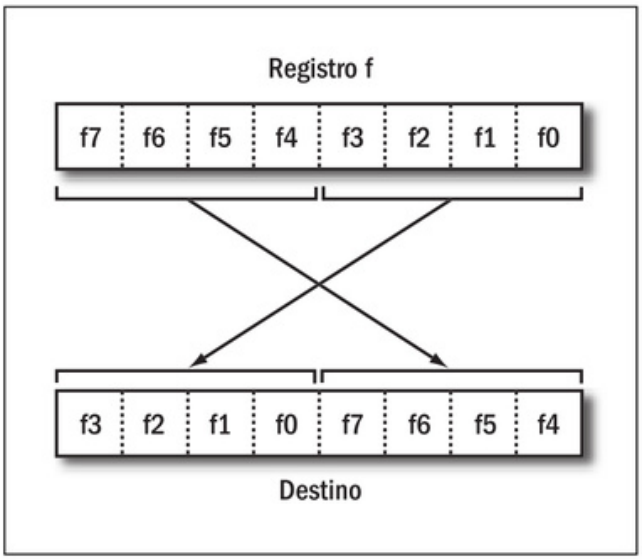


Figura 22. *La instrucción swapf intercambia los nibbles alto y bajo del registro f.*

Ejemplo:

```
swapf PORTB, 1
```

Se realiza el intercambio de los nibbles alto y bajo del registro PORTB y el resultado se almacena en el mismo registro PORTB.

xorwf f,d (Exclusive OR W with f)

Esta instrucción lleva a cabo una operación lógica XOR entre los bits de los registros W y f. Almacena el resultado en el registro dado por d.

Si **d = 0** el resultado se almacena en el registro W.
Si **d = 1** el resultado se almacena en el registro f.

Banderas afectadas:

Z: si el resultado de la operación ha sido 0, se activa la bandera de 0.

DESTINO PREDETERMINADO

En las instrucciones en donde se debe especificar el bit de destino (**d**) es común, en ocasiones, olvidarse de colocarlo. El programa ensamblador colocará de manera predeterminada el bit de destino **d** con un valor de 1, es decir, automáticamente se guardará el resultado de la operación en el registro **f,** por lo tanto, es importante no olvidar el bit de destino cuando se requiera.

Ciclos de máquina: 1.

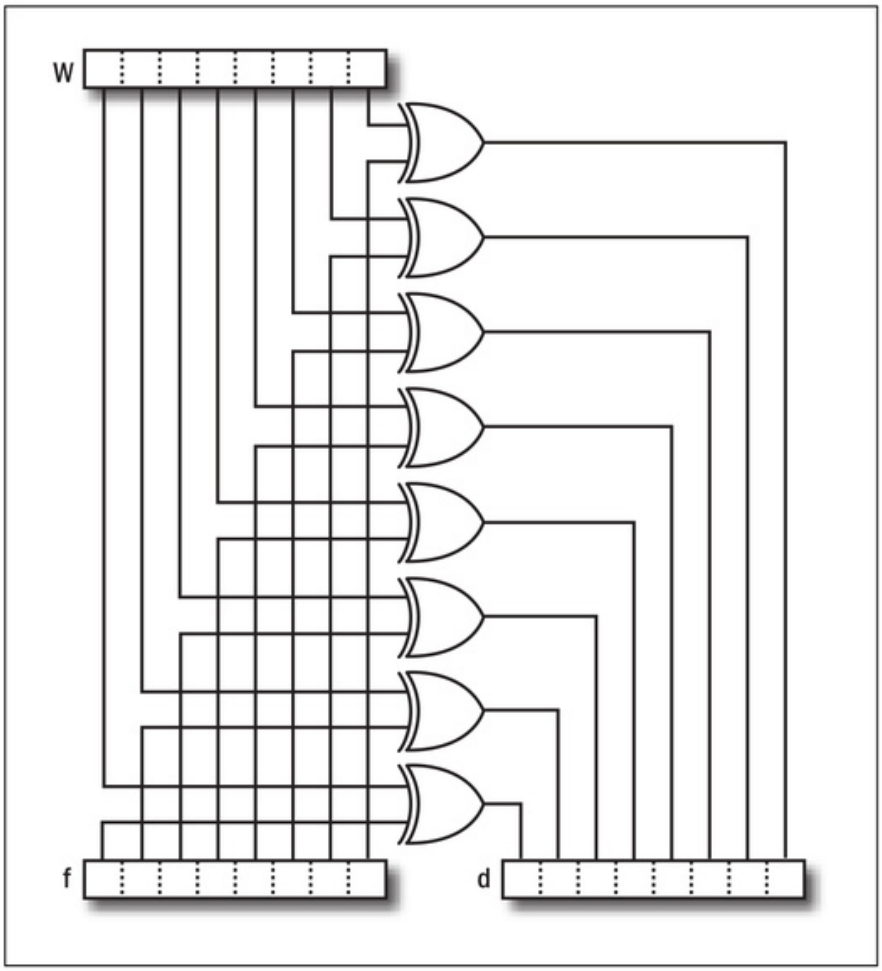


***Figura 23.** La instrucción xorwf realiza una operación lógica XOR entre W y f.*

Ejemplo:

```
xorwf temp, 1
```

Realiza la operación XOR bit a bit entre el registro W y el registro llamado **temp**. El resultado se almacena, en este caso, en temp.

III ARQUITECTURAS DE LOS PIC

Las arquitecturas de los PIC son: x12, x14, x16 y x16 enhanced, lo que significa la longitud en bits de las instrucciones que ejecutan. La arquitectura x12 tiene un set de 33 instrucciones de 12 bits, la x14 de 35 instrucciones de 14 bits, la x16 de 58 instrucciones de 16 bits, y la x16 enhanced de 77 instrucciones de 16 bits. El PIC16F84A es de la familia x14.

Operaciones orientadas a bits

Las **operaciones orientadas a bits** son cuatro y están destinadas al manejo de un solo bit de cualquier registro de la memoria de datos.

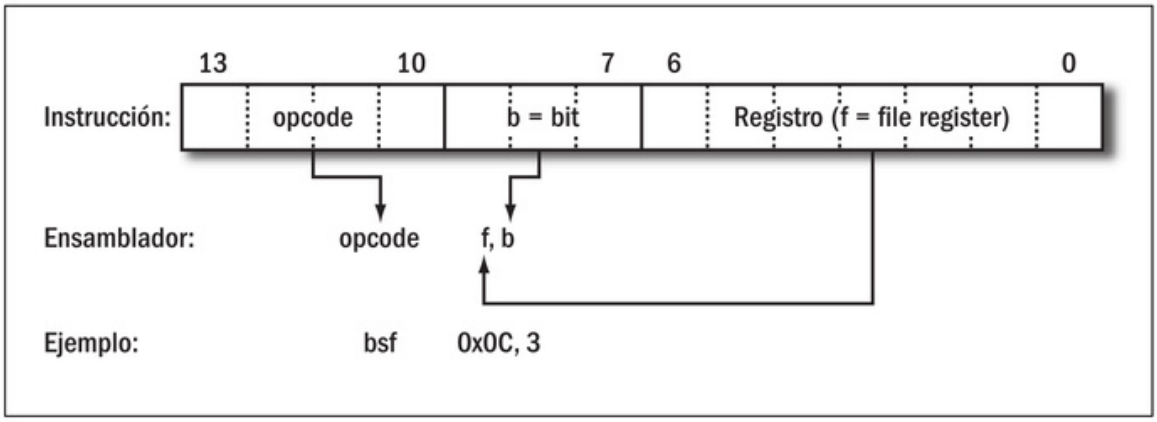


Figura 24. *Las instrucciones orientadas a bits trabajan con un solo bit de cualquier registro.*

En esta ocasión, el código de operación es de cuatro bits. Los bits del 7 al 9 indican cuál es el bit en el que la instrucción va a operar. Se requiere de tres bits para seleccionar mediante su número cualquier bit del registro en el cual se va a operar, que debe ser un número entre 0 y 7. Nuevamente, los 7 bits más bajos (0 a 6) indican la dirección del registro sobre el cual tendrá efecto la instrucción. Como mencionamos al principio, las operaciones orientadas a bits son sólo cuatro. Veamos cuáles son.

bcf f,b (Bit clear f)

Esta instrucción borra (pone a 0) el bit b del registro indicado por f.

Banderas afectadas: ninguna.

Ciclos de máquina: 1.

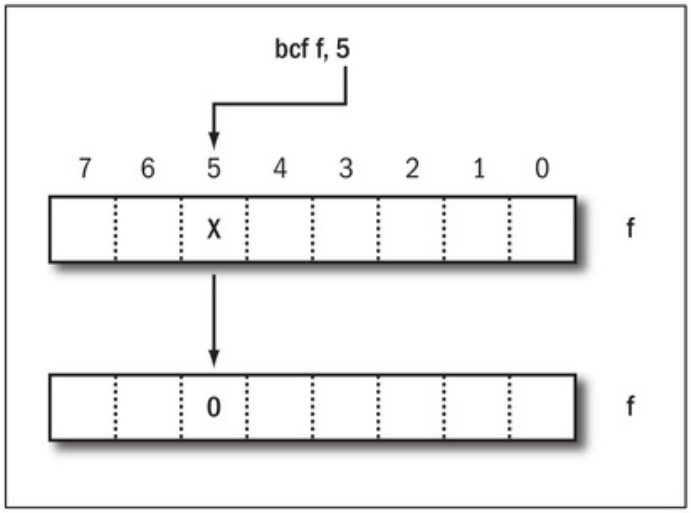


Figura 25. *Ejemplo del funcionamiento de la instrucción bcf.*

Ejemplo:

```
bcf STATUS, 2
```

Se borra o pone a 0 el bit número 2 del registro STATUS. En el ensamblador, la dirección del bit se pone en decimal. Es importante recordar que la numeración de los bits comienza desde 0, por lo que en este ejemplo se está operando con el tercer bit del registro que corresponde al bit C o acarreo.

bsf f,b (Bit set f)

Esta instrucción activa (pone a 1) el bit b del registro indicado por f.

Banderas afectadas: ninguna.

Ciclos de máquina: 1.

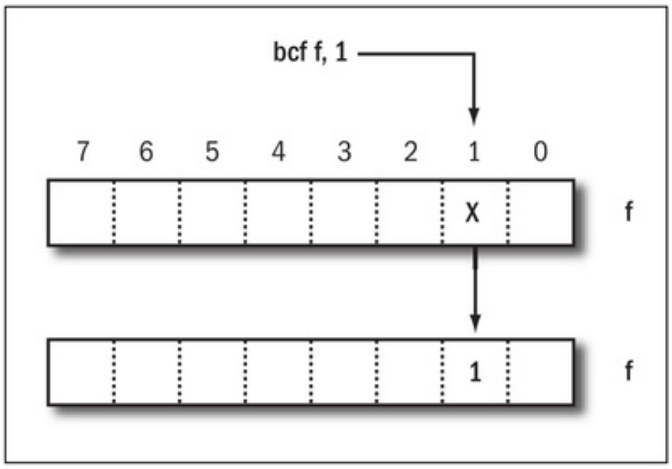


Figura 26. *Ejemplo del funcionamiento de la instrucción bsf.*

Ejemplo:

```
bsf TRISB, 7
```

Se activa o pone a 1 el bit número 7 (el bit más significativo) del registro TRISB.

btfsc f,b (Bit test f, skip if clear)

Esta instrucción realiza una verificación del estado del bit b del registro f. Si este bit es 1, ejecuta la instrucción siguiente y, si es 0, salta la instrucción siguiente.

Banderas afectadas: ninguna.

Ciclos de máquina: 1 cuando la condición no ha sido cumplida y no hay salto, y 2 cuando la condición se cumple y se realiza el salto.

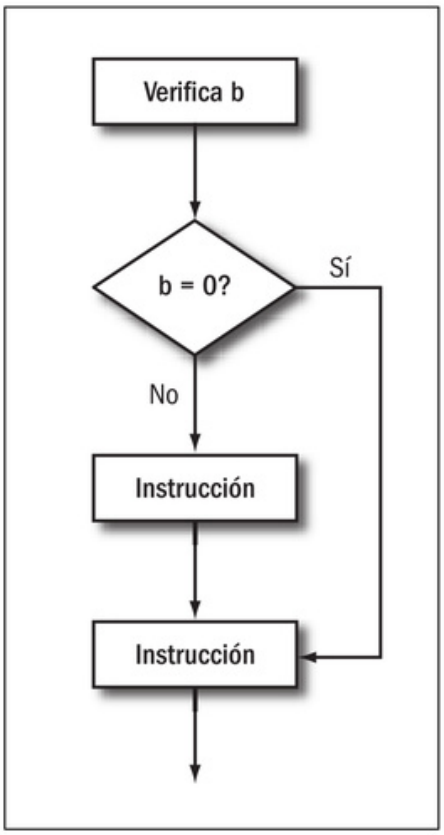


Figura 27. *La instrucción btfsc hace un salto en función de si el bit b es cero.*

Ejemplo:

```
btfsc PORTB, 5
[instrucción 1]
[instrucción 2]
```

Verifica el estado del bit número 5 del registro PORTB. Si el bit es 1, ejecuta la instrucción 1, y si es 0 salta y ejecuta la instrucción 2.

btfss f,b (Bit test f, skip if set)

Esta instrucción realiza una verificación del estado del bit b del registro f. Si este bit es 0, ejecuta la instrucción siguiente, y si es 1 salta la instrucción siguiente.

Banderas afectadas: ninguna.

Ciclos de máquina: 1 cuando la condición no ha sido cumplida y no hay salto, y 2 cuando la condición se cumple y se realiza el salto.

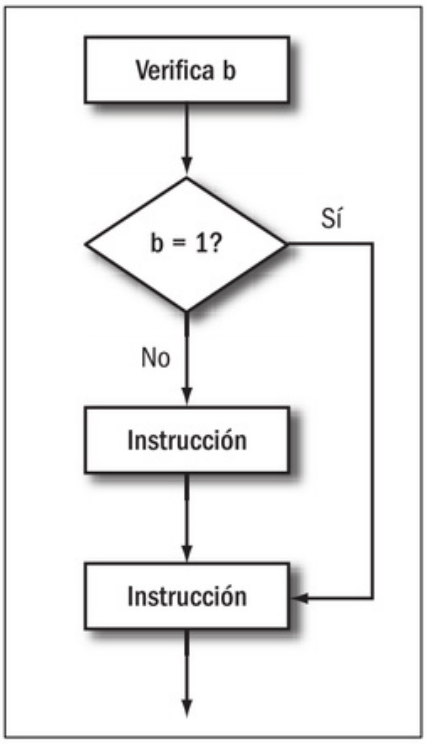


***Figura 28.** La instrucción btfss hace un salto en función de si el bit b es uno.*

Ejemplo:

```
btfss PORTB, 7
[instrucción 1]
[instrucción 2]
```

Verifica el estado del bit número 7 del registro PORTB. Si el bit es 0, ejecuta la instrucción 1, y si es 1 salta y ejecuta la instrucción 2.

Operaciones orientadas a literales y de control

El último grupo de operaciones que estudiaremos en este capítulo pertenece a las **instrucciones de literales**, es decir, en las cuales se opera sobre valores dados por

III HEXADECIMAL PREDETERMINADO

Si no se especifica ningún formato para los valores numéricos, el ensamblador tomará en forma predeterminada el hexadecimal, así que será importante indicar cuál es el formato cuando necesitemos establecer valores en decimal, binario, octal o ASCII, para evitar errores o mal funcionamiento del programa. Aun en hexadecimal, debemos indicarlo adecuadamente para mayor claridad.

la propia instrucción. Lo que debemos saber es que la literal se representa por la letra **k** y es un valor fijo sobre el cual actuará la instrucción. Además, en este grupo se incluyen algunas instrucciones de control.

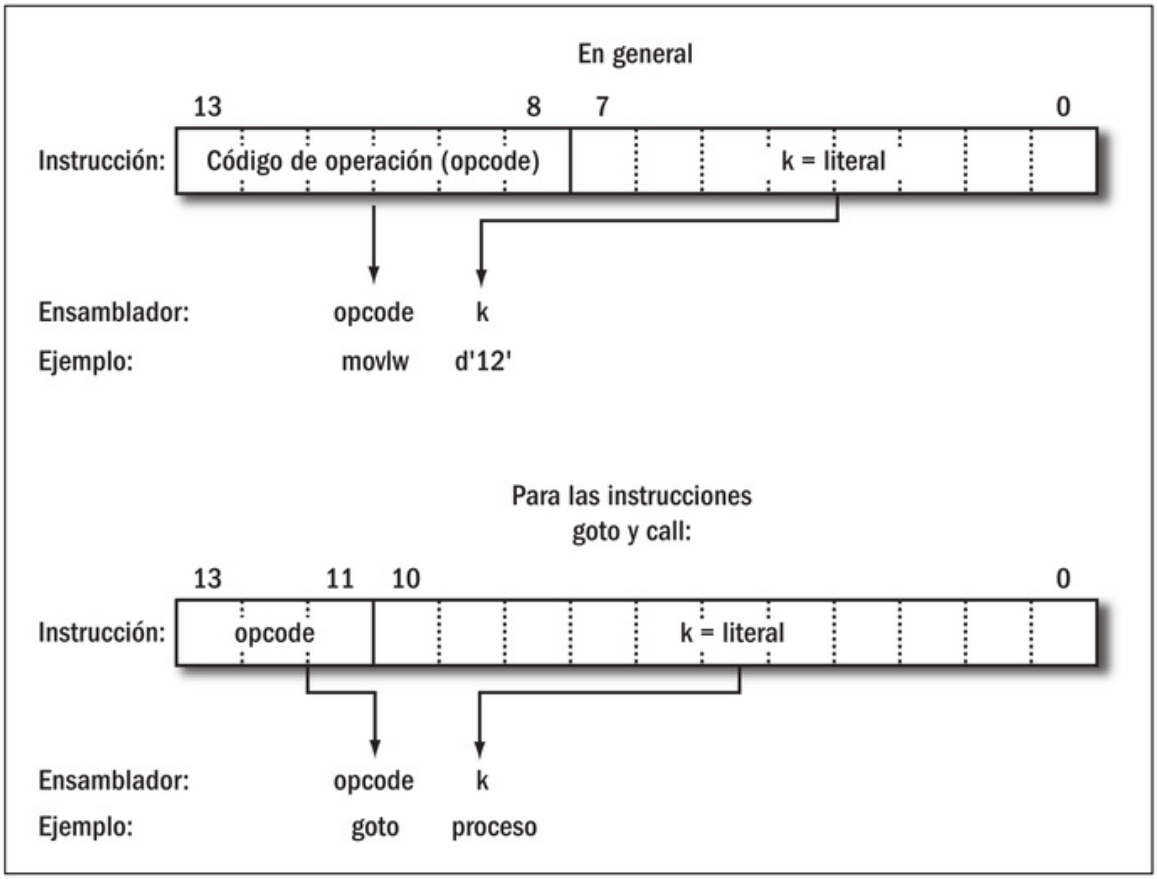


***Figura 29.** Las instrucciones de literales tienen un formato general. Las instrucciones goto y call tienen un formato especial.*

En este grupo de instrucciones se opera con literales o constantes, es decir, con valores fijos definidos en la propia instrucción. En general, ahora el código de operación es de 6 bits, y los 8 bits más bajos (0 a 7) de la instrucción corresponden precisamente a la literal o constante sobre la cual se va a operar. En las instrucciones **goto** y **call** se requiere de una dirección de la memoria de programa, ya que estas instrucciones son de salto, por lo que para poder direccionar los saltos se requiere de los 11 bits más bajos donde estará la dirección de dicho salto, quedando el código de operación de sólo tres bits en estos casos. En teoría, se puede acceder a 2048 direcciones con 11 bits aunque, como sabemos, el PIC16F84A sólo tiene 1024.

Un dato importante que debemos conocer es que cuando utilizamos estas instrucciones manejaremos literales o constantes que debemos escribir de alguna forma en particular, ya sea en valores binarios, decimales, hexadecimales, etcétera. Para ello, debemos seguir un formato determinado, no aleatorio. En la **Tabla 2** resumimos los formatos para las constantes.

CONSTANTE	FORMATO	EJEMPLO
Decimal	D'<cantidad_decimal>'	D'125'
	d'<cantidad_decimal>'	d'125'
	. <cantidad_decimal>	.125
Binario	B'<cantidad_binaria>'	B'01011010'
	b'<cantidad_binaria>'	b'01011010'
Hexadecimal	H'<cantidad_hexadecimal>'	H'2F'
	h'<cantidad_hexadecimal>'	h'2F'
	0x<cantidad_hexadecimal>	0x2F
Octal	O'<cantidad_octal>'	O'122'
	o'<cantidad_octal>'	o'122'
ASCII	A'<caracter>'	A'G'
	a'<caracter>'	a'G'
	'<caracter>'	'G'

***Tabla 2.** Formato para las constantes en lenguaje ensamblador.*

De esta manera, podemos indicar un número o constante para escribir los operandos en estas instrucciones en el ensamblador. Para los valores hexadecimales que comiencen con una letra, es mejor agregar un 0 al principio para evitar que puedan ser confundidos con etiquetas. Por ejemplo, en lugar de A3h usaremos 0A3h. Veamos cuáles son las instrucciones de este grupo.

addlw k (Add literal and W)

Esta instrucción suma el contenido del registro W con el de la literal o constante definida por k y almacena el resultado en el registro W.

Banderas afectadas:

C: si el resultado de la suma ha sido mayor a FF entonces se activa el bit de acarreo.

DC: significa que si ha habido un acarreo del nibble bajo al nibble alto, se activa la bandera de acarreo de dígito.

III ETIQUETAS

Las direcciones de la memoria de programa a donde se realizarán los saltos en las instrucciones de salto como **call** o **goto** se definen en el ensamblador mediante etiquetas. De esta forma, es muy fácil escribir los programas, ya que el ensamblador generará automáticamente las direcciones correctas al momento de ensamblar. Más adelante lo veremos en detalle.

Z: si el resultado de la suma ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1

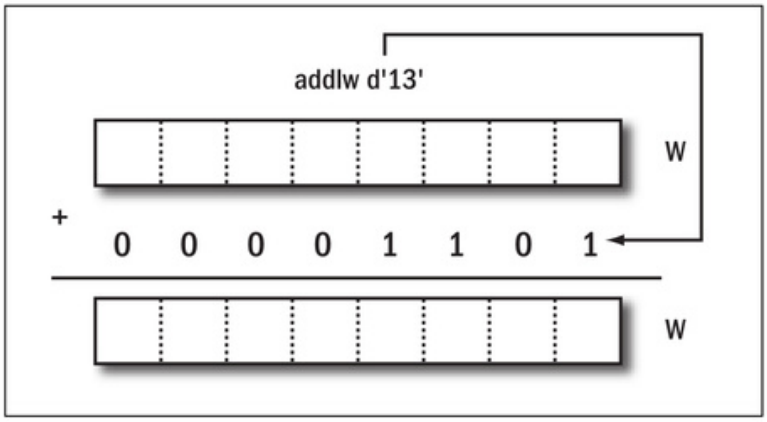


***Figura 30.** Ejemplo de la instrucción addlw, que suma el contenido de W y la literal.*

Ejemplo:

```
addlw d'1F'
```

Suma el contenido del registro W y el número 1Fh. En este caso, como podemos apreciar, el valor está indicado en hexadecimal.

andlw k (AND literal with W)

Esta instrucción realiza una operación AND bit a bit con el contenido del registro W y el de la literal o constante definida por k y almacena el resultado en el registro W.

Banderas afectadas:

Z: si el resultado de la operación ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1

Ejemplo:

```
andlw b'10011010'
```

Realiza una operación AND entre los bits del registro W y el valor de la constante 10011010. En este caso se indica en binario.

iorlw k (Inclusive OR literal with W)

Esta instrucción realiza una operación OR bit a bit con el contenido del registro W y el de la literal o constante definida por k y almacena el resultado en el registro W.

Banderas afectadas:

Z: si el resultado de la operación ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1

Ejemplo:

```
iorlw h'A8'
```

Realiza una operación OR entre los bits del registro W y el valor de la constante A8. En este caso se indica en hexadecimal.

movlw k (Move literal to W)

Esta instrucción mueve el valor dado por la literal k al registro W.

Banderas afectadas: ninguna.

Ciclos de máquina: 1

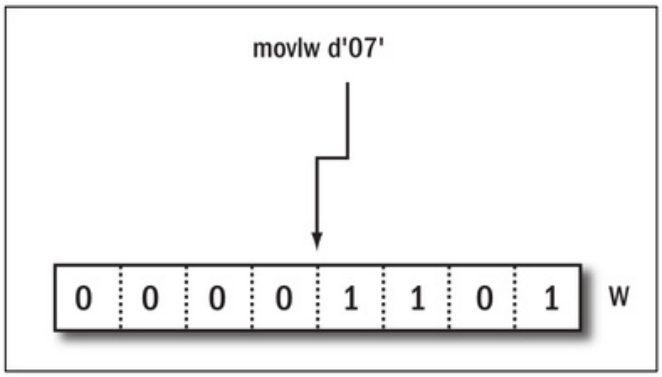


***Figura 31.** La instrucción movlw carga el registro W con el valor de la literal.*

Ejemplo:

```
movlw b'00010110'
```

Mueve el valor binario 00010110 al registro W sobrescribiendo su contenido.

sublw k (Substract W from literal)

Esta instrucción realiza una resta entre el contenido del registro W y el valor de la literal k (k – W), y almacena el resultado en W.

Banderas afectadas:

C: si el resultado de la resta es positivo, este bit se pone a 1, y si ha sido negativo, a 0.

DC: si ha habido un acarreo del nibble bajo al nibble alto, se activa la bandera de acarreo de dígito.

Z: si el resultado de la resta ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1

Ejemplo:

```
sublw .12
```

Realiza la resta de 12 menos el valor contenido en W y almacena el resultado en W.

xorlw k (Inclusive OR literal with W)

La instrucción **xorlw** se encarga de realizar una operación XOR bit a bit con el contenido del registro W y el de la literal o constante definida por k, y almacena el resultado en el registro W.

Banderas afectadas:

Z: si el resultado de la operación ha sido 0, se activa la bandera de 0.

Ciclos de máquina: 1

Ejemplo:

```
xorlw b'11010011'
```

Al analizar en detalle el ejemplo podemos observar que la instrucción **xorlw** realiza una operación XOR entre los bits del registro W y el valor de la constante 11010011, y almacena el resultado en W.

goto k (Go to adress)

Este es una instrucción de salto incondicional. Cada vez que el programa la encuentra, salta a la dirección dada por k. En este caso, k representa una dirección de la memoria de programa que es a donde el programa saltará.

Banderas afectadas: ninguna.

Ciclos de máquina: 2.

Ejemplo:

```
      goto proceso_2
      .
      .
      .

proceso_2
      .
      .
      .
```

Como podemos apreciar, al ejecutar esta instrucción, el programa saltará a la dirección de la memoria de programa definida por la etiqueta **proceso_2** y continuará la ejecución del programa desde esa nueva dirección.

call k (Call subroutine)

Esta instrucción realiza también un salto a la dirección apuntada por k. La diferencia es que llama a una subrutina, en la cual debe haber una instrucción de regreso para continuar la ejecución del programa en la dirección siguiente. En capítulos posteriores hablaremos en detalle de las subrutinas.

Banderas afectadas: ninguna.

Ciclos de máquina: 2.

return (Return from subroutine)

Mediante esta instrucción se regresa de una subrutina llamada previamente por la instrucción call k. La instrucción **return** salta a la siguiente instrucción después de **call**.

Banderas afectadas: ninguna.

Ciclos de máquina: 2.

Ejemplo de call/return:

```
      .
      .
      .
      call proceso
      nop
      .
      .
proceso
      .
      .
      .
      return
```

Cuando el programa llega a la instrucción **call**, llama a la subrutina proceso y salta hasta ella, ejecuta las instrucciones siguientes a la etiqueta proceso (que es una dirección de la memoria de programa) y cuando encuentra la instrucción **return** salta a la instrucción inmediata después de la instrucción **call**. En este caso, la instrucción **nop**.

retlw k (Return with literal in W)

Esta instrucción regresa también de una subrutina llamada con **call**, con la diferencia que pone un valor definido por k en el registro W antes de regresar.

Banderas afectadas: ninguna.

Ciclos de máquina: 2.

Ejemplo:

```
      .
      .
      .
      call proceso
      nop
      .
      .
```

```
proceso
      .
      .
      .
      retlw d'55'
```

Esencialmente es el mismo que el ejemplo anterior, sólo que esta vez la instrucción **retlw** pone el valor decimal 55 en el registro W y luego regresa a la instrucción **nop**.

retfie (Return from interrupt)

También es una instrucción de retorno de subrutina, pero esta vez se debe usar cuando la subrutina es provocada por una interrupción. En un capítulo posterior hablaremos detalladamente de las interrupciones.

Banderas afectadas: ninguna.

Ciclos de máquina: 2.

clrwdt (Clear watch dog timer)

Esta instrucción borra (pone a 0) el contenido del temporizador de perro guardián (*Watch dog timer*). Posteriormente estudiaremos en detalle qué es el WDT.

Banderas afectadas:

TO': este bit se pone a 1.

PD': este bit se pone a 1.

Ciclos de máquina: 1.

sleep (Go into stanby mode)

Mediante esta instrucción, el microcontrolador entra en el modo de **bajo consumo** (*Standby*) en el cual el oscilador se detiene. Esto sirve para ahorrar energía ya que al entrar en este modo el microcontrolador sólo consume unos pocos microamperes. Para "despertar" de nuevo el microcontrolador, debemos hacerlo por medio de un reset o una interrupción. En un capítulo posterior hablaremos más de este modo **sleep**.

Banderas afectadas:

TO': este bit se pone a 1.

PD': este bit se pone a 0.

Ciclos de máquina: 1.

Éstas son las 35 instrucciones que componen el repertorio para los microcontroladores con arquitectura x14, es decir, con instrucciones de 14 bits. Son fáciles de usar y la mayoría sólo requiere un ciclo de máquina para ejecutarse. En los capítulos siguientes hablaremos más a fondo de algunas para entender completamente su funcionamiento.

Ahora que conocemos las instrucciones para nuestro microcontrolador, en el próximo capítulo estudiaremos dónde las escribiremos, es decir, el uso del software MPLAB.

RESUMEN

En este capítulo hemos estudiado el repertorio de instrucciones del PIC16F84A, compuesto de sólo 35 instrucciones, para así poder escribir nuestros programas, y cómo funcionan en detalle la mayoría de ellas. En capítulos posteriores estudiaremos más detalles de algunas de las instrucciones. En el siguiente, abordaremos el estudio del programa MPLAB, donde escribiremos nuestro código fuente y lo ensamblaremos.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es el lenguaje máquina?

2 ¿Qué es un mnemónico?

3 ¿Qué es el lenguaje ensamblador?

4 ¿El lenguaje ensamblador es un lenguaje de alto o de bajo nivel?

5 ¿Cómo se llama el proceso de traducir los mnemónicos a lenguaje máquina?

6 ¿Cuántos ciclos de reloj componen un ciclo de máquina en el PIC16F84A??

7 ¿Cuántas instrucciones componen el repertorio del PIC16F84A?

8 ¿Para qué sirve el bit de destino (d)?

9 ¿Para qué sirve la instrucción incf?

10 ¿Para qué sirve la instrucción goto?

Capítulo 4

El entorno de desarrollo MPLAB IDE

Ahora que conocemos el repertorio de instrucciones de nuestro microcontrolador, necesitaremos saber dónde colocaremos esas instrucciones para formar el código fuente de nuestros programas, para luego poder ensamblarlos. El entorno de desarrollo MPLAB contiene las herramientas para escribir y ensamblar los programas que diseñemos para nuestro PIC, por lo que en este capítulo nos dedicaremos a estudiarlo.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

INTRODUCCIÓN A MPLAB

En este capítulo aprenderemos a utilizar el entorno **MPLAB IDE** (IDE = *Integrated Development Environment*), un software que cuenta con todas las herramientas necesarias para desarrollar los programas para nuestro microcontrolador. Este programa funciona bajo plataformas Windows y podemos descargarlo de forma gratuita desde la página web del fabricante de los microcontroladores PIC, Microchip: **www.microchip.com**. Al abrir la página principal de Microchip debemos hacer clic en el enlace **MPLAB® IDE**, que nos llevará a la página de descarga. Una vez que lo bajamos a nuestra PC, lo instalamos como cualquier otro programa para Windows. Luego, ya estaremos listos para comenzar a utilizarlo.

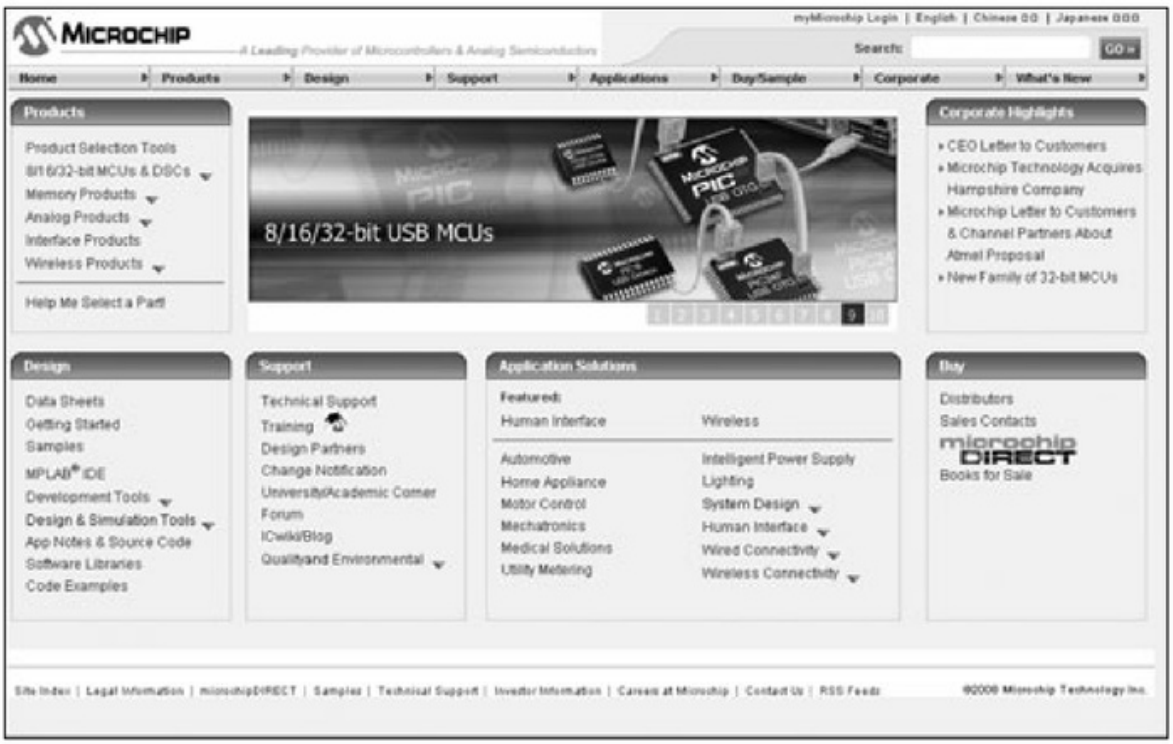


***Figura 1.** Desde la web de Microchip podemos descargar el programa MPLAB, las hojas de datos y otra información que nos puede ser de utilidad.*

Una vez instalado el programa, aparecerá en el **Escritorio** de Windows un acceso directo con el nombre **MPLAB IDE v8.30**, con el que podemos abrir la aplicación. Hablaremos aquí de la versión **8.30**, que es la más reciente al momento de escribir este libro. Las características principales que utilizaremos en este programa serán: el **editor de texto**, que nos servirá para escribir nuestros códigos fuente, el ensamblador llamado **MPASM**, que se encargará de ensamblar nuestros programas, es decir, de generar el código máquina que luego nos permitirá grabarlo en la memoria de nuestro PIC16F84A, y un simulador llamado **MPLAB SIM**, el cual nos permitirá simular nuestros programas para asegurarnos de que hagan lo que realmente deben hacer.

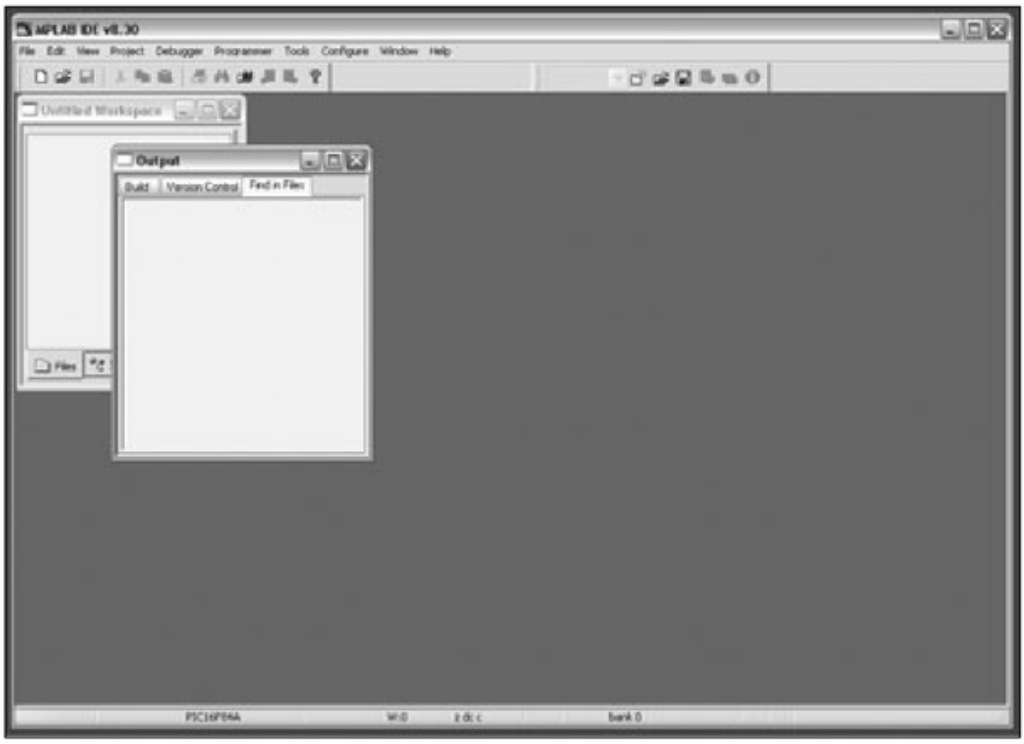


Figura 2. *Al abrir MPLAB IDE podemos ver la interfaz del programa y un par de ventanas pequeñas llamadas* ***Untitled workspace*** *y* ***Output****.*

Al abrir el programa por primera vez, aparecerá una ventana como la que se muestra en la **Figura 2**. Esta ventana es la interfaz principal. Desde ahí trabajaremos para comenzar, pero antes de hacerlo es conveniente llevar a cabo algunas configuraciones. El entorno MPLAB contiene las herramientas para trabajar con cualquier microcontrolador de la firma Microchip, así que debemos configurarlo según el PIC que vayamos a utilizar. Para ello iremos al menú **Configure/Select Device...** , donde se abrirá una pequeña ventana llamada **Select Device**, como vemos en la **Figura 3**.

Figura 3. *En la ventana* ***Select Device*** *elegiremos el microcontrolador que vamos a usar, en nuestro caso, el PIC16F84A.*

En esta ventana, elegiremos en el cuadro marcado **Device Family:** la opción **Mid-Range 8-bit MCUs (12/16/MCP)** y luego en el cuadro de selección marcado como **Device:** el **PIC16F84A**. Por último, hacemos clic en el botón **OK** para cerrar la ventana. Con esto ya tendremos configurado nuestro PIC. Si vamos a trabajar con otro PIC diferente en posteriores ocasiones, debemos repetir este procedimiento para elegir otro dispositivo.

CREAR UN NUEVO ARCHIVO FUENTE

Para comenzar a escribir un programa debemos crear un nuevo archivo fuente. Esto lo haremos desde el menú **File/New**, al presionar en nuestro teclado la combinación **CTRL+N**, o bien, al presionar el pequeño botón en la barra de herramientas que tiene la imagen de una hoja en blanco. Después de realizar cualquiera de estas acciones, aparecerá una ventana con el título **MPLAB IDE Editor**, que es donde escribiremos nuestro programa. Si deseamos, podemos maximizarla para verla en la pantalla completa de la ventana del programa.

***Figura 4.** MPLAB incluye un editor de texto llamado MPLAB IDE Editor para poder escribir nuestros códigos fuente en él.*

Pero antes de comenzar a trabajar con el editor debemos guardar el archivo fuente como un archivo con la extensión **.asm**. Para eso, vamos al menú **File/Save As...**,

CUIDADO CON LAS RUTAS...

La ruta del archivo fuente no debe exceder los 62 caracteres, y debemos guardar el archivo en una carpeta cercana a la raíz de la unidad donde trabajemos. Por ejemplo, podemos crear una carpeta **C:/ProyectosPIC** o alguna similar, ya que si la ruta excede los 62 caracteres máximos, al momento de intentar ensamblar el programa nos dará un error y no se llevará a cabo el procedimiento.

donde se abrirá el cuadro de diálogo para elegir en qué carpeta lo guardaremos. Tendremos que escribir el nombre que le daremos a nuestro archivo fuente. Debemos recordar poner el nombre y la extensión **.asm**, por ejemplo, **Mi archivo fuente.asm** y guardarlo en la carpeta elegida.

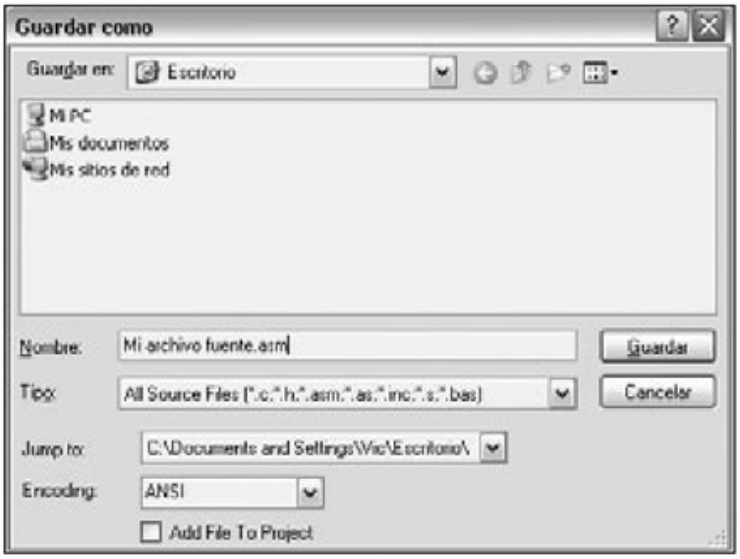


***Figura 5.** Debemos guardar nuestros archivos como .asm, que es la extensión para los archivos de código fuente en ensamblador.*

Es conveniente que el editor numere en forma automática las líneas de nuestro código para trabajar más cómodamente y localizar los errores de forma más rápida y fácil. Para activar esta opción vamos al menú **Edit/Properties...** para abrir el cuadro de diálogo **Editor Properties**, donde iremos a la pestaña llamada **File Type**, marcaremos la opción **Line Numbers** y presionaremos el botón **Aceptar**. Ahora sí ya estamos listos para comenzar a escribir nuestros códigos fuente en MPLAB. Veamos cómo hacerlo.

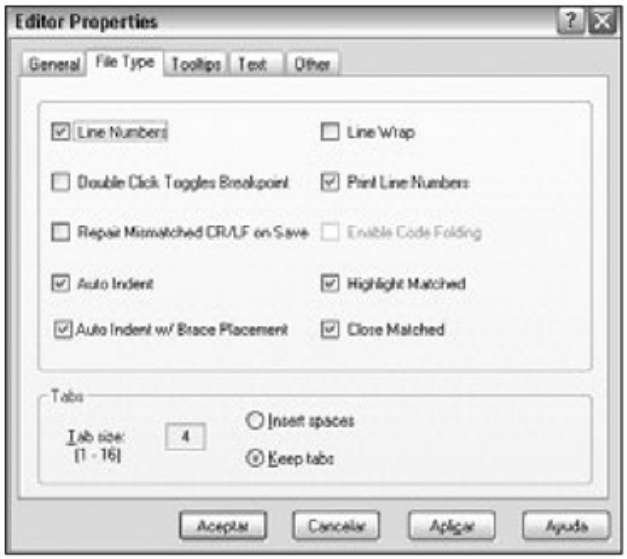
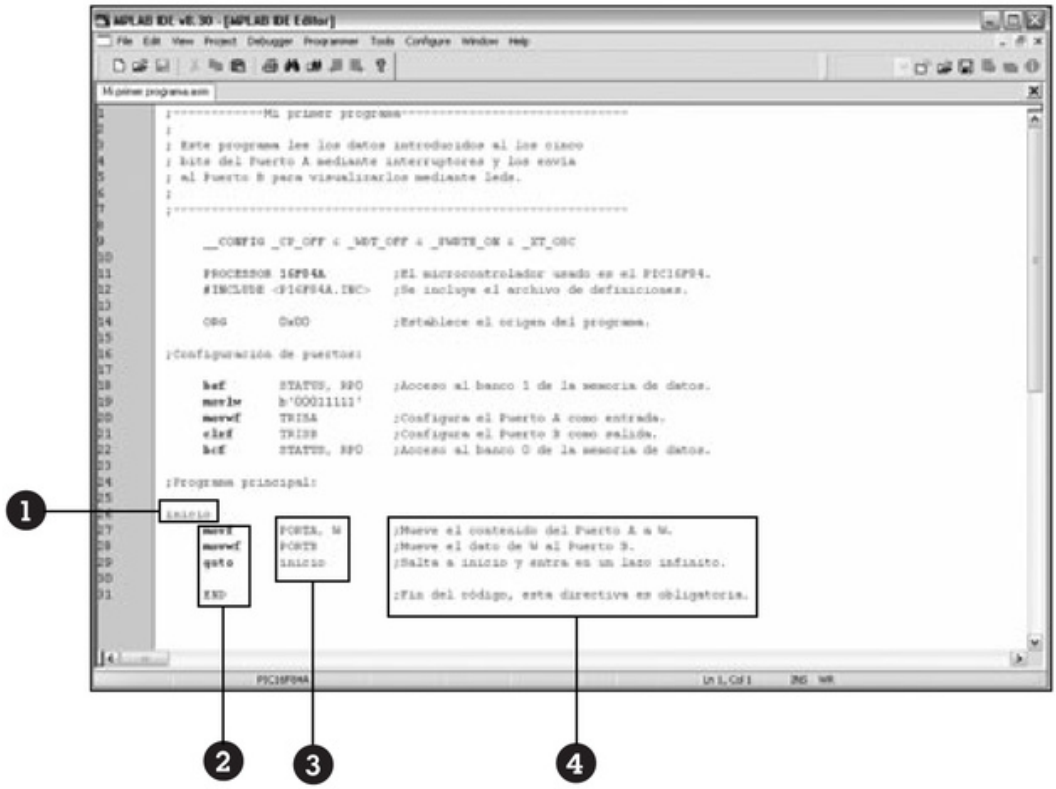


***Figura 6.** Es conveniente activar la numeración automática de las líneas de nuestro código fuente si es que no está activada de manera predeterminada.*

El formato del código fuente

Para que el ensamblador pueda reconocer correctamente los mnemónicos y realizar el ensamblado, debemos seguir un formato para escribir nuestros programas. Esto nos permitirá, además, lograr mayor claridad para nosotros mismos o alguien más que intente entender, corregir, o modificar el programa. En esencia, el código fuente está compuesto por una serie de líneas donde escribiremos las instrucciones y demás elementos. Cada una de estas líneas puede tener uno o más de los elementos o campos que describimos a continuación.

El formato del código fuente GV



❶ **Campo de etiquetas**: en el primer campo colocaremos las etiquetas que marcarán una línea de nuestro programa. Esencialmente, se usan para definir los saltos mediante un nombre para que las instrucciones de salto sepan dónde hacerlo. En este ejemplo, la instrucción **goto inicio** saltará a la línea donde se encuentra la etiqueta **inicio**.

❷ **Campo de código**: en este campo escribiremos los códigos o mnemónicos de nuestras instrucciones. Se separa del anterior mediante tabulaciones o espacios.

❸ **Campo de operandos**: en este campo pondremos los operandos necesarios de nuestras instrucciones que definen los registros, bits, destinos y direcciones, según la instrucción. Se puede separar del anterior mediante tabulaciones o espacios.

❹ **Campo de comentarios**: en este último campo podemos colocar comentarios para documentar nuestros programas. Los comentarios deben comenzar con punto y coma (;) para que el ensamblador ignore todo lo que sea precedido del punto y coma al momento de ensamblar. Esto es útil para describir los procesos llevados a cabo por el programa. De esta forma, podremos hacer modificaciones o corrección de errores más fácilmente en ocasiones posteriores.

El editor MPLAB le coloca automáticamente diferentes colores a los elementos del código para mayor claridad. Por ejemplo, los códigos o mnemónicos son de color azul y los comentarios son de color verde.

Además del formato, debemos tomar en cuenta algunas otras reglas:

Para las etiquetas:

1. En primer lugar, las etiquetas siempre deben empezar con una letra. Los siguientes caracteres pueden ser letras, números o guión bajo (_), y no pueden contener espacios intermedios. Por ejemplo: **Proceso 1** no es una etiqueta válida, el ensamblador dará error al momento del ensamblado. Una etiqueta válida sería, por ejemplo, **Proceso_1** o **Proceso1**.
2. No debemos poner tabuladores ni espacios antes de una etiqueta, ya que si lo hacemos el ensamblador intentará tomarla como una instrucción.
3. No podemos usar palabras reservadas para instrucciones o directivas, como por ejemplo, **goto**, **sleep**, **END** como etiquetas. Tampoco podemos usar nombres reservados para registros del SFR, bits o banderas, como por ejemplo, **C** (que está reservado a la bandera de acarreo), STATUS, TRISA, etcétera.

Es conveniente, también, que los nombres de las etiquetas ayuden a entender el funcionamiento del programa, por ejemplo, si bien podemos colocar una etiqueta llamada **etiqueta1**, no nos brinda mucha información por sí misma, por lo que

III CAMBIAR COLORES DEL TEXTO

Si queremos cambiar el color de los textos y también la fuente del editor, debemos ir al menú **Edit/Properties...** mientras tenemos abierto algún archivo de código fuente, y en la ventana **Editor Properties** ir a la pestaña **Text**. Ahí podremos cambiarlos a nuestro gusto, o regresar los colores a la configuración predeterminada.

en su lugar podríamos usar **LeePuertoB**, que nos indica que estamos en una rutina para leer datos del puerto B, y de esa forma la etiqueta nos ayuda a tener más claridad. Las etiquetas representan direcciones de la memoria de programa, y el ensamblador colocará automáticamente las direcciones adecuadas al momento del ensamblado en los lugares donde encuentre las etiquetas. De esta forma, no tenemos que preocuparnos por hacerlo nosotros.

Para los campos:

Los campos pueden separarse con espacios o tabulaciones, por ejemplo, en el siguiente código están separados por tabulaciones:

```
inicio      movf    PORTA, W      ;Mueve el Puerto B a W.
            movwf   PORTB         ;Mueve el dato al Puerto A.
            goto    inicio        ;Reinicia el ciclo de lectura.
```

Sería lo mismo si lo escribiéramos así:

```
inicio      movf PORTA, W ;Mueve el Puerto B a W.
            movwf PORTB ;Mueve el dato al Puerto A.
            goto inicio ;Reinicia el ciclo de lectura.
```

En este segundo ejemplo sólo los campos de etiquetas y de códigos están separados por tabulaciones y los demás únicamente por un espacio. Es cuestión de gustos elegir una forma u otra. También podemos dejar líneas en blanco para dar mayor forma a nuestro código fuente, ya que el ensamblador también ignorará todas las líneas en blanco al momento de ensamblar. El primer campo siempre es para las etiquetas, el segundo para los mnemónicos, y el tercero para los operandos de las instrucciones. El cuarto es para los comentarios, aunque realmente éstos pueden colocarse en cualquier lugar, si los necesitamos.

UN TRUCO DE EDICIÓN...

Si estamos modificando o corrigiendo un programa, a veces necesitamos eliminar algunas líneas del código para hacer pruebas. Pero si las borramos, después será difícil recordarlas o escribirlas de nuevo. En su lugar, podemos colocar punto y coma al principio, para convertirlas en comentarios y así el ensamblador las ignorará sin tener que borrarlas.

Para los comentarios:

Como ya mencionamos, los comentarios deben ser precedidos por punto y coma, para que el ensamblador los ignore. De esta manera, podemos colocar los comentarios en cualquier lugar del código fuente donde los necesitemos. En el ejemplo de la **Guía visual: El formato del código fuente** podemos ver que hay comentarios en el primer campo. Si utilizamos varias líneas para escribir comentarios, cada una de ellas debe iniciar con punto y coma, como podemos observar en las primeras líneas del programa.

Figura 7. *Los comentarios son esenciales para colocar detalles del funcionamiento o instrucciones de cómo usar un programa.*

Los comentarios son importantes para documentar lo suficiente nuestro programa y hacer más fácil su interpretación para nosotros mismos o para alguien más que lo intente leer o modificar. En los ejemplos y códigos fuente de este libro haremos un uso muy amplio de los comentarios, de modo que podamos comprender bien el funcionamiento de los programas, ya que ése es nuestro fin.

III BUENAS COSTUMBRES

Es muy importante documentar los códigos fuente con comentarios suficientes que aclaren su funcionamiento. Al estar creando un programa tenemos muy claro cómo funciona, pero después de unas semanas o incluso días de no trabajar con él, puede ser indescifrable, aunque lo hayamos escrito nosotros mismos. Es allí donde los comentarios cobran una gran importancia.

DIRECTIVAS

Además de las instrucciones que ya estudiamos, también podemos usar **directivas** en nuestro código fuente. Las directivas son **palabras reservadas** o **comandos** que controlarán el proceso de ensamblado del código fuente. No debemos confundirlas con las instrucciones, ya que no tienen equivalente en código máquina, sino que sólo sirven para controlar el ensamblado al momento de realizarlo y, además, para facilitar la escritura del código fuente. Es por eso que la gran mayoría de las directivas son opcionales y por lo tanto podemos usarlas o no, es nuestra decisión. El ensamblador MPASM tiene muchas directivas diferentes (alrededor de 60), pero en esta sección estudiaremos sólo las principales, que son las más útiles. Estudiaremos otras en secciones o capítulos posteriores.

END (*end program block*)

Esta es la única directiva que es obligatoria. La directiva **END** le indica al ensamblador dónde finaliza el código, para que se detenga al momento de ensamblar. Es obligatoria ya que si no la colocamos, el ensamblador no sabrá dónde debe detenerse, nos dará un error, y el proceso de ensamblado no se llevará a cabo. Normalmente pondremos esta directiva después de la última línea de nuestro código fuente, pero podemos ponerla en cualquier otro lugar. Las líneas de código que estén después de la directiva **END** serán ignoradas y no se ensamblarán.

EQU (*define an assembler constant*)

Sintaxis: **[etiqueta] EQU exp**

Esta directiva sirve para asignar etiquetas a constantes u otros elementos. Lo que pongamos en **exp** será asignado a la etiqueta. Por ejemplo, podemos usar esta directiva para asignar nombres a registros de la memoria de datos:

```
contador    EQU 0x0C
            .
            .
            .
            incf contador, 1
```

En este ejemplo, el valor **0x0C** es asignado a la etiqueta **contador**. De esta forma, cada vez que escribamos **contador** en nuestro programa, el ensamblador lo sustituirá por el valor **0x0C**, que es la dirección del primer registro de la memoria de datos en

la sección GPR. Así, no tenemos que colocar la dirección, si no la etiqueta que le asignamos, lo cual hará más fácil usar y recordar cuál es la función de ese registro. En la instrucción **incf**, el registro **0x0C** será incrementado y el resultado se almacenará en él. Otro ejemplo del uso de esta directiva sería:

```
siete       EQU d'7'
            .
            .
            .
            addlw siete
            .
            .
            movlw siete
```

Para evitar escribir los valores directamente, podemos asignarlos a etiquetas y así hacer más fácil la escritura de los programas al usar etiquetas en lugar de direcciones o valores. Usualmente se asignan las etiquetas al principio, pero podemos hacerlo en cualquier parte del programa.

CBLOCK (*define a block of constants*) y ENDC (*end an automatic constant block*)

Sintaxis:

```
CBLOCK dir
[etiqueta 1]
[etiqueta 2]
.
.
.
ENDC
```

¿MAYÚSCULAS O MINÚSCULAS?

Normalmente, las directivas para el ensamblador se deben escribir en mayúsculas para que sean reconocidas correctamente. Además, esto nos ayuda a diferenciarlas de otros elementos como los mnemónicos, que deben ser escritos en minúscula. De todos modos, si escribimos las directivas en minúscula el ensamblador también las reconocerá.

Muchas veces necesitaremos definir más de una etiqueta para la asignación de nombres a los registros de la memoria de datos, y aunque podemos hacerlo mediante varias directivas **EQU**, también tenemos la posibilidad de realizarlo mediante las directivas **CBLOCK** y **ENDC**. La directiva comienza con **CBLOCK** y la dirección (**dir**) de origen para la asignación. Es decir, esta dirección se asignará a la etiqueta 1, la dirección siguiente se asignará a la etiqueta 2, y así sucesivamente, y el bloque se cerrará o terminará con **ENDC**. Veamos un ejemplo del uso de esta directiva:

```
    CBLOCK 0x0C
    uno
    dos
    tres
    ENDC
```

En este ejemplo, la dirección **0x0C** es asignada a la etiqueta **uno**, la dirección siguiente, **0x0D**, es asignada a la etiqueta **dos**, y la dirección **0x0E** es asignada a la etiqueta **tres**. Este ejemplo es equivalente a:

```
uno EQU 0x0C
dos EQU 0x0D
tres EQU 0x0E
```

Las directivas **CBLOCK** y **ENDC** son útiles cuando definimos muchas constantes en nuestro programa. También podemos poner las etiquetas en una sola línea separadas por comas. Si lo deseamos, podemos tener más de un bloque de constantes en nuestro programa. Si no especificamos ninguna dirección después de **CBLOCK**, el ensamblador tomará automáticamente la última dirección del bloque anterior y continuará desde ella. Por ejemplo, si colocamos otro bloque de constantes después del anterior, de esta forma:

```
    .
    .
    .
    CBLOCK
    cuatro
    cinco
    seis
    ENDC
```

La directiva **CBLOCK** no tiene dirección, por lo que se asignará la que sigue a la última del bloque anterior, y entonces se asignará el valor **0x0F** a la etiqueta **cuatro**, **0x10** a **cinco**, y así sucesivamente. Si no colocamos una dirección en el primer bloque se tomará de manera predeterminada la dirección **0x00**.

ORG (*set program origin*)

Sintaxis: **[etiqueta] ORG dir**

La directiva **ORG** le indica al ensamblador el origen de las instrucciones siguientes en la dirección de memoria de programa definida por **dir**. Es decir, le señalará al ensamblador a partir de qué dirección se pondrán las siguientes instrucciones de nuestro código. Esto es útil cuando deseamos especificarlo, como en el siguiente código:

```
        ORG 05h
        bsf STATUS, RP0
        .
        .
        .
```

La línea con la instrucción **bsf** se pondrá en la dirección de memoria de programa **05h**, y a partir de ahí las demás que le sigan. Si no usamos la directiva **ORG** el ensamblador tomará la primera dirección de la memoria de programa (**00h**), y a partir de ahí pondrá las instrucciones. Podemos usar la directiva **ORG** todas las veces que sea necesario en nuestro código, para definir diferentes direcciones de origen para diferentes bloques del programa. Si colocamos una etiqueta se le asignará el valor **dir** especificado:

```
dir_1       ORG 10h
```

Le asignará el valor de **10h** a la etiqueta **dir_1**, además de especificar el origen.

CBLOCK Y LAS DIRECCIONES

Si no especificamos ninguna dirección de comienzo para el primer bloque de constantes definidas mediante la directiva **CBLOCK** en nuestro código fuente, el ensamblador tomará en forma predeterminada la dirección **0x00**, la cual, como sabemos, pertenece al bloque de los SFR y entonces las etiquetas se asignarán a ellos y nuestro programa no funcionará correctamente.

PROCESSOR (*set processor type*)

Sintaxis: **PROCESSOR [microcontrolador]**

Esta directiva especifica el tipo de microcontrolador que se utilizará. No es necesario que la utilicemos en caso de que ya hayamos definido el microcontrolador mediante el menú **Configure/Select Device...** de MPLAB, tal como vimos antes. De todas formas, se acostumbra colocarla siempre para asegurar que el tipo de microcontrolador sea el adecuado, y como referencia del microcontrolador para el que se escribe el programa. Ejemplo:

```
PROCESSOR 16F84A
```

RADIX (*specify default radix*)

Sintaxis: **RADIX [raíz]**

Esta directiva especifica la raíz para los valores de datos, que pueden ser en decimal (**dec**), hexadecimal (**hex**), u octal (**oct**). Por ejemplo:

```
      RADIX dec

      movlw 10h                    ;El valor se toma en hexadecimal
      movlw o'10'                  ;El valor se toma en octal
      movlw 10                     ;Se toma en decimal

      RADIX hex

      movlw d'22'                  ;Se toma en decimal
      movlw o'22'                  ;Se toma en octal
      movlw 22                     ;Se toma en hexadecimal
```

III CÓDIGO FUENTE SIN MPLAB

Los archivos de código fuente son, en esencia, archivos de texto. Si estamos en una PC que no tiene instalado MPLAB, aun así podremos abrir nuestros archivos ***.asm.** Para eso, podemos utilizar cualquier editor de texto, como Wordpad, Word, o hasta el propio Bloc de notas de Windows. Incluso podemos hacer modificaciones en ellos aunque, por supuesto, no podremos ensamblarlos.

Como podemos observar en el código anterior, si no se especifica la raíz, el valor se tomará con la raíz indicada por la directiva **RADIX**. Si no se usa la directiva **RADIX**, el ensamblador tomará de manera predeterminada la raíz hexadecimal para todos los valores que no la especifiquen.

LIST (*listing options*)

Sintaxis: **LIST [opción1], [opción2], ...**

Esta directiva permite algunas opciones de listado. Las que más usaremos son:

r=[raíz]

Especifica la raíz para los valores de los datos, igual que con la directiva **RADIX**.

p=[microcontrolador]

Especifica el microcontrolador y es equivalente a la directiva **PROCESSOR**. Ejemplo:

```
LIST p=16F84A, r=hex
```

Especifica el PIC16F84A como microcontrolador utilizado y una raíz hexadecimal para los datos que no lo especifiquen. Se puede usar la opción **LIST** en lugar de **RADIX** y **PROCESSOR**, ya que en ella se definen ambas cosas en una sola línea, aunque esencialmente es lo mismo.

#DEFINE (*define a text substitution label*)

Sintaxis: **#DEFINE [etiqueta] exp**

Esta directiva define una cadena de texto para una etiqueta. En cualquier lugar donde el ensamblador encuentre la etiqueta, lo sustituirá por **exp**. Podemos usarla para asignar instrucciones completas a etiquetas, como por ejemplo:

```
#DEFINE carga_W movlw d'100'
```

En este ejemplo, a la etiqueta **carga_W** se le asigna la cadena de texto **movlw d'100'**, que es una instrucción completa. De esa forma, cada vez que escribamos **carga_W** en nuestro programa, el ensamblador lo sustituirá por la instrucción asignada.

Esta directiva debe comenzar con el símbolo **#**, aunque si no lo ponemos es también reconocida por el ensamblador. Dado que se puede asignar cualquier cadena de texto, podríamos usar también esta directiva en lugar de **EQU**, como por ejemplo:

```
#DEFINE tres d'3'
```

#UNDEFINE (*delete a substitution label*)

Sintaxis: **#UNDEFINE [etiqueta]**

Esta directiva borra la asignación que hayamos hecho antes con la directiva **#DEFINE**. Por ejemplo, para el código anterior, si colocamos posteriormente:

```
#UNDEFINE carga_W
```

Borra la asignación del valor que tenía la etiqueta **carga_W**. De esta forma, podemos volver a definir otro valor para ella si lo necesitamos.

#INCLUDE (*include additional source file*)

Sintaxis:

#INCLUDE [archivo] o **#INCLUDE ["archivo"]** o **#INCLUDE [<archivo>]**

Esta es una directiva muy importante, ya que permite incluir un archivo fuente completo en el lugar donde la coloquemos. Es decir, al especificar un archivo, su contenido será colocado en ese lugar, tal como si lo hubiéramos escrito ahí.

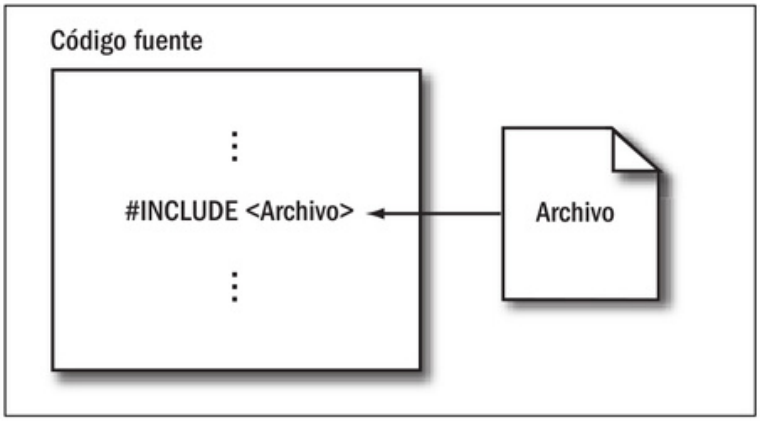


***Figura 8.** Con #INCLUDE insertamos el contenido de un archivo en nuestro código fuente.*

De esa forma, podemos insertar códigos y definiciones de constantes desde otros archivos sin tener que escribirlos nuevamente. También podemos tener una colección de librerías con subrutinas que utilicemos constantemente y así evitar escribirlas de nuevo. En el **Capítulo 6** hablaremos más de las librerías. Ahora presentamos un ejemplo del uso de esta directiva:

```
#INCLUDE <P16F84A.INC>
```

En el ejemplo insertamos el contenido del archivo llamado **P16F84A.INC** en el lugar donde está indicado por la directiva. Podemos señalar el archivo a incluir entre comillas (**""**), entre los símbolos **<>** , o sin ellos, aunque si el nombre del archivo contiene espacios es mejor hacerlo entre comillas o mediante **<>**. Si especificamos la ruta completa del archivo (por ejemplo, **<C:/Mis documentos/Mi archivo.inc>**), el ensamblador sólo lo buscará en esa ruta. Si no especificamos ninguna ruta para el archivo (colocamos únicamente su nombre), entonces el ensamblador lo buscará en este orden:

- En la carpeta de trabajo actual.
- En la carpeta del archivo fuente.
- En la carpeta de instalación del ensamblador **MPASMWIN.exe**.

La carpeta de instalación del ensamblador es generalmente **C:\Archivos de programa\Microchip\MPASM Suite**, a menos que lo hayamos instalado en otra unidad o carpeta. Si en cualquier caso el ensamblador no encuentra el archivo, nos dará un error. Los archivos con extensión **.inc** son precisamente archivos de tipo **include**, y se usa esa extensión para diferenciarlos de los archivos **.asm**. El archivo **P16F84A.INC** nos será de gran utilidad, por lo que a continuación hablaremos de él.

El archivo P16F84A.INC

En este archivo, que se instala automáticamente junto con MPLAB, están las definiciones de todas las etiquetas de los registros SFR del PIC16F84A. El archivo **P16F84A.INC** se encuentra en la carpeta de instalación del ensamblador, que es normalmente **C:\Archivos de programa\Microchip\MPASM Suite\P16F84A.INC**.

Como hemos mencionado, podemos asignar etiquetas para los distintos registros de la memoria de datos, ya sea de la sección GPR o SFR, para asignar nombres y hacer más fácil su utilización en nuestros programas. Por ejemplo, para definir una etiqueta para el registro STATUS, lo haremos de la siguiente forma:

```
STATUS EQU h'03'
```

Cada vez que escribamos **STATUS** en nuestro código, se interpretará como **03h**, que es la dirección de este registro. De la misma manera podemos proceder en los demás registros del SFR o los bits. Por ejemplo, para cambiar de banco de memoria usaremos el bit 5 del registro STATUS que es llamado RP0, por lo que podemos especificar:

```
RP0 EQU h'05'
```

Y así podemos escribir la instrucción:

```
bsf STATUS, RP0
```

Para indicar que deseamos poner a uno el bit RP0, es más fácil recordar los nombres STATUS y RP0 que sus direcciones, pero tendríamos que definir todas estas etiquetas al inicio de nuestro programa, lo que nos puede demorar bastante. Es por eso que el archivo llamado **P16F84A.INC** nos será de gran utilidad, sólo basta agregarlo en nuestro código fuente mediante la directiva **#INCLUDE**.

En este archivo están definidos los nombres de todos los registros del SFR, por lo tanto debemos escribirlos como están en él. A continuación, transcribimos parte del contenido del archivo **P16F84A.INC**:

```
;==========================================================================
;
;       Register Definitions
;
;==========================================================================

W                               EQU     H'0000'
F                               EQU     H'0001'

;----- Register Files------------------------------------------------------

INDF                            EQU     H'0000'
TMR0                            EQU     H'0001'
PCL                             EQU     H'0002'
STATUS                          EQU     H'0003'
FSR                             EQU     H'0004'
PORTA                           EQU     H'0005'
```

```
PORTB                           EQU     H'0006'
EEDATA                          EQU     H'0008'
EEADR                           EQU     H'0009'
PCLATH                          EQU     H'000A'
INTCON                          EQU     H'000B'

OPTION_REG                      EQU     H'0081'
TRISA                           EQU     H'0085'
TRISB                           EQU     H'0086'
EECON1                          EQU     H'0088'
EECON2                          EQU     H'0089'

;----- STATUS Bits --------------------------------------------------

IRP                             EQU     H'0007'
RP1                             EQU     H'0006'
RP0                             EQU     H'0005'
NOT_TO                          EQU     H'0004'
NOT_PD                          EQU     H'0003'
Z                               EQU     H'0002'
DC                              EQU     H'0001'
C                               EQU     H'0000'

;----- INTCON Bits --------------------------------------------------

GIE                             EQU     H'0007'
EEIE                            EQU     H'0006'
TOIE                            EQU     H'0005'
INTE                            EQU     H'0004'
RBIE                            EQU     H'0003'
TOIF                            EQU     H'0002'
INTF                            EQU     H'0001'
RBIF                            EQU     H'0000'

;----- OPTION_REG Bits ----------------------------------------------

NOT_RBPU                        EQU     H'0007'
INTEDG                          EQU     H'0006'
TOCS                            EQU     H'0005'
TOSE                            EQU     H'0004'
```

```
PSA                             EQU     H'0003'
PS2                             EQU     H'0002'
PS1                             EQU     H'0001'
PS0                             EQU     H'0000'

;----- EECON1 Bits --------------------------------------------------

EEIF                            EQU     H'0004'
WRERR                           EQU     H'0003'
WREN                            EQU     H'0002'
WR                              EQU     H'0001'
RD                              EQU     H'0000'

;=========================================================================
;
;       RAM Definition
;
;=========================================================================

        __MAXRAM H'CF'
        __BADRAM H'07', H'50'-H'7F', H'87'

;=========================================================================
;
;       Configuration Bits
;
;=========================================================================

_CP_ON                          EQU     H'000F'
_CP_OFF                         EQU     H'3FFF'
_PWRTE_ON                       EQU     H'3FF7'
_PWRTE_OFF                      EQU     H'3FFF'
_WDT_ON                         EQU     H'3FFF'
_WDT_OFF                        EQU     H'3FFB'
_LP_OSC                         EQU     H'3FFC'
_XT_OSC                         EQU     H'3FFD'
_HS_OSC                         EQU     H'3FFE'
_RC_OSC                         EQU     H'3FFF'

        LIST
```

Podemos observar cómo están definidos, por ejemplo, los bits de destino como F o W, por lo que en las instrucciones donde necesitemos especificar un registro de destino, a partir de ahora podemos hacerlo con F o W, en lugar de 1 ó 0. Por ejemplo:

```
#INCLUDE <P16F84A.INC>
contador EQU 0x0C
.
.
.
addwf contador, W
```

De esta manera, se indica que el destino es el registro W en lugar de tener que escribir 0. Por supuesto, antes de ejecutar esta instrucción debemos recordar incluir el archivo **P16F84A.INC** mediante la directiva **#INCLUDE**. Es conveniente familiarizarnos con los nombres asignados en él para poder usarlos. Para eso, podemos abrir el archivo en MPLAB, para ver su contenido completo y estudiarlo.

NUESTRO PRIMER PROGRAMA

Con lo estudiado hasta ahora ya podemos comenzar a escribir nuestros programas. Veamos, por ejemplo, cómo podemos configurar los puertos para que funcionen como entrada o salida de datos. Ésta será una tarea de prácticamente todos los programas, por lo que es conveniente estudiarla con detenimiento. Sabemos que las líneas de los puertos de entra/salida del PIC16F84A se pueden configurar de manera independiente cada una. Los registros relacionados con los puertos son:

PORTA para el Puerto A
PORTB para el Puerto B
TRISA para la configuración de las líneas del Puerto A
TRISB para la configuración de las líneas del Puerto B

Sabemos que los registros PORTA y PORTB están en el banco 0 de la memoria de datos, y que los registros TRISA y TRISB están en el banco 1. Además, sabemos que en cada bit de los registros TRISA y TRISB al poner un cero, la línea correspondiente se configurará como salida, y al colocar un uno, lo hará como entrada. Ahora supongamos que necesitamos un circuito como el de la **Figura 9**, en el cual tenemos cinco interruptores en el Puerto A, el cual por lo tanto configuraremos como entrada, y en el Puerto B tendremos 5 leds, por lo que los configuraremos como salida.

De esta manera, en nuestro programa leeremos los datos en el Puerto A y los pasaremos directamente al Puerto B, para visualizarlos en los leds.

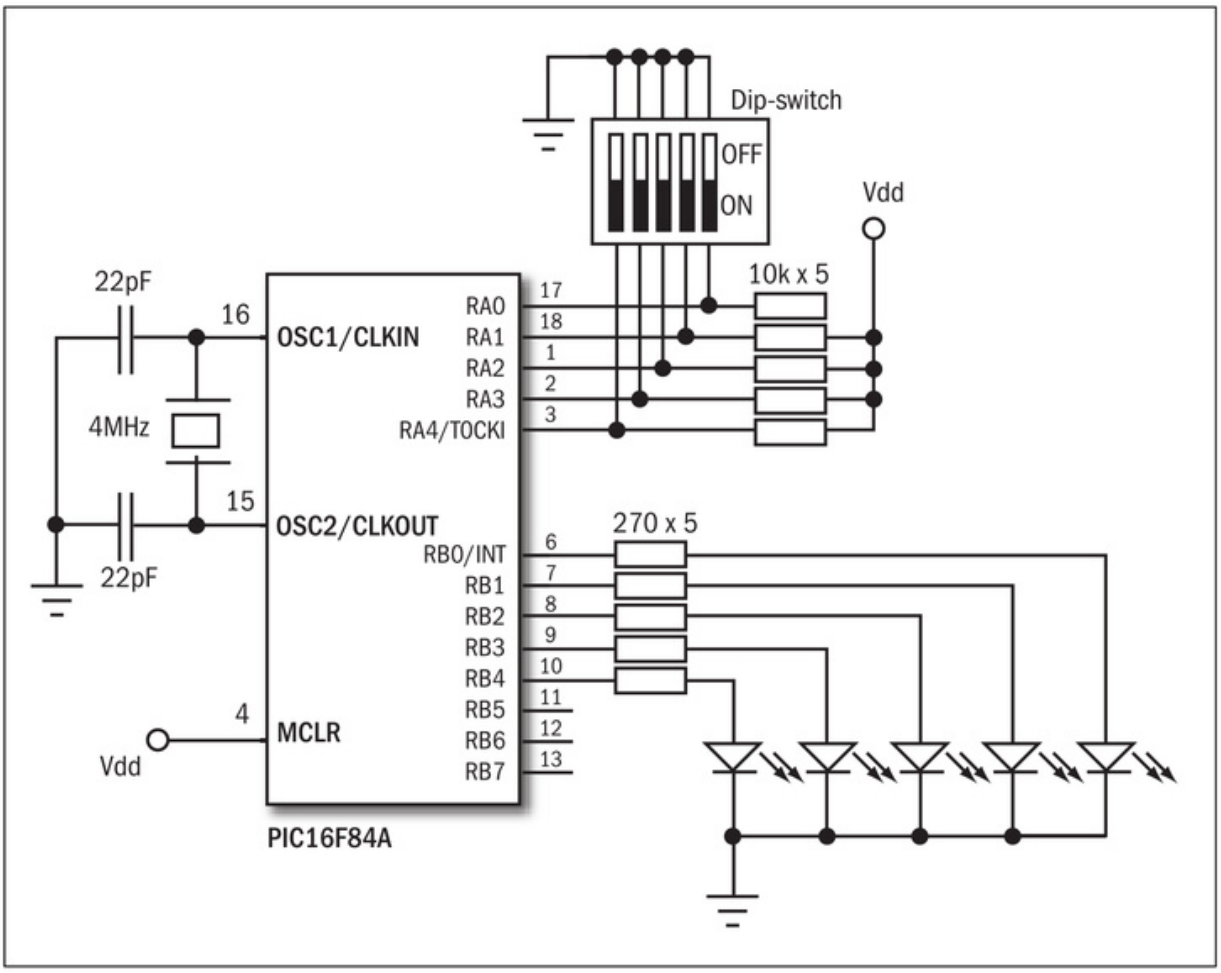


Figura 9. *Este circuito nos servirá para escribir nuestro primer programa para el PIC16F84A y para estudiar, a su vez, el uso de los puertos.*

Nuestro programa quedará de esta manera:

```
;------------Mi primer programa--------------------------
;
; Este programa lee los datos introducidos al los cinco
; bits del Puerto A mediante interruptores y los envía
; al Puerto B para visualizarlos mediante leds.
;
;---------------------------------------------------------

     __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

     PROCESSOR 16F84A              ;El microcontrolador usado es el
```

```
PIC16F84.
     #INCLUDE <P16F84A.INC>        ;Se incluye el archivo de definiciones.

     ORG            0x00           ;Establece el origen del programa.

;Configuración de puertos:

     bsf            STATUS, RP0    ;Acceso al banco 1 de la memoria de datos.
     movlw          b'00011111'
     movwf          TRISA          ;Configura el Puerto A como entrada.
     clrf           TRISB          ;Configura el Puerto B como salida.
     bcf            STATUS, RP0    ;Acceso al banco 0 de la memoria de datos.

;Programa principal:

inicio
     movf           PORTA, W       ;Mueve el contenido del Puerto A a W.
     movwf          PORTB          ;Mueve el dato de W al Puerto B.
     goto           inicio         ;Salta a inicio y entra en un lazo
infinito.

     END                           ;Fin del código, esta directiva
es obligatoria.
```

Analicémoslo con detenimiento para entender su funcionamiento:

Las primeras líneas son comentarios con el título, y una breve descripción del programa. Como ya mencionamos, es importante documentar nuestros programas para poder entenderlos en posteriores ocasiones, y saber qué función realizan. A partir de ahora, en los códigos del libro omitiremos los encabezados para ahorrar espacio, pero en los archivos *.asm podremos verlos.

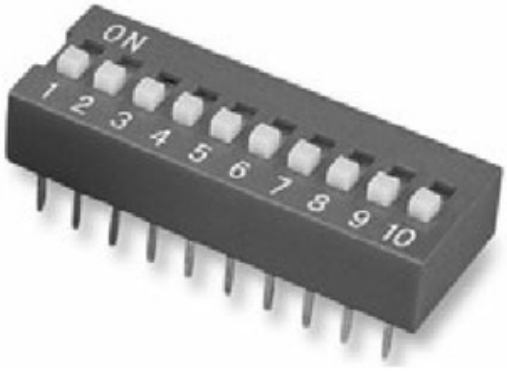


Figura 10. *Utilizaremos un dip-switch para introducir datos por el Puerto A al PIC16F84A.*

Después de la sección de comentarios tenemos la directiva **__CONFIG** para configurar los bits de la palabra de configuración (esto lo estudiaremos en detalle en el **Capítulo 5**). En la línea siguiente definimos el microcontrolador usado mediante la directiva **PROCESSOR**. Después

está la directiva **#INCLUDE**, que define el archivo **P16F84A.INC** para incluirlo, y así tener las definiciones de las etiquetas para los registros SFR. La siguiente línea establece el origen del código en la memoria de programa mediante la directiva **ORG**. Podemos omitirla ya que el origen en este caso es cero, pero la pondremos para tenerla como ejemplo de su uso y para acostumbrarnos a ella. A continuación, ya comienza el programa en sí. Luego encontramos la sección de configuración de los puertos. Veamos cómo funciona:

En primer lugar, mediante la instrucción **bsf** pondremos a 1 el bit RP0 del registro STATUS. De esta forma, estamos accediendo al banco 1 de la memoria de datos, ya que necesitamos escribir en los registros TRISA y TRISB, que están precisamente en el banco 1. Después cargamos al registro W con el valor binario 00011111 mediante la instrucción **movlw**; luego, con la instrucción **movwf**, enviamos este dato al registro TRISA, y con esto quedan configuradas las cinco líneas del Puerto A como entradas, ya que hemos puesto un 1 en cada una de ellas. Luego, con la instrucción **clrf** borramos o ponemos a 0 todos los bits del registro TRISB para configurar todas las líneas del Puerto B como salidas. Una vez hecho esto, con la instrucción **bcf** ponemos de nuevo el bit RP0 a 0 para tener acceso ahora al banco 0 de la memoria de datos. De esta forma, se configuran las líneas de los puertos, ya sea como entradas o como salidas, según necesitemos, escribiendo en sus registros de configuración.

Después de la configuración de puertos, tenemos el cuerpo principal de nuestro programa. Vamos a analizarla en detalle: con la instrucción **movf** leemos el valor que tenga en ese momento el Puerto A y lo almacenamos en W; luego, mediante la instrucción **movwf**, pasamos el valor de W al registro del Puerto B para que se refleje a la salida de sus líneas. Una vez hecho esto tenemos finalmente una instrucción de salto **goto**, la cual indica un salto hacia el lugar donde está la etiqueta **inicio**. De este modo, se crea un bucle que se repite siempre, leyendo los valores de entrada y luego enviándolos al puerto de salida. Finalmente tenemos la directiva **END**, que le indica al ensamblador el final del código (debemos recordar siempre ponerla al final).

{} OTROS ARCHIVOS .INC

Al instalar MPLAB, junto con él se instala el archivo de definiciones **P16F84A.INC** para que podamos usarlo. Además, se instalan muchos otros archivos de tipo **include** para muchos otros microcontroladores PIC, ya que cada uno tiene sus propios nombres para los SFR, por lo que cada PIC tendrá su propio archivo **.INC**.

Una vez escrito nuestro programa en el editor MPLAB, lo guardaremos como un archivo **.asm** con un nombre que nos recuerde para qué sirve. Por ejemplo, llamaremos a este código **Mi primer programa.asm**.

ENSAMBLADO DE LOS PROGRAMAS

Después de escribir nuestros programas, estamos listos para grabarlos en nuestro microcontrolador para que sean ejecutados en él. Pero, como sabemos, primero hay que traducirlo a lenguaje máquina, es decir, **ensamblarlo**. De esto se encargará el ensamblador MPASM. Como ya tenemos nuestro primer programa escrito y guardado como un archivo **.asm**, veamos ahora el procedimiento para ensamblarlo. Para hacerlo debemos tener abierto MPLAB y el archivo a ensamblar. Podemos usar el archivo del ejemplo anterior, **Mi primer programa.asm**, que debemos guardar en una carpeta con una ruta no mayor a 62 caracteres.

Figura 11. *Para ensamblar los programas lo haremos desde el menú Project de MPLAB.*

Una vez que tenemos nuestro archivo abierto en MPLAB, lo ensamblaremos. Si vamos al menú **Project/Quickbuild Mi primer programa.asm** notaremos cómo, en forma automática, MPLAB colocará el nombre del archivo **.asm** que estamos intentando ensamblar después de la palabra **Quickbuild**. Una vez elegido el comando, comenzará el proceso de ensamblado.

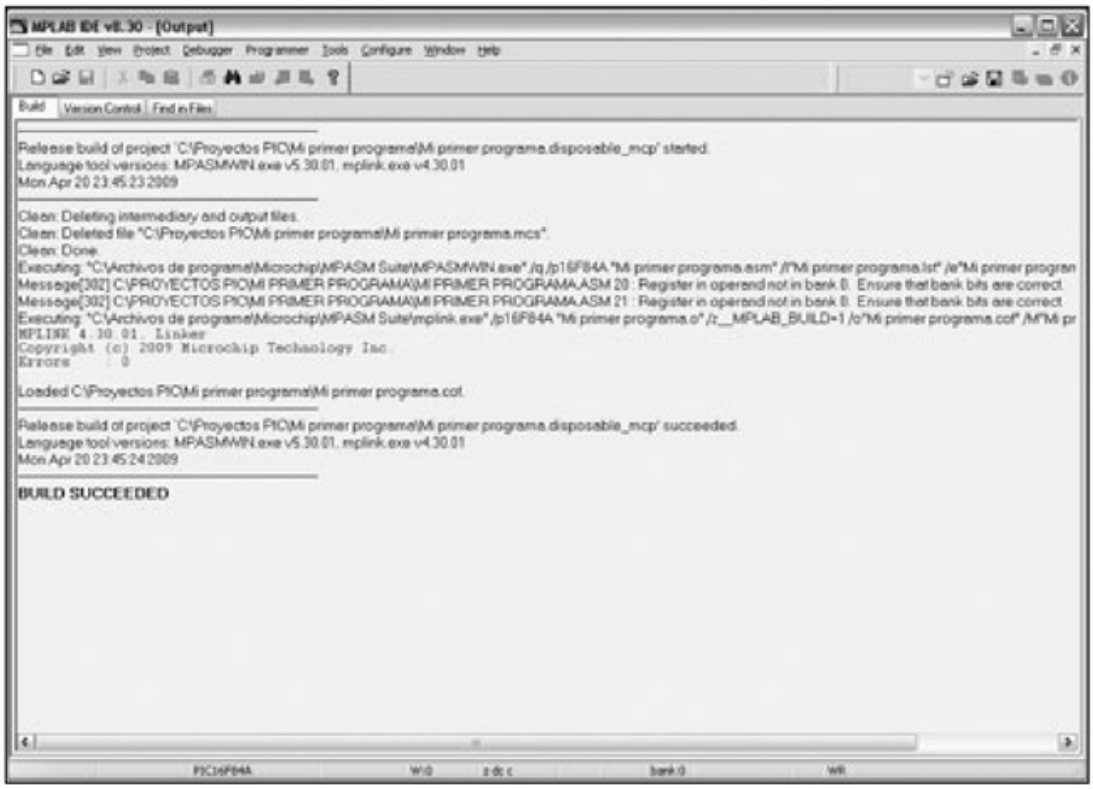


***Figura 12.** Durante el ensamblado aparecerá paulatinamente un reporte del proceso.*

Mientras se lleva a cabo el proceso de ensamblado de nuestro programa, aparecerá una ventana con el título **Build**, en la que veremos un reporte que nos indicará el estado de avance del mismo. Después de unos segundos, el proceso se completará, y si todo estuvo bien, al final del reporte del ensamblado deberemos encontrar la leyenda **BUILD SUCCEEDED**, que nos indica que el ensamblado se llevó a cabo con éxito. En el reporte del ensamblado también pueden aparecer otros mensajes para informarnos de eventos ocurridos durante el ensamblado, advertencias o errores. Si el programa detecta una falla al momento de intentar ensamblar, el proceso se detendrá y al final del reporte aparecerá el mensaje **BUILD FAILED** (**Figura 13**) en color rojo, señalando que el ensamblado falló y no se llevó a cabo. En el mismo reporte de ensamblado podemos ver cuál es el error o los errores que se detectaron, para corregirlos e intentarlo nuevamente.

III LOCALIZAR LOS ERRORES

Si nuestro código fuente tiene errores, el proceso de ensamblado se detendrá y se mostrarán los errores encontrados en el reporte, y el número de línea en el que se encuentra cada uno. Si hacemos doble clic en alguna línea del reporte que nos indica un error, esto nos llevará a la línea del código fuente donde se encuentra la falla, para poder corregirla.

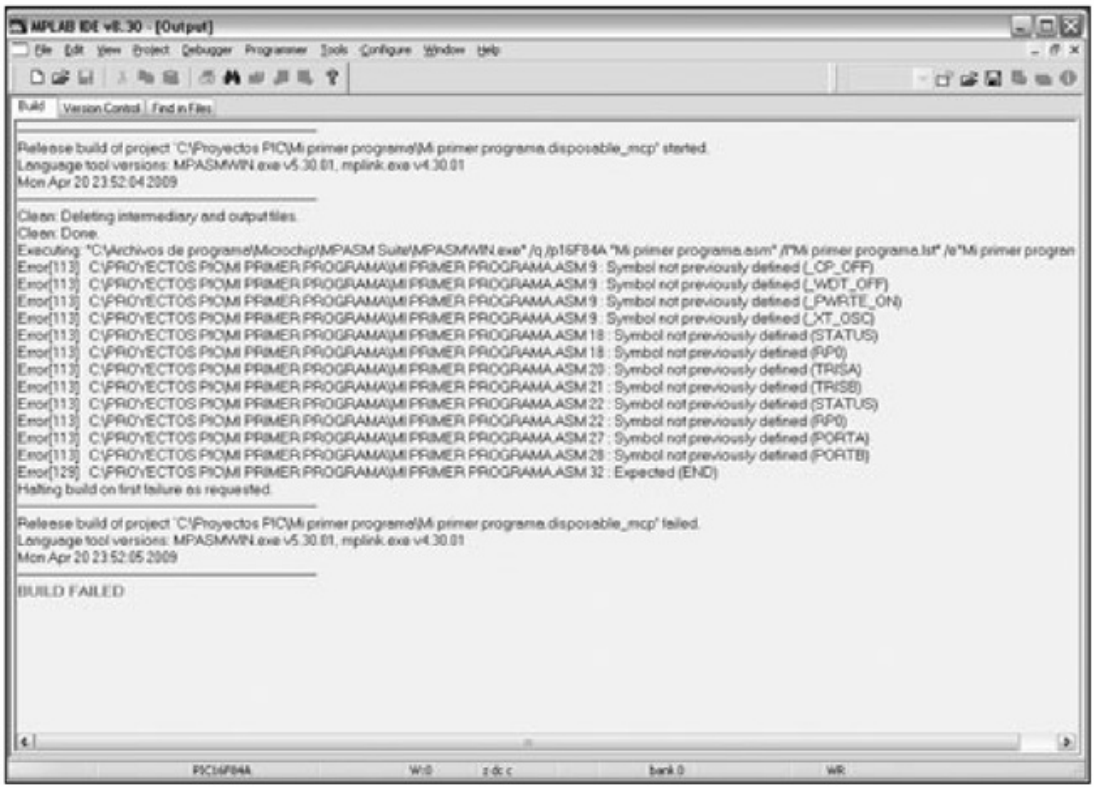


Figura 13. *Errores que detienen el proceso de ensamblado y que habrá que corregir.*

Resultado del ensamblado

El objetivo del ensamblado de nuestro código fuente es el de generar el código máquina para poder grabarlo en nuestro PIC. Después de que el proceso de ensamblado se ha llevado a cabo en forma exitosa, el código máquina será creado en un nuevo archivo con la extensión **.hex** y el mismo nombre del archivo fuente. El archivo **.hex** y otros más se crearán en la misma carpeta donde se encuentra el archivo fuente **.asm**.

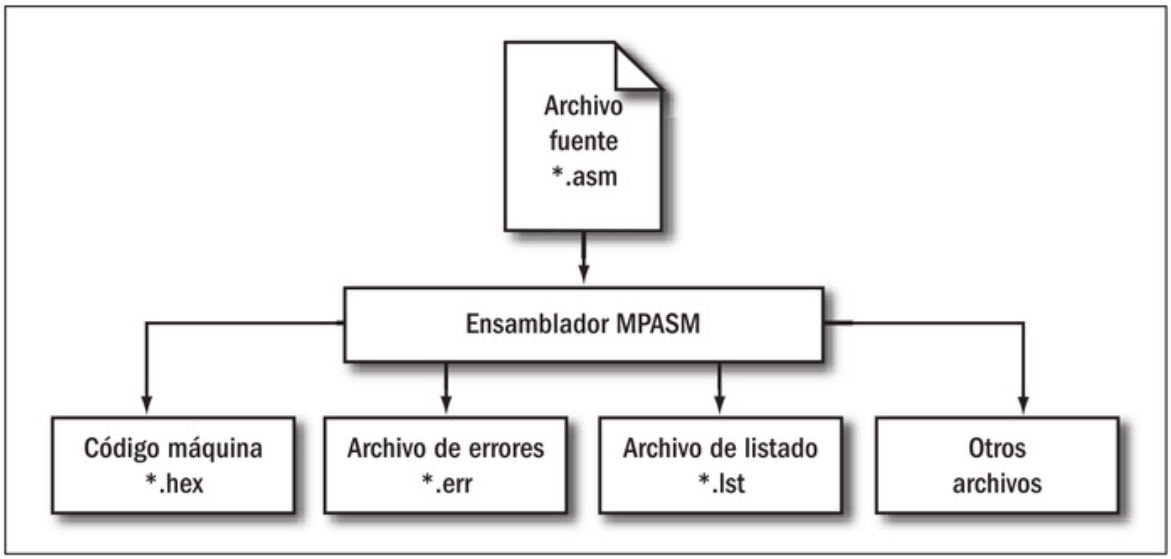


Figura 14. *El proceso de ensamblado generará varios archivos con diferentes extensiones en la carpeta del archivo fuente.*

Algunos de los archivos generados por el proceso de ensamblado son:

- **.hex**: es el archivo de código máquina. Contiene los valores binarios (aunque realmente están expresados en formato hexadecimal) que corresponden a nuestras instrucciones ya listas para ser grabadas en la memoria de programa del PIC16F84A. En el **Capítulo 5** estudiaremos en detalle el uso de este archivo.
- **.err**: este archivo contiene una lista de los errores encontrados durante el proceso de ensamblado del programa. Podemos abrirlo con el mismo MPLAB si necesitamos verificar cuáles son y dónde están dichos errores.
- **.lst**: este es un archivo de listado. Contiene la lista de todo el proceso, es decir, una copia del código fuente, los errores producidos, el código ensamblado (código máquina), detalles de uso de la memoria, y demás datos. Podemos abrirlo en el propio MPLAB para ver su contenido.

Además de los archivos mencionados, pueden generarse otros que no son relevantes o no tienen ninguna utilidad para nosotros. Cada vez que ensamblemos el mismo programa, es decir, si ensamblamos de nuevo el mismo código fuente, el ensamblador borrará automáticamente todos los archivos generados por el último proceso y los creará de nuevo.

SIMULACIÓN EN MPLAB SIM

Una vez escrito y ensamblado nuestro programa, ya podemos usarlo en nuestro PIC, pero ¿cómo sabemos si nuestro código realmente hace lo que se supone que debe hacer? Podemos grabarlo en el microcontrolador y probarlo, pero si hay errores y el programa no funciona correctamente debemos corregirlo y probar de nuevo. Este proceso de prueba y error puede ser muy lento. Por eso, para resolver esto, tenemos en el propio MPLAB un simulador que nos permitirá analizar el funcionamiento de nuestros programas para ver si hacen lo correcto, o para detectar y corregir fallas y errores en él. Para poder simular un programa debemos tener en cuenta algunos detalles:

III EL REPORTE DE ENSAMBLADO

Durante el proceso de ensamblado pueden aparecer en el reporte las leyendas: **Warning** (atención) o **Message** (mensaje), las cuales no son errores en sí, sino que nos indican algo que ocurrió para poner atención, o algo que debemos vigilar o verificar. Si sólo aparecen estos mensajes, el programa se ensamblará, pero si aparece algún error, el proceso de ensamblado se detendrá.

- El código fuente debe estar abierto en MPLAB.
- Debemos ensamblarlo y asegurarnos de que fue ensamblado con éxito.
- Debemos elegir el microcontrolador adecuado (como ya vimos antes).

Después de asegurarnos de estos detalles debemos habilitar el simulador **MPLAB SIM**. Para ello iremos al menú **Debugger/Select Tool** y en la lista que aparece elegiremos la opción **MPLAB SIM**.

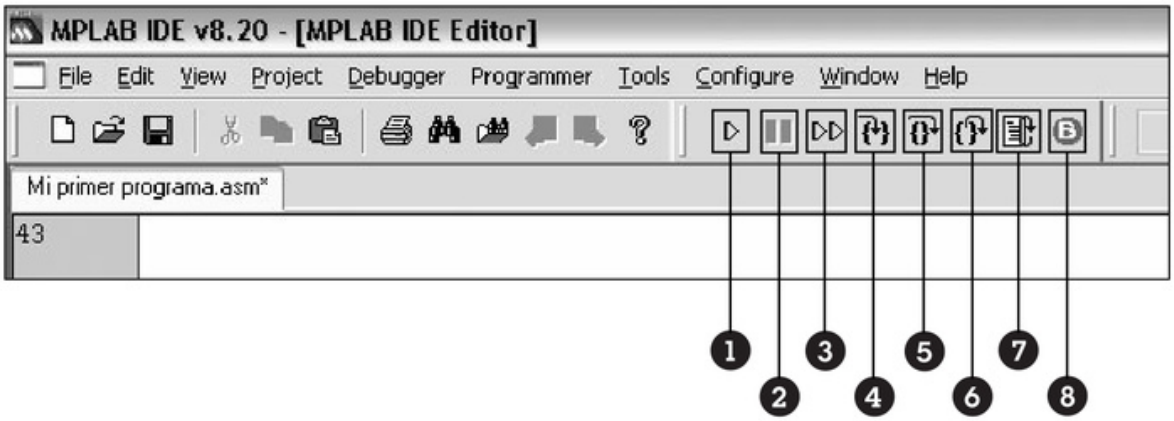
***Figura 15.** Debemos habilitar el simulador para poder usarlo. Es posible que tengamos que hacerlo cada vez que abrimos MPLAB.*

Una vez que habilitamos el simulador MPLAB SIM aparecerán en la barra de herramientas los controles que nos permitirán manejar la simulación. Además, en el menú **Debugger** aparecerán más opciones. Veamos la función de los principales botones de simulación en la siguiente **Guía Visual**.

III OTROS SIMULADORES

Aunque en este capítulo sólo estudiaremos la simulación con **MPLAB SIM**, podemos usar otros simuladores para corregir o depurar nuestros programas. Algunos de ellos, como **Proteus**, permiten ver una animación en tiempo real del circuito completo para ver su funcionamiento. La desventaja es que este tipo de simuladores no son gratuitos y pueden ser muy costosos.

Los controles de simulación GV



1. **Run** (ejecutar): este botón sirve para ejecutar el programa de forma continua. Si lo presionamos, la simulación correrá libre y realmente no veremos nada hasta que se detenga.
2. **Halt** (alto): si presionamos este botón mientras la simulación está corriendo, se detendrá. Este botón no estará activo hasta que corramos la simulación.
3. **Animate** (animar): al presionar este botón, la simulación correrá paso a paso por cada línea. Una flecha de color verde del lado izquierdo del código fuente indicará la línea de programa que se está ejecutando.
4. **Step Into** (paso a paso): con este botón podemos ejecutar manualmente el programa un paso a la vez. Cada vez que lo presionemos se ejecutará una línea del programa. Nuevamente, la flecha verde nos indicará la línea que estamos ejecutando.
5. **Step Over** (paso a paso sin subrutinas): este botón funciona de manera similar al botón **Step Into**, pero con la diferencia de que al encontrar una instrucción **call** (llamado a subrutina) no entrará en ella, sino que la ejecutará como una sola instrucción.
6. **Step Out** (salir de subrutina): si hemos entrado en la ejecución de una subrutina, este botón permite salir de ella ejecutándola por completo en un solo paso.
7. **Reset** (reset): este botón es equivalente a dar un reset al microcontrolador. Si lo presionamos iniciaremos de nuevo el programa desde el principio, es decir, en la primera línea.
8. **Breakpoints** (puntos de ruptura): este botón abre una ventana para poder observar una lista con los puntos de ruptura y, desde ella, crearlos, eliminarlos, activarlos o desactivarlos. Veremos los detalles más adelante en este mismo capítulo.

Antes de ejecutar la simulación, además debemos asegurarnos de que tenemos elegida la frecuencia correcta para el oscilador de nuestro microcontrolador. Esto será importante sobre todo cuando necesitemos medir tiempos para ver cuánto dura el programa o partes de él. Para configurar la frecuencia del oscilador debemos ir al

menú **Debugger/Settings...** y en la ventana **Simulator Settings** debemos elegir la frecuencia adecuada en la pestaña **Osc / Trace** en la sección marcada como **Processor frequency**. En la mayoría de los casos usaremos 4 MHz.

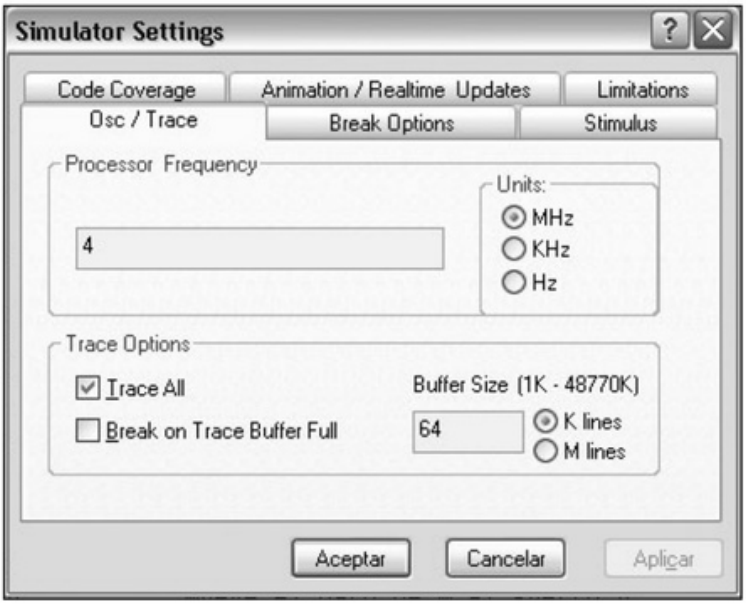


***Figura 16.** En la ventana Simulator Settings elegiremos la frecuencia de oscilación adecuada para nuestro caso.*

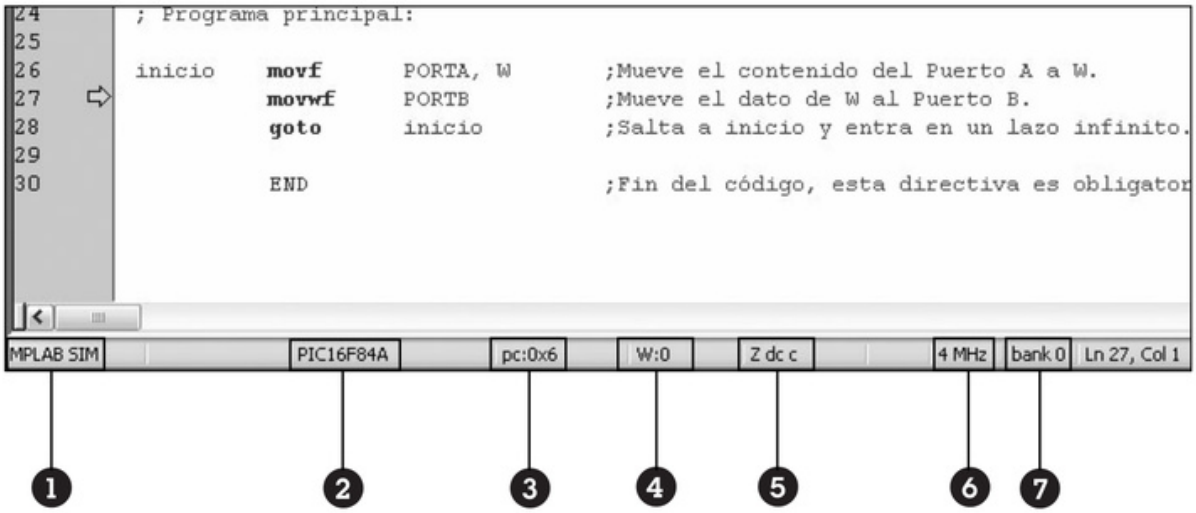
Desde la ventana **Simulator Settings** podemos configurar la frecuencia eligiendo las opciones de **Hz**, **KHz** o **MHz** y escribiendo el valor deseado. Esta frecuencia de oscilación realmente no tiene efecto en la velocidad de la simulación, sólo es para que los tiempos marcados sean correctos con respecto a nuestro circuito de oscilador que utilicemos en nuestro PIC.

Mediante los controles podemos correr la simulación para observar el funcionamiento del programa, pero hasta ahora no hemos observado demasiado. El objetivo de la simulación es ver si el programa realiza las operaciones correctas. Lo primero que podemos observar es la **barra de estado**, que está en la parte inferior de la ventana de MPLAB. En ella tenemos indicados varios parámetros que cambiarán durante la simulación.

III VELOCIDAD DE SIMULACIÓN

Podemos cambiar la velocidad de la simulación del botón **Animate**. Para ello, vamos al menú **Debugger/Settings...** y en la ventana **Simulator Settings** vamos a la pestaña **Animation / Realtime Updates**. En ella veremos un control deslizante llamado **Animate Step Time**, y arrastrándolo podremos hacer la animación más rápida (**Fastest**) o más lenta (**Slowest**).

Guía Visual: La barra de estado GV



1. **MPLAB SIM**: si esta leyenda está presente, indica que MPLAB SIM está activado. Si no está, el simulador no está activado. Si el simulador no está activado, algunos de los elementos siguientes puede que no aparezcan tampoco.
2. **Microcontrolador**: indica el microcontrolador elegido con el que estamos trabajando.
3. **Contador de programa (PC)**: muestra en todo momento el valor del contador de programa (PC).
4. **Registro de trabajo (W)**: muestra el valor contenido por el registro de trabajo **W**.
5. **Banderas del registro STATUS**: muestra los estados que tienen los bits o banderas Z, DC y C del registro STATUS. Si están en minúsculas, indican un cero, y en mayúsculas un uno. Por ejemplo **Z=1** y **z=0**.
6. **Frecuencia del oscilador**: se indica la frecuencia del oscilador que hemos configurado. Debe coincidir con la frecuencia que tenemos pensado usar realmente.
7. **Banco de memoria**: indica cuál es el banco de la memoria de datos que está seleccionado.

De esta forma tendremos información de lo que está sucediendo con estos elementos mientras corremos la simulación del programa. Los elementos cambiarán automáticamente para indicarnos su estado.

Visualización de registros

Además de la barra de estado podemos tener más detalles sobre lo que está sucediendo al correr la simulación. Por ejemplo, podemos observar mediante otras ventanas los estados que toman los registros de la memoria de datos para ver si el programa funciona como lo esperamos.

Visualización de los registros de función especial (SFR)

Si vamos a **View/Special Function Registers** se abrirá una ventana donde tendremos una lista de los registros de función especial. Podemos ver los valores que van tomando estos registros durante la simulación y apreciar si toman los valores que esperamos.

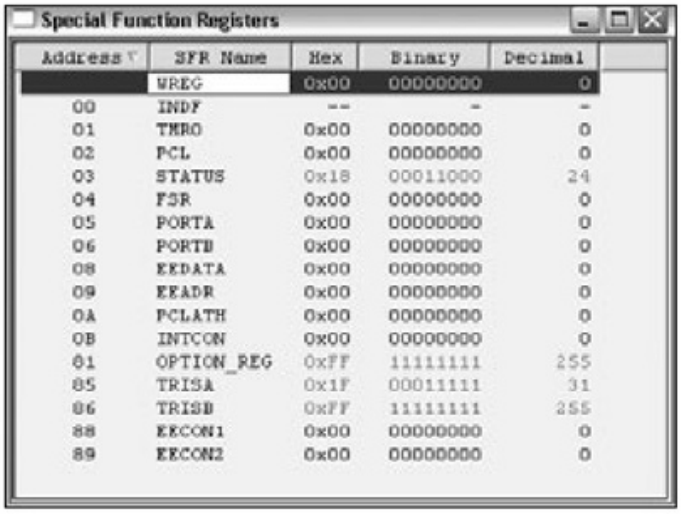


Figura 17. *Aquí podemos vigilar el estado de estos registros durante la simulación.*

Registros de la memoria de datos

Para observar los registros de la memoria de datos vamos al menú **View/File Registers**, donde aparecerá una ventana (**Figura 18**) con un mapa de todos los registros de la memoria de datos. Si presionamos la pestaña **Symbolic**, debajo de esta ventana se mostrará una lista de registros con su nombre y su valor contenido. Esto es útil para ver los valores que tienen los registros de la memoria de datos durante la simulación. Los registros del área GPR sólo mostrarán nombre a los que les hayamos asignado uno mediante etiquetas en el código fuente.

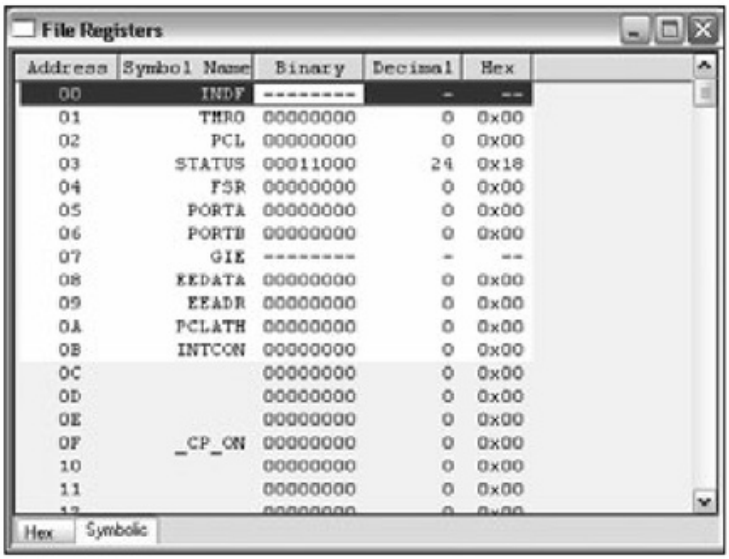


Figura 18. *La ventana File Registers muestra los valores de todos los registros de la memoria de datos, incluyendo los GPR y SFR.*

La ventana Watch

Si vamos al menú **View** y activamos la opción **Watch** aparecerá una ventana con ese título que al principio se encontrará vacía.

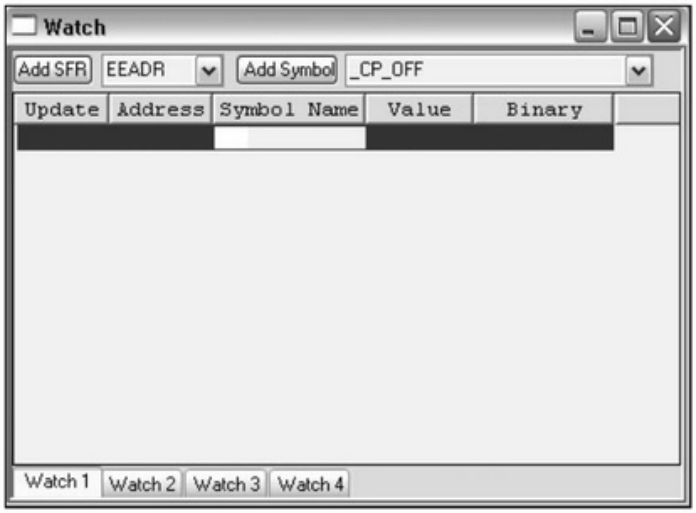


***Figura 19.** La ventana Watch sólo sirve para mostrar los registros que usemos con mayor frecuencia.*

La ventana **Watch** nos será útil para observar únicamente los registros o bits individuales que necesitemos. Es decir, podemos personalizar en ella los registros o bits de la memoria de datos que queremos vigilar. Esto lo hacemos al elegir de las listas desplegables el elemento deseado, y luego con un clic en el botón **Add SFR** o **Add Symbol**, respectivamente. La ventana **Watch** tiene cuatro pestañas abajo, por lo que podemos tener diferente configuración en cada una de ellas. En estas ventanas podemos ver los valores que toman los registros durante la simulación y así poder analizar si el programa realmente hace lo que necesitamos o no. Cuando hay un cambio en alguno de los registros, sus valores contenidos se pondrán en color rojo para indicarlo.

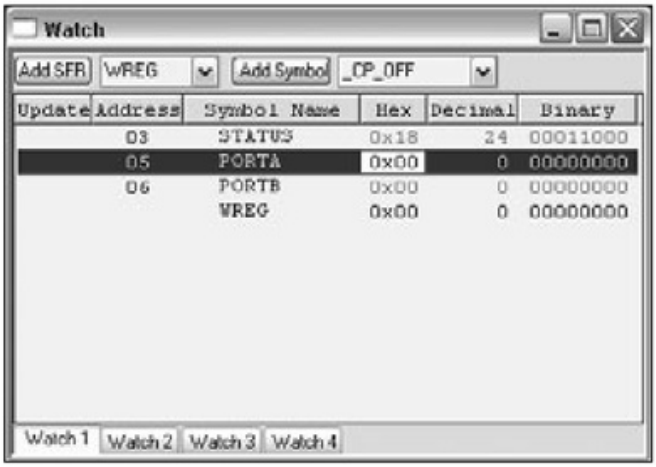


***Figura 20.** La ventana Watch con algunos registros agregados, los puertos, el registro STATUS y el registro W.*

En cualquier ventana de visualización de registros podemos hacer clic con el botón derecho del mouse en los títulos de las columnas para agregar otras o quitarlas. Por ejemplo, podemos agregar una columna para ver los valores en decimal, o también podemos arrastrarlas para cambiarlas de lugar.

Puntos de ruptura (*breakpoints*)

Un **punto de ruptura** o **breackpoint** es un punto en el que la simulación se detendrá automáticamente. De esta forma, podemos analizar los valores de los registros en ese momento. Para agregar un punto de ruptura, haremos un doble clic en la línea del código fuente donde queremos colocarlo, y aparecerá a la izquierda un **círculo de color rojo** con una letra **B**, que indicará el punto de ruptura insertado. Para remover un punto de ruptura haremos también doble clic en la línea donde exista alguno y se borrará.

```
15
16      ; Configuración de puertos:
17
18  ⇨           bsf       STATUS, RP0    ;Acceso al banco 1 de la memoria de datos.
19              movlw     b'00011111'
20  B           movwf     TRISA          ;Configura el Puerto A como entrada.
21              clrf      TRISB          ;Configura el Puerto B como salida.
22  B           bcf       STATUS, RP0    ;Acceso al banco 0 de la memoria de datos.
23
24      ; Programa principal:
25
26  B   inicio  movf      PORTA, W       ;Mueve el contenido del Puerto A a W.
27              movwf     PORTB          ;Mueve el dato de W al Puerto B.
28              goto      inicio         ;Salta a inicio y entra en un lazo infinito.
29
30              END                      ;Fin del código, esta directiva es obligatoria.
```

***Figura 21.** Un punto de ruptura marca la detención de la simulación en ese punto del programa.*

Podemos insertar todos los puntos de ruptura que necesitemos. Cuando corremos la simulación mediante el botón **Run** o **Animate** ésta se detendrá automáticamente en el primer breackpoint que encuentre. Para reanudar la simulación a partir de ahí, sólo basta con presionar nuevamente el botón **Run** o **Animate**. Es

III MODIFICAR LOS CONTENIDOS

En las ventanas de visualización de los registros, si detenemos la simulación y hacemos doble clic en el valor de cualquiera de los registros en ellas, podemos cambiar su valor y escribir el que necesitemos. Pero esto no es válido para simular entradas de datos en los registros de los puertos configurados como entradas, para ello debemos usar **estímulos**.

importante tener en cuenta que la línea donde se encuentra el punto de ruptura no será ejecutada, así que debemos colocar un punto de ruptura después de la última instrucción que necesitemos que sea ejecutada. Si hacemos clic con el botón derecho del mouse en un punto de ruptura, podemos desactivarlo con la opción **Disable Breackpoint**, para no tener que borrarlo, y de esta forma no funcionará hasta que lo activemos de nuevo.

```
15
16      ;Configuración de puertos:
17
18              bsf     STATUS, RP0     ;Acceso al banco 1 de la memoria de datos.
19              movlw   b'00011111'
20              movwf   TRISA           ;Configura el Puerto A como entrada.
21              clrf    TRISB           ;Configura el Puerto B como salida.
22              bcf     STATUS, RP0     ;Acceso al banco 0 de la memoria de datos.
23
24      ;Programa principal:
25
26      inicio  movf    PORTA, W        ;Mueve el contenido del Puerto A a W.
27              movwf   PORTB           ;Mueve el dato de W al Puerto B.
28              goto    inicio          ;Salta a inicio y entra en un lazo infinito.
29
30              END                     ;Fin del código, esta directiva es obligatoria.
```

Figura 22. *Ejemplo de algunos breackpoints desactivados, los cuales ya no funcionarán.*

Si presionamos el botón **Breakpoints** de la barra de herramientas (que describimos en la **Guía visual: Los controles de simulación**) se abrirá una ventana con el mismo título, en la cual aparecerá una lista con todos los puntos de ruptura (**Figura 23**). En ella podemos activarlos o desactivarlos marcándolos o demarcándolos de la lista. Podemos activarlos todos (**Enable all**), desactivarlos todos (**Disable all**), eliminar alguno de ellos (**Remove**), o todos (**Remove all**).

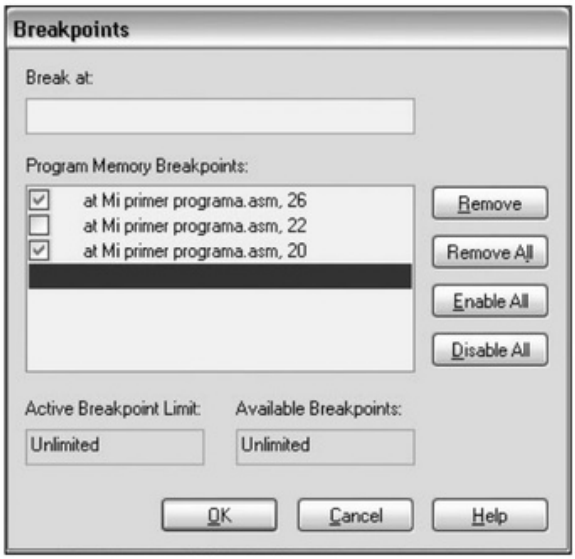


Figura 23. *La ventana Breakpoints nos permitirá administrar nuestros puntos de ruptura.*

Estímulos

Podemos cambiar los valores de los registros con un doble clic en ellos y escribiendo el valor que necesitemos. Pero cuando tengamos un puerto de entrada de datos, esto no será válido. Para simular los datos en los pines de entrada de los puertos debemos hacerlo mediante **estímulos** (*stimulus*). A través de los estímulos podemos simular entradas de datos o señales en los puertos que tengamos configurados como entradas. Para configurar los estímulos necesarios vamos al menú **Debugger/ Stimulus/New Workbook** y se abrirá la ventana con el título **Stimulus – [Untitled]**.

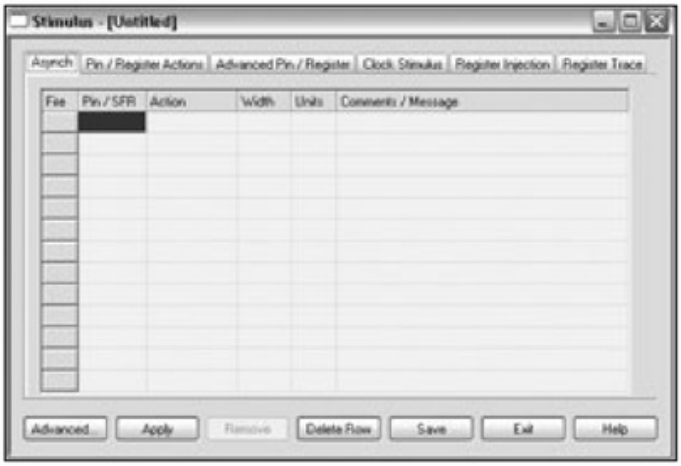


***Figura 24.** Desde aquí podemos configurar las señales para nuestros pines o puertos de entrada.*

Seleccionaremos la pestaña **Asynch** y veremos una serie de celdas. Si hacemos clic en la primera celda de la columna llamada **Pin / SFR**, podemos elegir un pin de entrada, por ejemplo, RA0. Si hacemos clic en la celda siguiente en la columna **Action** podemos elegir la acción que necesitamos, entre las siguientes opciones:

Set High: pone el pin a uno.
Set Low: pone el pin a cero.
Toggle: cambia el valor actual del pin.
Pulse High: da un pulso en alto.
Pulse Low: da un pulso en bajo.

En las primeras dos opciones, se coloca el estado correspondiente (alto o bajo). La opción **Toggle** cambiará el estado actual, es decir que si había un cero lo pondrá a uno, y si había un uno lo pondrá a cero. Las últimas dos opciones darán un pulso en alto o bajo, según corresponda, sin importar el estado actual. En estas opciones también hay que especificar la duración de dicho pulso en las siguientes dos columnas **Width** y **Units**, ya sea en ciclos de reloj o tiempo. El cambio se llevará a cabo con un clic en el botón de la columna **Fire** mientras la simulación está corriendo. Podemos definir todas las entradas necesarias en las siguientes filas, de acuerdo con lo que necesitemos.

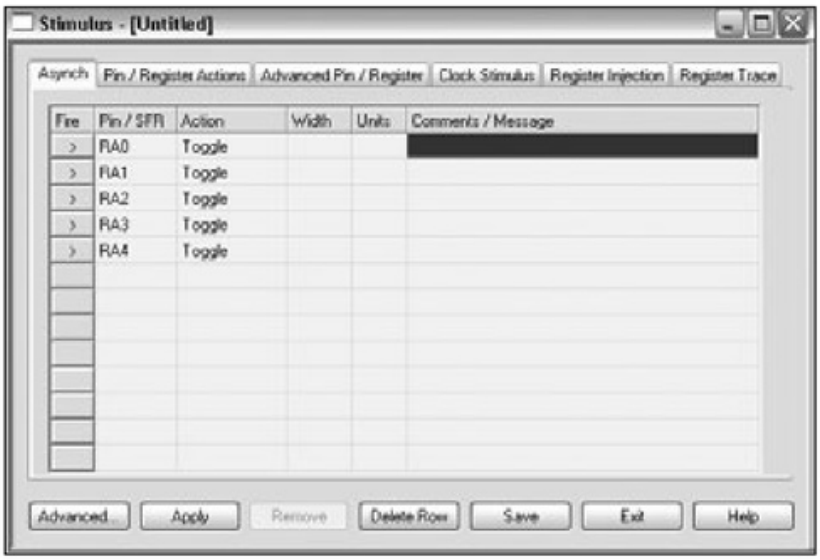


***Figura 25.** Las cinco líneas del Puerto A configuradas en modo Toggle.*

Otras funciones de simulación útiles

Otras funciones que nos pueden resultar útiles para la simulación de nuestros programas se encuentran en el menú contextual al hacer clic con el botón derecho del mouse en cualquier lugar del código fuente:

- **Run To Cursor**: correrá la simulación tal como si hubiéramos presionado el botón **Run**, pero sólo hasta donde se encuentra el cursor. Las líneas que se ejecutarán comenzarán en donde se encuentre el contador de programa (la flecha de color verde) y la posición actual del cursor. Debemos tener en cuenta que al hacer clic con el botón derecho del mouse, el cursor se ubicará en el lugar donde hicimos clic.
- **Set PC at Cursor**: coloca el contador de programa en la línea del programa donde se encuentre el cursor. Es útil para cambiar el PC a donde necesitemos en el código.
- **GoTo...**: al elegir esta opción se abrirá un cuadro de diálogo con el título **Go To**, en el cual podemos elegir algún punto del código fuente al cual queremos ir. Esto será especialmente útil cuando nuestros códigos fuente sean muy extensos. Tenemos la opción de colocar un número de línea (**Line**) o elegir una etiqueta (**Label**) del código fuente a la cual queremos ir al presionar el botón **Go To**. La opción **GoTo...** también está disponible en el menú **Edit** de la barra de menús de MPLAB.

***Figura 26.** Aquí localizaremos puntos específicos dentro de nuestros códigos fuente.*

Simulación de nuestro primer programa

Para simular nuestro primer programa, basta con abrir el archivo **Mi primer programa.asm** en MPLAB y ensamblarlo para que esté listo para simularse. Pero antes de hacerlo debemos tomar en cuenta algunos aspectos:

- El proceso de ensamblado debe ser exitoso, es decir, sin errores.
- Debemos habilitar MPLAB SIM, si no está habilitado aún.
- Debemos configurar la frecuencia del oscilador.
- Debemos haber elegido el PIC16F84A.

Además de lo anterior, es bueno poder ver los registros necesarios para verificar el funcionamiento del programa. Para ello abriremos la ventana **Watch** y en ella agregaremos los registros PORTA y PORTB para poder observar sus valores durante la simulación. Como el Puerto A es la entrada donde usaremos interruptores, necesitaremos usar estímulos para poder entrar los datos en él. Para ello abriremos la ventana **Stimulus** y elegiremos las 5 líneas del Puerto A (RA0 a RA4) y las configuraremos en modo **Toggle**.

Es conveniente acomodar las ventanas de tal forma que las veamos todas a la vez. Una vez que está todo listo podemos presionar el botón **Animate**, para que la simulación corra. Veremos la flecha de color verde recorriendo las líneas del programa. Con los botones **Fire** de la ventana **Stimulus** podemos cambiar el estado del puerto de entrada para ver cómo afectan a los valores de la ventana **Watch**. Como es de esperarse, si todo está bien, al modificar los estados de las líneas del Puerto A, el cambio se reflejará en el registro PORTA, y luego serán enviados al puerto de salida que es el Puerto B, ya que ese es el objetivo del programa, confirmando así que el funcionamiento es correcto. También podemos correr la simulación con el botón **Step Into** para hacerlo paso a paso.

... RESUMEN

En este capítulo hemos aprendido las bases para usar el software MPLAB IDE, que es la herramienta para escribir nuestros programas, ensamblarlos y simularlos. Dado que la escritura de programas es fundamental para el uso de microcontroladores, es necesario dominar bien la programación y las herramientas para hacerlo. En capítulos posteriores continuaremos estudiando el uso del software, las directivas y demás funciones para entenderlas mejor.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Para qué sirve el entorno MPLAB?

2 ¿Cuántos campos se usan para la estructura del código fuente?

3 ¿Para qué sirve el campo de etiquetas?

4 ¿Para qué sirve el campo de comentarios?

5 ¿Cómo deben comenzar los comentarios en el código fuente?

6 ¿Qué son y para qué se usan las directivas?

7 ¿Cuál es la única directiva que es obligatoria y para qué sirve?

8 ¿Para qué sirve el archivo P16F84A.INC?

9 ¿Cómo se ensambla un programa en MPLAB?

10 ¿Cuál es la extensión del archivo de código máquina resultante del ensamblado?

PRÁCTICAS

1 Modifique el código de Mi primer programa, de tal forma que ahora los bits se muestren a la salida en el Puerto B invertidos con respecto a los de entrada en el Puerto A. Consejo: se debe cambiar una sola instrucción.

2 Después de hacer la modificación anterior, guarde el programa con otro nombre, ensámblelo y simúlelo para comprobar que funcione correctamente.

Capítulo 5

Grabadores de PIC

Después de escribir nuestro programa y de ensamblarlo para generar el código máquina, estamos en condiciones de grabarlo en la memoria de programa del microcontrolador para que nuestros circuitos queden listos para funcionar. Para escribir el código máquina en el PIC, debemos usar un circuito especial que será el encargado de hacerlo. Estos circuitos o dispositivos son llamados grabadores. En este capítulo estudiaremos los grabadores de microcontroladores PIC.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

GRABACIÓN DE MICROCONTROLADORES PIC

Como ya sabemos, el PIC16F84A y todos los PICs de gama baja y media que en su nombre incluyen una **F**, cuentan con una memoria de programa del tipo Flash, que puede ser grabada o borrada en múltiples ocasiones, y es no volátil. Es esta memoria es donde grabaremos el código máquina generado por el ensamblador, que es el programa que el PIC ejecutará. La escritura de esta memoria de programa se realiza de forma **serial**. El protocolo utilizado para ello se denomina **ICSP** (*In Circuit Serial Programming*) y con él se puede grabar o leer en la memoria de datos y de programa del microcontrolador.
La escritura o lectura en la memoria Flash del microcontrolador es **serial síncrona**, es decir, todos los datos entran o salen en serie por un solo pin del circuito y el proceso se sincroniza mediante una señal de reloj.

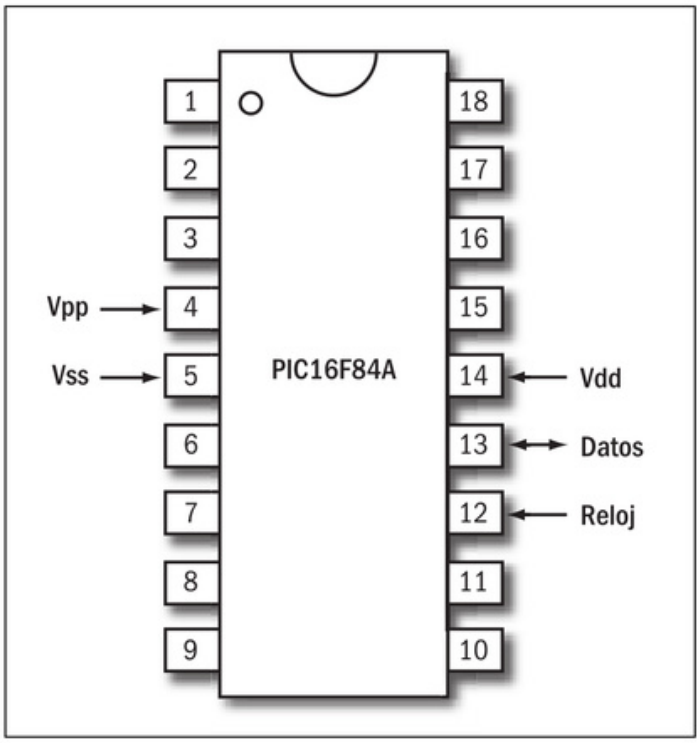


***Figura 1.** Configuración de los pines del PIC16F84A para la grabación de la memoria de programa.*

Algunos de los pines del microcontrolador toman ahora una función alternativa especial que permite leer o escribir en la memoria de datos y de programa. El pin Master Clear (4) ahora será la entrada del **voltaje de programación** o **Vpp**. Cuando llevamos este pin a un flanco de subida desde 0 hasta un voltaje de entre 12 y 14 V, el microcontrolador entra en el modo llamado **Program/Verify**, en el cual ya se puede tener acceso para la lectura o escritura de datos. Los pines Vdd (14) y Vss (5) son para el voltaje de alimentación de 5 V. El pin RB7 (13) es la entrada y salida de datos en serie. Y el pin RB6 (12) sirve para introducir la señal de reloj que sincronizará los procesos.

Esta es la forma básica de escribir y leer en la memoria de programa del PIC. La comunicación se realiza mediante comandos, que controlan los procesos de grabación, lectura, borrado, etcétera. Pero no los estudiaremos ahora, ya que no pretendemos adentrarnos en los detalles, sino que sólo nos interesa el concepto básico de la grabación de microcontroladores PIC.

GRABADORES

Para enviar o grabar el programa en la memoria del microcontrolador se utiliza un dispositivo llamado **grabador**. Este grabador es un circuito en el cual colocaremos el microcontrolador para que pueda recibir los datos del archivo **.hex** que grabaremos en su memoria, con la ayuda de una computadora que será quien los envíe hacia éste siguiendo el protocolo de comunicación ICSP. La conexión se realiza mediante alguno de los puertos de comunicación de la computadora, como pueden ser, el puerto serial, el puerto paralelo, o por un puerto USB.

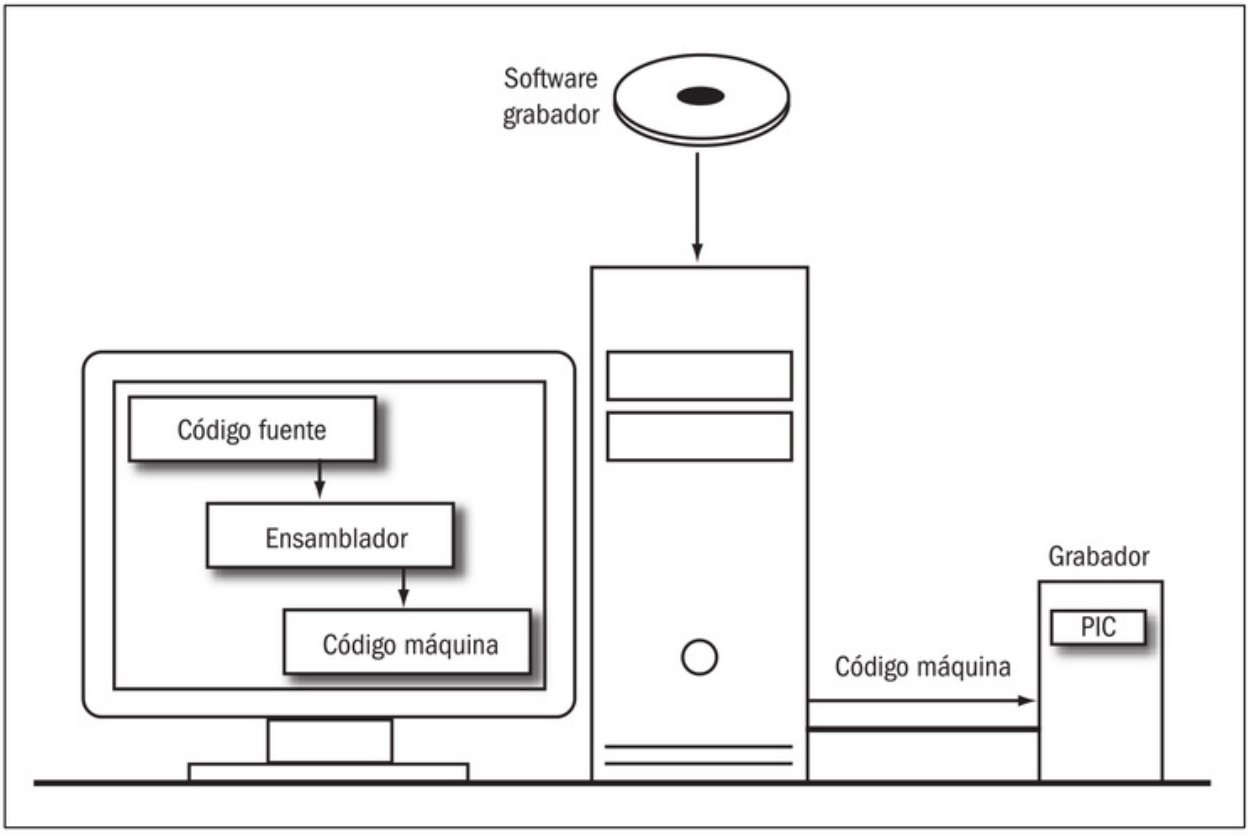


***Figura 2.** Un grabador es el dispositivo donde se coloca al PIC para que pueda recibir los datos del programa que grabaremos en él.*

Ahora que conocemos en qué consiste el grabador, veamos cuáles son los elementos básicos que necesitaremos para enviar el programa hacia nuestro PIC:

- Una computadora, donde escribiremos el código fuente del programa, y lo ensamblaremos para generar el archivo **.hex**, que es el código máquina a grabar, y desde ella lo enviaremos al grabador.
- Un grabador, que es un circuito o dispositivo especial para colocar el microcontrolador en él, y que recibirá el código fuente desde la PC para grabarlo en el PIC.
- Un programa o software que sea capaz de controlar la comunicación entre la computadora y el grabador para poder enviar con éxito el código máquina.
- El microcontrolador que colocaremos en el grabador para llevar a cabo la grabación.

El proceso de escribir el código fuente en la computadora y ensamblarlo para generar el archivo de código máquina **.hex** ya lo estudiamos en el **Capítulo 4**, por lo tanto, ahora vamos a estudiar las otras partes que necesitamos para grabar un PIC, es decir, los grabadores de PIC, y su utilización.

LOS GRABADORES PROFESIONALES

Como el uso de microcontroladores PIC es muy popular hoy en día, existen multitud de opciones en cuanto a los grabadores que podemos encontrar. La forma más profesional, aunque también es la más costosa, es adquirir un grabador profesional, que son principalmente comercializados por el propio fabricante de los microcontroladores PIC. Existen varios modelos disponibles, algunos de los más nuevos hoy en día son el **PICkit 2** ó el **PICkit 3**, que se conectan a la PC por medio de un puerto USB.

***Figura 3.** El PICkit 3 es fabricado por Microchip y es un grabador eficiente y muy confiable.*

Con estos grabadores podemos trabajar la gran mayoría de los microcontroladores PIC. La ventaja de adquirir un grabador como estos es la alta confiabilidad y compatibilidad en el funcionamiento, ya que está diseñado por el propio fabricante de los microcontroladores y eso asegura un correcto funcionamiento en todo momento. También existen otros grabadores profesionales, como **MPLAB PM3**, que puede grabar prácticamente todos los microcontroladores PIC. Este tipo de grabadores se conecta a la PC por medio de un puerto USB y también tiene la posibilidad de funcionar por sí solo, es decir, sin necesidad de la computadora, aunque el precio de estos dispositivos es muy elevado.

Las ventajas de usar estos grabadores profesionales son, como ya mencionamos, la total compatibilidad con los dispositivos a grabar, la confianza de que funcionarán correctamente y el hecho de que se compran absolutamente listos para usar. Pero la gran desventaja es que puede ser difícil encontrarlos en las tiendas locales y además los precios son elevados. Quedará como alternativa de cada uno si puede adquirir un grabador de este tipo. Para usarlos hay que estudiar los instructivos que incluyen.

Figura 4. *MPLAB PM3 es un grabador universal profesional que puede grabar prácticamente cualquier PIC.*

GRABADORES DE BAJO COSTO

El uso de un grabador profesional se recomienda cuando se trabaja constantemente con microcontroladores PIC y realmente se requiere de su confiabilidad y compatibilidad. Para los aficionados que no deseen desembolsar tanto dinero, existen otras alternativas. Por un lado están los grabadores que podemos adquirir en las tiendas de electrónica, que son generalmente más económicos que los profesionales. Otra alternativa es construir nuestro propio grabador, si así lo deseamos.

Grabadores JDM

Afortunadamente, existe mucha información en Internet acerca de grabadores de microcontroladores PIC económicos y fáciles de construir. Por ejemplo, los grabadores **JDM** y sus múltiples variantes son muy populares hoy en día. En la página web de su creador podemos encontrar la información sobre sus características: **www.jdm.homepage.dk/newpics.htm**. También podemos adquirir en alguna tienda de electrónica un grabador basado en los JDM, que son muy sencillos y económicos. Uno de los más populares es el conocido como **T20-SE**, el cual se conecta a la PC por medio del puerto serial.

Figura 5. *El grabador T20-SE basado en JDM es un grabador de bajo costo y lo podemos adquirir armado y listo para funcionar.*

También podemos encontrar otros grabadores económicos que no están basa-

dos en los JDM que mencionamos antes, pero que se conectan por medio del puerto USB, lo cual los hace muy sencillos de implementar. Además, la ventaja de comprar un grabador de bajo costo es que ya está listo para usar. No obstante, en caso de adquirir uno de estos grabadores, deberemos también leer la documentación que lo acompaña para aprender a utilizarlo.

CONSTRUCCIÓN DE UN GRABADOR DE PICS

Para quien así lo desee, propondremos ahora la construcción de un grabador de microcontroladores PIC de bajo costo. Si ya contamos con un grabador, ya sea armado previamente por nosotros mismos o comprado en alguna tienda del ramo, entonces podremos saltear esta sección. Antes de avanzar, debemos tener muy en cuenta que armar un grabador como el que describiremos tiene sus riesgos: un error puede ocasionar daños en los microcontroladores que intentemos grabar, en los componentes del propio grabador, o en la computadora donde lo conectemos, por lo tanto debemos tener mucho cuidado al construirlo, y la responsabilidad de daños correrá por nuestra cuenta. El grabador propuesto se conecta a la computadora por medio del puerto serial, por lo que si no contamos con uno en nuestra computadora, entonces será mejor adquirir o construir un grabador que pueda conectarse en otro tipo de puerto, por ejemplo, en un puerto paralelo o USB.

El grabador que proponemos aquí tiene las siguientes características:

- Se conecta a la PC a través del puerto serial.
- No necesita alimentación externa de voltaje.
- Puede grabar una gran variedad de microcontroladores PIC.
- Los componentes son fáciles de encontrar y económicos.

Además, nuestro grabador está basado en los JDM, por lo que es compatible con cualquier otro que construyamos o compremos también basado en JDM.

{} ¿GRABADOR, PROGRAMADOR O QUEMADOR?

Existen diversos nombres para los dispositivos que se usan para escribir en la memoria de programa de los PIC. Algunos los llaman **programadores**, pero el término se puede confundir con el acto de escribir un programa o programar. Otros los llaman **quemadores** de PIC, aunque en realidad no hay nada que se "queme". El término más aceptado es simplemente **grabador**.

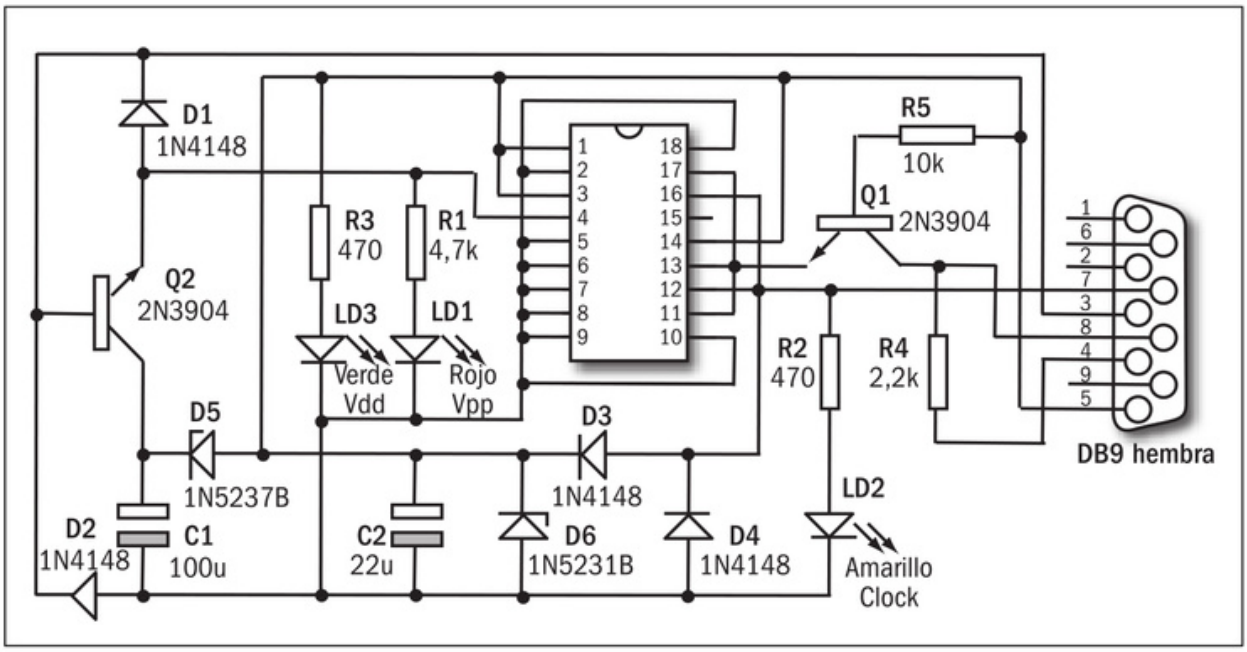


Figura 6. *El diagrama completo del circuito de nuestro grabador de bajo costo.*

La lista de materiales necesarios para construir el grabador es la siguiente:

R1: Resistor 4.7k
R2-R3: Resistores 470
R4: Resistor 2.2k
R5: Resistor 10k
C1: Capacitor electrolítico 100uf / 16V
C2: Capacitor electrolítico 22uf / 16V
Q1-Q2: Transistores NPN 2N3904
D1-D4: Diodos 1N4148
D5: 1N5237B Diodo zener 8.2V
D6: 1N5231B Diodo zener 5.1V
LD1: LED 3mm rojo
LD2: LED 3mm amarillo
LD3: LED 3mm verde
DB9: Conector DB9 hembra
ZIF18: Zócalo ZIF de 18 pines

Como podemos observar, estos componentes son bastante comunes, por lo que no tendremos mayor problema para encontrarlos en las tiendas locales de electrónica. En caso de no hallar los diodos zener indicados, podemos reemplazarlos por cualquier otro diodo zener de 8.2 V y 5.1 V de medio Watt, respectivamente. Todos los resistores son a 1/4 de Watt y 5% de tolerancia. Para quienes quieran construir un circuito impreso para el grabador, en el sitio web de la editorial (**www.redusers.com**) podemos descargar el archivo llamado **Grabador.pdf**, con el dibujo del circuito listo para imprimir.

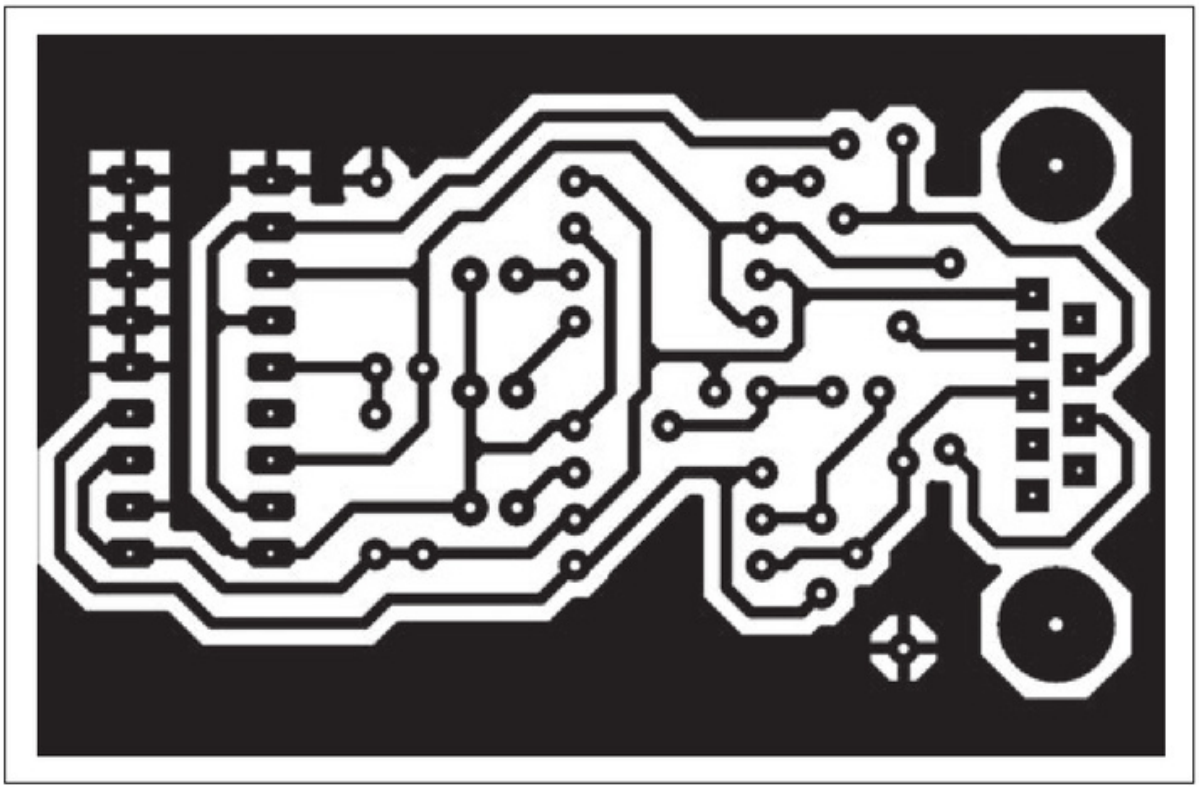


Figura 7. *Podemos construir un circuito impreso para el grabador basándonos en el archivo* ***Grabador.pdf****.*

Para armar el grabador en el circuito impreso, necesitaremos un conector DB9 hembra en ángulo recto, y un cable serial para conectarlo al puerto de la computadora.

También necesitaremos un **zócalo** o base para circuito integrado tipo **ZIF** de 18 pines. Si no logramos conseguir una base de tipo ZIF, podemos colocar en su lugar una base para circuito integrado común de 18 pines que es muy fácil de conseguir en cualquier tienda de electrónica. Aunque una base del tipo ZIF es sumamente útil para poder colocar y retirar los microcontroladores del grabador con mucha comodidad.

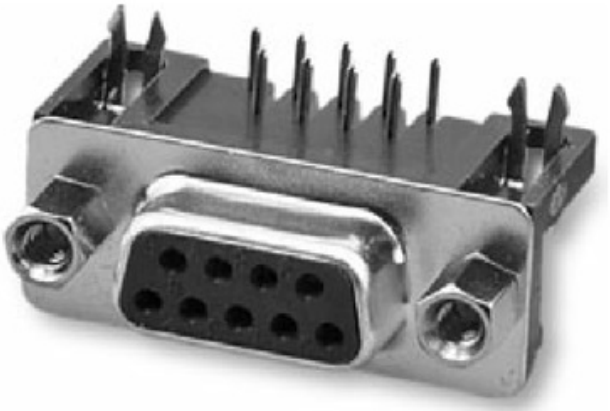


Figura 8. *Un conector DB9 hembra servirá para unir el grabador a un puerto serial (COM) de la PC.*

Como podemos observar, el circuito del grabador contiene tres leds. El led rojo indica el voltaje de programación Vpp, es decir, cuando el PIC esté siendo grabado, este led deberá encender. El led amarillo es la señal de reloj (Clock). Y, por último, el led verde indica el voltaje de alimentación de 5 V.

En la **Figura 11** podemos observar la ubicación y la forma de conexión de los componentes sobre la placa de circuito

impreso, si es que lo construimos. Debemos poner mucha atención al colocar los elementos, la orientación correcta de los diodos, leds y sobre todo de los capacitores electrolíticos. Los elementos marcados como **J1**, **J2** y **J3** son simples alambres que sirven como puentes de conexión.

Figura 9. *Una base de cero fuerza de inserción (ZIF) nos será de utilidad para colocar el PIC en el grabador.*

Figura 10. *Una base para circuito integrado común puede sustituir la base tipo ZIF si no la encontramos en las tiendas.*

Si no tenemos la posibilidad de construir el circuito impreso, no debemos preocuparnos ya que podemos armar el grabador en una **placa universal** o incluso en un **protoboard**, asegurándonos de que las conexiones sean correctas.

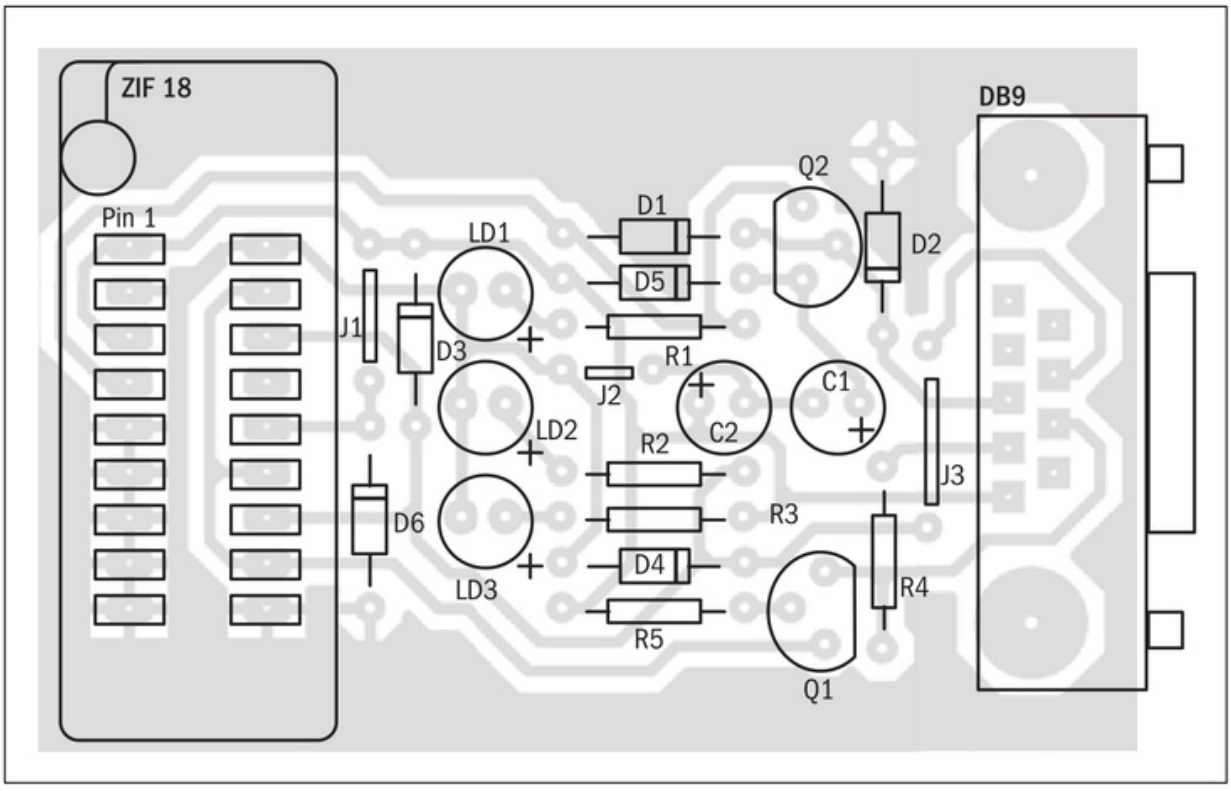


Figura 11. *Orientación y colocación de los componentes del grabador en la placa de circuito impreso.*

Este grabador puede trabajar con microcontroladores PIC de las siguientes familias: PIC12Fxxx, que son de 8 pines, y PIC16F62x y PIC16Fxx, que son de 18 pines, entre los que, por supuesto, está el PIC16F84A. Es importante colocar de forma adecuada el PIC en el zócalo del grabador, ya que si lo colocamos invertido podemos arruinarlo y dejarlo inservible. La forma de colocar los microcontroladores, tanto los de 18 pines como los de 8 pines, es con el pin 1 orientado hacia la palanca de liberación del zócalo ZIF. Como referencia, en la **Figura 11** indicamos cuál es el pin 1 del zócalo. Si utilizamos una base para circuito integrado común, entonces habrá que colocarla de tal forma que coincida también con la orientación adecuada.

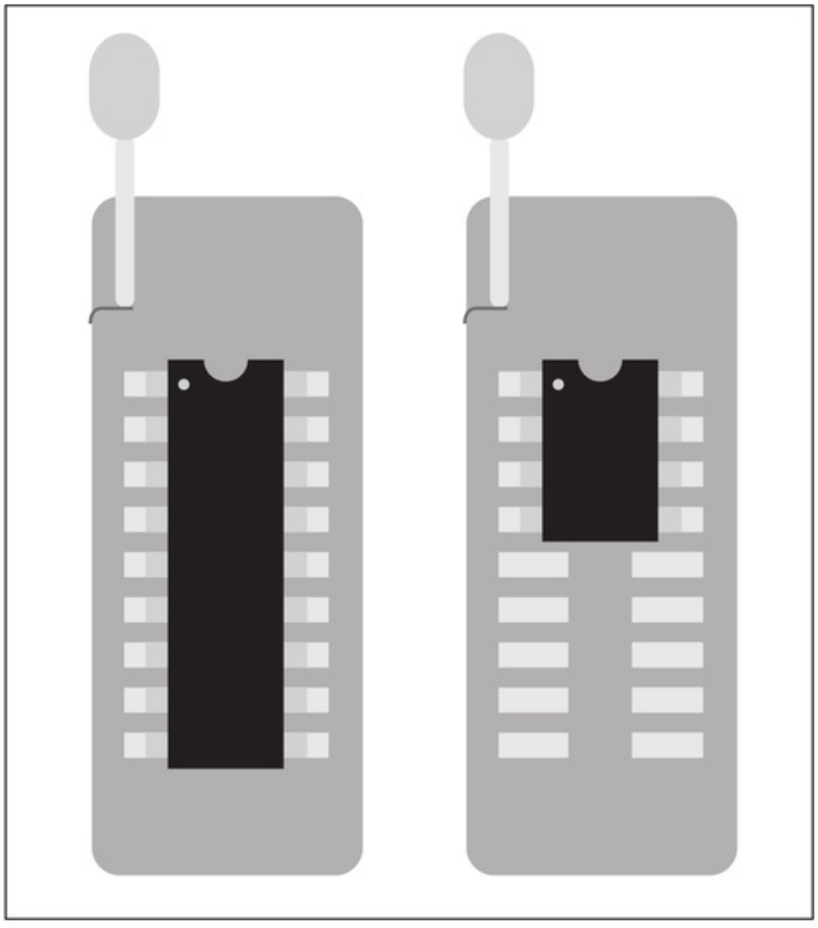


***Figura 12.** Colocación correcta de los microcontroladores de 18 y 8 pines en la base ZIF del grabador.*

Utilización del grabador

Una vez que hayamos construido nuestro grabador, veremos cuál es la forma de utilizarlo. Primero necesitaremos un programa que sea capaz de enviar el código fuente a grabar en el microcontrolador. Para ello emplearemos un programa gratuito llamado **IC-Prog**, el cual podemos descargar de su página web oficial: **www.ic-prog.com**. Este programa es muy fácil de utilizar, es completamente gratuito y es compatible con los grabadores basados en JDM.

Instalación y configuración de IC-Prog

IC-Prog es un programa muy pequeño y fácil de usar que no requiere de instalación especial, sólo hay que descargar desde su página web el programa y el driver, que están comprimidos en archivos tipo **.zip**, descomprimir los archivos **icprog.exe** e **icprog.sys** (que es el driver para Windows 2000 o XP), y guardarlos en una misma carpeta. Con esto ya habremos logrado instalar IC-Prog y estará listo para usar.

También podemos colocar un acceso directo a IC-Prog en el **Escritorio** para el archivo **icprog.exe**. La primera vez que lo ejecutemos, veremos una ventana que nos advertirá, precisamente, que es la primera vez que estamos utilizando el programa, y nos pedirá que configuremos nuestro hardware (el grabador).

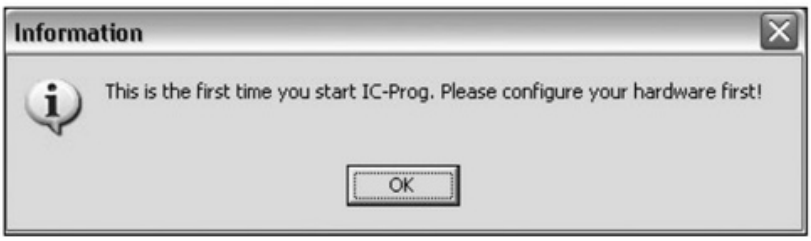


***Figura 13.** IC-Prog nos pedirá configurar el grabador la primera vez que intentemos abrirlo.*

Al presionar **OK**, se abrirá la ventana de configuración llamada **Hardware settings**, donde deberemos configurar el grabador que usaremos. En este caso, de la lista **Programmer:** elegiremos la opción **JDM Programmer**. En la sección **Ports** elegiremos el puerto serial en donde tendremos conectado nuestro grabador, que generalmente es **Com 1**. Las demás opciones las configuraremos tal como vemos en la **Figura 14**. Es muy importante que lo hagamos de la forma indicada para el correcto funcionamiento.

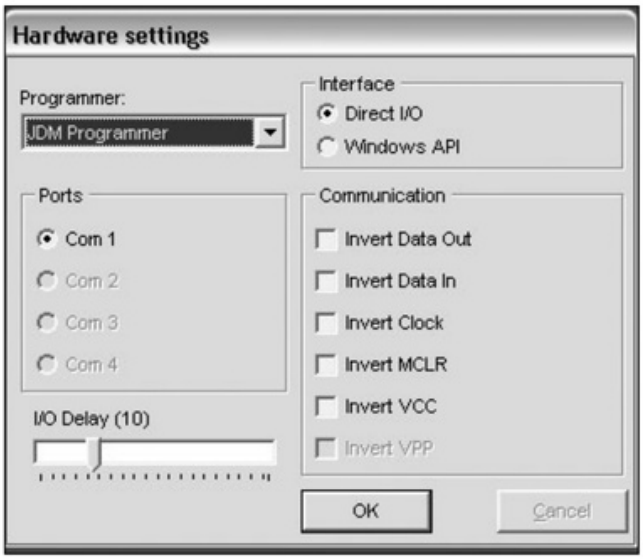


***Figura 14.** En la ventana Hardware settings elegiremos el grabador a usar y el puerto donde estará conectado a la PC.*

Al presionar **OK** en la ventana **Hardware settings**, y si estamos usando Windows 2000 o XP, aparecerá una pequeña ventana con el texto **Privileged instruction**, que nos advierte que necesitamos instalar el driver o controlador para estos sistemas operativos. Más adelante veremos cómo hacerlo.

***Figura 15.** Si utilizamos Windows 2000 o XP, debemos instalar el controlador icprog.sys para que funcione el programa.*

Al presionar **Aceptar** es posible que veamos otras ventanas de error, ya que todavía no hemos instalado el driver necesario para nuestro sistema operativo. Sólo debemos presionar **Aceptar** en ellas hasta que se abra el programa (**Figura 16**).

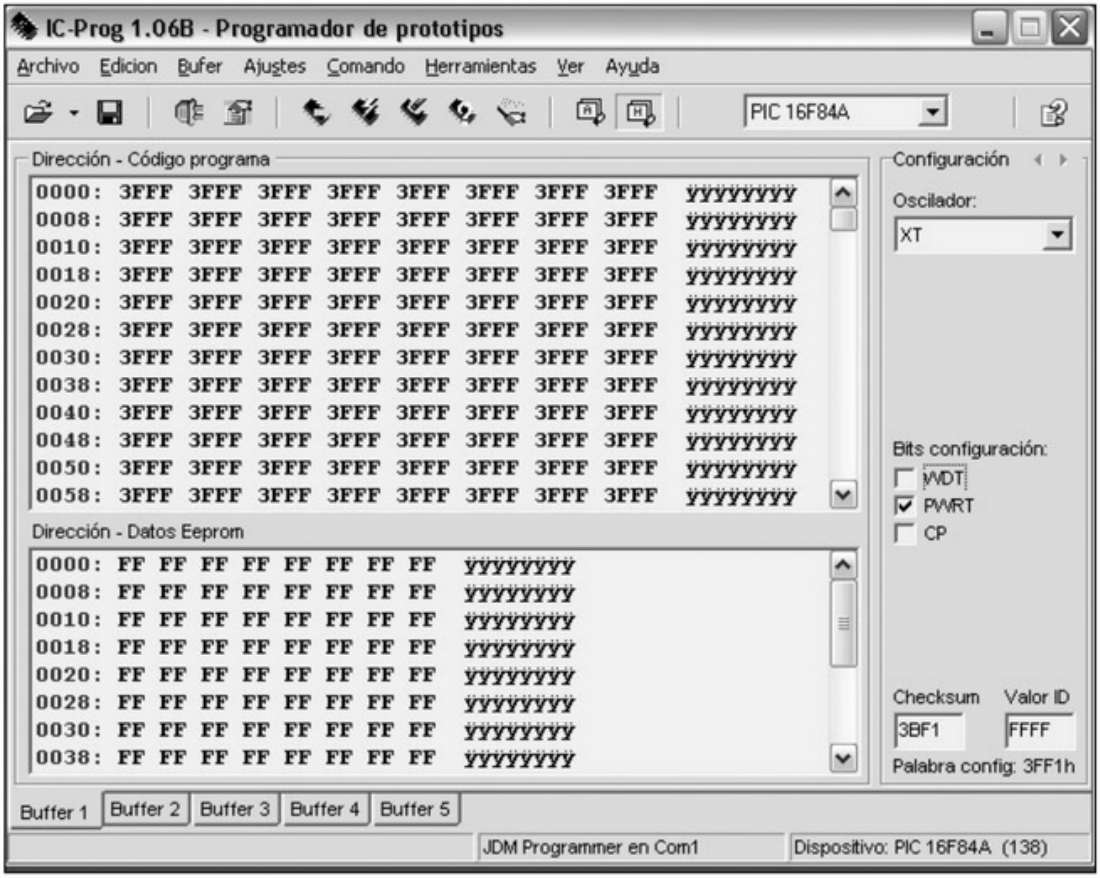


***Figura 16.** IC-Prog nos permitirá enviar nuestros códigos máquina al grabador para ser escritos en el PIC.*

De aquí en adelante, cada vez que abramos el programa aparecerá la ventana con la leyenda **Privileged instruction**, de modo que para evitarla debemos instalar el con-

trolador, que es el archivo **icprog.sys**, y que debe estar en la misma carpeta que el archivo **icprog.exe**. Después de asegurarnos de llevar a cabo este procedimiento, vamos al menú **Ajustes/Opciones** y en la ventana titulada **Opciones** hacemos clic en la pestaña **Miscelánea**. En ella activaremos la opción **Habilitar Driver NT/2000/XP** (**Figura 17**) y al hacerlo aparecerá un mensaje que nos preguntará si deseamos reiniciar IC-Prog para que el driver quede instalado, a lo cual diremos que sí.

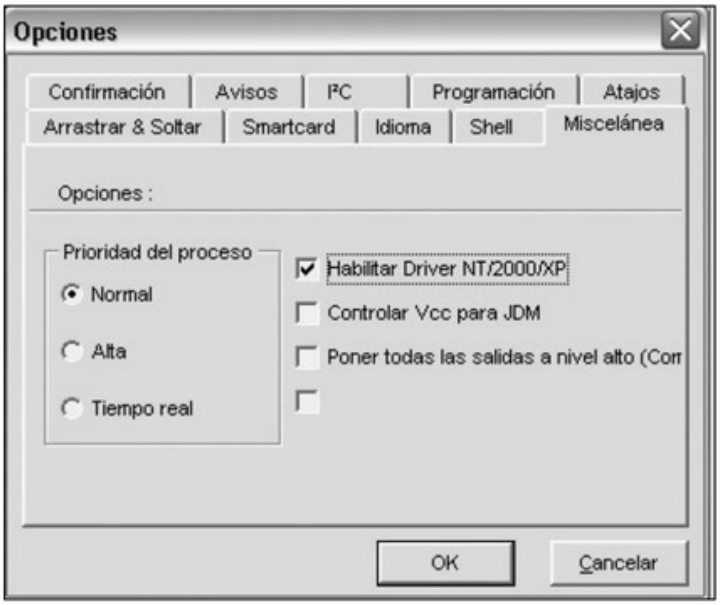


***Figura 17.** No debemos olvidar colocar el archivo icprog.sys en el lugar adecuado antes de habilitarlo en las opciones.*

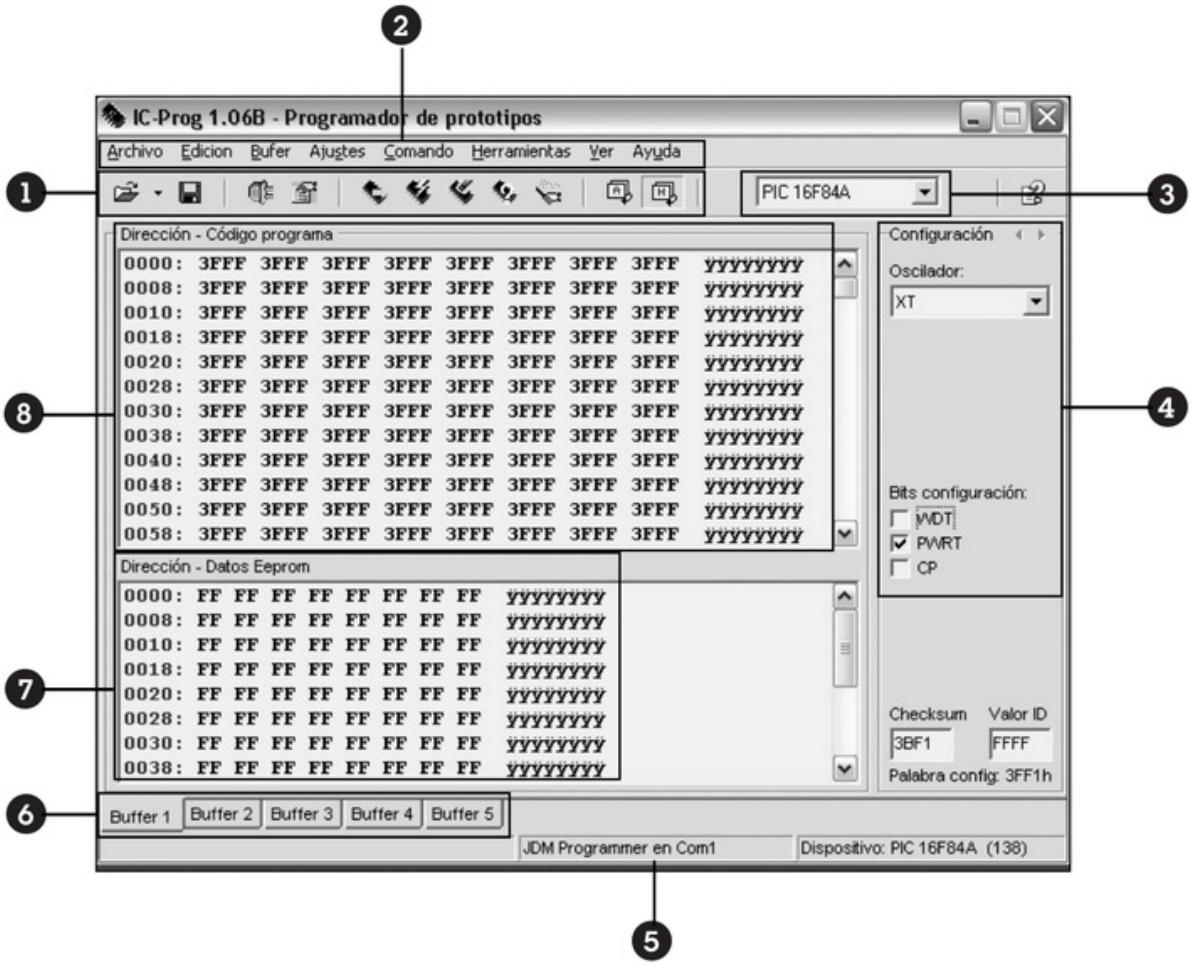
El programa se reiniciará en forma automática y nos preguntará si deseamos instalar el controlador, a lo cual obviamente le diremos que sí en la ventana que aparece, llamada **Confirm**. A partir de ese momento se abrirá normalmente el programa, y ya no aparecerá más el aviso de **Privileged instruction** las veces siguientes que lo usemos. Ahora sí ya estamos en condiciones de comenzar a usar IC-Prog para grabar nuestros microcontroladores PIC. Para otros grabadores el programa de grabación puede ser diferente usando nuestro grabador que construimos o cualquier otro basado en JDM, que sea compatible con IC-Prog. Veamos ahora cuáles son los elementos de la interfaz de IC-Prog para familiarizarnos con ellos.

III DE INGLÉS A ESPAÑOL

IC-Prog debe instalarse en el mismo idioma del sistema operativo, pero si por alguna razón esto no es posible, podemos cambiarlo de idioma en forma manual. Para ello vamos al menú **Settings/Options** y en la pestaña **Language** seleccionamos un idioma de la lista. En este caso elegimos la opción **Spanish**, y al hacer clic en el botón **Ok** ya tendremos IC-Prog en español.

La ventana principal de IC-Prog

GV



❶ **Barra de herramientas**: aquí tenemos diferentes botones que controlan principalmente las funciones de grabación, lectura, borrado, etcétera.

❷ **Barra de menús**: es la barra de menús típica de cualquier programa, desde donde accederemos a múltiples funciones del programa.

❸ **Selección de dispositivo**: en esta lista desplegable elegiremos el dispositivo con el que estamos trabajando. En nuestro caso debemos elegir el PIC16F84A.

❹ **Campo de configuración**: en esta zona elegiremos las opciones de la palabra de configuración del dispositivo. Más adelante hablaremos de esto.

❺ **Barra de estado**: en esta barra encontraremos información importante, como el dispositivo seleccionado o el grabador configurado.

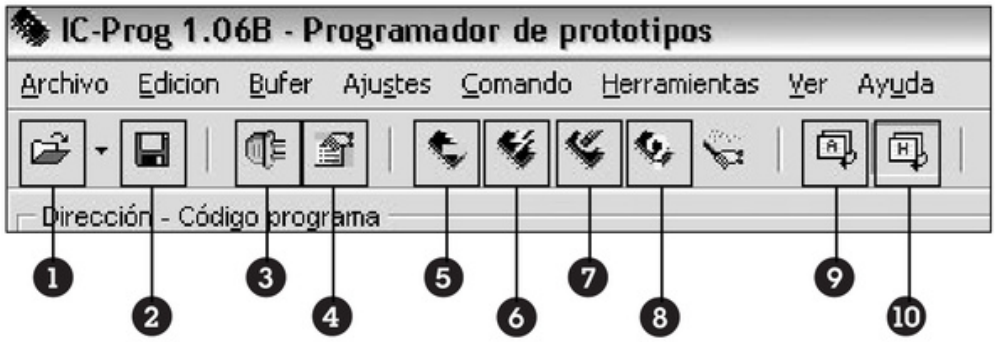
❻ **Pestañas de Buffers**: estas pestañas, llamadas **Buffer**, nos sirven para elegir una de ellas. En cada una podemos tener un archivo de código máquina abierto para trabajar con varios a la vez, si lo necesitamos.

❼ **Datos de EEPROM**: en esta zona es donde se muestran los datos que se grabarán o que han sido leídos de la memoria EEPROM de datos del PIC. Los datos se muestran en hexadecimal.

❽ **Código de programa**: en esta zona se muestran los datos del código de programa o código máquina que vamos a grabar o que hemos leído en nuestro microcontrolador. Se muestran en hexadecimal.

Podemos ver que la interfaz del programa es sencilla y no hay mayor problema para entenderla. Veamos ahora para qué sirven los botones de la barra de herramientas.

La barra de herramientas de IC-Prog GV



❶ **Abrir**: este botón nos permite abrir el archivo **.hex** que vamos a grabar. Al presionarlo aparecerá la ventana que nos permitirá explorar en las carpetas de nuestro disco duro hasta encontrar el archivo deseado y abrirlo. Al hacerlo, los datos del archivo aparecerán en la zona de código de programa.

❷ **Guardar como**: este botón nos permite guardar un archivo de código fuente **.hex**. Esto es útil si modificamos algún dato de la zona de código de programa y queremos guardar esos cambios, o guardar el archivo leído de un microcontrolador.

❸ **Configurar el hardware**: este botón nos da acceso a la configuración del programador, para elegir qué programador usaremos y el puerto en donde está conectado, como ya vimos antes.

❹ **Opciones**: con este botón abriremos la ventana **Opciones** para hacer ajustes en la configuración de diferentes parámetros del programa.

❺ **Leer todo**: con este botón podemos leer los datos del PIC que coloquemos en nuestro grabador, es decir, leeremos el programa almacenado en él, así como los datos de la memoria EEPROM y los bits de configuración. Si el PIC está protegido no podremos leerlo.

❻ **Programar todo**: este botón envía el archivo que tengamos abierto en el buffer activo al microcontrolador, para ser grabado en él.

❼ **Borrar todo**: este botón borrará todos los datos almacenados en el microcontrolador que tengamos colocado en el grabador. Debemos prestar atención a esta función ya que se perderá lo que esté grabado en el microcontrolador.
❽ **Verificar todo**: con este botón se realizará una verificación de los datos grabados en el PIC, es decir, se leerán los datos del PIC y se compararán con los del buffer activo para ver si son iguales.
❾ **Vista en ensamblador**: mediante la activación de este botón veremos en la ventana de código de programa el código fuente, es decir, se hará una traducción del código máquina a código fuente del archivo en el buffer activo.
❿ **Vista en hexadecimal**: mediante la activación de este botón veremos en la ventana de código de programa el código máquina codificado en hexadecimal.

LOS BITS DE CONFIGURACIÓN

Existe un registro especial en la memoria de programa del PIC16F84A, que está en la dirección **2007h**, que se encuentra fuera de la memoria de programa de usuario y pertenece a un espacio especial en ella. Esta palabra o **registro de configuración** sólo puede ser escrita durante el proceso de grabación, no puede modificarse mediante el programa. Existe en todos los microcontroladores PIC, pero en cada uno su configuración será diferente. Mediante esta palabra de configuración se pueden modificar o configurar algunos parámetros del funcionamiento del microcontrolador. En la **Tabla 1** tenemos representada la palabra de configuración.

Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
CP	CP	CP	CP	CP	CP	CP	CP	CP	CP	PWRTE'	WDTE	FOSC1	FOSC0

***Tabla 1.** La palabra de configuración del PIC16F84A.*

Las funciones de los bits son los siguientes:

Bit 13 a 4 (**CP**, bits de protección de código): estos bits sirven para configurar la protección del código o el programa grabado en el microcontrolador. Si se ponen a 1, el código no estará protegido, pero si se ponen a 0, el código estará protegido y no podrá ser leído. Esto sirve para proteger el programa grabado en nuestro PIC, para que otras personas no puedan tener acceso a él.
Bit 3 (**PWRTE'**, bit de habilitación del Power-up timer): este bit sirve para activar el

temporizador llamado **Power-up timer**, que estudiamos en el **Capítulo 2**. Si ponemos este bit a 1, el PWRTE estará deshabilitado, y con un 0 lo habilitaremos.
Bit 2 (**WDTE**, bit de habilitación del Watch dog timer): con este bit habilitaremos o deshabilitaremos el temporizador de perro guardián (*Watch dog timer*). Con un 1 se habilita, y con un 0 se deshabilita. En un capítulo posterior estudiaremos en detalle el tema del WDT, por ahora siempre lo deshabilitaremos mientras no lo utilicemos.
Bit 1-0 (bit de configuración del oscilador): estos dos bits sirven para configurar el tipo de oscilador que vayamos a utilizar en nuestro microcontrolador. En el **Capítulo 2** hablamos de los tipos de osciladores que podemos usar. Mediante estos bits seleccionaremos uno de ellos como mostramos a continuación:

00 = Modo LP
01 = Modo XT
10 = Modo HS
11 = Modo RC

De esta forma debemos configurar adecuadamente los bits de la palabra de configuración, antes de grabar el microcontrolador, ya que esta palabra o registro no puede modificarse posteriormente (a menos que grabemos de nuevo el PIC).

La directiva __CONFIG

Esta directiva nos ayudará precisamente a establecer los bits de la palabra de configuración desde nuestro código fuente. Si bien ya vimos un ejemplo de esta directiva en el **Capítulo 4**, cuando estudiamos nuestro primer programa, ahora veremos cómo funciona esta directiva en detalle.

Lo primero que hay que notar es que esta directiva inicia con **dos guiones bajos**. Es importante recordarlo para escribirla correctamente. Para poder usarla debemos incluir el archivo **P16F84A.INC** mediante la directiva **#INCLUDE**, ya que los parámetros para el uso de **__CONFIG** están definidos en este archivo. Si no lo incluimos, al ensamblar nos dará error. Un ejemplo del uso de la directiva **_CONFIG** es el siguiente:

III VERIFICACIÓN DESPUÉS DE LA GRABACIÓN

Normalmente, luego de grabar un PIC, se hace una verificación de los datos grabados, es decir, se leen los datos que se acaban de grabar en el microcontrolador y se comparan con los del archivo **.hex** que se utilizó. Si todos los datos son iguales, significa que se han grabado correctamente. De todas formas, podemos deshabilitar la verificación en las opciones del programa.

```
__CONFIG _CP_OFF & _PWRTE_ON & _WDT_OFF & _XT_OSC
```

Observamos que los parámetros se separan mediante el símbolo **&** para definir cada configuración de forma independiente. Se pueden colocar los elementos en cualquier orden, sólo siguiendo la correcta sintaxis, que es:

_CP_ON = Protección de código activada
_CP_OFF = Protección de código desactivada

_PWRTE_ON = Power-up timer habilitado
_PWRTE_OFF = Power-up timer deshabilitado

_WDT_ON = WDT habilitado
_WDT_OFF = WDT deshabilitado

_LP_OSC = Oscilador LP
_XT_OSC = Oscilador XT
_HS_OSC = Oscilador HS
_RC_OSC = Oscilador RC

De esta forma, en el ejemplo anterior tendremos el código de protección desactivado, el Power-up timer habilitado, el WDT deshabilitado y el oscilador en modo XT. La ventaja de usar la directiva **__CONFIG** está en que ya no necesitaremos configurar los bits de la palabra de configuración al momento de grabar el PIC, ya que al ensamblar automáticamente quedará configurada en el propio código máquina.

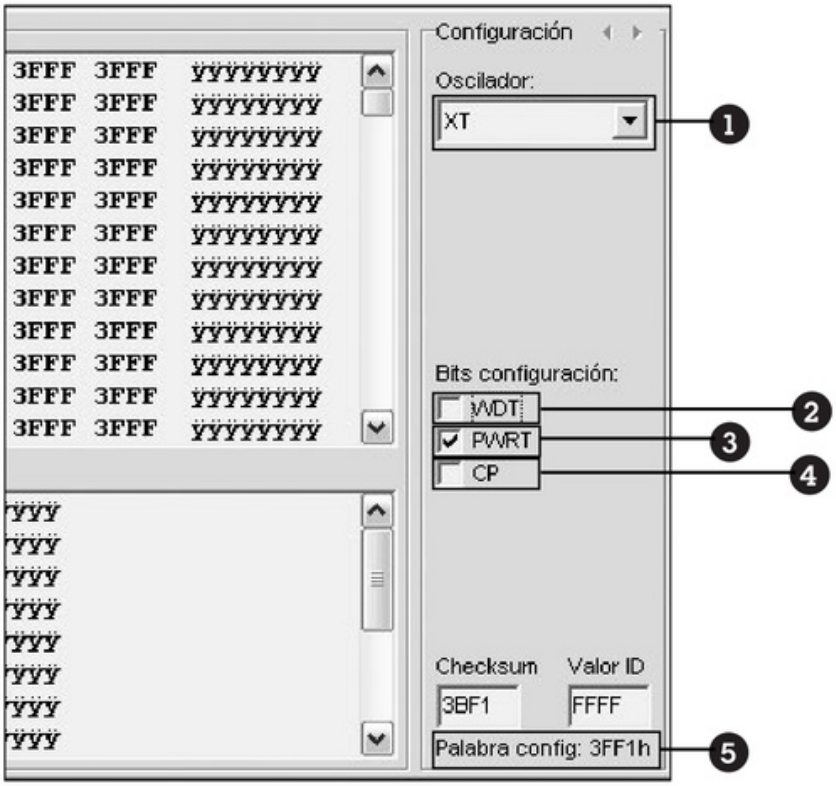
Los bits de configuración en IC-Prog

Como vimos, hay una zona a la derecha de la ventana principal de IC-Prog con el título **Configuración**, en la cual podemos definir los bits de la palabra de configuración del dispositivo elegido. Veámoslo en detalle:

III TODOS LOS BITS EN UNO

Si no definimos en algún momento los bits de la palabra de configuración, de manera predeterminada se pondrán todos a 1, por lo que la configuración predeterminada será: CP desactivado, PWRTE deshabilitado, WDT habilitado, oscilador en modo RC. Debemos poner mucha atención en esto o nuestros proyectos no funcionarán adecuadamente.

Los bits de configuración en IC-Prog GV



❶ **Oscilador**: esta lista nos permite elegir el tipo de oscilador que utilizaremos.
❷ **WDT**: esta opción nos permite habilitar el WDT. Si tiene marca es que está habilitado y si no la tiene es porque está deshabilitado.
❸ **PWRTE**: en esta opción podemos habilitar o deshabilitar el Power-up timer.
❹ **CP**: esta opción habilita o deshabilita la protección de código (Code Protection).
❺ **Palabra config**: aquí se muestra el valor en hexadecimal que será escrito en la palabra de configuración del microcontrolador, según las opciones elegidas.

Si definimos la configuración en el código fuente mediante la directiva **__CONFIG**, entonces al momento de abrir el archivo **.hex** en IC-Prog podemos observar cómo,

III CLONES DEL PICKIT 2

En la página web de Microchip podemos encontrar la documentación del grabador **PICkit 2**, la cual incluye el diagrama del circuito. Para los más atrevidos, es posible construirlo a partir de él. En Internet podemos encontrar algunos grabadores conocidos como **clones** de este grabador, que pueden ser una buena alternativa si queremos armar un grabador que se conecte por USB.

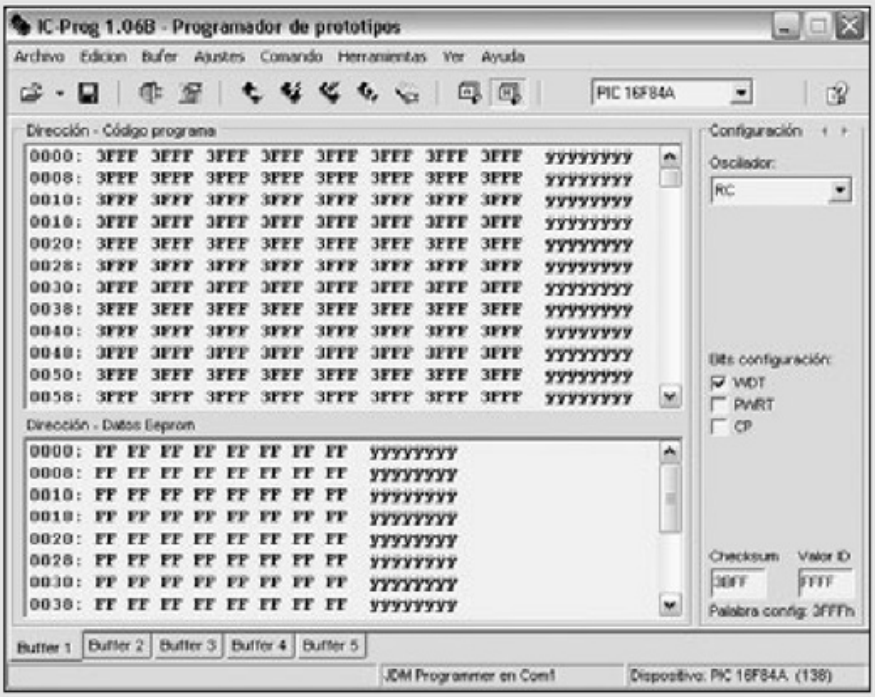
en el área de configuración, los valores activados en ella se corresponden con los que definimos en el código fuente. De esta forma ya no tenemos que configurar en IC-Prog, y nuestro código máquina estará completo para grabarse en el microcontrolador. Si lo necesitamos, también podemos modificar las opciones de la palabra de configuración en IC-Prog antes de grabar el microcontrolador.

GRABAR NUESTRO PRIMER PROGRAMA

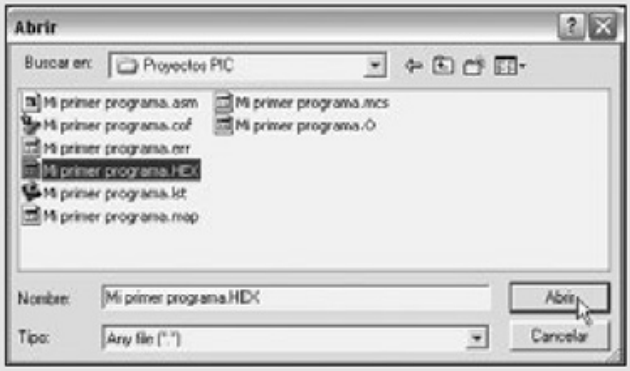
En el capítulo anterior estudiamos la programación y escribimos nuestro primer programa para el PIC16F84A. Pero el proceso no está completo hasta que grabemos el programa en código máquina en nuestro microcontrolador, para poder armar el circuito y que funcione completamente. Para eso estamos aprendiendo a diseñar los programas, para que gobiernen un circuito real. Veamos cómo hacer para grabar el microcontrolador con el archivo **.hex** que ya generamos, y el grabador que construimos, o cualquier otro basado en JDM. El primer paso es, por supuesto, tener conectado el grabador a la PC y con el microcontrolador a grabar colocado de manera adecuada en el zócalo.

Paso a paso: Grabar Mi primer programa con IC-Prog

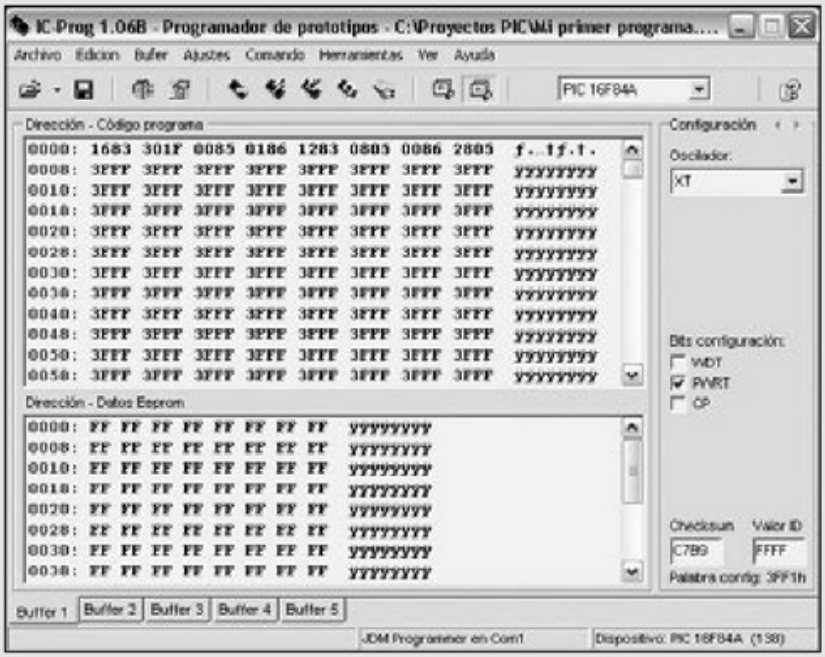
1 Una vez que tenga listo el archivo **.hex** resultante del ensamblado desde MPLAB de **Mi primer programa.asm**, abra IC-Prog.



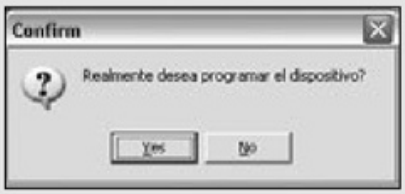
2 Vaya a **Archivo/Abrir archivo...** para acceder a la ventana **Abrir** y explore en su disco hasta encontrar el archivo **.hex** que desea grabar. Selecciónelo y haga clic en **Abrir**.



3 Una vez abierto el archivo a grabar notará cómo aparecen los datos en hexadecimal en la zona de **Código de programa**. Y si ha usado la directiva **__CONFIG**, los bits de configuración estarán tal como los configuró en ella.

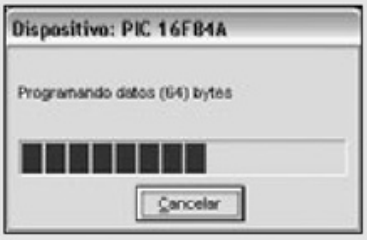


4 Para grabar el programa, haga clic en el botón **Programar todo** (el que tiene la imagen de un pequeño circuito integrado y un relámpago amarillo). Aparecerá una ventana de confirmación que le preguntará si desea grabar el dispositivo.



5 Una vez que presionó **Yes** (sí), aparecerá una pequeña ventana que le indicará el proceso de grabación del código del programa.

6 Después aparecerá otra ventana que indicará que se están grabando los datos en la memoria EEPROM, si es que los hay. Luego verá otra ventana que informará que se está grabando la palabra de configuración.



7 Una vez grabado el programa en el PIC, aparecerá una ventana que señalará que la verificación se está llevando a cabo. Esta sólo la veremos si tenemos activada la opción de verificar después de grabar en la configuración del programa.

8 Por último, aparecerá una ventana que indicará si la verificación se ha llevado a cabo con éxito. Si es así, entonces la grabación del microcontrolador está completa.

Una vez que hemos grabado exitosamente el programa en el PIC16F84A, entonces estamos listos para retirarlo del grabador y colocarlo en el circuito que propusimos en el **Capítulo 4**, que precisamente nos serviría para escribir nuestro primer programa. De esta forma, el proceso de diseño, escritura del programa y grabación estará completo y tendremos ya nuestro primer circuito con el PIC16F84A en funcionamiento.

Leer un microcontrolador

Como ya vimos antes, existe un botón en la barra de herramientas de IC-Prog llamado **Leer todo**, el cual nos servirá para leer el programa o código máquina grabado en nuestro microcontrolador. Esto puede ser útil para recuperar un programa del PIC si no contamos con el archivo fuente o con el propio código máquina. Para leer el programa sólo bastará con tener el PIC que vayamos a leer

colocado en el grabador y presionar el botón **Leer todo**, para que se realice la lectura. Al finalizar aparecerá el código leído en el **Buffer** activo.

***Figura 18.** Podemos leer el programa y los datos que contiene un PIC en su memoria, aunque sólo si no está protegido.*

Una vez que hemos leído el código máquina desde el PIC, podemos guardarlo desde IC-Prog como un archivo **.hex**. Para ello, iremos al menú **Archivo/Guardar como...** y aparecerá la ventana **Guardar como**, en donde exploraremos en nuestro disco hasta encontrar la carpeta deseada, pondremos un nombre al archivo en la sección **Nombre:** y lo guardaremos. También podemos grabar este código leído en otro microcontrolador, si lo deseamos. Sólo bastará con cambiar el microcontrolador en el grabador y llevar a cabo el proceso de grabación como ya conocemos.

Borrar un microcontrolador

Por supuesto, si queremos eliminar un programa (sin grabar otro) de nuestro microcontrolador, también podemos hacerlo. Esto se lleva a cabo mediante el botón **Borrar todo** de la barra de herramientas de IC-Prog. Al presionarlo, si hemos activado la opción de confirmación, aparecerá una ventana que nos pedirá que confirmemos si queremos borrar el dispositivo. En caso de elegir la opción **Yes**, entonces IC-Prog procederá con el borrado de la memoria del microcontrolador. Aparecerá una ventana que nos informará que el proceso de borrado se ha llevado a cabo con éxito. Si intentamos leer el dispositivo, entonces el buffer quedará con todas las casillas con el valor **3FFF**, dado que el proceso de borrado pone todos los bits de la memoria con unos.

... RESUMEN

En este capítulo hemos conocido la forma de escribir o grabar nuestros programas en la memoria de programa del microcontrolador. Propusimos la construcción de un sencillo y económico grabador de PIC basado en los grabadores JDM, y estudiamos en detalle su utilización junto con el programa IC-Prog. Conocimos, además, algunas alternativas como los grabadores profesionales. Con lo visto, ya estamos listos para comenzar a construir poderosos proyectos con microcontroladores PIC.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Para qué debemos grabar un PIC?

2 ¿Cómo se llama el protocolo de grabación de microcontroladores PIC?

3 ¿Qué es un grabador?

4 ¿En qué están basados los grabadores de bajo costo más populares?

5 ¿Cómo se llama el programa que nos ayuda con la grabación del PIC?

6 ¿Para qué sirve la palabra de configuración del PIC16F84A?

7 ¿Para qué se utiliza la directiva __CONFIG?

8 ¿Para qué sirve la opción llamada CP?

9 ¿Para qué sirve la opción Borrar todo de IC-Prog?

10 ¿Qué botón debemos presionar si queremos leer el programa grabado ya en un PIC?

PRÁCTICAS

1 Grabe el programa Mi primer programa.hex, que diseñó en el Capítulo 4, en el PIC16F84A; arme el circuito y compruebe que funcione correctamente.

2 En las prácticas del Capítulo 4 propusimos una modificación a Mi primer programa. Una vez realizada, ensamble el programa para generar el código máquina, grabe ahora este nuevo programa en el PIC16F84A y compruebe que funcione correctamente en el circuito.

Capítulo 6

Técnicas de programación en ensamblador

Para poder escribir programas eficientes para nuestros proyectos con microcontroladores PIC es importante estudiar algunas técnicas de programación.

En este capítulo veremos algunas de ellas y conoceremos los trucos de programación en lenguaje ensamblador, orientados principalmente al PIC16F84A.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

PROYECTOS CON MICROCONTROLADORES PIC

Para el diseño de un proyecto de manera eficiente, debemos seguir un orden. Por eso, en esta sección estudiaremos el proceso para poder diseñar nuestros proyectos de forma ordenada. Aunque los pasos expuestos no son todos obligatorios ni tampoco necesariamente debemos seguir este esquema, nos puede servir de guía al momento de intentar diseñar un proyecto.

1. **Definir el proyecto global**: lo primero que debemos hacer es decidir el proyecto, es decir, definir claramente qué es lo que queremos o necesitamos. Podemos preguntarnos: ¿qué es lo que necesito o deseo que haga mi proyecto?
2. **Diseño del circuito**: una vez que tenemos clara la función que debe cumplir el proyecto, podemos diseñar un circuito, el cual por supuesto será gobernado por el microcontrolador. En esta fase debemos diseñar por completo el circuito definiendo cuál es la tarea del PIC, ya que en base a esto escribiremos el programa. Es por eso que es muy importante determinar perfectamente el circuito.
3. **Definición del programa**: una vez diseñado nuestro circuito electrónico, estamos listos para comenzar a pensar en el programa que llevará su control. Para eso, primero debemos definir cuál es la función que cumplirá y cómo lo hará. Podemos utilizar una herramienta de diseño, como los **diagramas de flujo**, que nos ayudará a diseñar el funcionamiento de nuestro programa. Más adelante hablaremos en profundidad sobre los diagramas de flujo.
4. **Escritura del programa**: una vez definido el programa mediante el diagrama de flujo, es momento de escribirlo. Para ello ya aprendimos cómo hacerlo en el editor de MPLAB. En él escribiremos nuestro código fuente. Para escribir nuestros programas debemos conocer algunas técnicas de programación. En las secciones posteriores de este capítulo veremos algunas técnicas y trucos de programación.
5. **Ensamblado y simulación del programa**: una vez escrito el programa (el código fuente) estamos listos para ensamblarlo y de esta forma generar el código máquina. Además de esto, en este punto también podemos hacer uso de algún simulador. Como ya estudiamos, el uso de MPLAB SIM puede sernos de mucha utilidad para evaluar si nuestros programas funcionan correctamente, aunque este paso es opcional. También podemos usar otros simuladores si lo deseamos.
6. **Grabación del programa**: después de haber terminado la escritura del código fuente y el proceso de ensamblado del programa, ya estamos listos para grabar el programa en la memoria del microcontrolador, como estudiamos en el **Capítulo 5**.
7. **Prueba**: la grabación del programa nos permitirá colocar el microcontrolador en el circuito que hemos diseñado y observar si todo funciona según lo esperado. Si todo funciona correctamente, entonces podemos saltear el paso siguiente.
8. **Depuración**: si el programa o el circuito no funciona como lo habíamos planeado, entonces es tiempo de hacer una **depuración** tanto en el circuito físico, si es

necesario, como en el programa, donde puede haber algo que omitimos o que no funciona completamente bien. En este caso habrá que regresar al editor MPLAB y hacer los cambios y correcciones pertinentes.

9. **Documentación**: una vez que nuestro proyecto está funcionando correctamente según lo planeado, es bueno elaborar la documentación para tener claro cómo funciona, cómo hay que operarlo y demás. Esto ayudará a facilitar el uso del proyecto, sobre todo si lo va a emplear otra persona además de nosotros. La documentación también nos facilitará futuras modificaciones.

Diagramas de flujo

Un diagrama de flujo, como su nombre lo indica, es un **gráfico** que nos conduce paso a paso a la resolución de un problema o un proceso y consiste en una serie de símbolos especiales que nos van indicando gráficamente los procesos necesarios.

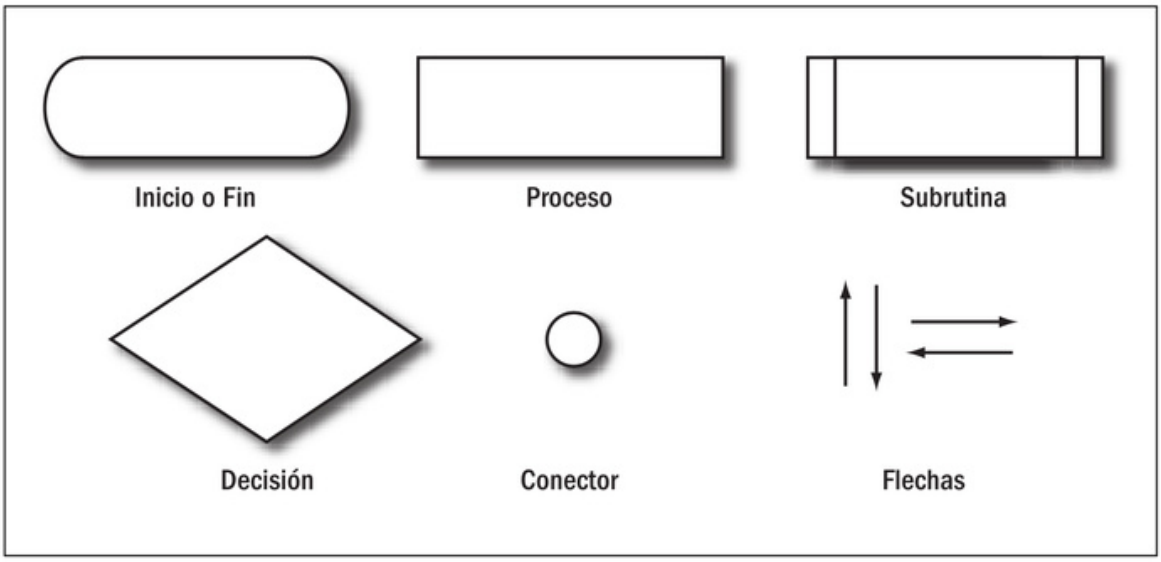


***Figura 1.** Los símbolos básicos usados en un diagrama de flujo definen los pasos de cómo se desarrollará el programa.*

En la **Figura 1** tenemos los principales símbolos utilizados para dibujar un diagrama de flujo. Para comprender el uso de cada uno veamos los siguientes ejemplos:

III OTROS USOS DE LOS DIAGRAMAS DE FLUJO

Aunque los diagramas de flujo se utilizan mucho en las tareas de programación, no son la única aplicación que tienen. En general, un diagrama de flujo puede emplearse para representar cualquier proceso o para resolver cualquier tipo de problema, por ejemplo, se puede dibujar un diagrama de flujo para describir el proceso de fabricación de un producto.

- **Inicio** o **fin**: este símbolo se usa para indicar el inicio o fin de un programa o un proceso. El diagrama de flujo siempre debe comenzar y terminar con este símbolo para definir perfectamente el inicio y el fin. De todos modos, algunos programas pueden no tener un final definido, por ejemplo, en algunos casos el programa o parte de él se repite constantemente, por lo que no hay un fin.

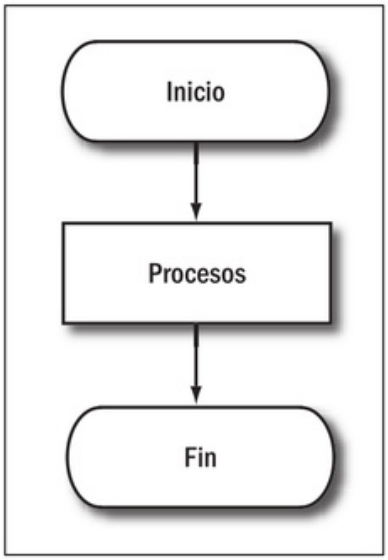


***Figura 2.** Los símbolos de inicio y fin enmarcan el programa o proceso completo.*

- **Proceso**: se usa para definir alguna operación o proceso individual o global. Se utiliza en la mayoría de los pasos del diagrama.

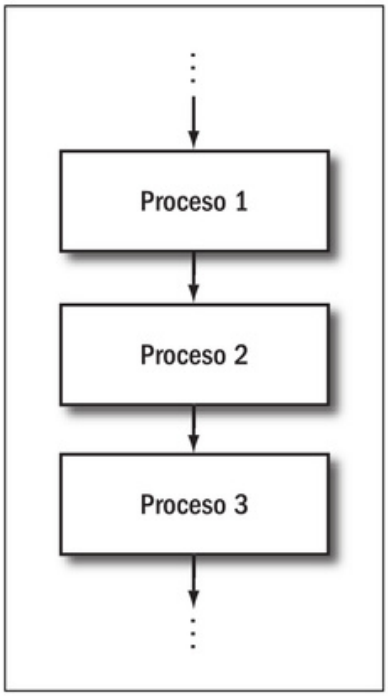


***Figura 3.** Los símbolos de procesos indican, en general, los procesos que se llevan a cabo en el diagrama de flujo.*

- **Subrutina**: se utiliza para indicar un llamado a una subrutina que llevará a cabo un proceso específico. La subrutina normalmente se detalla en un diagrama independiente, si se considera necesario hacerlo.

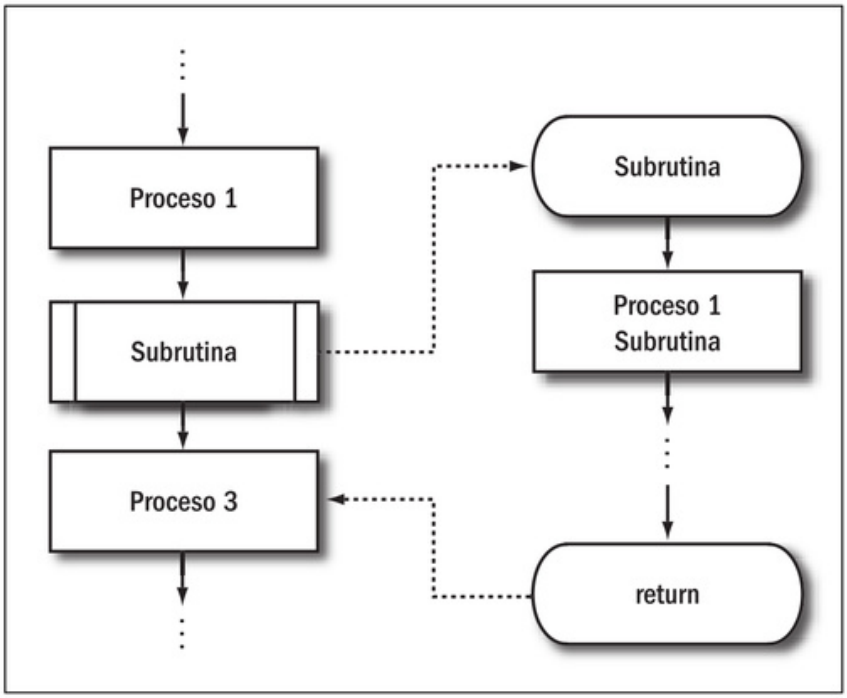


Figura 4. *Las subrutinas se indican en el cuerpo del programa y se detallan externamente si es necesario.*

- **Decisión**: se utiliza en los procesos en donde hay una decisión que tomar y en base a ella se seguirá un camino u otro, dependiendo si la condición fue cumplida o no.

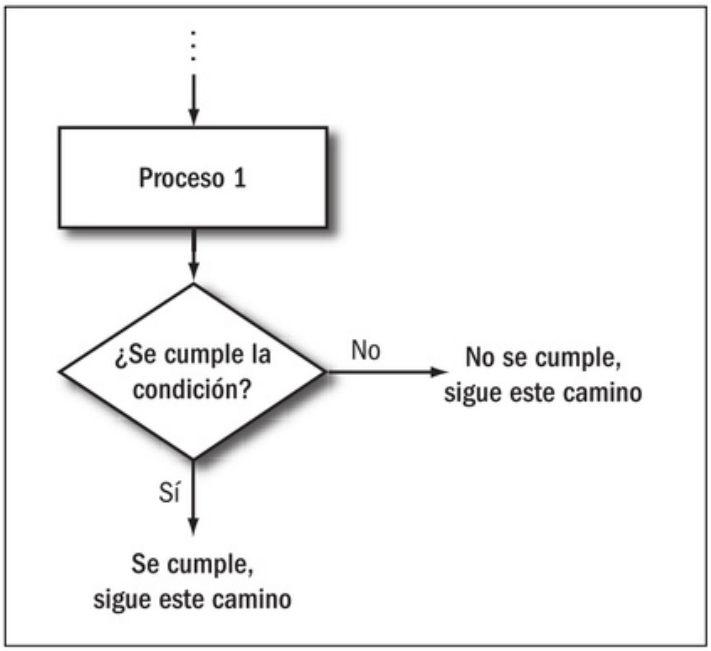


Figura 5. *El símbolo de decisión bifurca el diagrama en dos caminos diferentes, dependiendo de una condición.*

- **Conector**: en ocasiones, el diagrama de flujo puede ser muy grande (dependiendo de la complejidad y la extensión del programa que se intenta diseñar), por lo que no cabrá en una sola hoja. Los conectores sirven para poder llevar un orden en el camino que sigue el diagrama en caso de que necesitemos dividirlo. Generalmente, para conectar una sección con otra del diagrama, se utilizan números.

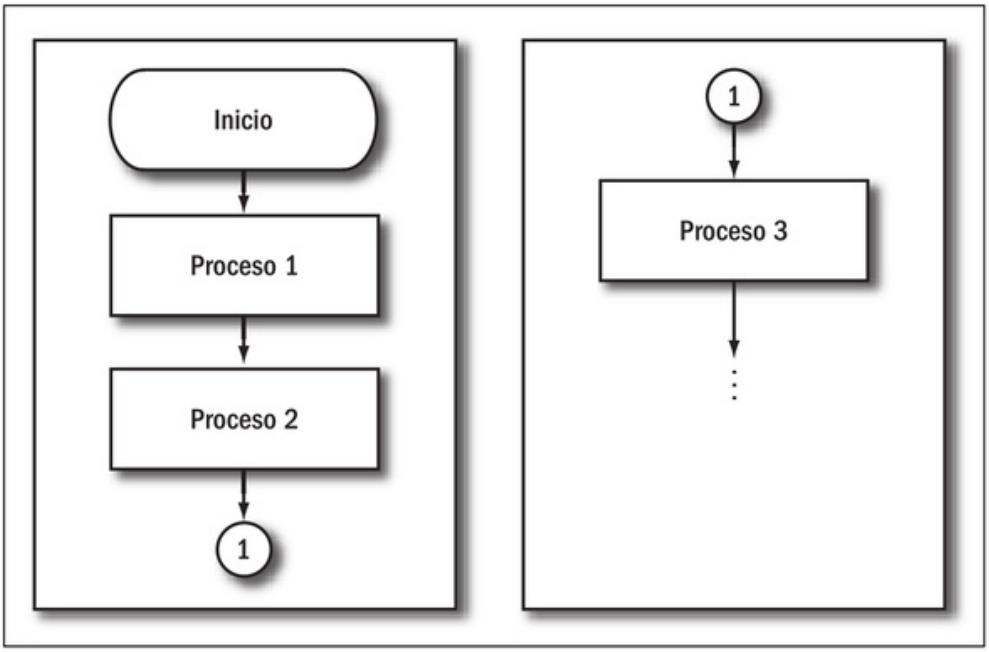


***Figura 6.** Los conectores se utilizan para interconectar un diagrama que está dividido en varias hojas o partes.*

- **Flechas**: indican hacia dónde sigue el flujo en el diagrama.

Es conveniente dibujar un diagrama de flujo antes de comenzar a escribir el programa, para tener un panorama claro de lo que se necesita y cómo se va a estructurar de manera global. De esta forma, será más fácil escribirlo. Para programas muy sencillos puede no ser necesario, ya que si el programa será corto no habrá mayor inconveniente para entender qué es lo que se requiere y cómo se debe hacer. La decisión de dibujar el diagrama de flujo es opcional, pero puede ser muy útil. Por ejemplo, podemos dibujar el diagrama de flujo de Mi primer programa,

{} OTROS SÍMBOLOS

Existen otros símbolos que podemos utilizar para diagramas de flujo, como la inclusión de documentos, salida o entrada de datos, etcétera. Sin embargo, no son tan empleados usualmente, o son utilizados en diagramas de flujo destinados para otros fines diferentes de los de la programación.

que estudiamos en los capítulos anteriores. En la **Figura 7** tenemos una posibilidad, el diagrama puede variar de acuerdo con los detalles que le agreguemos.

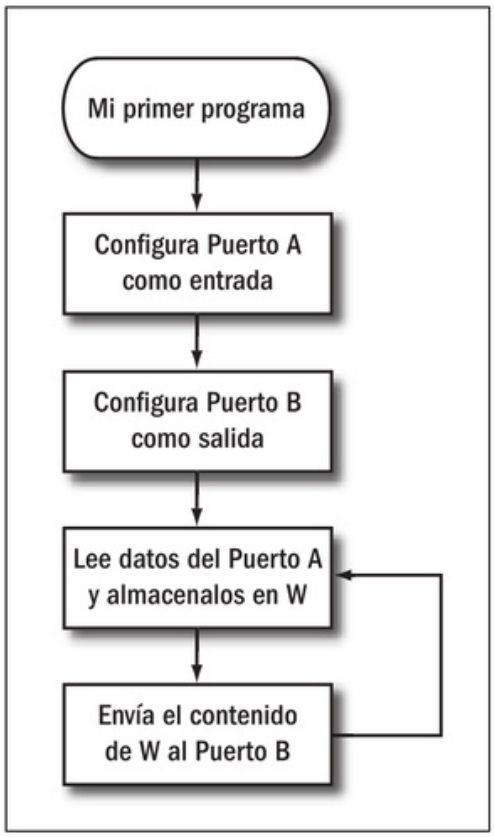


***Figura 7.** El diagrama de flujo de Mi primer programa es muy sencillo y corto, pero a la vez muy explícito.*

Podemos observar que el diagrama de flujo del programa es realmente muy sencillo, describe los pasos uno a uno para poder ver la estructura general del programa. Pero para escribir programas para nuestros microcontroladores PIC, no sólo basta con conocer las instrucciones, por lo que en las siguientes secciones estudiaremos algunas técnicas para la programación en ensamblador, lo cual puede resultar sumamente útil al momento de diseñar y escribir nuestros programas. Como un ejemplo más podemos ver en la **Figura 8** un diagrama de flujo con los pasos de diseño de proyectos con microcontroladores que hemos visto antes.

III VENTAJAS DE LAS SUBRUTINAS

Como veremos, existen muchas ventajas en usar una subrutina. Por ejemplo: reduce la duplicación innecesaria de código, permite el uso del código en diferentes programas, mejora la comprensión del programa y lo hace más manejable, divide la programación en pasos pequeños más simples, se puede dividir el trabajo en varios programadores o etapas, y se reduce la posibilidad de errores.

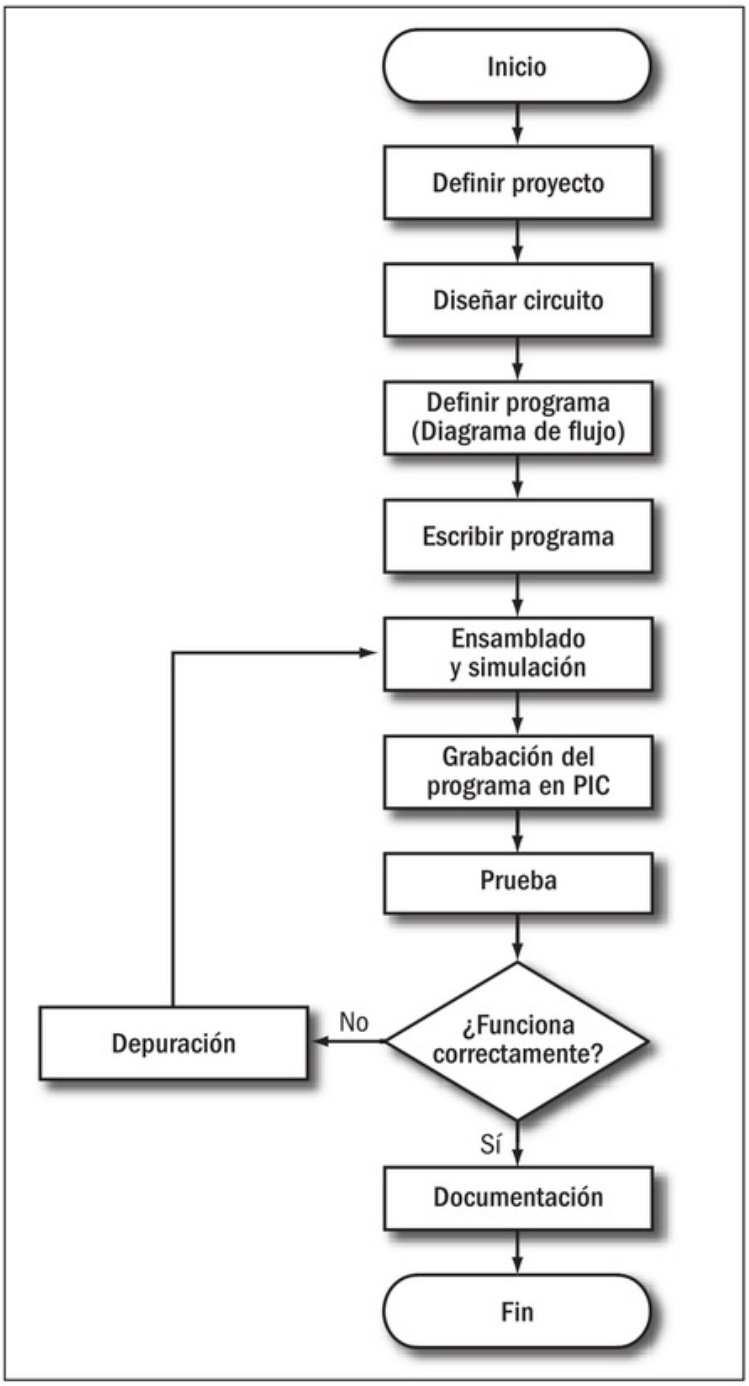


Figura 8. *El procedimiento de diseño expresado como un diagrama de flujo.*

SUBRUTINAS

En un programa que sigue una secuencia totalmente lineal, cada vez que se requiera un proceso especial, por ejemplo un cálculo específico, se inserta el código que realiza dicho cálculo o proceso. Esto representa la repetición del fragmento de código cada vez que se necesite, lo cual ocupa mucho espacio en la memoria de programa, dificulta un poco la lectura y, sobre todo su modificación o depuración.

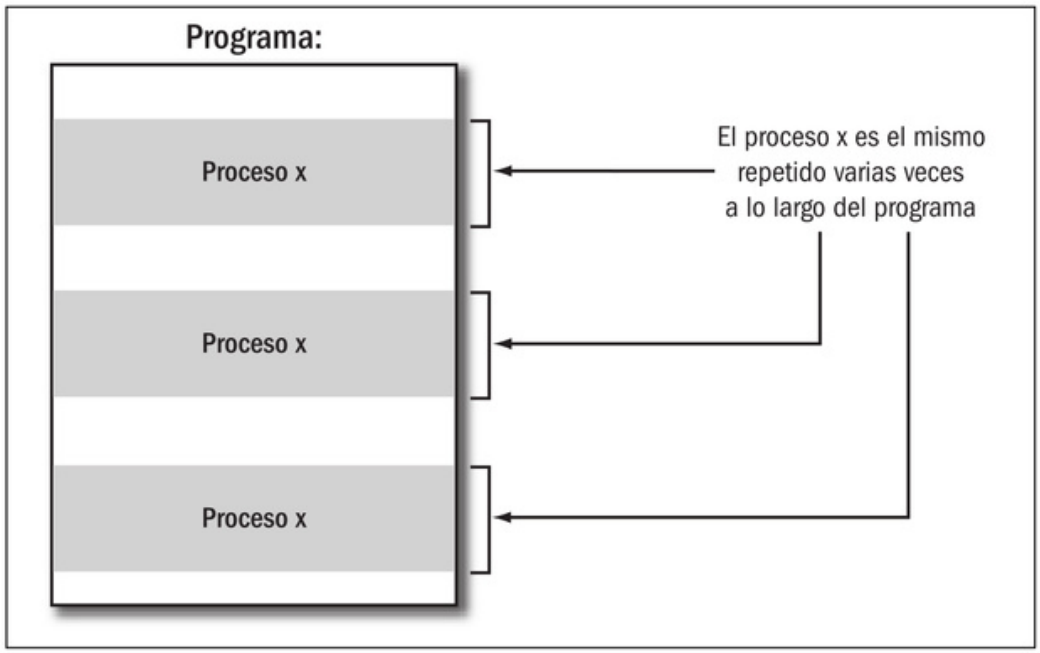


Figura 9. *En un programa sin subrutinas se repite código innecesariamente, haciendo el programa más largo.*

Para hacer los programas más cortos y más eficientes, se hace uso de **subrutinas**. Una subrutina es una parte de código separado del cuerpo principal del programa que lleva a cabo un proceso definido, y al cual se puede acceder en cualquier parte del programa mediante un **llamado a subrutina**. Como ya estudiamos, los llamados a subrutina se realizan mediante la instrucción **call**.

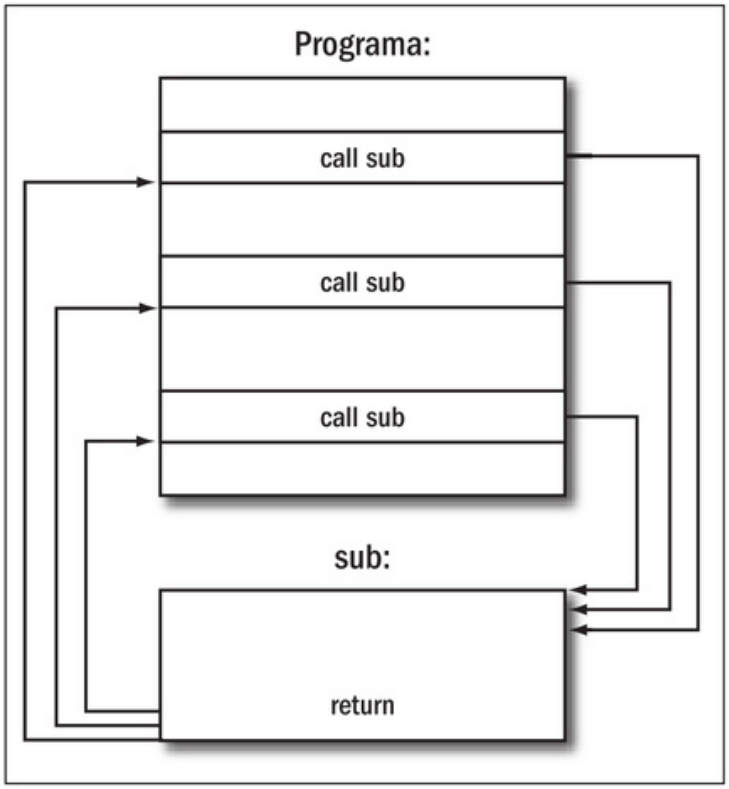


Figura 10. *Un programa con subrutinas es más eficiente y corto.*

Cuando llamamos a una subrutina, se realiza un salto hacia la dirección donde comienza la subrutina con el código necesario para llevar a cabo el proceso que se requiera. Una vez que se ha completado, se regresa a la siguiente instrucción después del llamado, es decir, a la siguiente instrucción después del **call** que llamó a la subrutina, y el programa principal continúa ejecutándose. El regreso de la subrutina se hace automáticamente mediante la instrucción **return**, por lo tanto, todas las subrutinas deben terminar con la instrucción **return** (o **retlw**). Se puede llamar a la subrutina todas las veces que sea necesario. De esta forma, el código de la subrutina sólo estará una vez. Y por supuesto, en el programa puede haber más de una subrutina. Un programa con una subrutina en él puede verse de la siguiente forma:

```
inicio      .
            .
            .
            call proceso        ;Llama a la subrutina llamada proceso
            .                   ;Regresa de la subrutina en esta instrucción
            .
            .
            call proceso        ;Llama a la subrutina llamada proceso
            .                   ;Regresa de la subrutina en esta instrucción
            .
            .

proceso     .                   ;Inicia la subrutina llamada proceso
            .
            .
            return              ;Instrucción de retorno, termina la
subritina

            .
            .
            .

            END
```

Subrutinas anidadas

Como hemos señalado, las subrutinas se pueden llamar en cualquier lugar del programa donde se necesite, incluso se puede llamar a una subrutina dentro de otra. Este proceso se conoce como **subrutinas anidadas**.

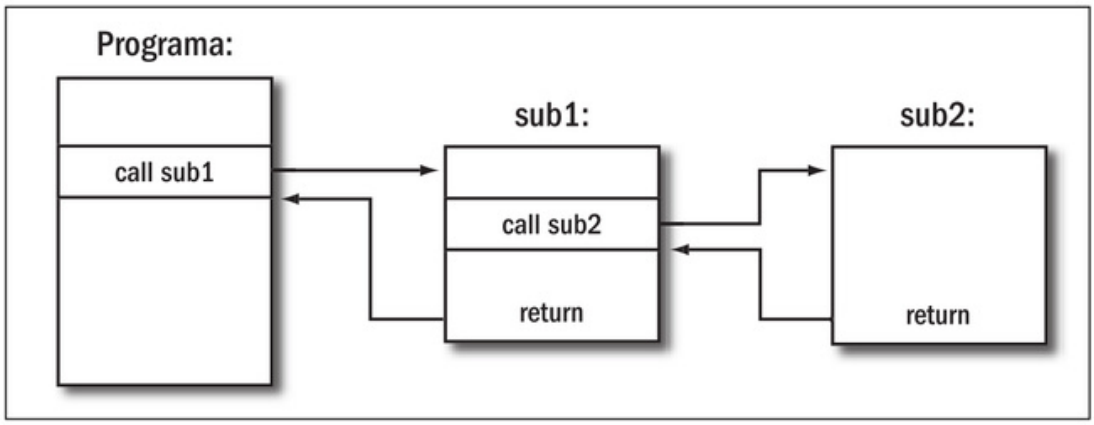


Figura 11. *El proceso de anidamiento de subrutinas tiene un límite.*

Cuando se llama a una subrutina dentro de otra, la última debe regresar al punto posterior al que fue llamada, y así hasta salir de la primera subrutina que contiene los llamados a las demás, y de esta forma regresar el control al programa principal.

La pila y las instrucciones call y return

Hemos visto el funcionamiento de las subrutinas y el proceso de anidamiento, pero ¿cómo sabe la subrutina cuál es el punto a donde debe regresar al final? Recordemos que en el **Capítulo 2** estudiamos la arquitectura interna del PIC16F84A. En la **Figura 1** del **Capítulo 2** tenemos el diagrama interno, donde se muestra arriba a la izquierda la **pila** (en inglés, *stack*), la cual tiene **8 niveles**. Esta pila es un espacio de memoria independiente que sirve para almacenar temporalmente direcciones de memoria. Está compuesta por registros de memoria de 13 bits de longitud cada uno y sirve para almacenar las direcciones de retorno cuando se llama a subrutinas. De esta forma, la subrutina sabe dónde debe regresar cuando llega a la instrucción **return**.

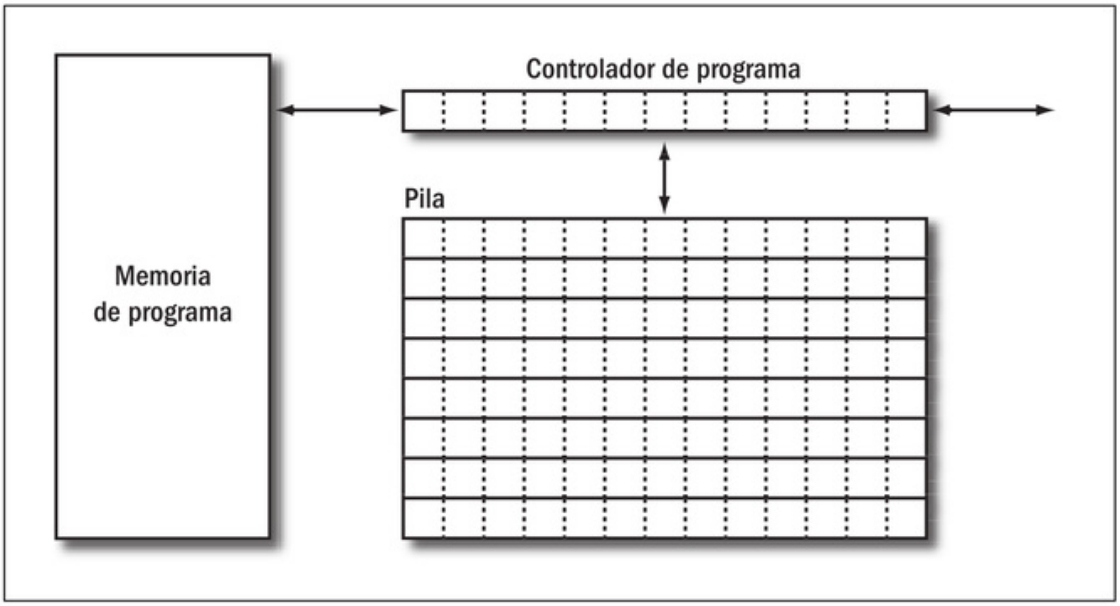


Figura 12. *La pila es una serie de registros donde se almacenan las direcciones de retorno de las subrutinas.*

La pila es de tipo **LIFO** (*Last In First Out*), es decir, el último dato que entra en ella es el primero en salir. Imaginemos la pila como un vaso en donde pondremos pequeñas pelotas (cada pelota representa un dato). Cuando ponemos la primera pelota en la pila (el vaso) podemos tomarla en cualquier momento, pero si ponemos una y luego otra, entonces no podemos sacar la primera que pusimos, ya que está debajo de la segunda, de modo que primero debemos sacar la segunda pelota, que es la última que pusimos. Así es como el último dato colocado es el primero que debe salir. Este es el concepto de una memoria tipo LIFO, como lo es la pila del PIC16F84A, que tiene 8 niveles. La operación de escribir un dato, o en este caso una dirección en la pila, se llama **Push**, y la operación de recuperarlo se llama **Pop**.

De esta forma, tenemos 8 posibles niveles de anidamiento para los llamados a subrutinas, ya que la pila del PIC16F84A es capaz de almacenar hasta 8 direcciones de retorno. De esta forma, la operación de las subrutinas es la siguiente:

- Cuando el programa encuentra por primera vez una instrucción **call**, suceden dos cosas: la primera es que la instrucción **call** ocasiona que se escriba el valor actual del contador de programa más uno (**PC+1**) en la pila; la segunda es que se coloca la dirección indicada en la instrucción **call** en el contador de programa (PC).
- De esta forma se provoca un salto hacia la dirección de la memoria de programa donde empieza la subrutina y ésta comienza a ejecutarse.
- Si dentro de la subrutina actual hay un llamado a otra subrutina, la nueva instrucción **call** coloca el valor actual del **PC+1** en la pila. Esto provoca que el valor anterior quede debajo de este último en la pila. Y nuevamente hace el salto hacia la dirección donde está la nueva subrutina.
- Si ya no hay más llamados a subrutina, la subrutina actual se ejecutará hasta encontrar la instrucción **return**, que indica el regreso de esta subrutina. La instrucción **return** hace que se tome la última dirección colocada en la pila, entonces se realiza el salto hacia esa dirección, y ahora sólo queda una dirección en ella.
- Como ya salimos de una subrutina, estamos ahora en la primera, y al encontrar la instrucción **return**, ésta tomará el valor que queda en la pila y saltará hacia esa dirección regresando al programa principal.

DESBORDAMIENTO DE LA PILA

Como hemos mencionado, la pila del PIC16F84A sólo tiene 8 niveles, por lo que sólo podemos colocar 8 direcciones en ella. Si intentamos colocar una novena dirección, ésta sobrescribirá la primera dirección en la pila, y se producirá un **desbordamiento**, lo que implica que hemos perdido una dirección y el programa se saldrá de control debido a esto.

En la **Figura 13** podemos ver gráficamente el proceso completo del funcionamiento de las subrutinas. De esta forma debemos asegurarnos de no utilizar más de 8 subrutinas anidadas, ya que la capacidad de la pila las limita a ese número.

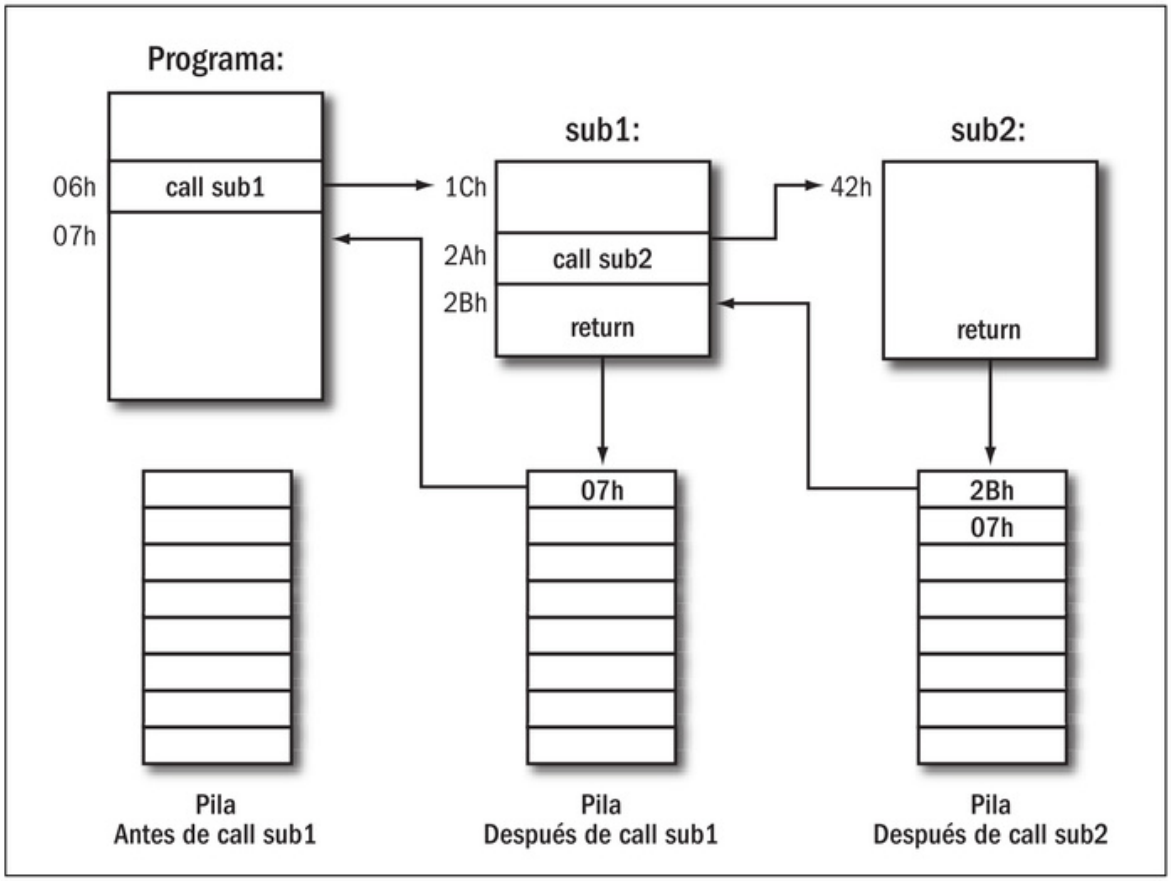


***Figura 13.** El funcionamiento detallado de las instrucciones call y return y su conjunción con la pila.*

SALTOS

Como aprendimos en el **Capítulo 3**, dentro del repertorio de instrucciones del PIC-16F84A tenemos algunas instrucciones de saltos, es decir, que llevan el programa a un lugar diferente. Existen variados tipos de saltos y algunas técnicas para usarlos. A continuación, veremos cuáles son.

Salto incondicional

La instrucción **goto** es un salto incondicional, es decir, no se requiere ninguna condición para llevar a cabo el salto. En cualquier lugar donde se encuentre una instrucción **goto**, el programa saltará al lugar donde la instrucción se lo indique. En el código fuente los saltos indican el lugar a donde saltarán mediante **etiquetas**.

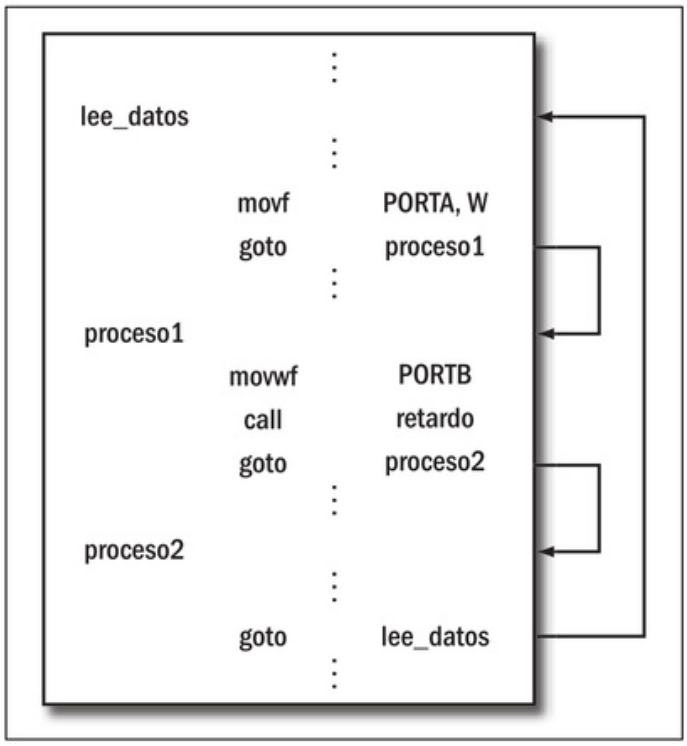


Figura 14. *La instrucción goto lleva el flujo del programa hacia otros sitios de forma incondicional.*

El operador $

El operador **$** devuelve el valor del contador de programa (PC) y puede ser útil para definir saltos sin usar etiquetas. Dado que **$** devuelve el PC, se puede hacer una suma o resta con él, para definir un salto hacia delante o atrás. Por ejemplo en este código:

```
aquí
     goto          aquí
```

Se genera un bucle que no sale de la instrucción **goto**. Si usamos el operador **$** entonces lo podríamos hacer simplemente así:

```
     goto          $
```

Lo cual es equivalente, ya que el operador **$** cargará la dirección del PC que es la dirección de la instrucción **goto**, por lo que se generará también un bucle sin salida. Para no generar el bucle sin salida podemos hacer una suma o resta de esta forma:

```
     goto          $+3
```

Lo cual ocasionará que al PC se sume un valor de 3, y entonces saltará a la dirección actual (la dirección del **goto**) más tres, es decir saltará a la tercera instrucción después del **goto**. También se puede hacer una resta para saltar hacia atrás, por ejemplo:

```
goto          $-6
```

El uso del operador **$** es igual que poner etiquetas, y es muy útil en saltos cortos.

Saltos condicionales

Un salto condicional es aquél en donde se lleva a cabo el salto sólo cuando una condición se cumple. En estos saltos hay una bifurcación del programa dependiendo de si la condición puesta se cumple o no, y el programa seguirá caminos diferentes.

Salto condicional en base a registros

Las instrucciones que permiten saltos condicionales en base a registros son **decfsz** e **incfsz**, y aunque ya analizamos su funcionamiento en el **Capítulo 3**, ahora veremos más detalles. Estas instrucciones permiten un salto condicional basado en el valor contenido en un registro después de incrementarlo o decrementarlo. Si el resultado en cuestión es diferente de 0, el programa tomará un curso, y si el contenido del registro es 0, el programa tomará un curso diferente.

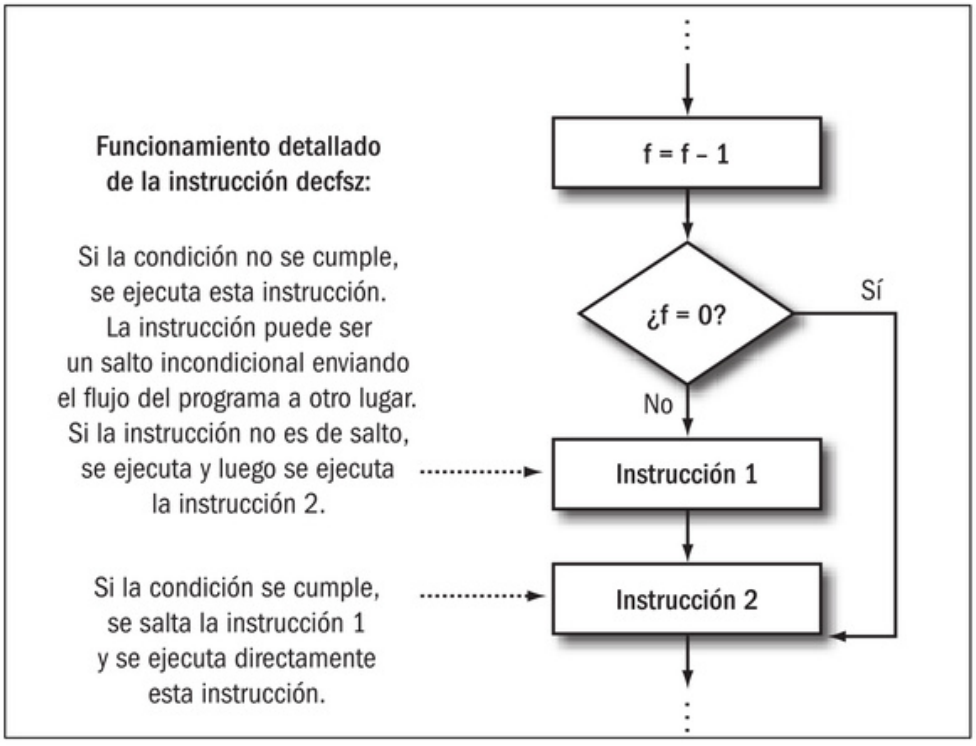


***Figura 15.** La instrucción decfsz es un salto condicional con decremento en un registro. Aquí podemos ver su funcionamiento en detalle.*

En la **Figura 15** tenemos una explicación gráfica detallada del funcionamiento de la instrucción **decfsz**. La instrucción **incfsz** es exactamente igual, excepto que en ésta se hace un incremento (**f = f +1**) en lugar de un decremento (**f = f - 1**). En ambas instrucciones hay un salto que evita una de las instrucciones cuando la condición de que el registro llega a 0 se cumple. Estas instrucciones son típicamente usadas para crear lazos o bucles que se ejecutan un número determinado de veces, como veremos más adelante en este capítulo.

Salto condicional en base a bits

También se puede hacer un salto condicional en base al valor de un solo bit de un registro. Las instrucciones que hacen esto son **btfss** y **btfsc**, en las cuales ahora la condición es que un bit sea 1 ó 0, respectivamente.

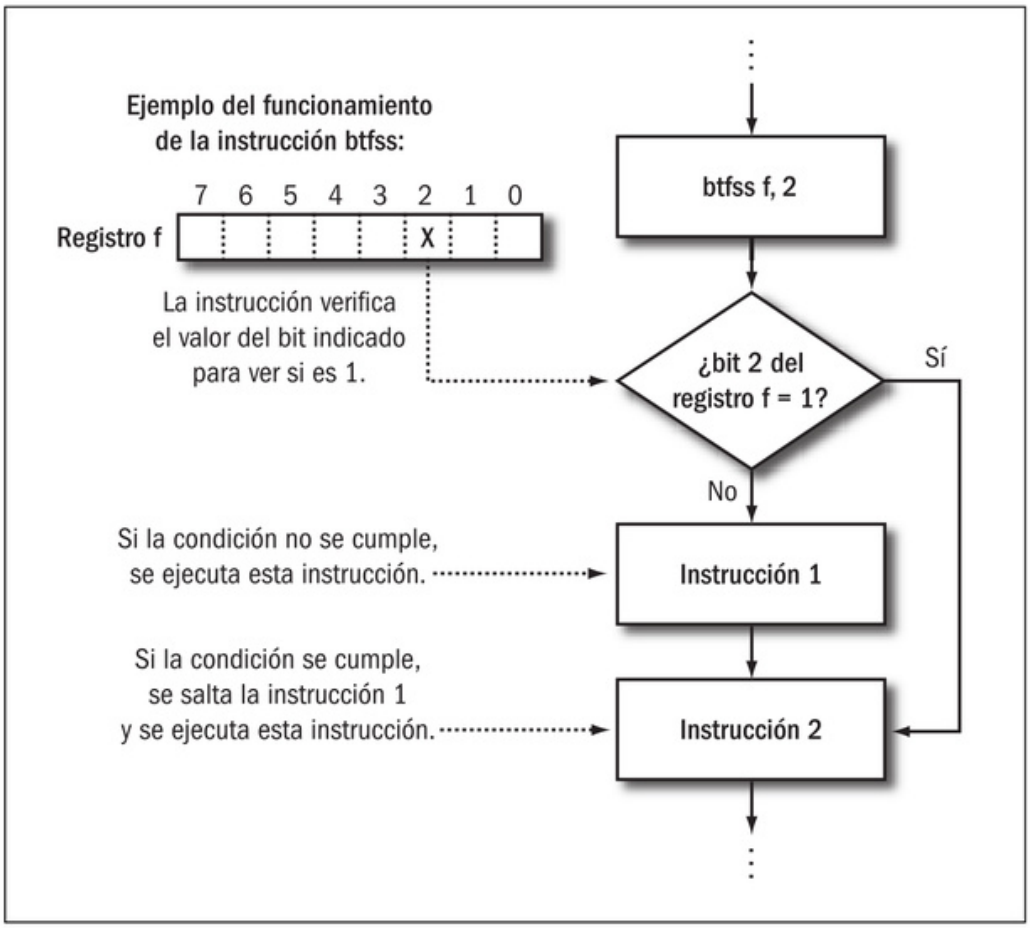


***Figura 16.** La instrucción btfss es un salto condicional con base en un solo bit. Aquí vemos su funcionamiento en detalle.*

De esta forma, podemos crear saltos condicionales en base al valor de un solo bit de cualquier registro de la memoria de datos, incluyendo los registros de la zona SFR y GPR. El valor del bit verificado no se altera en la ejecución de la instrucción. La instrucción **btfsc** funciona de manera idéntica, excepto que en ésta la condición a cumplir es que el bit verificado sea 0 en lugar de 1.

Comprobar si un registro contiene cero

Los saltos condicionales en base a bits pueden servirnos para hacer pruebas y comparaciones con registros. Veamos una técnica para determinar si un registro contiene un valor 0 o no. Esto se logra mediante la prueba del bit Z en el registro STATUS, que ya estudiamos antes.

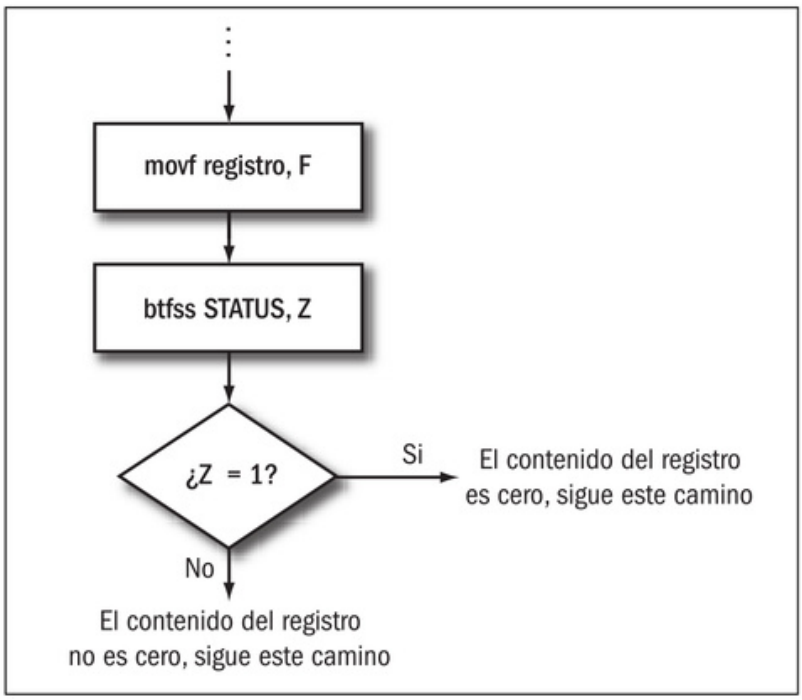


Figura 17. *Se utiliza la instrucción btfss con el bit Z del registro STATUS para comprobar si un registro contiene el valor 0.*

En la **Figura 17** tenemos el funcionamiento de la comprobación de 0. La instrucción **movf** lo que hace es mover el contenido del registro hacia sí, lo cual ocasiona que la bandera de 0 (Z) en el registro STATUS se active (se ponga a 1) si el contenido del registro es 0. De esta forma, al observar el valor que contiene el bit Z se sabe si el contenido del registro es 0 ó no. El fragmento de programa sería algo como:

```
            movf   registro, F      ;Mueve el registro a el mismo
            btfss  STATUS, Z        ;Comprueba el bit Z
            goto   noCero           ;Si Z es cero salta
esCero      .                       ;Si Z es uno ejecuta desde aquí
            .
            .

noCero      .                       ;Si Z es cero ejecuta desde aquí
            .
            .
```

Comprobar si un registro es igual a otro

Para comprobar que un registro sea igual (contiene el mismo número) que otro, simplemente haremos una resta de ambos. Si el contenido es el mismo en los dos, el resultado será 0. Si no, será un número diferente de 0. De todas maneras, si observamos el valor que toma el bit Z del registro STATUS sabremos si son iguales o no.

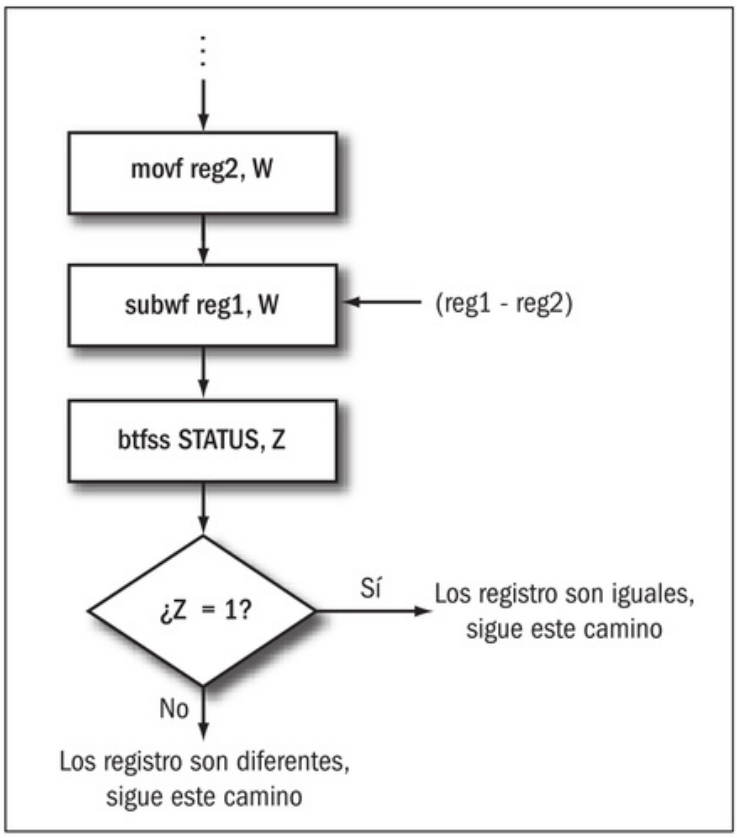


***Figura 18.** Una simple resta entre dos registros nos permite comprobar si su contenido es igual o diferente.*

En este caso el código sería:

```
            movf  reg2, W       ;Mueve el reg2 a el mismo
            subwf reg1, W       ;resta reg1- reg2
            btfss STATUS, Z     ;Comprueba el bit Z
            goto  noIgulaes     ;Si Z es cero salta
iguales     .                   ;Si Z es uno ejecuta desde aquí
            .
            .

noIguales   .                   ;Si Z es cero ejecuta desde aquí
            .
            .
```

Comprobar si un registro es mayor o menor que otro

Podemos usar los saltos condicionales en base a bits para verificar si el contenido de un registro es mayor o menor que otro. En este caso haremos la resta de ambos registros: si el resultado es positivo significa que el primer registro es mayor o igual que el otro y si es negativo, el primer registro es menor. En este caso comprobaremos el valor del bit de acarreo C. Recordemos que en una resta, si el valor obtenido es positivo, el bit de acarreo se activa (se pone a 1) y en caso contrario se desactiva (se pone a 0).

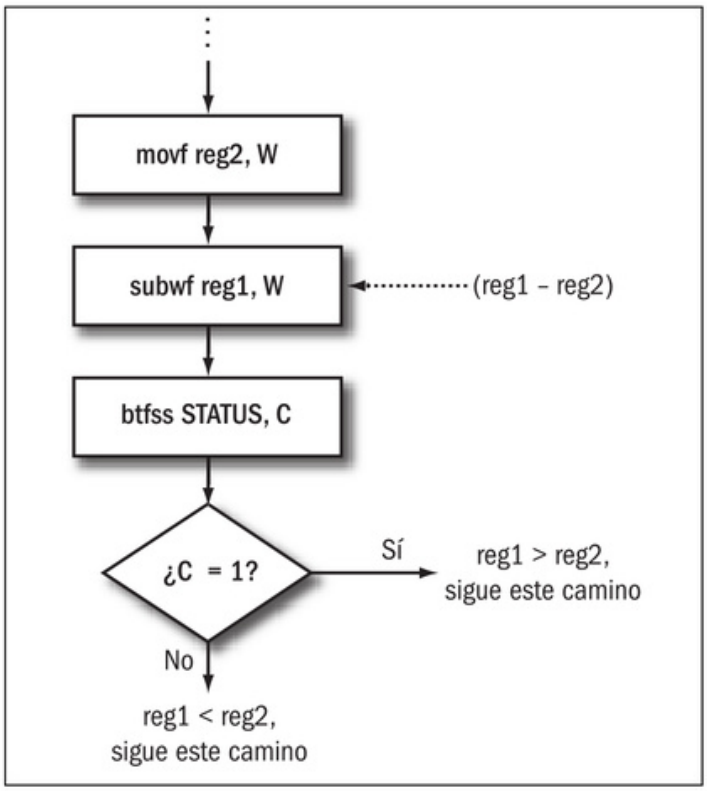


***Figura 19.** El bit de acarreo nos da información acerca de la comparación de dos registros.*

Debemos tener en cuenta que si el resultado es positivo (incluyendo el 0) puede significar que la resta pudo dar 0. En este caso, los contenidos de los registros son iguales, por lo que si necesitamos saber si son iguales, debemos llevar a cabo la comprobación de 0. El código de comparación de dos registros sería como este:

FUNCIONA TAMBIÉN CON XOR

Para la comprobación de registros iguales, se puede usar la operación XOR (**xorwf**) en lugar de la resta (**subwf**) y obtener el mismo resultado. Si hacemos una operación XOR entre los bits de ambos registros, el resultado será todos los bits a 0, si los registros son iguales, y cualquier otro diferente de 0 si no lo son.

```
            movf        reg2, W       ;Mueve reg2 a W
            subwf       reg1, W       ;Resta (reg1-reg2)
            btfss       STATUS, C     ;Comprueba el bit C
            goto        menor         ;Si C es cero salta
mayor.                                ;reg1>reg2 ejecuta desde aquí
            .
            .

menor       .                         ;reg1<reg2 ejecuta desde aquí
            .
            .
```

BUCLES O LAZOS

Dentro de nuestros programas hay ocasiones que necesitaremos que una parte de las instrucciones se repitan o ejecuten varias veces consecutivas. A este tipo de proceso se lo denomina **lazo** o **bucle** (en inglés se puede llamar *loop*) ya que es la repetición de una sección de código mediante un salto hacia atrás. De esta forma se vuelven a ejecutar las líneas de código anteriores. A continuación veremos cuáles son los distintos tipos de bucles que podemos utilizar en nuestros programas.

Bucles infinitos

Un **bucle infinito** es aquél en el que se entra en determinado punto del programa pero ya no puede salir de él. Este bucle se ejecutará siempre, repitiendo una y otra vez las instrucciones que se encuentren en él. El bucle será infinito siempre y cuando no haya dentro suyo alguna instrucción que lo haga salir, por ejemplo, un salto hacia otro lugar fuera del bucle.

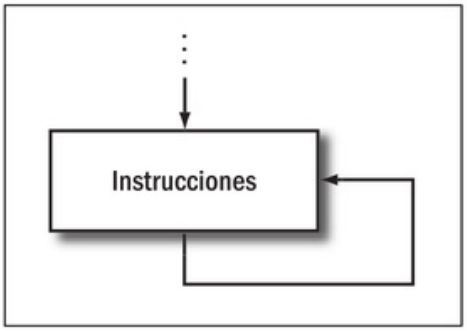


Figura 20. *Un salto incondicional hacia atrás con la instrucción goto puede generar un bucle infinito.*

Un claro ejemplo de un bucle infinito es el que utilizamos en Mi primer programa. Si analizamos de nuevo su funcionamiento, podemos darnos cuenta de que hay un bucle infinito al final. En él, el programa queda encerrado en el proceso de leer los datos que entran por el Puerto A y los envía al Puerto B. Como está repitiéndose constantemente, entonces siempre monitoreará los cambios en la entrada y los reflejará a la salida.

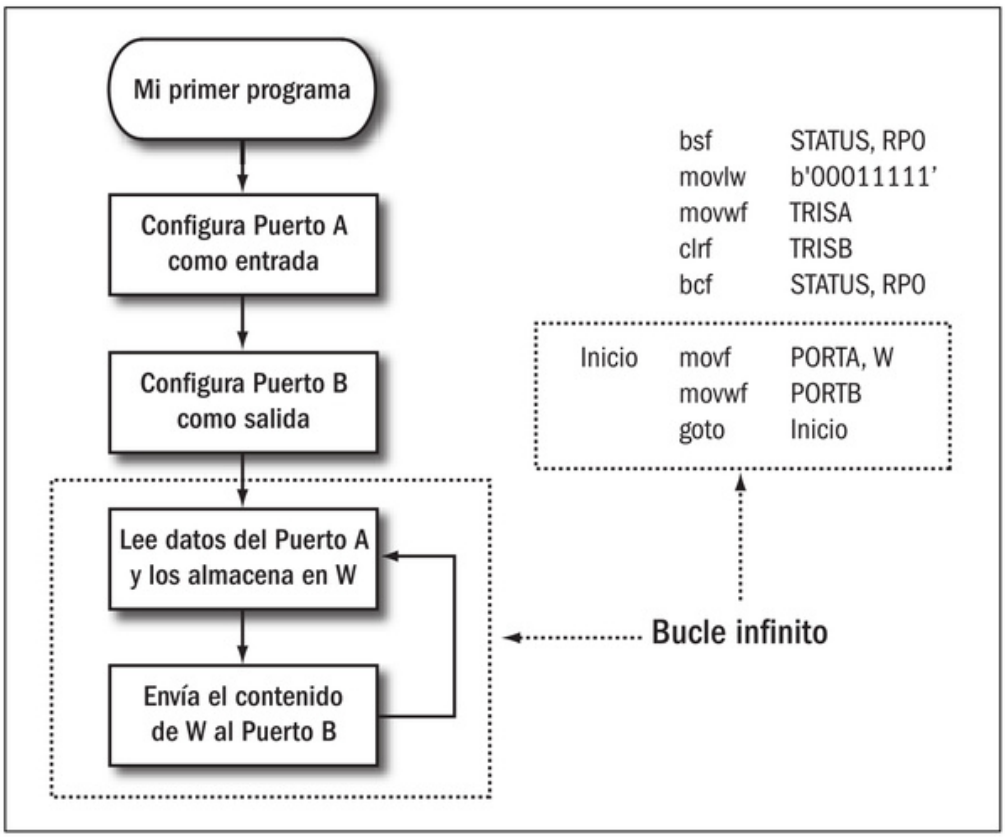


***Figura 21.** Un bucle infinito permite ejecutar constantemente el mismo grupo de instrucciones en nuestros programas.*

Bucle condicional

Podemos utilizar las instrucciones de salto condicional para generar **bucles condicionales**, es decir, que se repiten hasta que una condición se cumple y entonces el flujo del programa sale del bucle. Veamos un ejemplo.

BUCLES INFINITOS NO DESEADOS

Los bucles infinitos no siempre son deseados en la programación (en cualquier tipo de lenguaje o aplicación). Es común cometer algún error durante la escritura del programa y generar, sin ser intencional ni darse cuenta, un bucle infinito en él, en el cual el programa queda atrapado, y por ende no funciona como debería.

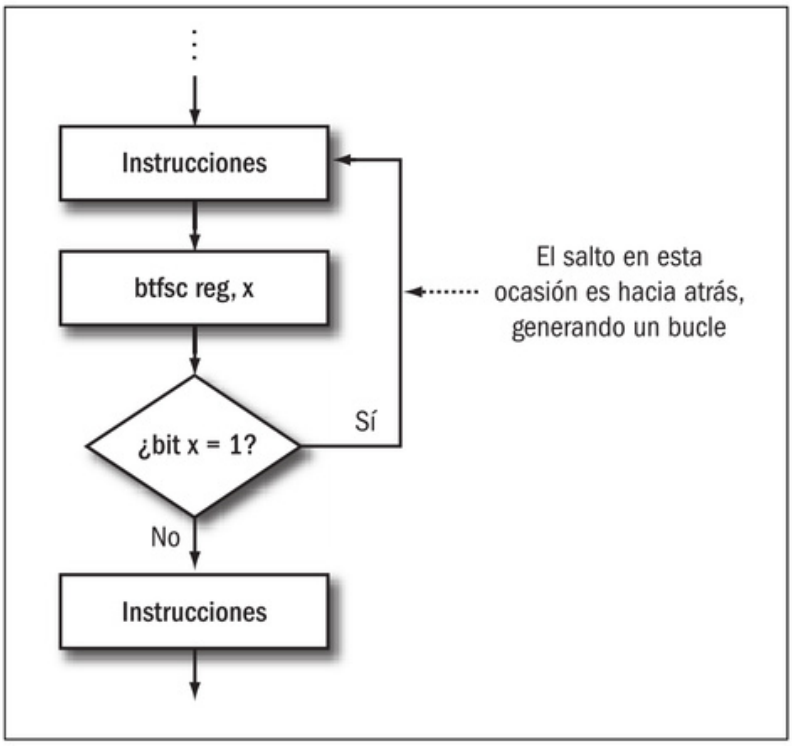


Figura 22. *En un bucle condicional no se sale de éste hasta que la condición se cumple.*

Un ejemplo de un bucle condicional es el monitoreo de un pulsador conectado a una línea de entrada de nuestro microcontrolador. Veamos un ejemplo de esto: supongamos que tenemos un pulsador o botón conectado a la línea RA4 del PIC16F84A y, a su vez, a tierra (**Figura 23**). El pin tiene un resistor de *pull-up* que lo obliga a tener un estado alto en él. Obviamente el pin RA4 debe ser configurado como entrada.

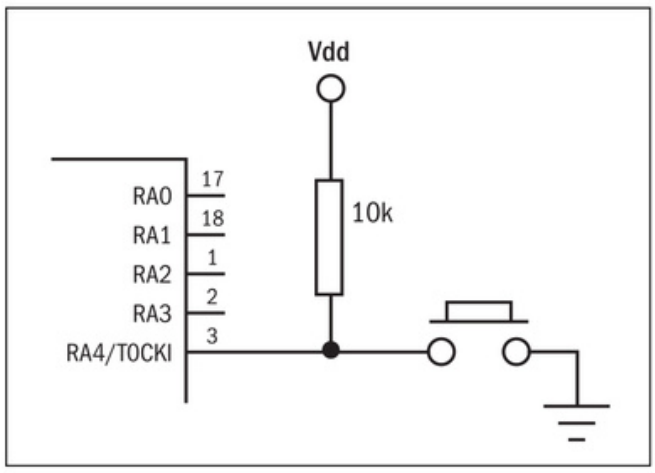


Figura 23. *Un bucle condicional puede servir como monitoreo de un pulsador en un pin de entrada.*

Mientras el pulsador no se presione, habrá un 1 en RA4, por lo que la condición no se cumplirá y seguirá en el bucle. Al accionar el pulsador, habrá un 0, lo cual cumple con la condición y saldrá del bucle hacia el proceso necesario. El código sería:

```
      .
      .
pulsador
      .
      .
      btfss          PORTA, 4      ;Verifica si el pulsador es accionado
      goto           proceso       ;Si fue accionado salta a proceso
      goto           pulsador      ;Si no es accionado genera un bucle
      .
      .
      .
proceso
```

Bucles fijos

Los **bucles fijos** son los que sólo se llevan a cabo un número fijo de veces y luego salen para continuar el flujo del programa. Para estos bucles se utiliza la instrucción **decfsz** o **incfsz**. De esta forma, se carga un número en algún registro que será quien se decremente o incremente, hasta llegar a 0, y cuando esto suceda se saldrá del bucle. Los bucles fijos son típicamente utilizados como retardos, para detener por algún tiempo el flujo del programa, y es precisamente el tema que estudiaremos a continuación.

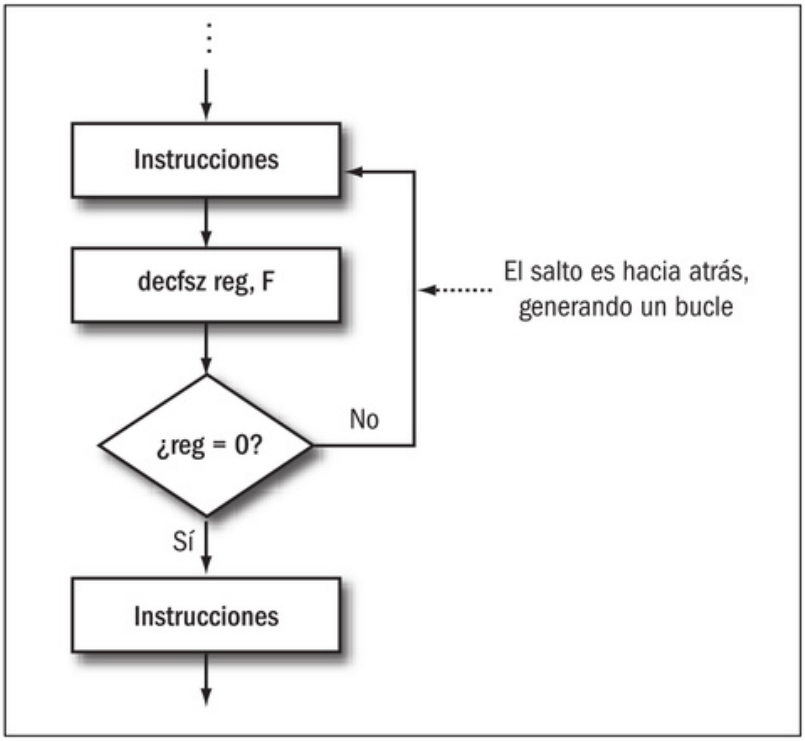


Figura 24. *Un bucle fijo se lleva a cabo un número conocido de veces.*

RETARDOS

Generalmente, la frecuencia de reloj usada en nuestro microcontrolador será muy alta, por lo que las instrucciones de nuestro programa se llevan a cabo de forma muy rápida. En ocasiones podemos requerir el retraso de algunos procesos para poder ver los efectos que causan o porque necesitamos que así sea. Esto se logra mediante **retardos**, que no es otra cosa que mantener ocupado al microcontrolador durante algún tiempo antes de que continúe con los procesos siguientes.

Como ya estudiamos antes, las instrucciones se ejecutan en ciclos de máquina, y cada uno de ellos es igual a cuatro ciclos de reloj. Si sabemos cuál es la frecuencia del oscilador y cuántos ciclos de máquina requiere cada una de las instrucciones, podemos saber cuál es el tiempo que le tomará a un programa o parte de él ejecutarse.

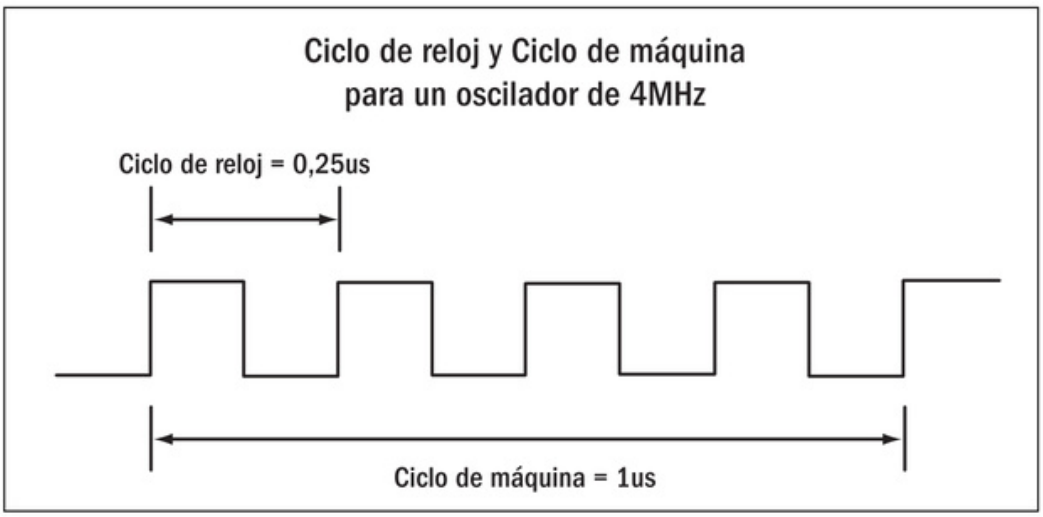


***Figura 25.** Si conocemos la frecuencia del reloj y el ciclo de máquina, podemos calcular los tiempos de ejecución.*

Para un oscilador de 4 MHz tenemos:

$$T = \frac{1}{f} = \frac{1}{4\text{MHz}} = 0{,}25\mu\text{s}$$

Donde:
T = Período de la señal de reloj
f = Frecuencia de la señal de reloj

Observamos cómo el período o un ciclo de la señal de reloj dura 0.25 microsegundos. Y si sabemos que el ciclo de máquina son 4 ciclos de la señal de reloj, entonces:

$$M = 4T = 4\ (0{,}25\mu s) = 1\mu$$

Donde:
M = Ciclo de máquina

Y así sabemos, entonces, que si utilizamos un oscilador de 4MHz, las instrucciones que se lleven a cabo en un ciclo de máquina se ejecutarán en un tiempo de 1 microsegundo cada una, mientras que las que se lleven a cabo en dos ciclos de máquina se ejecutarán en 2 microsegundos.

Podemos darnos cuenta de que la ejecución de cada instrucción es muy rápida y en gran número de ocasiones necesitaremos que el microcontrolador "espere" un poco antes de continuar con los procesos siguientes. Imaginemos esta situación: necesitamos diseñar un programa que nos muestre en el puerto de salida una cuenta de 1 a 10, por dar un ejemplo. Es fácil: sólo colocamos un valor de 1 en un registro, lo enviamos al puerto de salida para visualizarlo, luego incrementamos el registro y lo enviamos de nuevo al puerto de salida. Pero debemos tener en cuenta que el tiempo de ejecución de las instrucciones es muy corto, e incluso el tiempo de ejecución del programa completo también lo es, por lo que los cambios en el puerto de salida serían demasiado rápidos como para poder verlos. Es entonces cuando necesitamos "retardar" el envío de los datos de tal forma que sean lo suficientemente lentos como para apreciarlos.

Cuando estudiamos el repertorio de instrucciones vimos que existe la instrucción **nop** (*no operation*), que no hace nada, sólo gasta un ciclo de máquina, no hay ninguna operación y no afecta a ningún registro ni bandera. Por lo que si colocamos una o más instrucciones **nop** en nuestros programas, lo que logramos es retrasar la ejecución de las siguientes instrucciones a un ciclo de máquina (1 microsegundo para un oscilador de 4 MHZ) por cada instrucción **nop**. Sin embargo, aunque la instrucción **nop** nos proporciona retardos, éstos son muy cortos, por lo que necesitaríamos usar muchas instrucciones **nop**, lo cual no es práctico. Para retardos mayores se utilizan bucles o lazos.

III EL ALCANCE DE CALL

La instrucción call coloca la dirección del salto directamente en los once bits más bajos del contador de programa (PC), con lo cual esta instrucción puede acceder a cualquier dirección de memoria en un rango de 2048 bytes (2 KB). Aunque, como sabemos, el PIC16F84A tiene sólo 1024 bytes (1 KB) de memoria de programa.

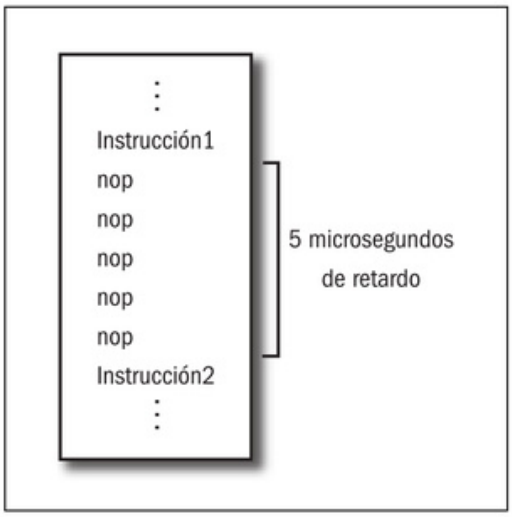


Figura 26. *La instrucción nop nos proporciona períodos de retardo de unos cuantos microsegundos.*

Retardo con bucle simple

Ya estudiamos cómo se puede generar un bucle o lazo fijo. Éstos nos sirven típicamente como retardos. Como un retardo suele utilizarse más de una vez en el programa, entonces se escribe como una subrutina para que el programa principal pueda llamarlo cada vez que lo requiera.

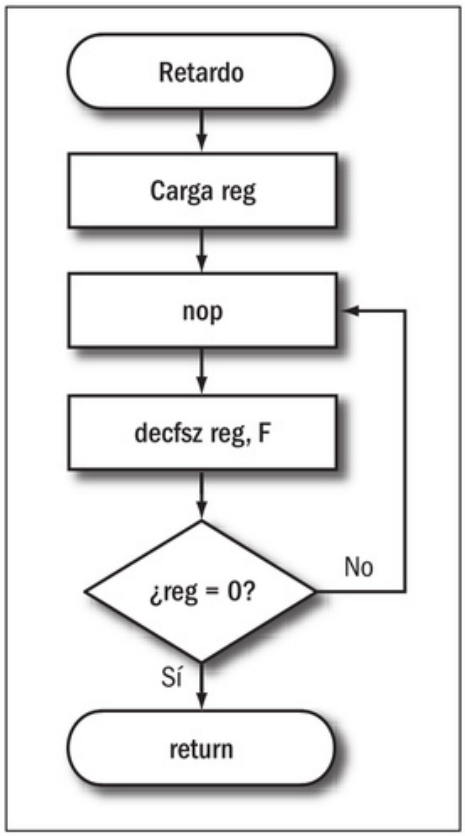


Figura 27. *Un bucle fijo es muy útil para crear una subrutina de retardo.*

En la **Figura 27** tenemos el diagrama de flujo de una subrutina de retardo con bucle simple. La subrutina puede ser escrita de esta forma:

```
     call          retardo       ;El llamado a subrutina aporta 2 ciclos
de máquina
     .
     .
     .

retardo                          ;Inicia subrutina de retardo

     movlw         d'K'          ;Aporta 1 ciclo
     movwf         contador      ;Aporta 1 ciclo

bucle
     nop                         ;Aporta K ciclos
     nop                         ;K
     nop                         ;K
     decfsz        contador, F   ;(K-1)+2 = K+1
     goto          bucle         ;2K-2
     return                      ;2

;Tiempo total de retardo = 6K + 5 (Ciclos de máquina)
```

Podemos observar cómo funciona el retardo y cómo calcular el tiempo que durará según el valor de **K**, que es cargado en el registro contador. El tiempo que dura este retardo se calcula con la ecuación:

$$M = 6K + 5$$

III AJUSTE DE TIEMPOS

Cuando escribimos rutinas de retardo, generalmente necesitamos un tiempo determinado. Aunque los tiempos aproximados pueden ser útiles en la mayoría de los casos, si necesitamos un tiempo muy exacto podemos calcular un retardo aproximado y ajustar luego con instrucciones **nop** para acercarnos al valor deseado.

Donde:
M = Ciclos de máquina
K = valor que cargaremos en el registro

Ejemplo: determinar cuántos ciclos de máquina tardará el retardo si colocamos un valor de 100 decimal en K:

M = 6 (100) + 5 = 605 ciclos

Si estamos utilizando un oscilador de 4 MHz, entonces sabemos que el ciclo de máquina es de 1 microsegundo, por lo que el retardo en este caso tardará 605 microsegundos.

El tiempo máximo para este retardo está dado cuando colocamos un valor de 0 en K, puesto que al realizarse el primer decremento, el valor del registro pasará a 255, y de esta forma el decremento del registro se llevará a cabo 256 veces:

M = 6 (256) + 5 = 1541 ciclos

Nos dará un retardo máximo de 1541 microsegundos ó 1.541 milisegundos con este retardo para una frecuencia de reloj de 4 MHZ. Si necesitamos tiempos de retardo aun más largos debemos recurrir a otra técnica.

Retardos con bucles anidados

Podemos lograr tiempos de retardo muy largos si anidamos bucles, es decir, si ponemos un bucle dentro de otro. De esta forma, un bucle simple se ejecutará muchas veces y logrará tiempos mucho más largos que con un solo bucle simple.

III LOS SALTOS Y EL PC

Las instrucciones de salto modifican el valor que contiene el contador de programa (PC), por lo que después de cualquier instrucción de salto, el programa continuará a partir de la nueva dirección que contiene el contador de programa. Todas las instrucciones que modifican el contador de programa directamente requieren de 2 ciclos de máquina para ejecutarse.

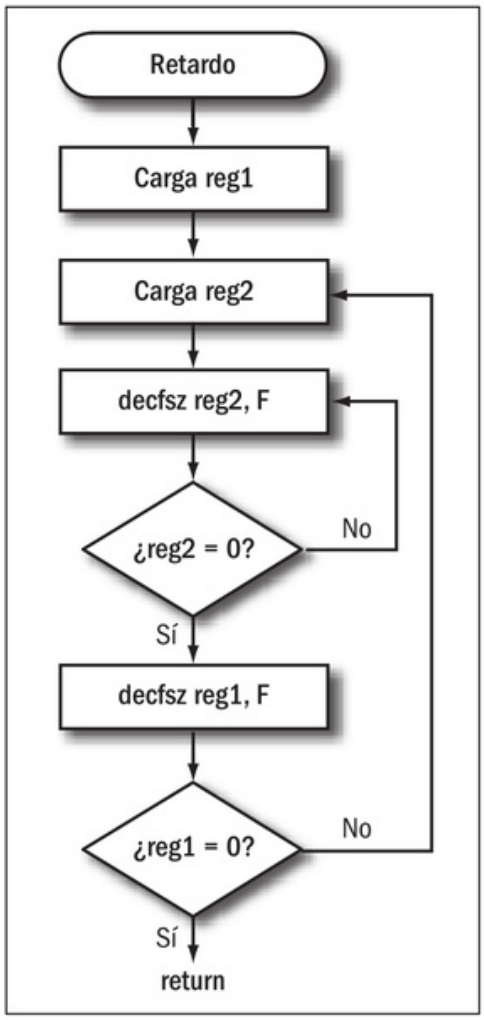


Figura 28. *En ocasiones necesitaremos retardos muy largos, y esto lo lograremos al anidar bucles o lazos.*

Como podemos apreciar en la **Figura 28**, tenemos un bucle simple dentro de otro. Así, el bucle simple se ejecuta muchas veces con lo cual multiplica el tiempo en que se ejecuta toda la subrutina de retardo. El código de un retardo con bucles anidados puede ser el siguiente:

```
     call          retardo              ;2
     .
     .
     .

retardo
     movlw         d'X'                 ;1
     movwf         cont_1               ;1
bucle_2
     nop                                ;X
```

```
      movlw         d'Y'                ;X
      movwf         cont_2              ;X
bucle_1
      nop                               ;XY
      decfsz        cont_2, F           ;XY+X
      goto          bucle_1             ;2XY-2X
      decfsz        cont_1, F           ;X+1
      goto          bucle_2             ;2X-2
      return                            ;2

;Tiempo total = 4XY + 5X +5 (Ciclos de máquina)
```

Por supuesto, podemos seguir anidando bucles o lazos para lograr tiempos aun mayores. Esto dependerá de lo que necesitemos en cada programa en particular.

Medir tiempos en MPLAB SIM

Podemos usar el simulador MPLAB SIM para medir tiempos, ya sea de nuestros programas completos, de una parte de ellos, o de los retardos en las simulaciones. Existe una función especial llamada **Stopwatch**, que es la que nos permite medir los tiempos de ejecución. Para usarla debemos antes habilitar el simulador MPLAB SIM dentro de MPLAB. Como vimos, lo haremos desde el menú **Debugger/Select Tool**. Una vez activado MPLAB SIM en el mismo menú **Debugger** elegiremos la opción **Stopwatch**.

***Figura 29.** Al activar MPLAB SIM aparecerán nuevas opciones en el menú Debugger, una de ellas es Stopwatch.*

Una vez que hemos seleccionado la opción **Stopwatch**, aparecerá una ventana con el mismo título, que nos permitirá monitorear los tiempos de nuestros programas. En la **Figura 30** tenemos el aspecto de esta ventana y los elementos que contiene.

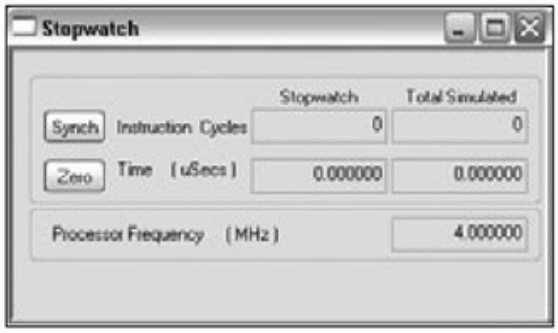


Figura 30. *La ventana Stopwatch sirve para medir tiempos y ciclos de máquina transcurridos en los programas.*

La medición de tiempos es automática en cualquiera de las opciones de simulación. Cuando corramos la simulación, ya sea en animación o con los controles paso a paso, veremos cómo en la ventana **Stopwatch** se irá reflejando el tiempo y ciclos de máquina (*instruction cycles*) ejecutados. Es importante notar que en la parte de abajo de la ventana aparece la leyenda **Processor Frecuency**, que nos indica la frecuencia del oscilador. Debemos configurar la frecuencia en el menú **Debugger/Settings...** en la sección **Processor Frecuency**. Es muy importante configurar la frecuencia correcta ya que los tiempos medidos dependen directamente de ella.

Observemos que hay un botón llamado **Zero**, que sirve para poner a cero los valores de la columna **Stopwatch** en cualquier momento que necesitemos. Esto es para medir desde el punto en donde lo requiramos. En la columna **Total Simulated** se cuentan el tiempo y los ciclos de máquina totales, sin importar cuántas veces hayamos presionado el botón **Zero**. Con el botón **Zynch** sincronizaremos los valores totales y los de la columna **Stopwatch**. Si aplicamos un **Reset** mediante el botón del simulador se borrarán todos los valores de la ventana **Stopwatch**, ya que el programa comenzará desde el principio como si hubiéramos aplicado un reset al microcontrolador.

Si necesitamos medir de forma rápida el tiempo de ejecución de una subrutina lo haremos así: primero simularemos el programa paso a paso hasta llegar a la instrucción **call** que llama a la subrutina deseada (también podemos colocar un punto de ruptura en la instrucción **call** y presionar el botón **Run**, con esto el programa se detendrá en el llamado a subrutina), presionamos el botón **Zero** y luego simplemente **Step Over** para ejecutar en un solo paso la subrutina, con lo cual el cursor de simulación (la flecha de color verde) pasará a la siguiente instrucción después del **call** y la ventana **Stopwatch** reflejará el tiempo que tomó la subrutina, incluyendo la instrucción de llamado **call**.

Programa para calcular retardos

Para calcular de forma sencilla nuestros retardos, podemos usar un pequeño programa llamado **PIC delayer**, que tenemos la posibilidad de descargar del sitio web de la editorial en **www.redusers.com**. El programa no necesita instalación, sino que simplemente al abrir el archivo **PICdel.exe** lo tendremos en funcionamiento.

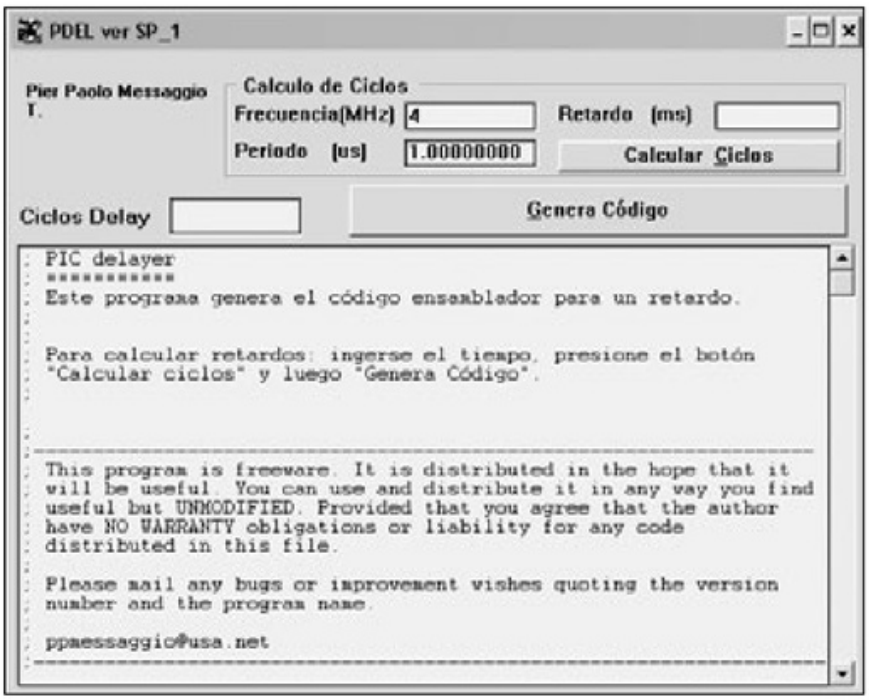


Figura 31. *El programa PIC delayer calcula el código de subrutinas de retardo en forma automática.*

Sólo tenemos que asegurarnos de tener la frecuencia adecuada del oscilador en la casilla **Frecuencia(MHz)**. Luego pondremos el tiempo que necesitamos para nuestro retardo en la casilla **Retardo (ms)**, que debe estar siempre en milisegundos. Por ejemplo, si necesitamos un retardo de medio segundo pondremos 500 ms. Presionamos el botón **Calcular Ciclos** y en la casilla **Ciclos Delay** aparecerá el número de ciclos de máquina necesarios para el retardo. Ahora, al presionar el botón **Genera Código**, el programa calculará el retardo y nos dará su código, que podemos copiar y pegar en el editor de MPLAB.

III OTRO CALCULADOR DE RETARDOS

El programa que estamos estudiando puede resultarnos sumamente útil para calcular nuestros retardos, pero no es el único. Por ejemplo, si vamos al sitio **www.golovchenko.org/cgi-bin/delay** encontraremos otro calculador de retardos gratuito que podemos usar en línea. El funcionamiento es muy parecido al PIC delayer.

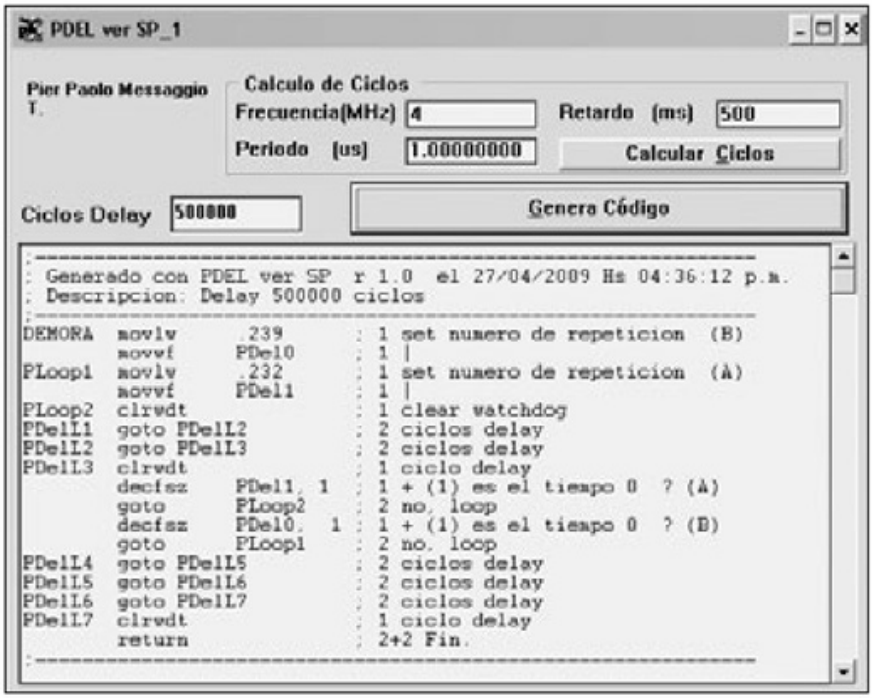


Figura 32. *Ejemplo del código calculado para un retardo de 500ms.*

Este programa nos indicará las constantes que debemos declarar en nuestro código fuente para que el retardo funcione correctamente. En los retardos calculados puede aparecer la instrucción **clrwdt**, pero si no estamos utilizando el WDT podemos cambiarlas por instrucciones **nop**, aunque si las dejamos con **clrwdt** no habrá problema.

Rebotes en pulsadores

En la mayoría de nuestros proyectos necesitaremos el uso de **interruptores** o **pulsadores** para introducir datos por los puertos, o para controlar o generar algún evento o proceso. Pero los pulsadores e interruptores son elementos mecánicos y, como elementos mecánicos, tienen un comportamiento un poco extraño: cuando son presionados o activados, se lleva a cabo el contacto eléctrico entre dos metales para cerrar un circuito, lo cual genera el efecto de **rebote** (**Figura 33**). Al accionar un pulsador, sus contactos literalmente rebotan al chocar y esto genera que la señal obtenida sea inestable por un momento.

III AJUSTAR RETARDO

Como mencionamos, los rebotes de los pulsadores dependen de varios factores, por lo cual puede que el tiempo necesario para eliminarlos varíe en nuestras aplicaciones. Si la rutina anti-rebotes no funciona bien con 30 ms, podemos intentar aumentar un poco el tiempo de retardo para lograr un correcto funcionamiento.

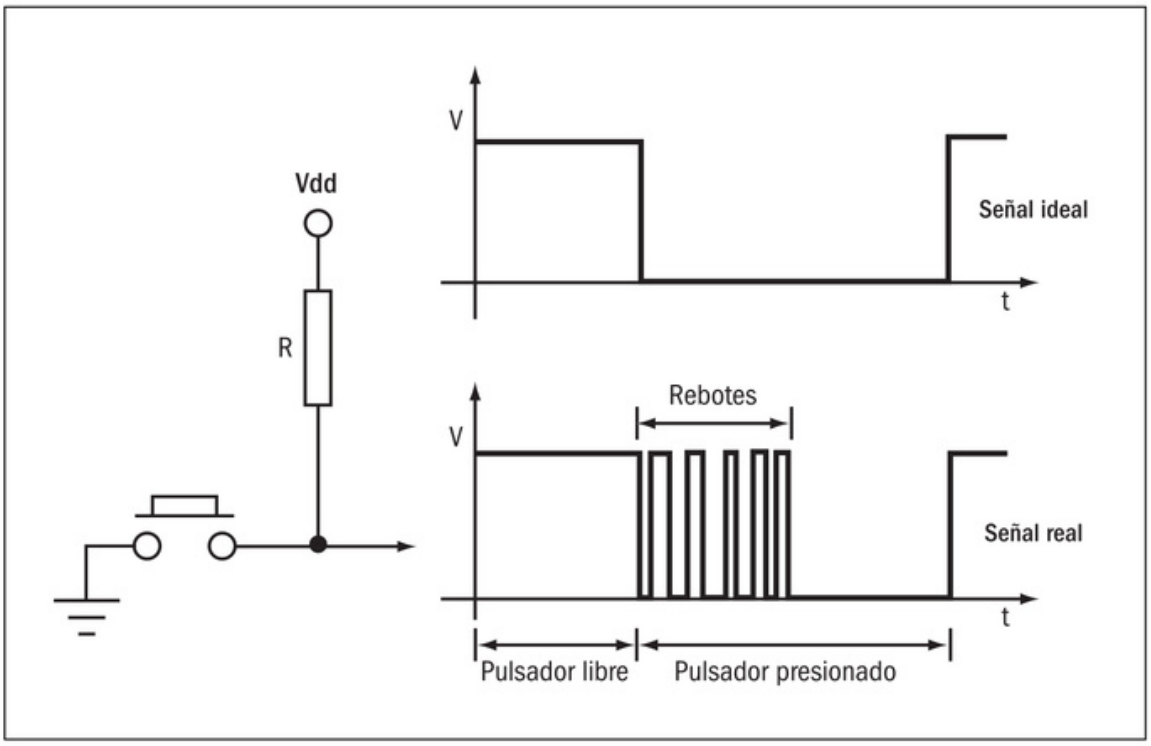


Figura 33. *El efecto de rebote en los pulsadores (en inglés,* ***bounce****) genera efectos no deseados.*

Con este choque mecánico, la señal que se obtiene rebota entre los dos estados y produce efectos no deseados en nuestro circuito, ya que es como si se presionara el pulsador muchas veces. Para resolver este problema existen métodos llamados **por hardware**, que son circuitos anti-rebotes que evitan este efecto.

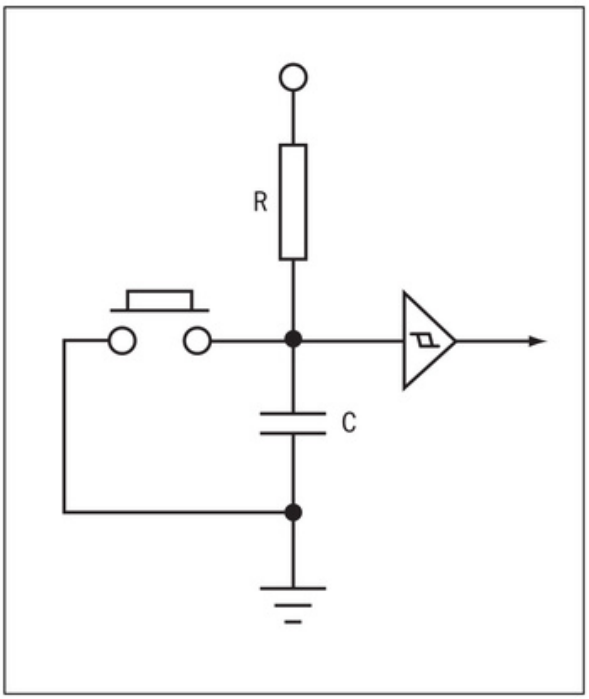


Figura 34. *Un ejemplo de un circuito anti-rebotes (****debouncing****), aunque no es necesario usarlo con los microcontroladores.*

Pero en nuestro caso no es necesario usar un circuito anti-rebotes, ya que podemos eliminar el efecto no deseado de los pulsadores mediante el propio programa (**por software**) y de esta forma ahorrarnos el costo y la complejidad de armar estos circuitos. La forma de eliminar los rebotes por software es muy sencilla, sólo se agrega un retardo cuando se detecta que el pulsador se ha presionado. De esta manera, se espera que los rebotes pasen, y así los hemos eliminado y no tendremos los efectos indeseados.

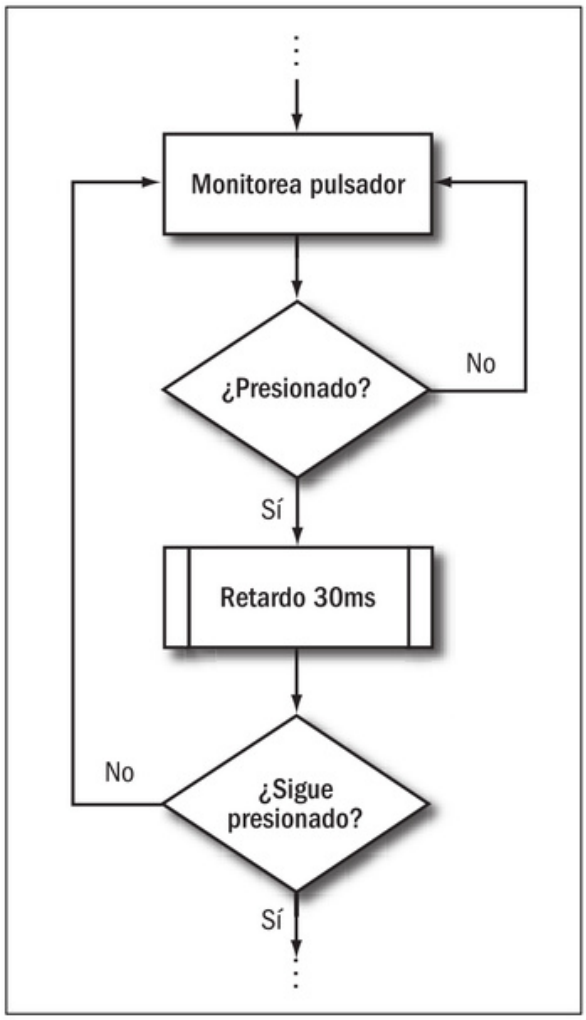


***Figura 35.** La rutina anti-rebotes asegura el buen funcionamiento de los pulsadores.*

{} DEBOUNCE POR HARDWARE

Los pulsadores o teclados de prácticamente todos los aparatos o circuitos electrónicos deben llevar alguna técnica anti-rebotes (debounce). Algunos usan la de **hardware**, es decir, un circuito que permite eliminar o minimizar el efecto de rebote en la señal. La técnica por hardware se usa sólo cuando no es posible hacerlo por software.

Debemos tener en cuenta que el tiempo de los rebotes del pulsador depende de varios factores, entre los cuales están el tipo de pulsador utilizado y la fuerza que se ejerce al presionarlo. Pero, en general, el tiempo en que ocurren los rebotes está entre 20 y 30 ms aproximadamente.

Un contador binario

Veamos un sencillo ejemplo de la aplicación de los retardos, anti-rebotes y subrutinas. Para esto, armaremos un contador binario controlado por un pulsador conectado al pin RA0 y 8 leds conectados como salida en el Puerto B.

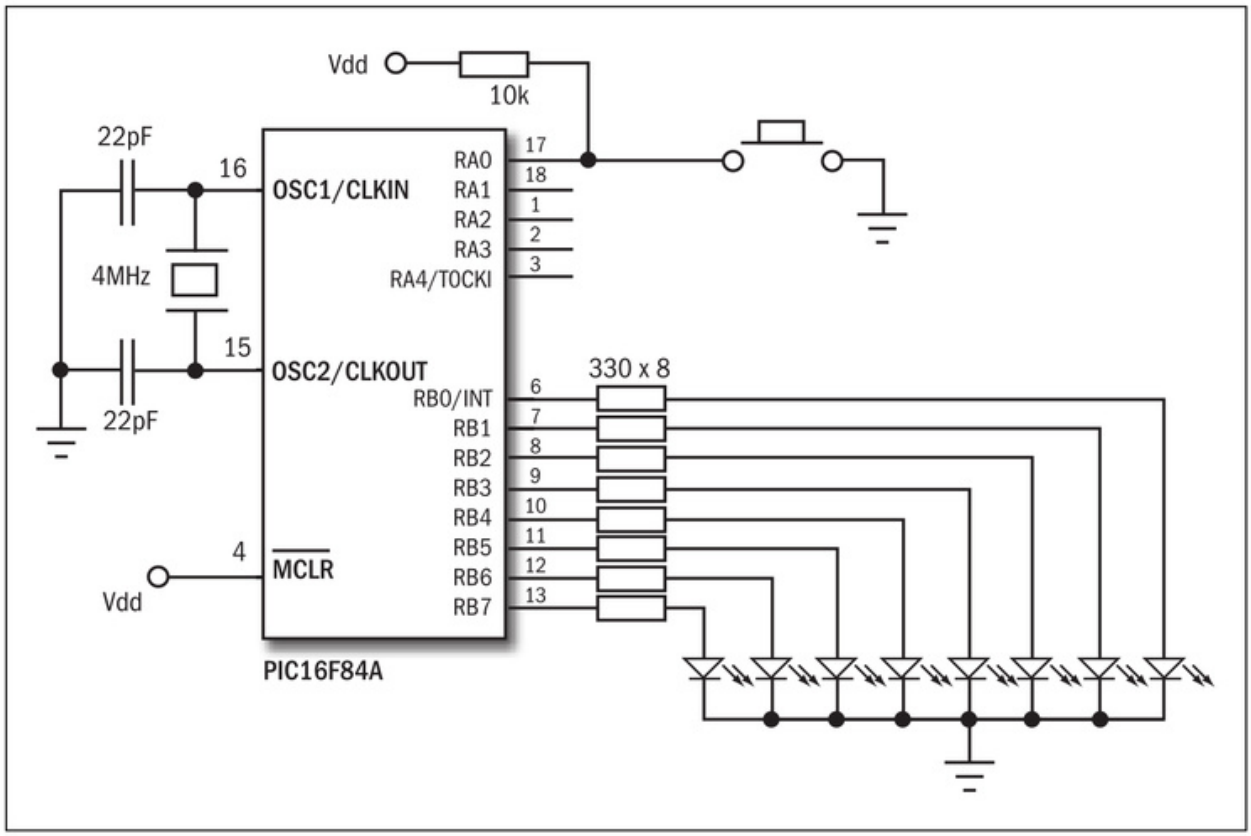


***Figura 36.** Un circuito contador de ejemplo para lo estudiado hasta ahora.*

Así, cada vez que presionemos el pulsador, la cuenta se incrementará en los leds conectados como salida. El diagrama de flujo sería como el de la **Figura 37**.

III TABLAS EN CUALQUIER LUGAR

Como sabemos, puede haber problemas si colocamos las tablas más allá de los primeros 256 bytes de la memoria de programa. Sin embargo, es posible colocarlas en cualquier lugar mediante el uso de una técnica especial. Si necesitamos implementarla podemos descargar del sitio web **www.microchip.com** la nota de aplicación **AN556 Implementing a Table Read**.

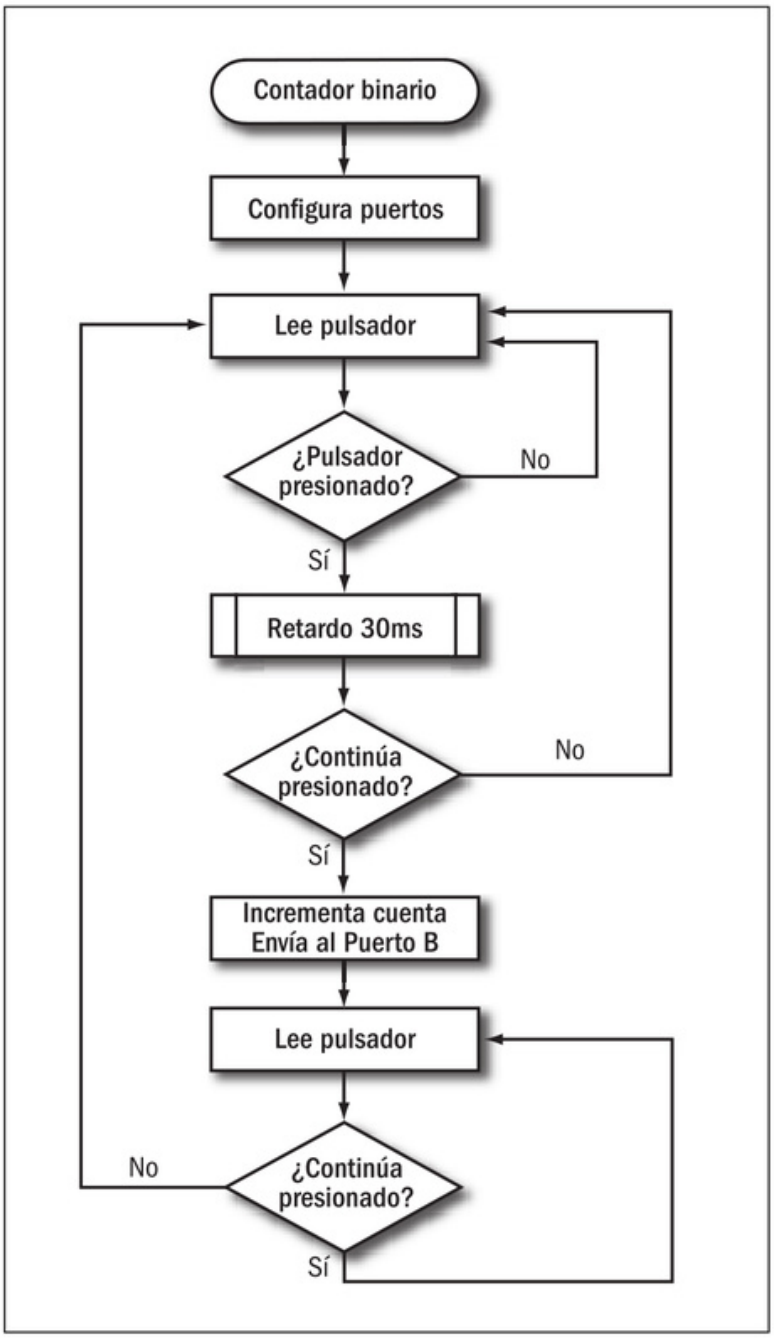


Figura 37. *Detalle del funcionamiento para nuestro contador con anti-rebotes.*

El código fuente es el siguiente:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

CBLOCK 0x0C
cuenta
```

```
     cont1
     cont2
     ENDC

     ORG            0x00                ;Establece el origen del programa

;Configuración de puertos:

     bsf            STATUS, RP0         ;Acceso al banco 1
     movlw          b'00000001'
     movwf          TRISA               ;Configura RA0 como entrada
     clrf           TRISB               ;Configura el Puerto B como salida
     bcf            STATUS, RP0         ;Acceso al banco 0

;Programa principal:

     clrf           cuenta              ;Borra el registro para la cuenta
     clrf           PORTB               ;Borra el Puerto B al inicio
inicio
     btfsc          PORTA, 0            ;¿Se ha presionado el pulsador?
     goto           inicio              ;No, genera un bucle
     call           retardo             ;Si, llama a subrutina "retardo"
para eliminar rebotes
     btfsc          PORTA, 0            ;¿Sigue pulsado?
     goto           inicio              ;No, falso disparo ve a inicio
     incf           cuenta, F           ;Si, incrementa la cuenta
     movf           cuenta, W
     movwf          PORTB               ;Envíala al Puerto B
pulsado
     btfss          PORTA, 0            ;¿Sigue pulsado?
     goto           pulsado             ;Si, genera un bucle hasta que el
pulsador sea liberado
     goto           inicio              ;Inicia de nuevo

retardo                                 ;Retardo anti-rebotes 30ms
aproximadamente
     movlw   d'100'
     movwf   cont1
bucle2
     movlw   d'100'
```

```
      movwf   cont2
bucle1
      decfsz  cont2, F
      goto            bucle1
      decfsz  cont1, F
      goto            bucle2
      return

      END
```

En el archivo **Contador binario.asm**, que podemos descargar de **www.redusers.com**, tenemos el código fuente para poder estudiarlo. En el ejemplo se usó una subrutina de retardo de unos 30 ms para eliminar los rebotes del pulsador. Tenemos también el archivo **Contador binario.hex** para grabarlo en el PIC y comprobar su funcionamiento.

TABLAS

El manejo de tablas grabadas en la memoria de programa (comúnmente llamado **tablas en ROM**) puede ser muy útil en ciertos casos en donde se debe disponer de una tabla para convertir códigos, para resolver tablas de verdad y otras tareas.

Salto indexado

Los **saltos indexados** se llevan a cabo con la instrucción **addwf PCL, F** para realizar un salto específico que depende del valor de W. Observamos cómo la instrucción de suma **addwf** lo que hace es sumar el valor que contenga W al registro PCL que, como ya vimos, es el registro correspondiente a los 8 bits más bajos del contador de programa (recordemos que éste es de 13 bits). Y de esta forma se obtiene una dirección nueva a donde se saltará. El valor contenido en W que controlará el salto se llama **offset**.

III EL SALTO INDEXADO Y EL CONTADOR DE PROGRAMA

Recordemos que debido a la arquitectura segmentada del procesador de los microcontroladores PIC, el contador de programa se incrementa mientras se está ejecutando la instrucción actual. Por lo tanto, el PC contiene realmente la dirección de la siguiente instrucción al **addwf PCL, F**, por lo que la suma se realiza con el valor de los bits de esta dirección.

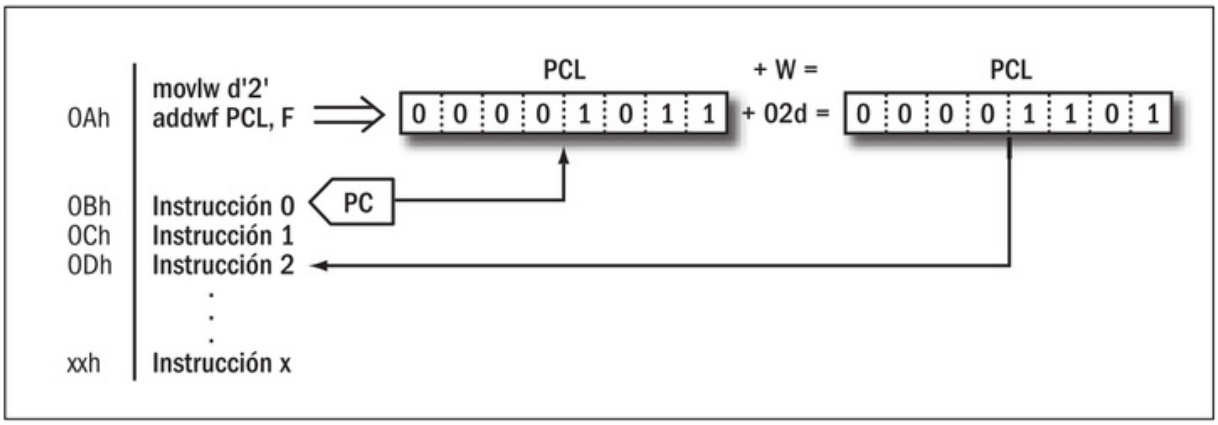


***Figura 38.** El salto indexado accede a una instrucción de una lista, determinada por el contenido de W.*

Al alterar el contenido del registro PCL al sumarle el valor de W (offset) se realiza un salto hacia la instrucción correspondiente de la lista, que generalmente es un salto (**goto**) hacia un proceso particular. Por ejemplo:

```
movf        PORTA, W          ;Lee el valor del Puerto A
andlw       b'00000011'       ;Se queda con los 2 bits más bajos
addwf       PCL, F            ;Salta al proceso adecuado
                              ; según el valor leído
goto        proceso0
goto        proceso1
goto        proceso2
goto        proceso3
```

Manejo de tablas

Ahora, si combinamos el salto indexado con la instrucción **retlw**, que es un retorno de subrutina similar a **return** (pero con la diferencia de que **retlw** coloca un valor en el registro W antes de regresar) tendremos una **tabla de datos**. Una tabla nos permite obtener datos de una lista dependiendo del valor del offset que coloquemos en W.

CIRCUITOS COMBINACIONALES Y TABLAS EN ROM

Las tablas de verdad, como ya sabemos, se usan para diseñar circuitos combinacionales con compuertas lógicas. Mediante la resolución con tablas en ROM nos damos cuenta de que, si lo deseamos, podemos sustituir dicho circuito combinacional por un microcontrolador, siendo más fácil y rápido de diseñar e implementar.

Nos puede ser útil para resolver tablas de verdad. Veamos un ejemplo: supongamos que tenemos que resolver una tabla de verdad, como la que vemos en la **Tabla 1**, en donde tenemos tres entradas y cinco salidas.

ENTADAS			SALIDAS				
C	B	A	X4	X3	X2	X1	X0
0	0	0	1	0	1	0	0
0	0	1	0	1	0	1	0
0	1	0	0	1	1	0	0
0	1	1	0	0	1	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	1	1	0
1	1	0	1	0	1	0	0
1	1	1	0	1	0	0	1

Tabla 1. *Tabla de verdad, ejemplo para la resolución mediante tablas en ROM.*

De esta forma, el offset estará dado por las entradas, y las salidas se obtendrán de la tabla. Podemos usar el mismo circuito de Mi primer programa que ya estudiamos, pero en este sólo se usarán tres de los interruptores conectados al Puerto A. El programa sería algo similar a esto:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

     PROCESSOR 16F84A              ;El microcontrolador usado es el PIC16F84
     #INCLUDE <P16F84A.INC>        ;Se incluye el archivo de definiciones

     ORG            0x00           ;Establece el origen del programa

;Configuración de puertos:

     bsf            STATUS, RP0    ;Acceso al banco 1
     movlw          b'00011111'
     movwf          TRISA          ;Configura el Puerto A como entrada
     clrf           TRISB          ;Configura el Puerto B como salida
     bcf            STATUS, RP0    ;Acceso al banco 0

;Programa principal:

inicio
```

```
     movf          PORTA, W      ;Lee los datos del Puerto A
     andlw         b'00000111'   ;Obtiene solo los tres primeros bits
     call          tabla         ;Llama a tabla para obtener el valor
correcto
     movwf         PORTB         ;Una vez obtenido el valor lo envía al
Puerto B
     goto          inicio        ;Crea un bucle infinito

;Subrutina que define la tabla en ROM:

tabla
     addwf PCL, F                ;Suma PCL + W para obtener el valor
de la tabla

     retlw         b'00010100'   ;Valor de salida para la entrada 000
     retlw         b'00001010'   ;Valor de salida para la entrada 001
     retlw         b'00001100'   ;Valor de salida para la entrada 010
     retlw         b'00000101'   ;Valor de salida para la entrada 011
     retlw         b'00011010'   ;Valor de salida para la entrada 100
     retlw         b'00001110'   ;Valor de salida para la entrada 101
     retlw         b'00010100'   ;Valor de salida para la entrada 110
     retlw         b'00001001'   ;Valor de salida para la entrada 111

     END
```

Desde **www.redusers.com** podemos descargar el archivo fuente llamado **Tabla verdad.asm**, para un análisis más detallado y el archivo **Tabla verdad.hex**, para su uso en el circuito para comprobar su funcionamiento. Podemos observar cómo el programa obtendrá el valor correcto de la tabla de datos según el valor de la entrada.

La directiva DT

Si queremos simplificar el manejo de tablas podemos hacer uso de la directiva **DT** (*Define Table*). Su sintaxis es la siguiente:

DT expr, expr, expr...

Cada **expr** es un valor que estará en la tabla generada por la directiva. La directiva **DT** generará automáticamente, al momento de ensamblar, la lista con los respectivos **retlw** por cada **expr** que coloquemos. Por ejemplo:

```
     DT d'63', d'25', d'127', d'88'

Es equivalente a escribirlo así:

     retlw  d'63'
     retlw  d'25'
     retlw  d'127'
     retlw  d'88'
```

La directiva **DT** es muy útil para definir caracteres o cadenas de código ASCII cuando lo necesitemos. Por ejemplo:

```
     DT "PIC"

Es equivalente a:

     retlw  0x50          ;P en código ASCII
     retlw  0x49          ;I en código ASCII
     retlw  0x43          ;C en código ASCII

O a esto:

     retlw  'P'
     retlw  'I'
     retlw  'C'
```

De esta forma, como vimos en el ejemplo anterior, mediante la directiva **DT**, es más fácil construir tablas en nuestros programas.

Las tablas están grabadas en la memoria de programa, por eso se las llama tablas en ROM y hay que tener en cuenta que los datos en ellas son fijos, no se pueden modificar, a menos que se grabe de nuevo el microcontrolador.

Los saltos indexados y el contador de programa

Cuando manejamos tablas de datos o saltos indexados en nuestros programas con la instrucción **addwf PCL, F** hay que tener en cuenta algunos detalles muy importantes para que funcionen correctamente. Veamos un par de recomendaciones para utilizar las tablas de datos de forma correcta:

Primero, cuando se ejecute una instrucción que tiene como destino el registro PCL, también se modificará el registro PCH, como vemos en la **Figura 39**. El registro PCL se puede leer y escribir directamente, pero el registro PCH no, por lo que debe escribirse mediante el registro **PCLATH** (*PC Latch High*).

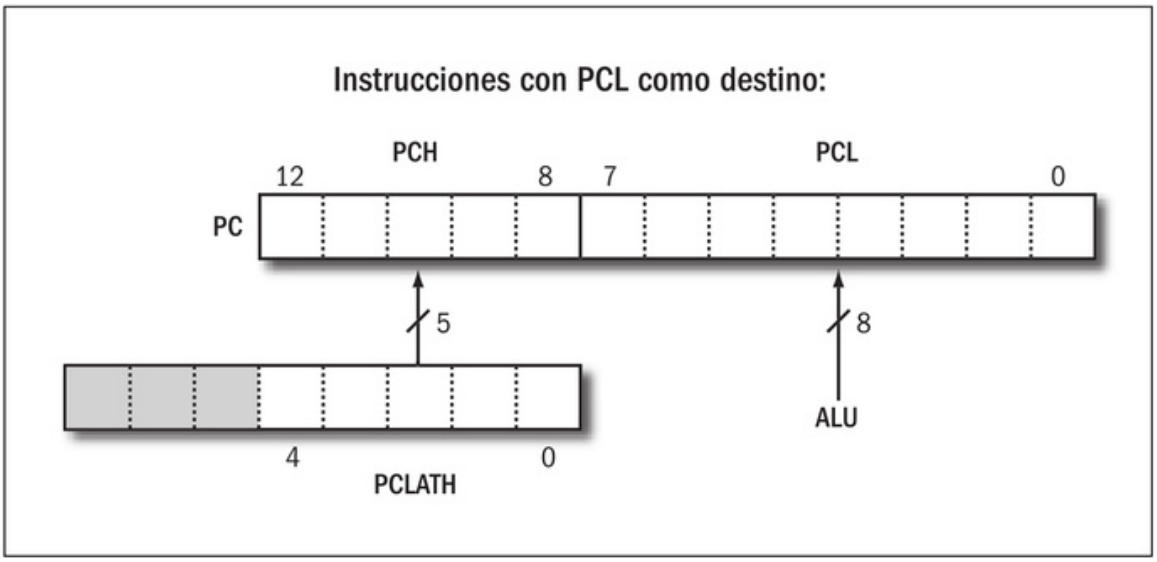


***Figura 39.** Las instrucciones que tienen como destino PCL modifican todo el contador de programa (PC).*

Al ejecutar la instrucción **addwf PCL, F** (o cualquier otra que tenga PCL como destino) realmente se estará escribiendo en todo el contador de programa, ya que PCL se escribirá directamente en los 8 bits más bajos del contador de programa y simultáneamente el registro PCLATH se escribirá en el registro PCH. Como el registro PCLATH toma el valor de 0 ante cualquier reset, si sólo modificamos el registro PCL, entonces únicamente podremos trabajar en los primeros 256 bytes de la memoria de programa, ya que PCLATH siempre tendrá el valor 0 mientas no lo modifiquemos. Por ejemplo, este fragmento de código no funcionará correctamente:

```
     ORG          0x22          ;Dentro de los primeros 256 bytes
     movlw        d'2'
     call         tabla

     ORG          0x100         ;Más allá de los 256 bytes
tabla
     addwf        PCL. F        ;Comienza la tabla

     retlw        d'10'
     retlw        d'11'
     retlw        d'12'
```

El origen de la tabla de datos está más allá de los primeros 256 bytes (100 h) de la memoria, por lo que la instrucción **addwf PCL, F** sólo modificará los primeros 8 bits del contador de programa, lo cual producirá un salto hacia alguna dirección dentro de estos primeros 256 bytes y el programa no funcionará.

El segundo punto a tener en cuenta es que si la tabla está en una posición cercana al límite de los primeros 256 bytes de la memoria de programa, al realizar la suma del PCL más W el registro PCL puede desbordarse. Y, de la misma forma, al no modificarse el PCH adecuadamente, el salto no será correcto y el programa no funcionará. Veamos un ejemplo de esto:

```
      ORG           00h           ;El origen esta en la dirección 0
      movlw         d'1'          ;Define el offset para el salto
      call          tabla
      .                           ;Regresa con el valor "B" en W
      .
      .
      movlw         d'3'          ;Define el offset para el salto
      call          tabla
      .                           ;No regresa...
      .
      .

      ORG           FDh           ;El origen de la tabla esta cerca
                                  ; del límite de los primeros 256
                                  ; bytes de la memoria
tabla
      addwf         PCL, F        ;Dirección FDh
                                  ; PCH=00000 PCL= 11111101
      retlw         'A'
      retlw         'B'           ;Dirección FFh
      retlw         'C'
      retlw         'D'           ;Dirección 101h el segundo salto
                                  ; debería ser aquí
```

Podemos observar en el ejemplo cómo, cuando se pone un offset de 1, el salto es adecuado, ya que al sumar el **PCL+1** entonces **PCH=00000** y **PCL=11111111**. Pero en el segundo salto, el offset tiene un valor de 3, por lo que al sumar el **PCL+3** el registro PCL se desbordará, y como el registro PCH no se incrementa, entonces **PCH=00000** y **PCL=00000001**. Por lo que el segundo salto se hará a la dirección 01 h

en lugar de hacerlo a la dirección 101 h y el programa ya no funcionará correctamente. En este ejemplo hemos colocado deliberadamente la tabla en un lugar no adecuado para poder ver el efecto. Por eso, se recomienda colocar las tablas siempre dentro de los primeros 256 bytes de la memoria de programa para evitar problemas con los saltos y que los programas no funcionen correctamente.

UN DADO ELECTRÓNICO

Con las técnicas de programación estudiadas hasta el momento, ya podemos comenzar a diseñar circuitos con el microcontrolador PIC16F84A. Veamos cómo diseñar y armar un **dado electrónico**. En primer lugar debemos definir nuestro circuito.

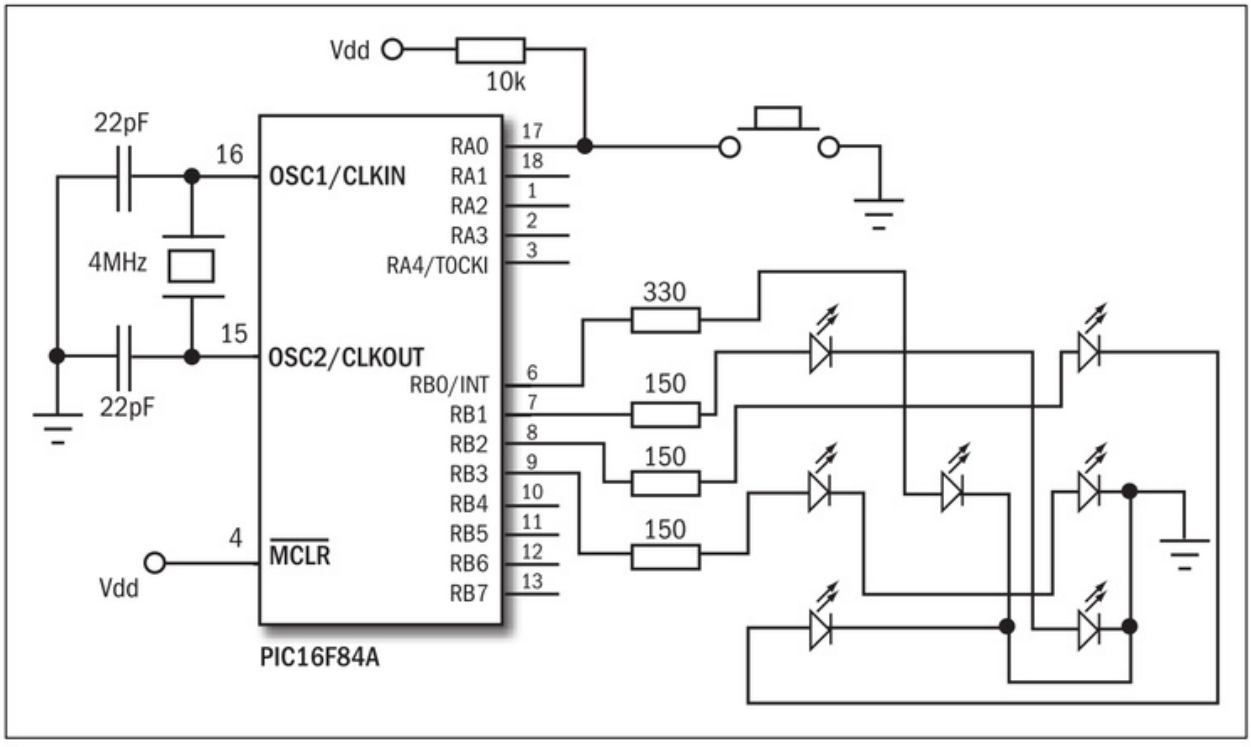


***Figura 40.** Un sencillo y práctico dado electrónico para comenzar a practicar con la programación.*

El circuito es realmente sencillo, tenemos un pulsador conectado al pin RA0 y 7 leds conectados a las cuatro primeras líneas del Puerto B. Mediante la combinación de salidas del Puerto B se encenderán los leds correspondientes a cada uno de los números del dado. Una vez que tenemos el circuito vamos a diseñar el programa.

Haremos uso de una tabla para obtener los valores de salida para los números del dado según las conexiones en el Puerto B y además de un retardo de unos 50 ms. Al presionar el pulsador, el dado "girará" y recorrerá los seis números, y volverá a

comenzar. Al soltarlo, el último número permanecerá en los leds. Como el cambio es muy rápido, no podemos saber cuál será el número elegido y esto hace que el dado genere un número al azar, tal como un dado real.

En la página web de la editorial (**www.redusers.com**) podemos descargar los archivos **Dado.asm** y **Dado.hex**, para estudiarlo y grabarlo en el microcontrolador respectivamente, y así tener nuestro dado electrónico funcionando. El código fuente está suficientemente comentado para su fácil análisis y comprensión. Con lo que hemos aprendido, podremos realizar todos los cambios que consideremos convenientes, para mejorar el funcionamiento y practicar.

LIBRERÍAS DE SUBRUTINAS

Como ya estudiamos en el **Capítulo 4**, podemos usar la directiva **#INCLUDE** para incluir otros archivos y al momento de llevar a cabo el ensamblado la directiva literalmente pegará el contenido del archivo en el lugar indicado. Esto nos resultará muy útil para crear **librerías de subrutinas**. Muchas subrutinas que diseñemos serán utilizadas en muchos de nuestros programas, por lo que es conveniente armar una librería de subrutinas para hacer más fácil y rápida la escritura de nuestros programas.

Para crear una librería sólo basta con crear un archivo nuevo en el editor de MPLAB y ahí escribir la subrutina deseada. Al tener el archivo con la subrutina en MPLAB lo guardaremos como un archivo ***.INC** y de esa forma ya tenemos nuestra librería lista para usar. Pero debemos seguir algunas reglas importantes para escribir librerías:

- Las librerías se escriben como cualquier programa, por lo tanto, deben especificar sus propias constantes, Esto debe hacerse mediante la directiva **CBLOCK** y hay que usar esta directiva también en el programa principal.

Como las constantes deben tener una dirección de la memoria de datos, no sabemos cuántas y cuáles utilizaremos en el programa principal, por lo que debemos definirlas en él mediante la directiva **CBLOCK**. De esta forma, en la librería también se usará la directiva **CBLOCK**, pero sin dirección, lo que permitirá que automáticamente se asigne a partir de la última dirección libre a las constantes de la librería. Si no usamos la directiva **CBLOCK** en el programa principal, en la librería se tomará la dirección predeterminada (0) y el programa no funcionará, por lo tanto, es obligatorio usar la directiva en el programa principal si vamos a incluir librerías que utilicen constantes. Si no empleamos constantes en el programa principal debemos aún colocar la directiva **CBLOCK**:

```
      CBLOCK 0x0C
      ENDC
```

Esto no define ninguna constante pero sí la dirección de inicio.

- Las librerías no deben terminar con la directiva **END**. De hacerlo, el ensamblado se detendrá en ese punto y no se ensamblará el resto del programa, si lo hay.
- En las constantes o etiquetas de la librería deben usarse nombres poco comunes para evitar repetirlos en el programa principal.

Para observar el uso de librerías, hemos creado una librería llamada **DEBOUNCE.INC**, que podemos descargar de **www.redusers.com**, junto con el archivo **Contador binario 2.asm**, en donde hemos usado esta librería a modo de ejemplo, para eliminar los rebotes del pulsador en el mismo ejemplo del contador binario que vimos antes. También proporcionamos el archivo **Contador binario 2.hex** para poder grabarlo en el PIC. Ahora, con el uso de la librería, el código fuente quedará de la siguiente forma:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      cuenta
      cont1
      cont2
      ENDC


      ORG           0x00          ;Establece el origen del programa

;Configuración de puertos:

      bsf           STATUS, RP0   ;Acceso al banco 1
      movlw         b'00000001'
      movwf         TRISA         ;Configura RA0 como entrada
      clrf          TRISB         ;Configura el Puerto B como salida
      bcf           STATUS, RP0   ;Acceso al banco 0
```

```
;Programa principal:

     clrf          cuenta        ;Borra el registro para la cuenta

clrf               PORTB         ;Borra el Puerto B al inicio
inicio
     btfsc         PORTA, 0      ;¿Se ha presionado el pulsador?
     goto          inicio        ;No, genera un bucle
     call          debounce      ;Si, llama a subrutina "debounce" para
eliminar rebotes
     btfsc         PORTA, 0      ;¿Sigue pulsado?
     goto          inicio        ;No, falso disparo ve a inicio
     incf          cuenta, F            ;Si, incrementa la cuenta
     movf          cuenta, W
     movwf         PORTB         ;Envíala al Puerto B
pulsado
     btfss         PORTA, 0      ;¿Sigue pulsado?
     goto          pulsado       ;Si, genera un bucle hasta que el
pulsador sea liberado
     goto          inicio        ;Inicia de nuevo

; Se incluye la librería de retardo anti-rebotes:

     #INCLUDE      <DEBOUNCE.INC>

     END
```

La librería **DEBOUNCE.INC** debemos colocarla en la carpeta adecuada para que el ensamblador la encuentre. Por ejemplo, en la carpeta donde tengamos el archivo fuente con el que estamos trabajando. También podemos descargar de **www.redusers.com** el archivo **RETARDOS_US.INC**, que es una librería de ejemplo que contiene retardos de 10 a 500 microsegundos. Podemos abrirla en MPLAB para estudiar su contenido.

RESUMEN

En este capítulo hemos aprendido algunas técnicas de programación en lenguaje ensamblador, para poder comenzar a diseñar programas eficientes para nuestro microcontrolador. Las técnicas aprendidas se utilizarán en prácticamente todos los programas.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un diagrama de flujo?

2 ¿Qué es una subrutina?

3 ¿Cuál es la instrucción que se utiliza para llamar a una subrutina?

4 ¿Para qué sirve la pila en el PIC16F84A?

5 ¿Qué es un bucle o lazo?

6 ¿Qué es un bucle infinito?

7 ¿Qué es un retardo?

8 ¿Para qué sirve un retardo?

9 ¿Qué es una tabla en ROM?

10 ¿Por qué no se deben colocar las tablas más allá de los primeros 256 bytes de la memoria de programa?

PRÁCTICAS

1 Diseñe una subrutina de retardo de exactamente 1 segundo, incluyendo el llamado a subrutina con la instrucción call.

2 Diseñe un programa que genere la siguiente secuencia en los 8 leds conectados al Puerto B:

00000000
00011000
00111100
01111110
11111111
11100111
11000011
10000001

Cada 100 ms aproximadamente, use una tabla para generar el efecto.

3 Tome el retardo que diseñó en el punto 1 y genere una librería llamada RETARD01S.INC siguiendo las pautas, para poder usar este retardo en futuros programas como librería.

4 Diseñe un programa que muestre una cuenta en binario en los 8 leds conectados al Puerto B de forma automática. La cuenta se incrementa automáticamente cada segundo. Use la librería de retardo que diseñó en el punto 3.

Capítulo 7

Displays LED y LCD

La comunicación del microcontrolador con el usuario es muy importante para nuestros proyectos, por lo que en este capítulo estudiaremos las formas de comunicación del PIC con el mundo exterior: los displays. Desarrollaremos el uso de los displays de 7 segmentosy los LCD.

DISPLAYS LED DE 7 SEGMENTOS

Los displays de 7 segmentos nos serán de utilidad en algunos proyectos en donde se requiera mostrar números a la salida. Un display de siete segmentos es un elemento que contiene 7 barras o segmentos en los cuales hay un LED en cada uno de ellos. De esta forma, al encenderlos selectivamente, se mostrarán números en él.

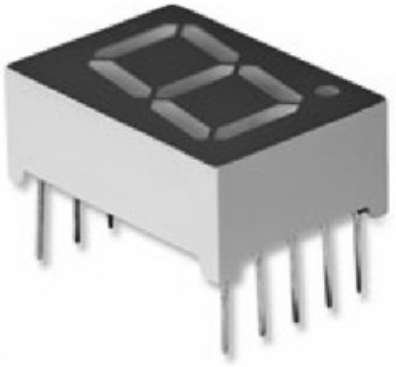


Figura 1. *Este display es útil para representar números mediante la iluminación de sus LEDs.*

Existen displays de **ánodo común** y de **cátodo común**, según la conexión de los LEDs internamente. Es decir, en los de ánodo común, todos los ánodos de los LEDs están interconectados y en los de cátodo común serán los cátodos los que estén interconectados.

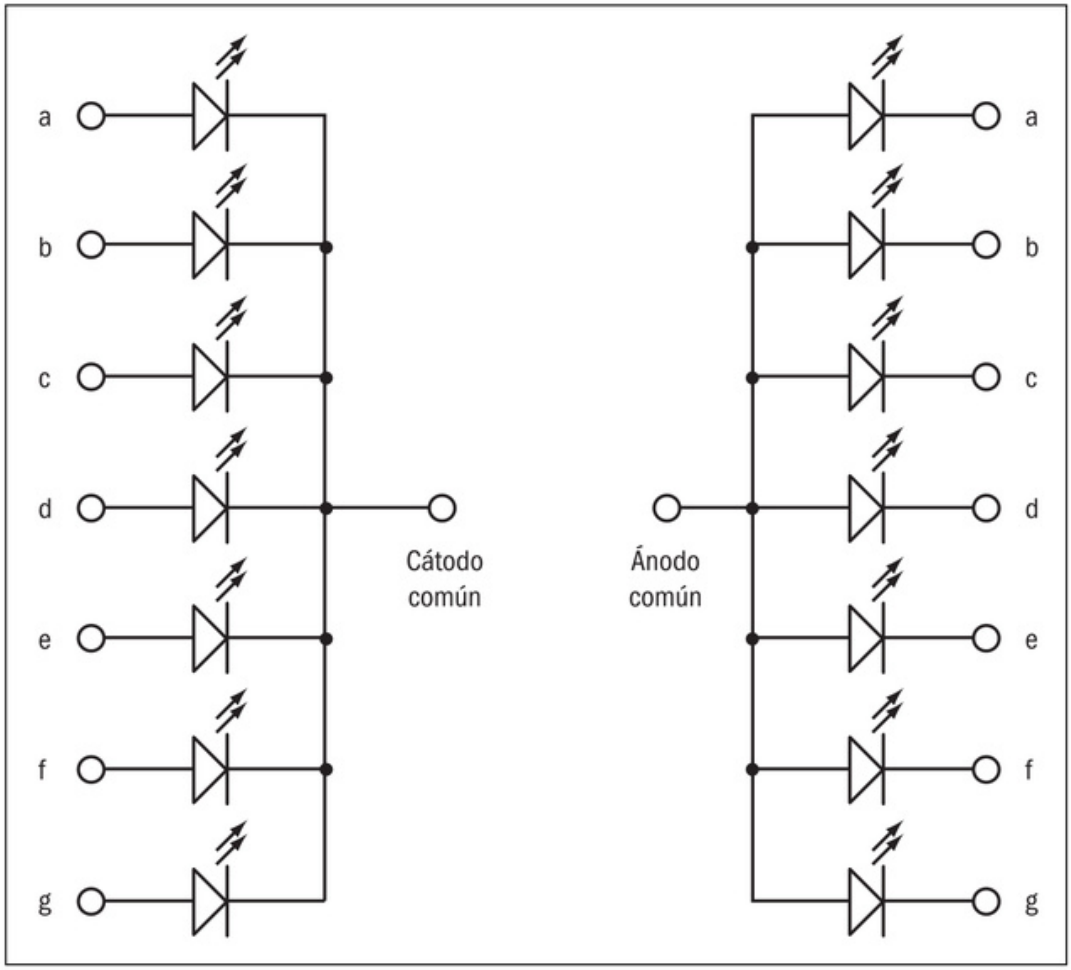


Figura 2. *Configuración interna de displays de ánodo y de cátodo común.*

Los segmentos del display encenderán con un 1 ó un 0 y debemos seguir un código correcto para encender los segmentos adecuados para representar los números en el display. Los segmentos del display se identifican mediante letras de la **a** a la **g**.

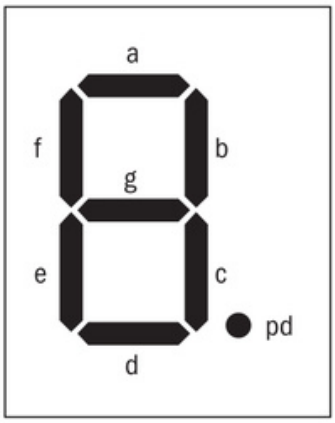


***Figura 3.** Identificación para el código de 7 segmentos. **pd** significa punto decimal.*

Dependiendo del número a representar deberemos encender los segmentos adecuados. En la **Figura 4** se muestran las posibilidades para representar números y algunas letras para poder representar cantidades decimales o hexadecimales.

gfedcba	00111111	00000110	01011011	01001111	01100110	01101101	01111101	00000111
Hex.	3Fh	06h	5Bh	4Fh	66h	6Dh	7Dh	07h
Display	0	1	2	3	4	5	6	7
gfedcba	01111111	01101111	01110111	01111100	00111001	01011110	01111001	01110001
Hex.	7Fh	6Fh	77h	7Ch	39h	5Eh	79h	71h
Display	8	9	A	b	C	d	E	F

***Figura 4.** Tabla de caracteres para display de cátodo común para representar valores en decimal o hexadecimal.*

III ¿Y LOS DE ÁNODO COMÚN?

En las tablas y ejemplos de éste capítulo hemos usado displays de cátodo común pero, por supuesto, también pueden ser usados displays de ánodo común, sólo hay que conectarlos correctamente e invertir los valores de salida con respecto a los mostrados, ya que en éstos el segmento encenderá con un 0 y se apagarán con 1.

Aunque también se pueden representar otros caracteres, éstos son los principales. Si necesitamos algún otro podemos generarlo al encender o apagar los segmentos necesarios. En la **Figura 4** se muestran las representaciones sólo para displays de cátodo común.

Conversión de binario a BCD

En algunos proyectos será necesario utilizar números representados en código BCD. Para ello, veremos una técnica para convertir un número binario de 8 bits.

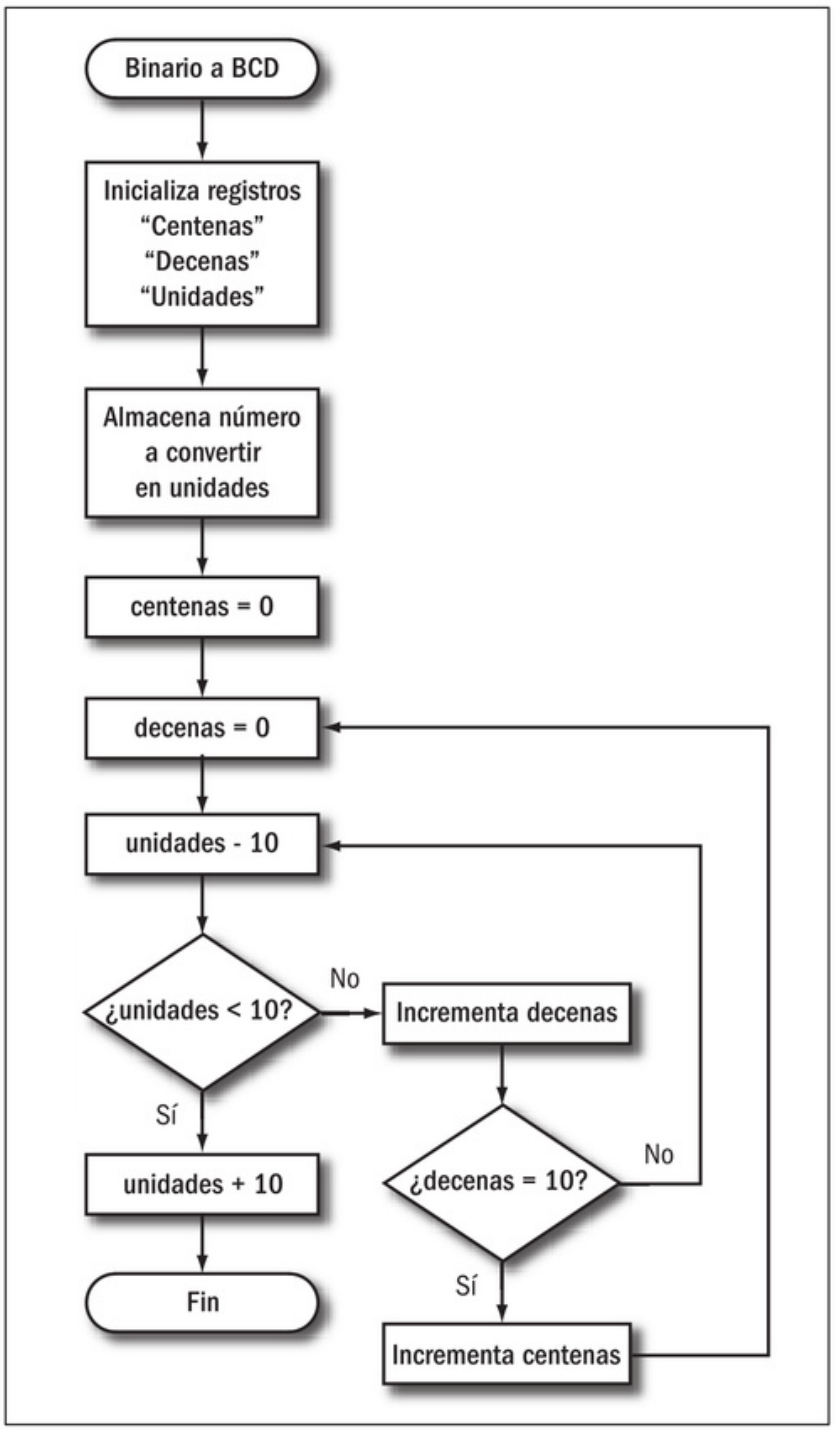


***Figura 5.** Diagrama de flujo para la conversión de binario a BCD.*

La conversión se realiza restando 10 consecutivamente al número a convertir. De esta forma, se van contando las decenas y las centenas (cuando se acumulan 10 decenas) en él. Cuando el número a convertir es menor a 10, la conversión está completa. Podemos tener nuestra rutina de conversión como una librería, como la siguiente:

```
      CBLOCK
      BCDcentenas
      BCDdecenas
      BCDunidades
      ENDC

BINaBCD
      movwf       BCDunidades          ;Mueve W al registro de unidades
      clrf        BCDcentenas
borra_decenas
      clrf        BCDdecenas

convBCD
      movlw       d'10'
      subwf       BCDunidades, F
      btfss       STATUS, C            ;¿unidades < 10?
      goto        BCDfin               ;Si, finaliza
      incf        BCDdecenas, F        ;No, Suma una decena
      subwf       BCDdecenas, W
      btfss       STATUS, Z            ;¿Se han acumulado 10 decenas?
      goto        convBCD              ;No, continúa
      incf        BCDcentenas, F       ;Si, incrementa las centenas
      goto        borra_decenas        ;Y borra el valor de las decenas
BCDfin
      addwf       BCDunidades, F       ;Recupera el valor de las unidades
      return                           ;Regresa
```

SÓLO HASTA 255

En la librería para la conversión de código binario a BCD, sólo se puede convertir como máximo un número de 255_{10} ya que, como sabemos, esa es la capacidad máxima de los registros de 8 bits. Al escribir nuestros programas que requieran esta conversión debemos tenerlo en cuenta.

Se llama a la subrutina **BINaBCD**, que devuelve el número convertido en los registros **BCDcentenas**, **BCDdecenas**, y **BCDunidades**. En cada uno se almacena el número en código BCD en el nibble bajo. En **www.redusers.com** tenemos la librería **BINABCD.INC**.

Displays multiplexados

Si necesitamos conectar más de un display para representar más de un dígito y no tenemos suficientes líneas en el PIC16F84A, podemos recurrir a la técnica de **displays multiplexados**. En este caso, se conecta más de un display al Puerto B y se activa uno a uno de forma muy rápida para dar la impresión de que todos están encendidos.

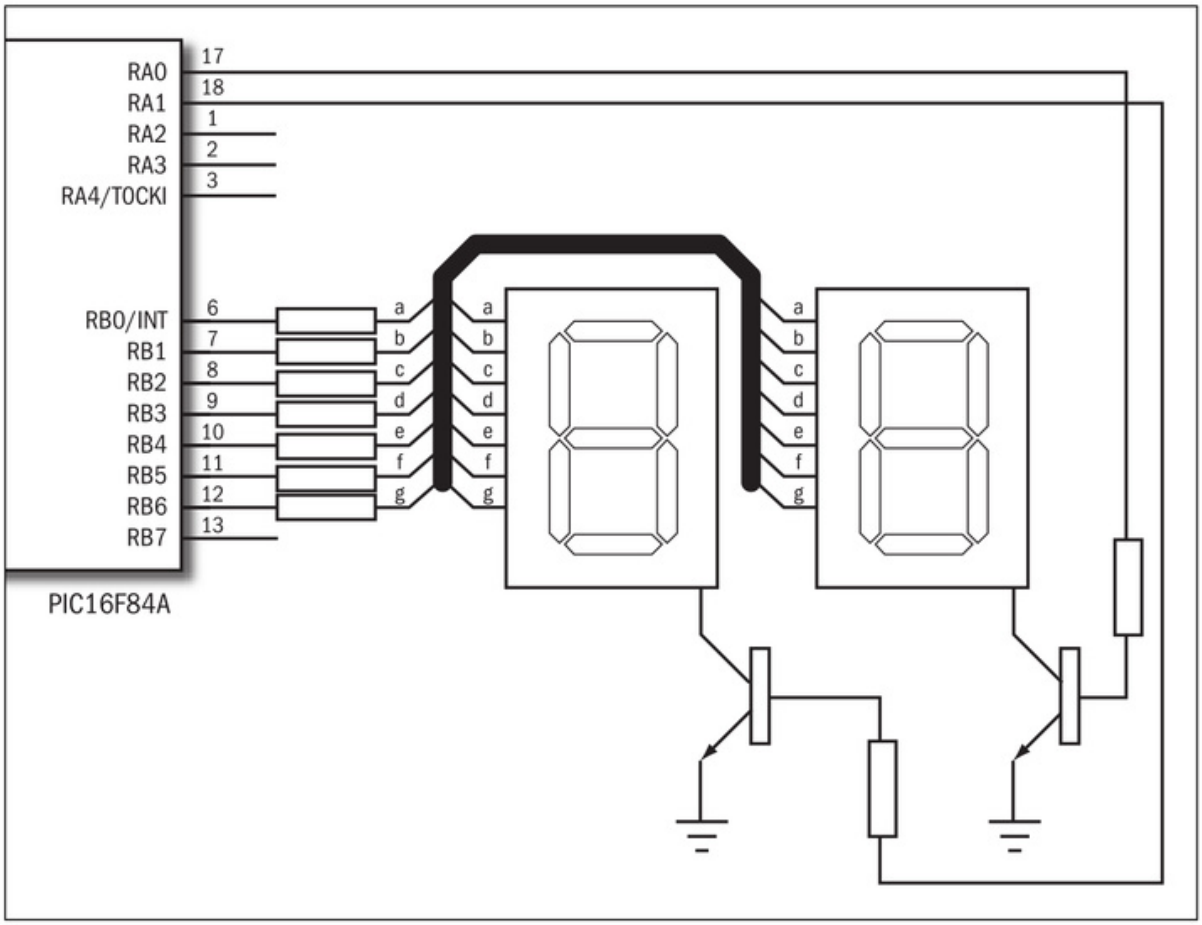


***Figura 6.** Conexión típica de un display multiplexado con PIC16F84A.*

LA MULTIPLEXACIÓN

La técnica de displays multiplexados es muy usada en todo tipo de aparatos electrónicos de consumo para mostrar información en las pantallas, ya que permite presentar mucha información con pocas líneas de comunicación. Además, ahorra energía, debido a que sólo un display o una parte de él está encendida a la vez.

Mediante los transistores activaremos uno de ellos, poniendo un 1 en la salida RA0 o RA1, y de esa forma pondremos un dígito primero en un display y luego en el otro. En el ejemplo de la **Figura 6** se muestran sólo dos, pero podemos conectar más.

Contador de turnos

Veamos un ejemplo de una aplicación con displays multiplexados. Todos hemos ido alguna vez a un banco o a un negocio donde debemos tomar un número y luego esperar hasta que nos toque el turno para ser atendidos. En una pantalla podemos ver el número de turno y a qué ventanilla deberemos ir. Vamos a realizar una de esas pantallas de un contador de turnos.

***Figura 7.** Con el PIC16F84A podemos construir un sencillo contador de turnos como los de los bancos o negocios.*

Para esta aplicación necesitaremos usar tres displays de 7 segmentos, los cuales serán multiplexados para mostrar los turnos, y tres pulsadores para tres ventanillas diferentes. En la **Figura 8** podemos ver el diagrama del circuito.

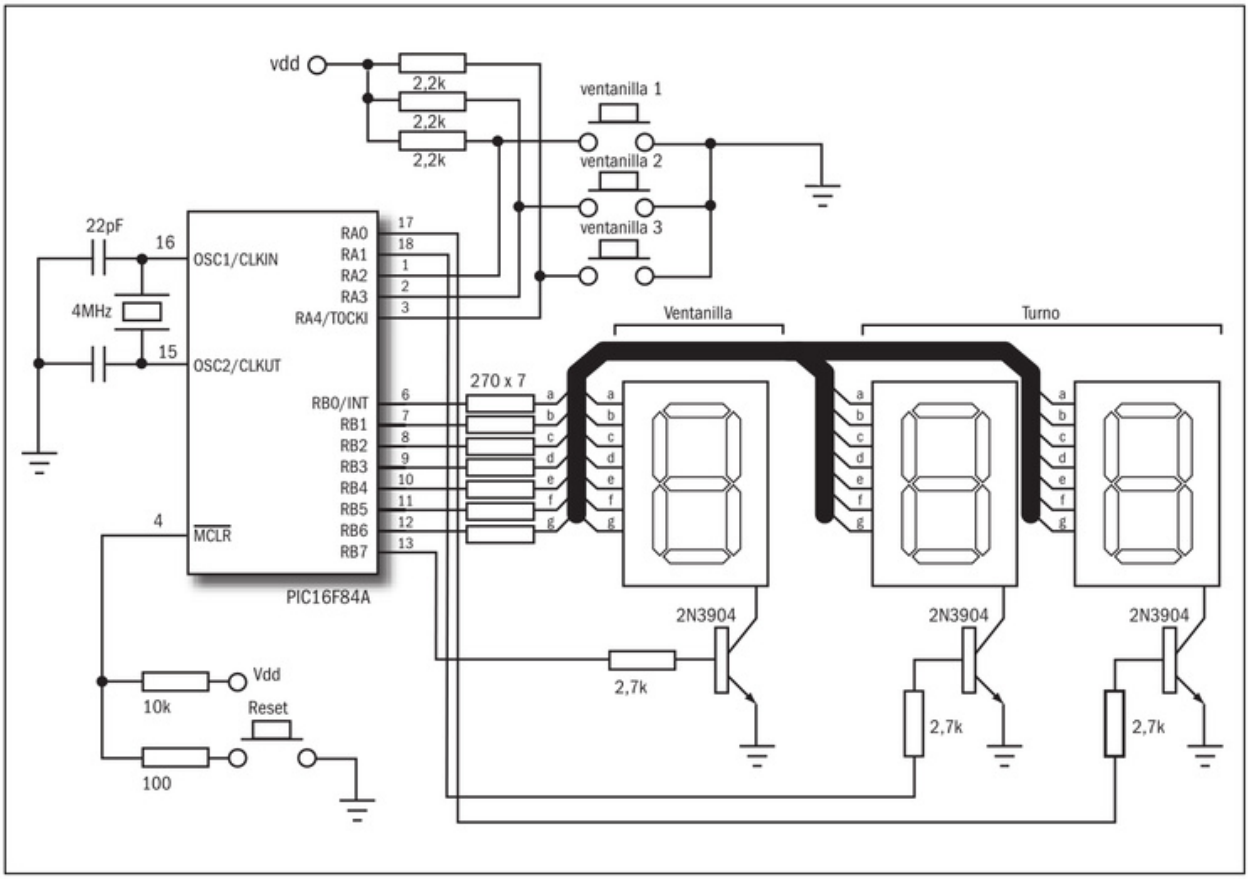


***Figura 8.** Diagrama de nuestro contador de turnos con microcontrolador PIC.*

El programa no tiene mayor dificultad, sólo hay que monitorear los pulsadores y cuando uno de ellos es presionado, se aumenta el valor del turno y se identifica el número de ventanilla. Luego se activan los displays de tal forma que enciendan uno tras otro de forma rápida. Podemos descargar el código fuente **Contador de turnos.asm** del sitio **www.redusers.com** para analizarlo y comprender su funcionamiento. Está suficientemente comentado, y con lo que hemos estudiado hasta ahora, no debemos tener mayores problemas para entenderlo. También podemos descargar el archivo **Contador de turnos.hex** para poder grabarlo en el PIC y comprobar el funcionamiento del circuito.

Como podemos observar, se llama constantemente a la subrutina para encender los displays durante todo el programa, ya que si no se hace así, se apagarán. Esta es una pequeña desventaja de usar displays multiplexados conectados directamente al PIC, además de que este tipo de displays está muy limitado en cuanto a los caracteres que puede mostrar más allá de números.

DISPLAY LCD

Podemos también usar un **display de cristal líquido** o **LCD** para mostrar cualquier carácter numérico o alfabético a la salida. Existen displays LCD que incluyen un circuito controlador para poder mostrar los caracteres en su pantalla. Los más populares son los basados en el controlador **Hitachi 44780**. Estos displays se pueden encontrar en configuraciones de 1 línea de 16 caracteres (16x1), dos líneas de 16 caracteres (16x2), o más, como 20x2 ó 40x4. Cada carácter está formado normalmente por una matriz de puntos de 5x7. En nuestro caso estudiaremos el display de 16 caracteres por 2 líneas.

***Figura 9.** Apariencia física de un display de 2 líneas y 16 caracteres por línea.*

III ILUMINANDO LA PANTALLA

Algunos displays cuentan con una luz de fondo (**back light**) para iluminar la pantalla y permitir una mejor visualización, o poder hacerlo en la oscuridad. Algunos tienen dos pines que sirven para la conexión de esta luz, y generalmente son LEDs. Es por eso que debemos consultar la hoja de datos para la correcta conexión de estos pines de la luz de fondo.

Aunque el modelo **LM016L** es el más popular, podemos usar cualquier otro que sea compatible con el controlador 44780. Este tipo de displays es ideal para poder presentar caracteres y números, ya que su consumo de potencia es muy reducido, lo que los hace ideales para el uso en circuitos alimentados con pilas o que requieran poco consumo en su operación. Hablaremos específicamente de los modelos con controlador Hitachi 44780, pero los ejemplos y programas pueden adaptarse también a otros displays que tengan un controlador compatible con éste, por ejemplo, los **JHD162A** que, aunque tienen un controlador diferente (KS0066), funcionan de manera casi idéntica.

Pines y sus funciones

El módulo LCD debe tener 14 ó 16 pines. La diferencia de tener 14 ó 16 es que en los de 16 hay dos pines que son usados normalmente para conectar la **luz de fondo** (*back light*) si el display cuenta con ella. El orden de numeración de los pines depende del modelo exacto de display que encontremos en las tiendas, y debe estar impreso en él como referencia para poder conectarlos adecuadamente.

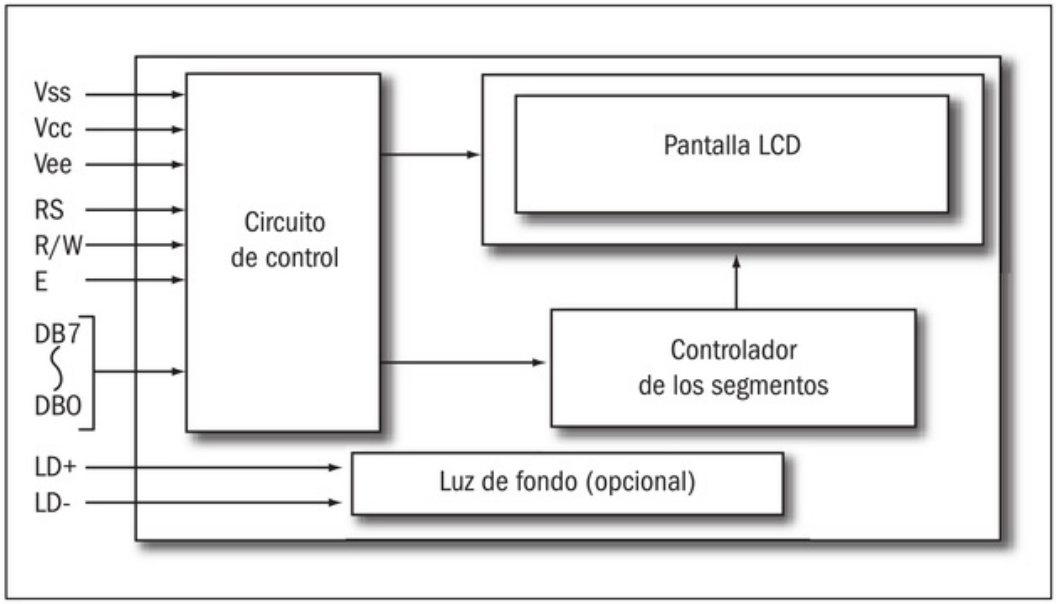


***Figura 10.** Diagrama de bloques del panel LCD.*

EL CONTRASTE

El contraste de la pantalla LCD se controla mediante un preset de 10 kohms, pero si no tenemos uno o no queremos usarlo, podemos llevar el pin Vee a tierra, y de esa forma tendremos siempre el máximo contraste en la pantalla, aunque esto puede ser incómodo para quien lo ve. Por el contrario, si llevamos el pin directamente a Vdd, el contraste estará al mínimo.

Debemos tener en cuenta que cada pin cumple una función especial. Es por eso que en la **Tabla 1** detallamos la función de los pines del módulo LCD.

PIN	SEÑAL	NOMBRE	FUNCIÓN
1	Vss	Ground	Tierra (masa)
2	Vcc	Power supply	Voltaje de alimentación
3	Vee	LCD contrast	Ajuste del contraste
4	RS	Register selecton	RS=0 Registro de instrucciones
			RS=1 Registro de datos
5	R/W	Read/Write	R/W=0 Escritura
			R/W=1 Lectura
6	E	Enable	E=0 Deshabilita el display
			E=1 Habilita el display
7~14	DB0~DB7	Data bus	Bus de datos

***Tabla 1.** Función de los pines del panel LCD.*

Veamos con más detalle la función de los pines:

1 Vss: este pin es para conectar tierra o masa.
2 Vcc: este pin recibirá el voltaje de alimentación para el display. Debemos alimentarlo con 5 V.
3 Vee: este pin controla el contraste de la pantalla. Podemos colocar un preset para controlarlo manualmente.

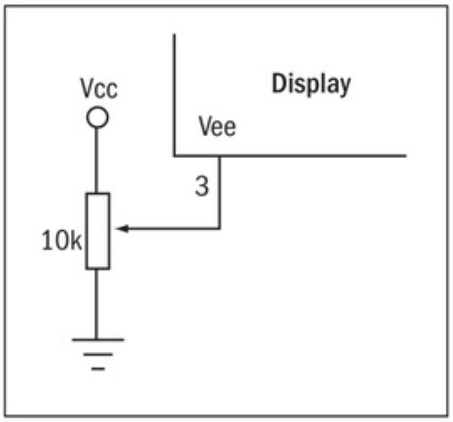


***Figura 11.** Podemos ajustar el contraste de la pantalla con un preset de 10 Kohms para una visualización correcta.*

4 RS (**register select**): el display cuenta con dos registros internos en los cuales podemos leer o escribir. El primero es el **registro de datos** (*data register*), que es donde enviaremos los códigos de los caracteres que se mostrarán en la pantalla del display. El segundo es el **registro de instrucciones** o comandos (*instruction register*), el cual recibirá instrucciones para el control o configuración del display. Mediante

el pin RS se selecciona en cuál de estos dos registros vamos a leer o escribir.
5 R/W (**Read/Write**): como en la mayoría de los dispositivos que contienen memorias, podemos leer o escribir en ellos. Este pin nos permite seleccionar si vamos a leer o escribir en el display.
6 E (**enable**): este pin habilita o deshabilita el display. Si ponemos un 0 en él, deshabilitaremos el display y el bus de datos entra en el modo de alta impedancia. Si colocamos un 1 habilitaremos el display para enviar o recibir datos.
7 a 14 DB0 a DB7: estas 8 líneas son el bus de datos que usaremos para escribir datos o instrucciones en el display, o leer datos de él.

Los caracteres de la CGROM

El display tiene una memoria interna llamada **CGROM** (*Character Generator ROM*), donde están almacenados los caracteres que puede mostrar. Es una memoria no volátil y fija. Debemos seguir los códigos de la **Figura 12** donde se representan los caracteres que el display puede mostrar y sus códigos en binario, que son los que debemos enviarle para que muestre los caracteres en la pantalla.

Figura 12. *Los 192 caracteres almacenados en la CGROM del display.*

Por ejemplo, para mostrar el carácter **E** debemos enviar el dato **01000101** al display. Las casillas marcadas con números entre paréntesis pertenecen a la **CGRAM** (*Character Generator RAM*) y son 8 espacios reservados para que el usuario pueda crear sus propios caracteres y almacenarlos ahí.

La DDRAM

El display cuenta con una memoria RAM que se utiliza para almacenar los caracteres que se muestran en la pantalla. Esta memoria es llamada **DDRAM** (*Data Display RAM*) y tiene una capacidad de 80 bytes, dividida en dos bancos de 40 bytes cada uno, precisamente un banco por cada línea del display. En la **Figura 13** podemos observar un diagrama de cómo se organiza la DDRAM.

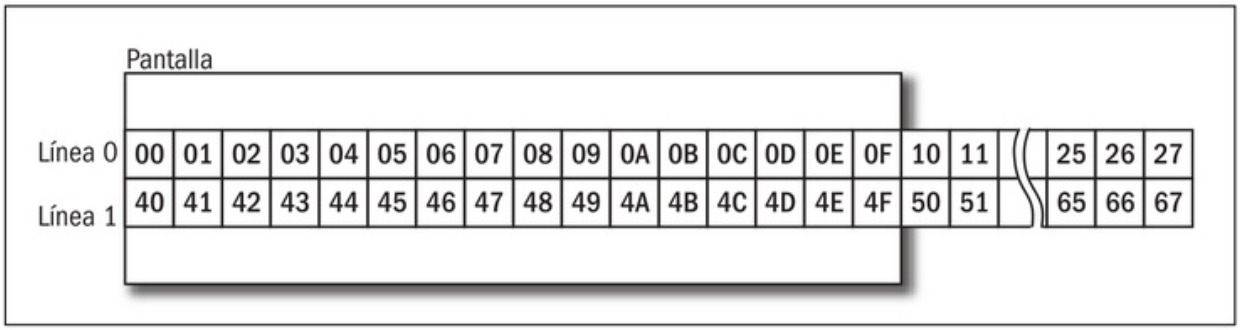


***Figura 13.** Organización de la DDRAM y su correspondencia con la pantalla del display.*

Como podemos apreciar en la **Figura 13**, la DDRAM se corresponde con los caracteres de la pantalla del display. La dirección **00h** es la primera de la línea 1 y la **40h** es la primera de la línea 2. Aunque la pantalla sólo mostrará 32 caracteres, la capacidad de la DDRAM es de 80.

Instrucciones o comandos

La comunicación entre el PIC y el display se hace a través de comandos y datos para controlar su funcionamiento. En la **Tabla 2** tenemos un resumen de los comandos y la forma de enviar datos o leerlos del display, especificados por el fabricante.

ACCIÓN	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	TIEMPO
Clear display	0	0	0	0	0	0	0	0	0	1	1.64ms
Return home	0	0	0	0	0	0	0	0	1	x	1.64ms
Entry mode set	0	0	0	0	0	0	0	1	**I/D**	**SH**	40us
Display control	0	0	0	0	0	0	1	**D**	**C**	**B**	40us
Cursor or display shift	0	0	0	0	0	1	**S/C**	**R/L**	x	x	40us
Function set	0	0	0	0	1	**DL**	**N**	**F**	x	x	40us
Set CGRAM adress	0	0	0	1	**CGRAM adress**						40us
Set DDRAM adress	0	0	1	**DDRAM adress**							40us
Read busy flag and address	0	1	**BF**	**DDRAM adress**							0
Write data to RAM	1	0	**Data**								40us
Read data from RAM	1	1	**Data**								40us

***Tabla 2.** Lista de los comandos para el display.*

A continuación, realizaremos una descripción detallada de cada uno de los comandos que incluimos en la **Tabla 2**, para comprender mejor su funcionamiento:

Clear display (borra display): borra la pantalla del display. Escribe 20h (espacio) en todas las posiciones de la DDRAM y coloca el cursor en la dirección 00h de la DDRAM, es decir, en la primera posición de la primera línea de la pantalla.
Return home (regresa al origen): devuelve el cursor a la posición 00h de la DDRAM.
Entry mode set (modo de entrada): movimiento del cursor y el display:

Bit **I/D** (*Increment or decrement of DDRAM*):
I/D=1 La posición del cursor se incrementa con cada escritura en la DDRAM.
I/D=0 La posición del cursor se decrementa con cada escritura en la DDRAM.

Bit **SH** (*Shift*):
SH=0 Los caracteres no se desplazan al escribir uno nuevo en la pantalla.
SH=1 Los caracteres se desplazan en la pantalla al escribir uno nuevo en la DDRAM, en la dirección indicada por el bit I/D.

Display control (control del display): controla algunos parámetros de visualización:

Bit **D** (*Display*):
D=0 El display se apaga, pero la DDRAM mantiene sus datos.
D=1 El display enciende.

Bit **C** (*Cursor*):
C=0 El cursor no se muestra.
C=1 El cursor se muestra.

Bit **B** (*Blink*):
B=0 No hay efecto en el cursor.
B=1 El cursor parpadea y tiene forma rectangular.

Cursor or display shift (desplazamiento del cursor o pantalla): controla el desplazamiento de la pantalla y el cursor. No afectan el contenido de la DDRAM.

S/C	R/L	OPERACIÓN
0	0	La posición del cursor es desplazada a la izquierda La DDRAM es decrementada en 1
0	1	La posición del cursor es desplazada a la derecha La DDRAM es incrementada en 1
1	0	La DDRAM y el cursor se desplazan a la izquierda
1	1	La DDRAM y el cursor se desplazan a la derecha

***Tabla 3.** Operación de los bits S/C y R/L.*

Function set (función): este comando epecifica algunos parámetros de configuración del funcionamiento del display, según los datos que especifiquemos en él:

Bit **DL** (*Data length*):
DL=0 Se usa una conexión de 4 bits.
DL=1 Se usa una conexión de 8 bits.

Bit **N** (*Number of lines*):
N=0 Una línea.
N=1 Dos líneas.

Bit **F** (*Font*):
F=0 Fuente o caracteres de 5x7 puntos.
F=1 Fuente o caracteres de 5x10 puntos.

Set CGRAM adress (dirección de CGRAM): especifica una dirección de la CGRAM sobre la cual se escribirá o leerá.
Set DDRAM adress (dirección de DDRAM): especifica una dirección de la DDRAM en la cual se escribirá o leerá.
Read busy flag and adress (lee el bit de ocupado y dirección de la DDRAM): permite leer una dirección de la DDRAM. Además, lee el bit de ocupado (*busy flag*). Este es el bit 7 (DB7) y sirve para saber si el display está realizando operaciones internas, en cuyo caso no podemos enviar datos ni comandos hasta que termine:

BF=0 El display está disponible.
BF=1 El display está ocupado y no puede recibir nuevos datos ni comandos.

Write data to RAM y **Read data from RAM** (escribir o leer datos de la RAM): estas dos últimas operaciones sirven para escribir o leer datos de la DDRAM.

El Busy flag

Como ya mencionamos, el bit llamado **Busy flag** es el bit 7 del bus de datos (DB7). Este **bit de ocupado** nos dirá si el display está realizando algún proceso interno y si es así, no podremos enviar otro dato o comando hasta que termine.
Existen dos modos de operar el display: con busy flag o sin él. Con busy flag debemos leer este bit antes de enviar datos o comandos para ver su estado. Si es 1, el display está ocupado y debemos esperar hasta que se ponga a 0 para realizar otro envío de datos. Si no queremos realizar la lectura del busy flag, podemos tomar los tiempos de la **Tabla 2**. Así, podemos agregar un retardo mayor a este tiempo para asegurar que la operación haya terminado. Si usamos el modo sin busy flag podemos conectar el pin R/W del display directamente a tierra, ya que no necesitaremos realizar operaciones de lectura, sólo de escritura, y así nos ahorraremos el uso de una línea de salida en el PIC.

Conexiones con 4 y 8 bits

Podemos conectar el PIC al display mediante un bus de 4 ó de 8 bits. Si lo hacemos sólo con 4 bits, nos ahorraremos 4 líneas de conexión, pero entonces tendremos que enviar los datos en dos paquetes de 4 bits al display.

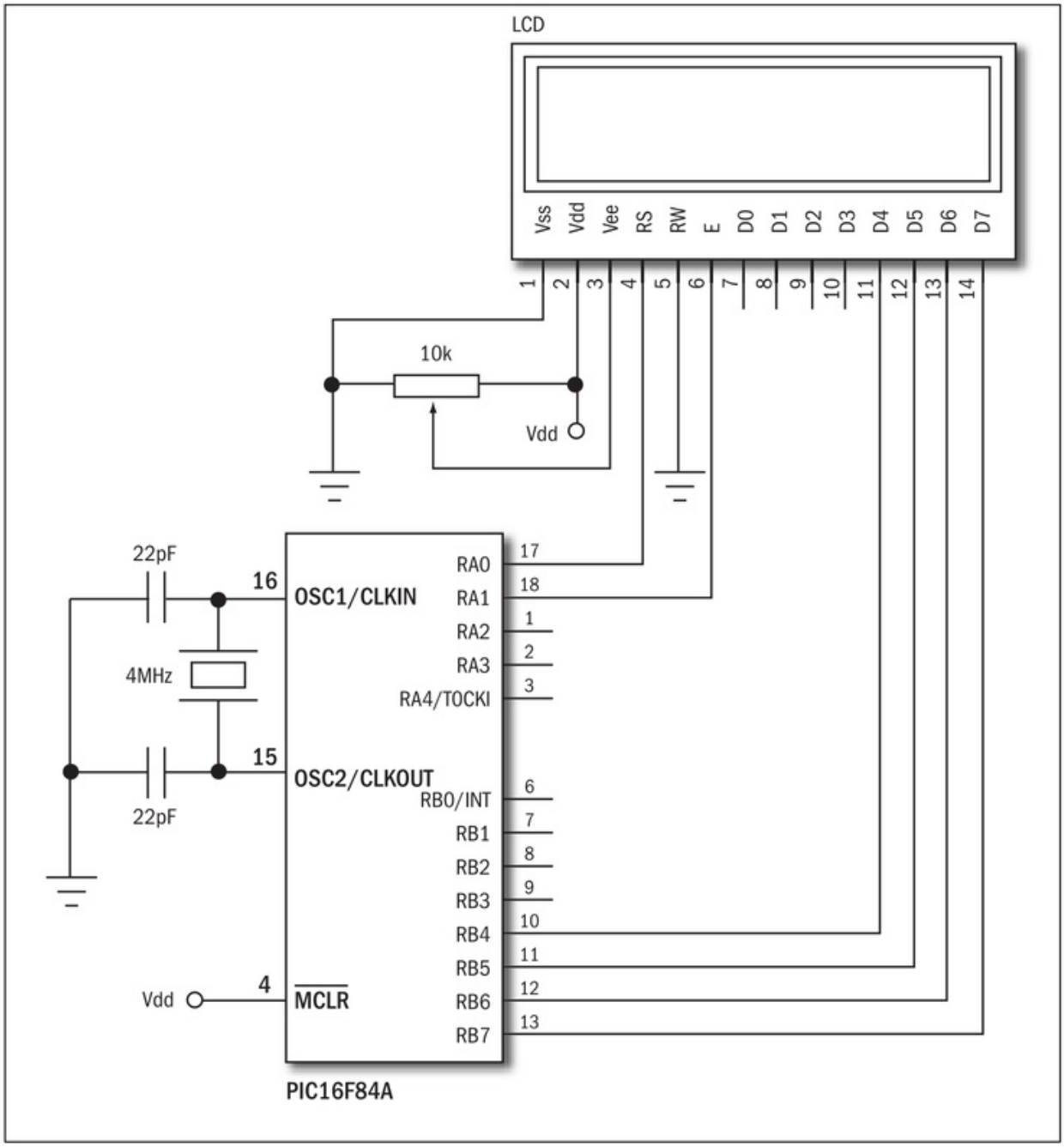


***Figura 14.** Conexión típica del display al PIC16F84A con un bus de sólo 4 bits sin usar el busy flag.*

En el diagrama de la **Figura 14** observamos cómo es posible la conexión del display con sólo 4 líneas para el bus de datos. En ella tenemos también conectado el pin R/W a tierra, sin la posibilidad de usar el busy flag. El modo de 4 bits es más lento, ya que hay que enviar dos paquetes de 4 bits al display para completar

un dato o comando. El modo de conexión de 8 bits es más eficiente ya que permite una mayor velocidad. Pero la principal desventaja del uso de 8 bits es que no nos quedan muchas líneas libres en el PIC16F84A.

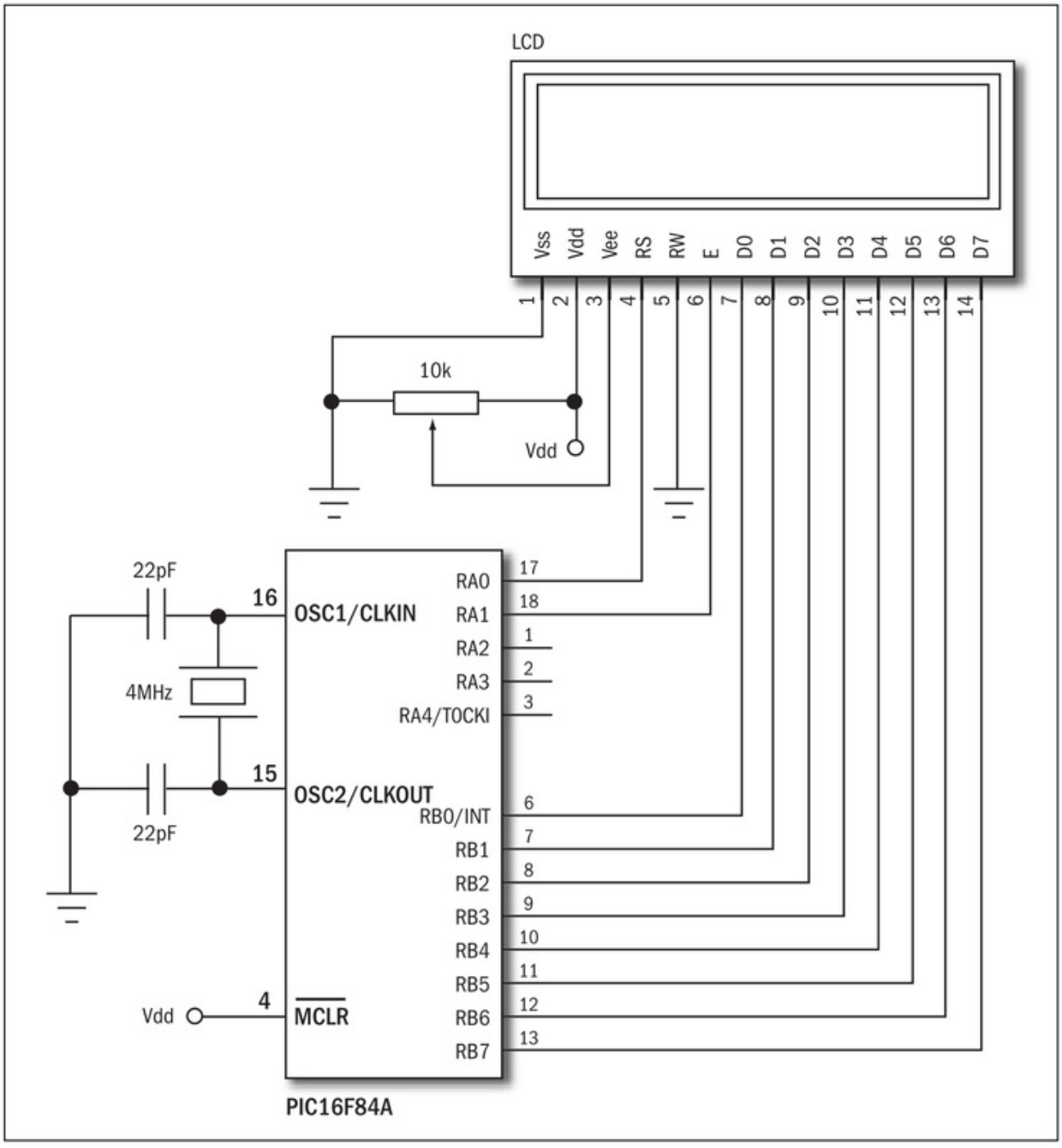


***Figura 15.** Conexión típica del display al PIC16F84A con un bus completo de 8 bits y sin uso del busy flag.*

Inicialización del display

Antes de poder comenzar a enviar datos o comandos al display, se debe llevar a cabo una rutina de inicialización para asegurar el correcto funcionamiento. La inicialización es diferente para la conexión de 4 ó de 8 bits.

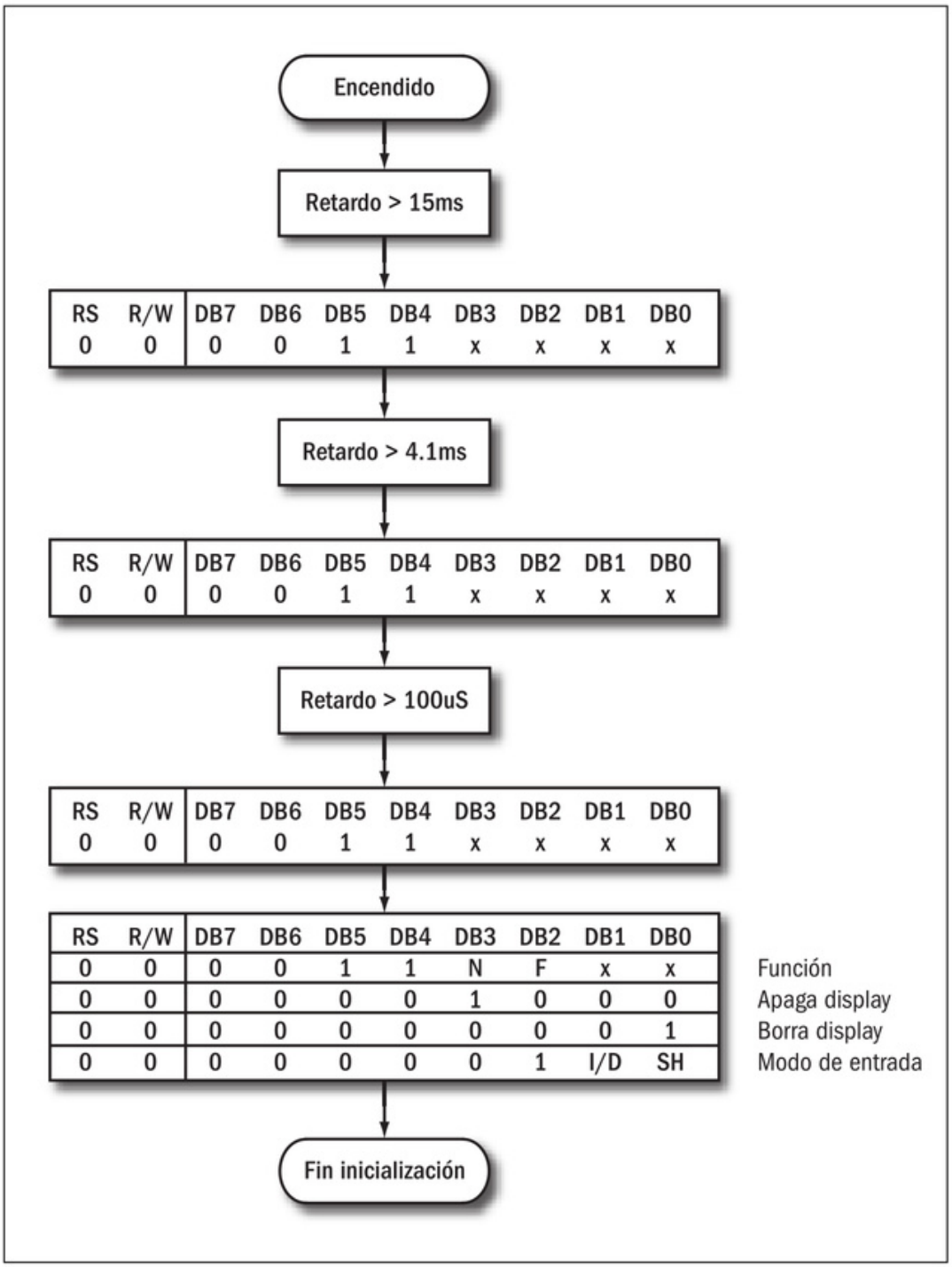


Figura 16. *Proceso de inicialización del display con modo de 8 bits.*

III VENTAJAS DEL BUSY FLAG

La principal ventaja que tenemos si utilizamos el bit de busy flag es que si monitoreamos el estado de este bit, podemos enviar otro dato en cuanto nos lo indique. Si colocamos retardos para no usarlo, puede que los procesos internos terminen mucho antes que el retardo, y eso hace nuestro programa un poco más lento.

Los procesos de inicialización son informados por el fabricante y debemos seguirlos para un correcto funcionamiento del display. Debemos incluir la rutina de inicialización al comienzo de cualquiera de nuestros programas donde usemos el display. Las **x** en los diagramas de flujo de inicialización indican que el estado de esos bits no afecta y es indistinto.

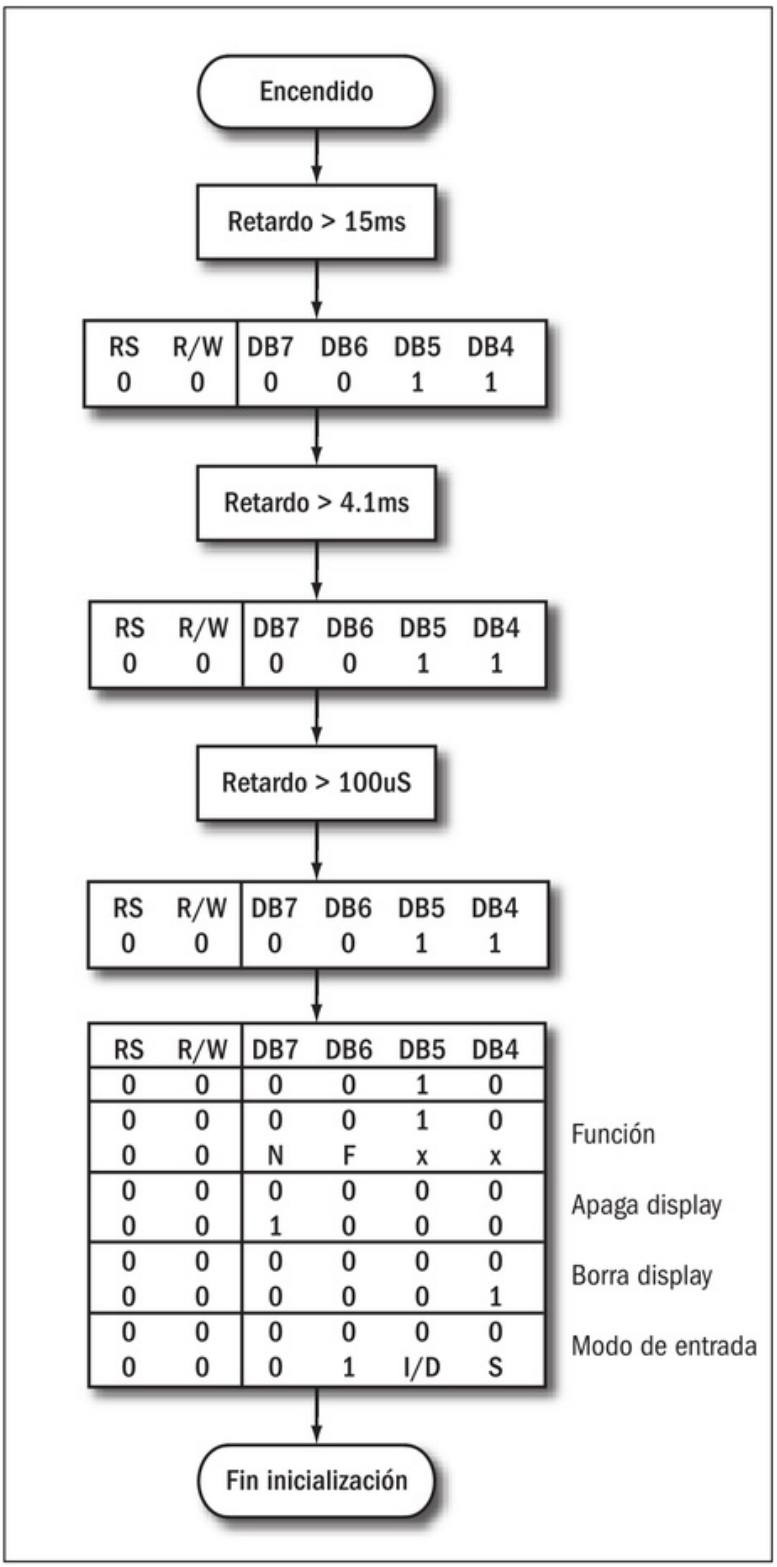


Figura 17. *Proceso de inicialización del display con modo de 4 bits.*

Librerías de control de LCD

Una vez que hemos estudiado cómo funciona el display LCD, es bueno tener nuestras librerías para incluirlas en todos los programas en donde hagamos uso del display. En el sitio **www.redusers.com** podemos descargar las librerías tanto para 8 como para 4 bits (**LCD8BITS.INC** y **LCD4BITS.INC**) y están suficientemente comentadas para que podamos estudiar el funcionamiento de cada una de ellas para comprenderlas.

Uso de las librerías

Ambas librerías usan los mismos nombres para las constantes, ya que se supone que no usaremos las dos en un mismo programa. Cada una contiene las rutinas de inicialización, de envío de caracteres y de comandos. En ambas hemos optado por no usar el bit de busy flag, así que la terminal R/W del display se conecta a tierra, tal como en los diagramas de las **Figuras 14** y **15**. En ambas librerías los procesos tienen el mismo nombre. Para el proceso de inicialización se debe llamar a la subrutina **LCD_inicializa**:

```
call          LCD_inicializa
```

Durante el proceso de inicialización se configura el display de la siguiente forma:

Function set: 8 ó 4 bits en cada caso, pantalla de 2 líneas y caracteres de 5x7.
Display control: se enciende el display, sin cursor visible y sin parpadeo del cursor.
Entry mode set: con incremento de la posición del cursor con cada escritura en la DDRAM y sin desplazamiento de la pantalla.

Éstas son las opciones de configuración predeterminadas, pero podemos modificarlas si lo necesitamos, o enviar desde el programa principal un comando para cambiarlas en el momento en que se requiera hacerlo.

Para enviar un comando, colocamos el valor del comando en el registro W y llamamos a la subrutina **LCD_comando**. Por ejemplo:

```
movlw   b'00001110'
call    LCD_comando
```

En este ejemplo enviamos el comando **Display control** con diferentes opciones que en el proceso de inicialización. En este caso se visualiza el cursor.

La mayoría de los comandos requiere de un tiempo de espera mayor a 40 us, excepto los de **Clear display** y **Return home**, que necesitan un tiempo mayor a 1.64 ms

(ver la **Tabla 2**). Es por eso que no deben llamarse con la subrutina **LCD_comando**, sino que tienen su subrutina especial que cumple con el tiempo de espera adecuado:

```
call        borra_display
call        LCD_origen1
```

La primera subrutina permite borrar la pantalla y colocar el cursor en la primera posición de la línea 1 de ésta. La segunda envía la posición del cursor a la primera posición de la línea 1, pero sin borrar el contenido de la DDRAM.

También tenemos la subrutina:

```
call        LCD_origen2
```

Envía el cursor a la primera posición de la línea 2 del display. Para enviar un carácter al display se coloca el código en el registro W y se llama a la subrutina **LCD_caracter**:

```
movlw   'A'
call    LCD_caracter
```

Los caracteres pueden darse mediante código ASCII, como en el ejemplo anterior, pero en la tabla de caracteres de la CGROM podemos ver que de la posición 126_{10} (01111110_2) en adelante los caracteres ya no corresponden con el código ASCII, por lo tanto, para usarlos hay que poner directamente su código, ya sea en binario, decimal o hexadecimal. Por ejemplo, para mostrar el símbolo de la raíz cuadrada ponemos:

```
movlw   b'11101000'
call    LCD_caracter
```

III DESVENTAJAS DEL BUSY FLAG

Las desventajas de usar el busy flag en programas que utilicen el display LCD, son que usamos una línea más del PIC para poder leerlo. Además, si el display falla, puede que nunca se libere el busy flag, provocando que el programa quede estancado esperándolo y no funcionará más. Si no lo usamos, el programa seguirá funcionando aun si el display no funciona.

Como ejemplo proponemos el siguiente programa que coloca exto en el display. Se usa la librería para manejo de LCD de 8 bits **LCD8BITS.INC**, pero puede ser también la de 4 bits, con la conexión adecuada del display al PIC16F84A. Nos sirve para conocer la forma de enviar caracteres al display, para mostrar un mensaje en él:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      ENDC

      ORG             0x00

      call            LCD_inicializa
inicio
      movlw           'P'
      call            LCD_caracter
      movlw           'I'
      call            LCD_caracter
      movlw           'C'
      call            LCD_caracter
      movlw           '1'
      call            LCD_caracter
      movlw           '6'
      call            LCD_caracter
      movlw           'F'
      call            LCD_caracter
      movlw           '8'
      call            LCD_caracter
      movlw           '4'
      call            LCD_caracter
      movlw           'A'

      sleep

      #INCLUDE        <LCD8BITS.INC>

      END
```

Mensajes en LCD

A continuación, veremos otra librería que nos permitirá mostrar mensajes en el display de una manera muy fácil y práctica para los programas. Mediante el uso de esta librería podremos mostrar mensajes de hasta 32 caracteres usando las dos líneas de la pantalla. En el sitio **www.redusers.com** podemos obtener esta librería llamada **LCDMENSAJES.INC** para su estudio y su uso.

Uso de la librería de mensajes en LCD

Para usar esta librería debemos agregarla en el programa principal, además de adicionar la librería para manejo de display, ya sea para 4 u 8 bits. Los mensajes debemos colocarlos en tablas como en este ejemplo:

```
mensajes
     addwf          PCL, F
mensaje_1
     DT             "Mensaje 1", 00h
mensaje_2
     DT             "Mensaje 2", 00h
     .
     .
     .
mensaje_x

     DT             "mensaje x", 00h
```

Debemos asegurarnos que las tablas queden dentro de los primeros 256 bytes de la memoria de programa y terminar cada mensaje con 00h, para indicar su final. Estas dos condiciones son muy importantes. Luego de hacer esto debemos colocar en el registro W la dirección (la etiqueta) del mensaje deseado y llamar a la subrutina **LCD_mensaje**, como por ejemplo:

```
     movlw          mensaje_2      ;Dirección del mensaje 2

     call           LCD_mensaje
```

De esta forma, la librería se encargará de borrar el display y mostrar el mensaje indicado. Es importante no sobrepasar los 32 caracteres, ya que si lo hacemos el mensaje no se verá completo. En la **Figura 18** tenemos el diagrama de flujo de la librería, que junto con el código fuente nos ayudará a entender su funcionamiento.

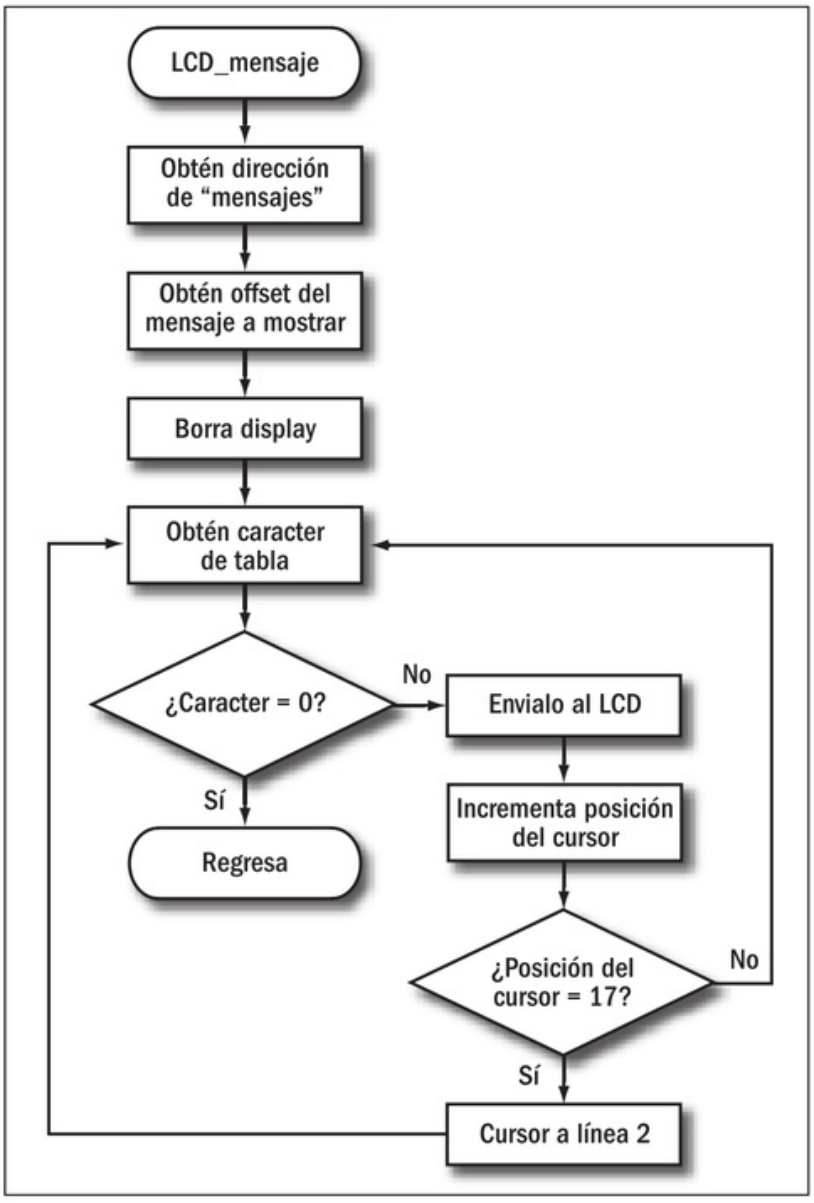


Figura 18. *Proceso para mostrar mensajes de hasta 32 caracteres mediante tablas.*

Como ejemplo del uso de la librería para mostrar mensajes en el display, podemos descargar de **www.redusers.com** el archivo fuente **Mensajes.asm**. Además, contamos con el archivo **Mensajes.hex**, para grabarlo en el PIC16F84A y comprobar su funcionamiento. Se usa la conexión de 8 bits, tal como en la **Figura 15**.

RESUMEN

Hemos aprendido el uso de displays para mostrar información desde el microcontrolador al exterior. Hemos estudiado el funcionamiento y el uso de displays de 7 segmentos y la forma de multiplexarlos para poder conectar varios de ellos al PIC16F84A. Además, desarrollamos el uso de un display LCD para mostrar mensajes de texto, números y símbolos.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es un display de 7 segmentos?

2 ¿Cuáles son los dos tipos de displays de 7 segmentos, según su conexión interna?

3 ¿En qué tipo de display de 7 segmentos se encienden los segmentos con un uno?

4 ¿Para qué sirve multiplexar los displays de 7 segmentos?

5 ¿Qué significa LCD?

6 ¿Para qué sirve el pin Vee en el display LCD?

7 ¿Para que sirve el pin RS en el display LCD?

8 El bus de datos del display se pone en alta impedancia cuando en el pin E hay un _____.

9 Para borrar el display hay que enviarle en binario el comando _____.

10 ¿Para qué sirve el busy flag?

PRÁCTICAS

1 Escriba el programa que está al final de la sección Librerías de control de LCD en MPLAB. Ensámblelo, y grábelo en el PIC16F84A para comprobar su funcionamiento.

2 Diseñe un programa que muestre un mensaje de hasta 16 caracteres en la segunda línea del display.

3 Diseñe un programa que muestre un mensaje menor a 16 caracteres en el display, de tal forma que cada carácter aparezca secuencialmente con un retardo de aproximadamente 200 ms con respecto al otro. Cuando el mensaje esté completo, debe esperar 2 segundos, borrar la pantalla y comenzar nuevamente. Sugerencia: puede usar también el programa del punto 1 como base.

4 Modifique las librerías LCD8BITS.INC y LCD4BITS.INC para que ahora, en lugar de retardos, se haga uso del busy flag. Recuerde que debe modificar la conexión del pin R/W. Guárdelas con nombres distintos para usarlas cuando las necesite.

5 Conecte dos pulsadores a dos pines libres del Puerto A. Diseñe un programa que muestre el nombre del pin del pulsador presionado, sólo mientras éste se mantiene presionado.

El Timer 0

El PIC16F84A tiene una función que resulta muy útil para ciertos proyectos. Esta función es un timer que puede servir como contador de eventos externos, o para hacer temporizaciones. En este capítulo analizaremos su estructura, su funcionamiento y sus aplicaciones.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

EL TIMER 0 DEL PIC16F84A

El PIC16F84A cuenta con un temporizador/contador denominado **Timer 0**, que es básicamente un **contador binario ascendente de 8 bits**, que se incrementa de dos formas posibles: una de ellas depende del oscilador del sistema y la otra de la señal entrante en el pin RA4/T0CKI. Precisamente de ahí el nombre **T0CKI**, que significa **Timer 0 Clock In**. Cuando se usa con la señal externa se puede incrementar ya sea con flanco de bajada o subida, según lo decida el usuario.

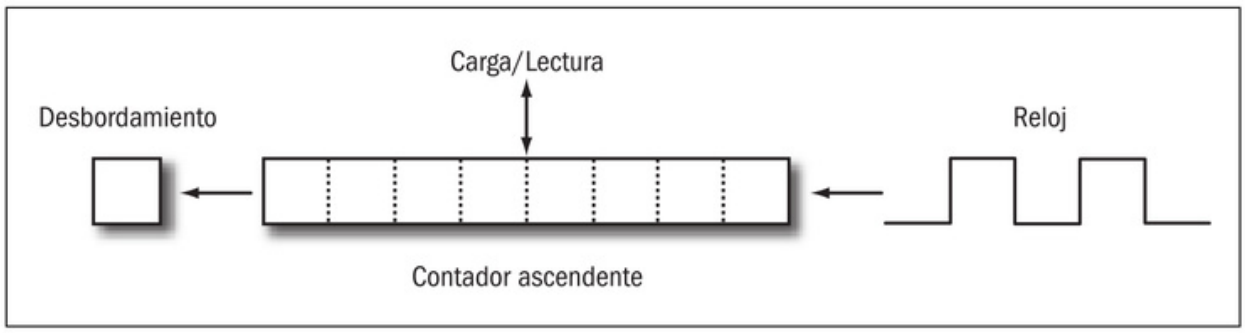


***Figura 1.** Estructura general de un temporizador/contador de 8 bits.*

Cuando el Timer 0 es incrementado con un flanco, ya sea de bajada o de subida, en el pin RA4/T0CKI se usa típicamente como **contador de eventos externos**, y cuando se incrementa con la señal de reloj, se usa como **temporizador**. Se puede escribir o leer en él en cualquier momento que lo necesitemos. Como el Timer 0 es un simple contador de 8 bits, tiene un límite de cuenta y cuando llega a él y se incrementa una vez más, se desborda y comienza a contar desde 0 nuevamente.

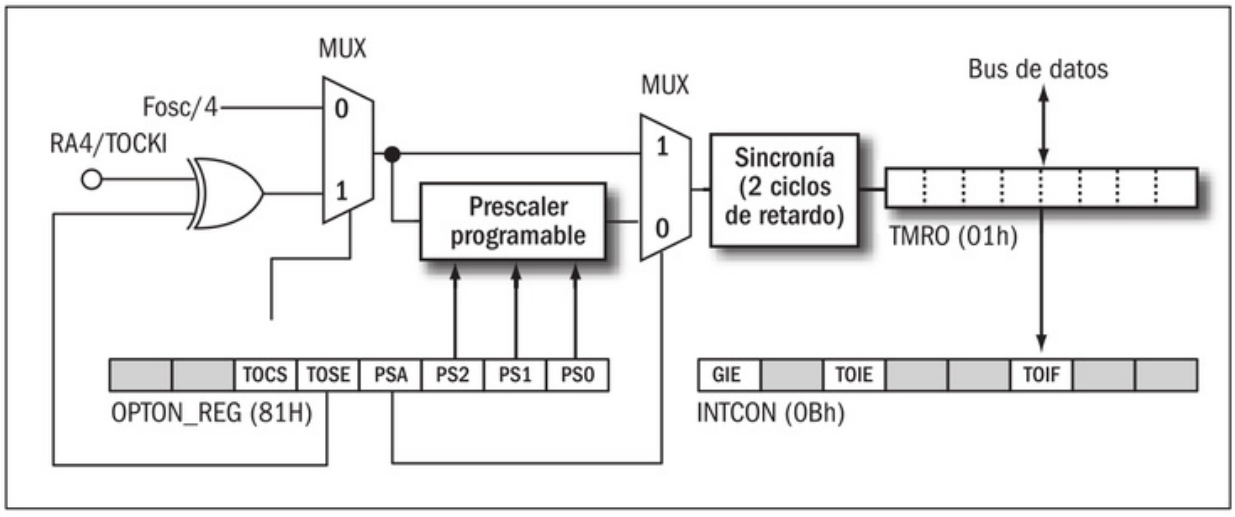


***Figura 2.** La estructura interna del Timer 0 del PIC16F84A, se puede abreviar como **TMR0**.*

Si bien nos puede parecer que el Timer 0 es una función muy sencilla, pronto descubriremos que puede sernos de gran utilidad en muchos proyectos.

El prescaler (divisor de frecuencia)

El Timer 0 sirve como contador o temporizador, aunque ya hemos visto que podemos medir o lograr tiempos específicos si usamos retardos. También podemos emplear el TMR0 para esta tarea. La ventaja de hacerlo es que el timer está contando automáticamente, mientras el PIC puede aprovechar para realizar otras tareas, cosa que con los retardos no es posible, o al menos no siempre. El TMR0 cuenta con un **divisor de frecuencia** o **prescaler programable** para dividir la frecuencia con la que es incrementado, y así lograr tiempos de conteo más largos.

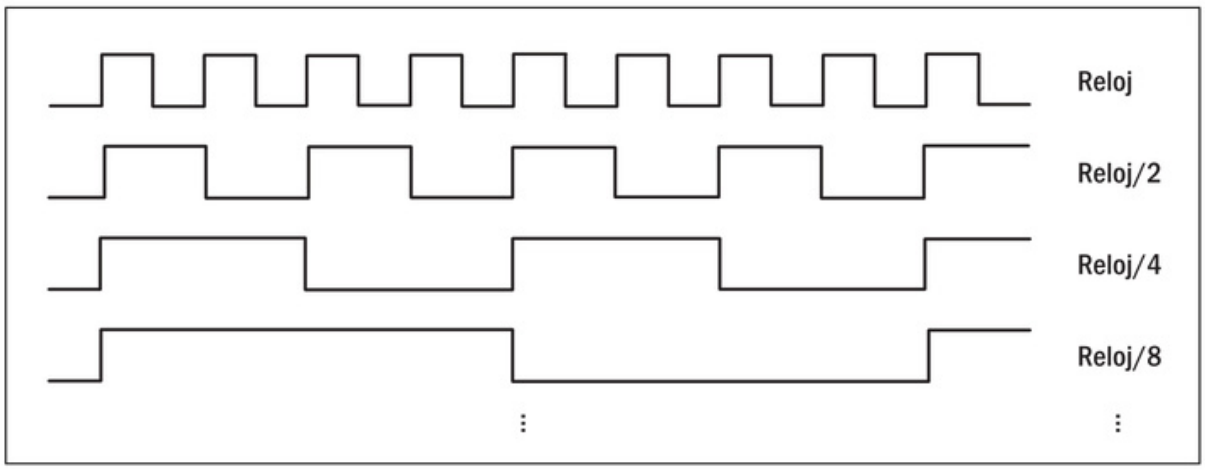


***Figura 3.** El prescaler divide la frecuencia del reloj en 2 cada vez, para lograr tiempos mayores.*

El PIC16F84A realmente tiene otro temporizador llamado **WDT** o *Watch dog timer*, aunque éste tiene un propósito diferente al Timer 0, por lo que hablaremos de él más adelante. El prescaler puede ser asignado a alguno de estos dos temporizadores. Luego, en la **Tabla 2**, veremos los rangos en que puede ser dividida la frecuencia.

LOS REGISTROS RELACIONADOS CON EL TMR0

El uso del TMR0 y su control se realiza mediante algunos registros del área SFR en la memoria de datos. Para comprender mejor su funcionamiento, a continuación veremos cuáles son los registros relacionados con el Timer 0 del PIC16F84A.

El registro TMR0

El registro llamado **TMR0** es el registro perteneciente al Timer 0. En él se reflejará en todo momento la cuenta, es decir, el valor que tenga todo el tiempo el Timer 0. Como pudimos apreciar en la **Figura 2**, el registro TMR0 está conectado al bus de datos, por lo que podemos escribir en él o leerlo cuando necesitemos.

El registro llamado TMR0 está ubicado en la dirección 01h del banco 0 del área SFR de la memoria de datos. Cuando usamos el Timer 0 como temporizador y cargamos un nuevo valor en él, el siguiente incremento se retrasará dos ciclos de reloj.

El registro OPTION

El registro **OPTION** que se encuentra ubicado en la dirección 81h en el banco 1 de la memoria de datos tiene varios bits que sirven de configuración y control del TMR0. En la **Tabla 1** tenemos la estructura de este registro.

Bit 7	Bit 6	Bit 5	Bit4	Bit3	Bit 2	Bit 1	Bit 0
RBPU'	INTEDG	**T0CS**	**T0SE**	**PSA**	**PS2**	**PS1**	**PS0**

***Tabla 1.** Estructura del registro OPTION.*

La función de los bits de este registro es:

PS2, **PS1**, **PS0** (**Prescaler rate select bits**): estos bits sirven para definir el rango del prescaler o divisor de frecuencia. En la **Tabla 2** vemos en detalle los rangos del prescaler para no cometer errores.

PS2 PS1 PS0	TMR0	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

***Tabla 2.** Asignación de rangos del prescaler.*

PSA (**Prescaler assignment bit**): determina a quién se le asignará el prescaler:

III SIN PRESCALER EN TMR0

En la **Tabla 2** podemos observar que si colocamos los bits PS2, PS1 y PS0 a 0, lo que obtenemos es un prescaler de 1:2 para el TMR0. Pero si no queremos dividir la frecuencia a la entrada del TMR0, debemos asignarle el prescaler al WDT y de esa forma no tendrá ningún efecto en el TMR0 y la frecuencia del reloj no será dividida.

PSA=0 El prescaler se asigna al TMR0
PSA=1 El prescaler se asigna al WDT

T0SE (**TMR0 Source Edge bit**): si estamos usando el TMR0 como contador, es decir, controlado por la señal en el pin RA4/T0CKI, con este bit se elige si el incremento en él se hará con flanco descendente o ascendente:

T0SE=0 El TMR0 se incrementa con cada flanco de subida de la señal en T0CKI
T0SE=1 El TMR0 se incrementa con cada flanco de bajada de la señal en T0CKI

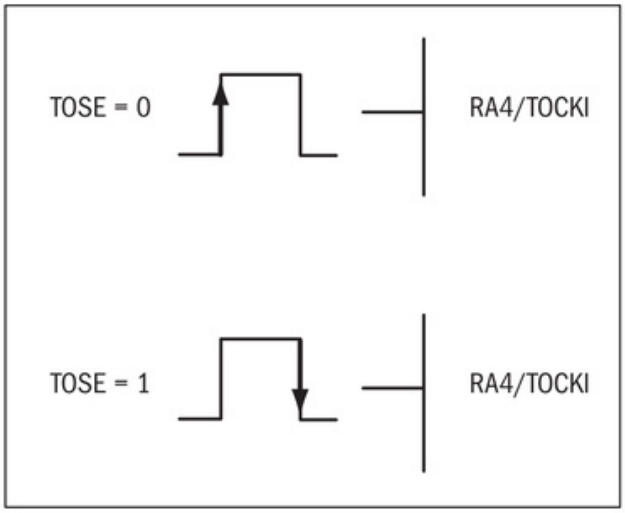


***Figura 4.** Con el bit T0SE se elige el flanco para el incremento del TMR0.*

T0CS (**TMR0 Clock source select bit**): con este bit se elige la forma con la que el TMR0 se incrementará:

T0CS=0 El TMR0 se incrementa con la señal de reloj interna
T0CS=1 El TMR0 se incrementa con cada flanco de la señal en el pin RA4/T0CKI

Los bits 6 y 7 del registro **OPTION_REG** son usados para otras tareas que no viene al caso detallar ahora, ya que las analizaremos posteriormente.

TIMERS EN OTROS PICS

El PIC16F84A sólo tiene un timer de 8 bits, denominado Timer 0. Muchos otros microcontroladores PIC tienen más de un timer, además de contar con diferentes características. Por ejemplo, algunos de esos timers son de 16 bits, tienen postscaler (además de prescaler), etcétera. Un ejemplo es el PIC16F628A que tiene 3 timers, denominados Timer 0, Timer 1 y Timer 2.

El registro INTCON

Este registro se usa principalmente para las interrupciones, de las cuales no nos explayaremos aquí ya que le dedicaremos el **Capítulo 10** para su desarrollo. Por ahora sólo nos interesa conocer un bit de este registro que está en la dirección 0Bh del banco 0 y se repite en la 8Bh del banco 1.

Bit 7	Bit 6	Bit 5	Bit4	Bit3	Bit 2	Bit 1	Bit 0
GIE	EEIE	**T0IE**	INTE	RBIE	**T0IF**	INTF	RBIF

***Tabla 3.** El registro INTCON.*

Este registro, en realidad, tiene dos bits asociados al TMR0, pero por ahora sólo estudiaremos uno de ellos:

T0IF (**TMR0 Overflow Interrupt flag bit**): este bit nos indica el estado de desbordamiento del TMR0:

T0IF=0 Indica que el TMR0 no se ha desbordado
T0IF=1 Indica que el TMR0 se ha desbordado

Cuando el TMR0 se desborda, entonces que se activa el bit T0IF, este bit debe ser borrado por software, es decir, no se borra automáticamente, sino que habrá que ponerlo a 0 mediante la instrucción **bcf** cuando sea necesario. Es importante tener en cuenta esto para que nuestros programas funcionen correctamente.

EL TIMER 0 COMO CONTADOR

Veamos el uso del Timer 0 como contador. Para ello utilizaremos el circuito mostrado en la **Figura 5**, de modo que al presionar un pulsador conectado al pin RA4/T0CKI un contador se incrementará con cada pulso y estaremos comprobando cómo el TMR0 aumenta con cada flanco de bajada en la entrada.

{} EL PRESCALER

Un prescaler o divisor de frecuencia es normalmente un contador binario. Como sabemos, un contador binario divide la frecuencia en dos en cada una de sus salidas al funcionar. Si llevamos nuestra señal de reloj al contador y tomamos las salidas de éste, en cada una habrá una frecuencia dividida en 2, 4, 8, 16, etcétera, y esto se aprovecha como prescaler.

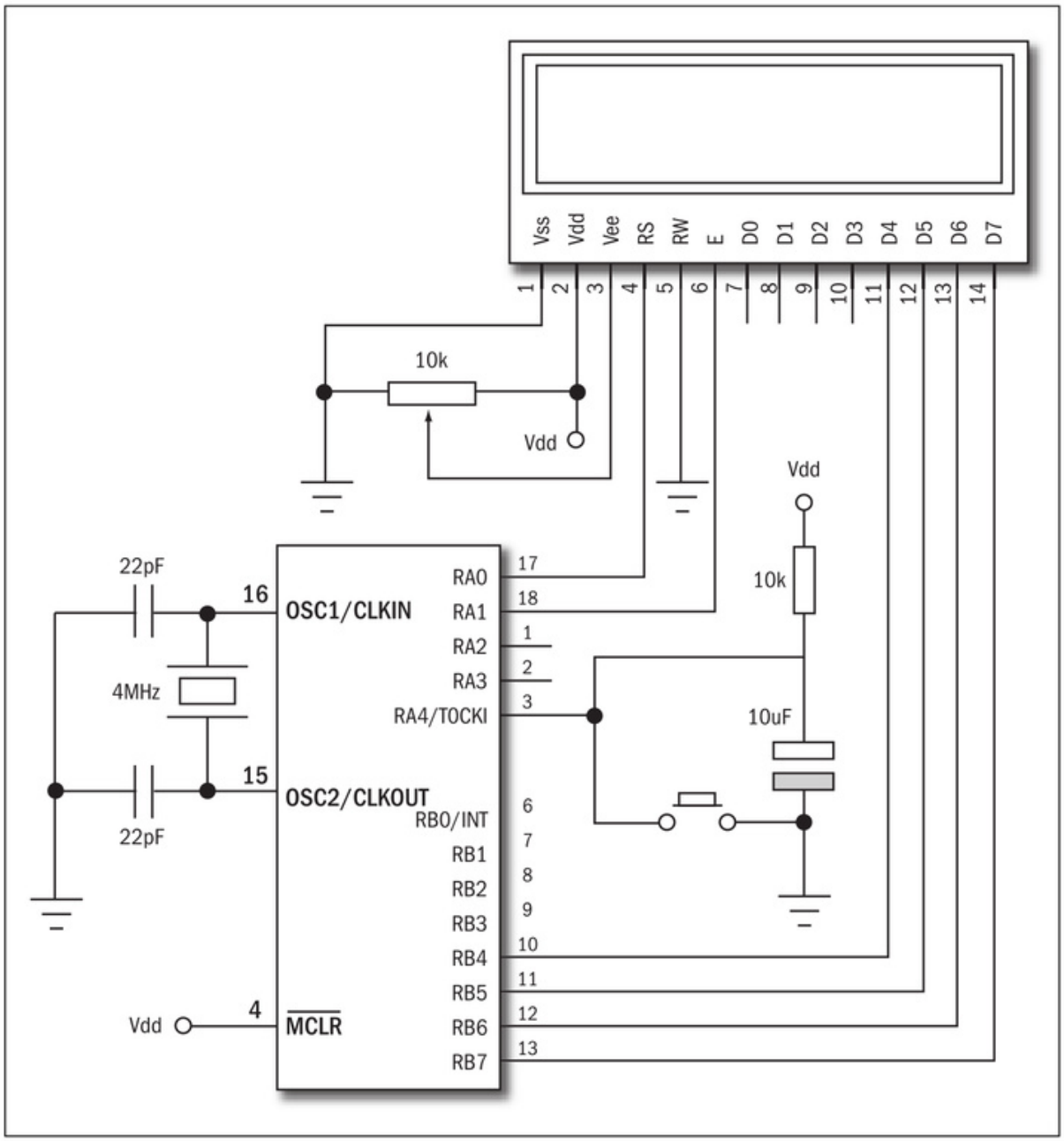


Figura 5. *Circuito de ejemplo para el programa* ***TMR0 contador.asm.***

Lo que estamos haciendo, en verdad, es contar los pulsos aplicados en RA4/T0CKI. El código del programa será el siguiente:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

CBLOCK 0x0C
ENDC
```

```
     ORG            0x00

     call           LCD_inicializa

     bsf            STATUS,RP0          ;Acceso al Banco 1
     movlw          b'00111000'
     movwf          OPTION_REG
     bcf            STATUS,RP0          ;Acceso al Banco 0


     clrf           TMR0                ;Borra el Timer0

inicio
     movf           TMR0, W             ;Lee el valor del TMR0
     call           BINaBCD             ;Lo convierte en BCD
     call           LCD_decimal         ;Subrutina para mostrar el valor
en decimal
     goto           inicio              ;Genera un bucle que lee el TMR0
constantemente

LCD_decimal
     call           LCD_origen1         ;Envía el cursor del LCD al
inicio de la línea 1

     movf           BCDcentenas, W      ;Pasa las centenas a W
     btfsc          STATUS, Z           ;¿Centenas = 0?
     goto           NOcentenas          ;Si, salta
     addlw          '0'                 ;No, suma "cero" para obtener el
valor ASCII
     goto           LCDcentenas
NOcentenas
     movlw          ' '                 ;Coloca un espacio si las
centenas valen cero
LCDcentenas
     call           LCD_caracter        ;Envíalo al LCD
decenas
     movlw          d'10'
     subwf          TMR0, W
     btfss          STATUS, C           ;¿Valor del TMR0 < 10?
     goto           NOdecenas           ;Si, salta
```

```
     movf          BCDdecenas, W        ;No, pasa las decenas a W
     addlw         '0'                  ;Suma "cero" para obtener el
     valor ASCII
     goto          LCDdecenas
NOdecenas
     movlw         ' '                  ;Coloca un espacio si las decenas
valen cero
LCDdecenas
     call          LCD_caracter         ;Envíalo al LCD
unidades
     movf          BCDunidades, W       ;Pasa las unidades a W
     addlw   '0'                        ;Suma "cero" para obtener el
     valor ASCII
     call          LCD_caracter         ;Envíalo al LCD
     return

     #INCLUDE      <BINABCD.INC>
     #INCLUDE      <LCD4BITS.INC>

     END
```

Podemos descargar el código fuente **TMR0 contador.asm** donde hay más comentarios del funcionamiento y, él código máquina **TMR0 contador.hex**, desde **www.redusers.com** para poder grabarlo en el PIC16F84A y comprobarlo.

Observemos cómo se configura el TMR0 de manera que la fuente de pulsos para su incremento sea externa, es decir, con la señal del pin T0CKI colocando un 1 en el bit T0CS. Con el bit T0SE a 1 se indica que el incremento se hará con un flanco de bajada. El prescaler se le asigna al WDT, por lo que no tendrá ningún efecto en el TMR0, ni los bits de asignación PS2, PS1 y PS0, por lo que no importa su valor. El WDT está desactivado, como lo hemos hecho siempre hasta ahora. El bloque principal

¿POR QUÉ OPTION_REG?

Dentro de las instrucciones de los microcontroladores PIC existe una llamada **option**, la cual ya no es recomendada para su uso, pero sigue siendo reconocida por el ensamblador. Es por eso que en el archivo de definiciones **P16F84A.INC** el registro **OPTION** es definido como **OPTION_REG** para evitar que el ensamblador lo confunda con la instrucción.

del programa es muy sencillo, sólo envía constantemente el valor contenido en TMR0 al display para que sea visualizado. La subrutina **LCD_decimal** convierte el valor de TMR0 a decimal, de tal forma que en el display podamos observarlo. Recomendamos estudiar bien el funcionamiento de esta subrutina en el código fuente.

FRECUENCÍMETRO

Si usamos el TMR0 como contador, lo primero que se nos ocurriría contar es precisamente el número de pulsos de una señal. Si hacemos esto durante un segundo, habremos medido la frecuencia de dicha señal y con esto podremos escribir un programa que funcione como un **frecuencímetro digital**.

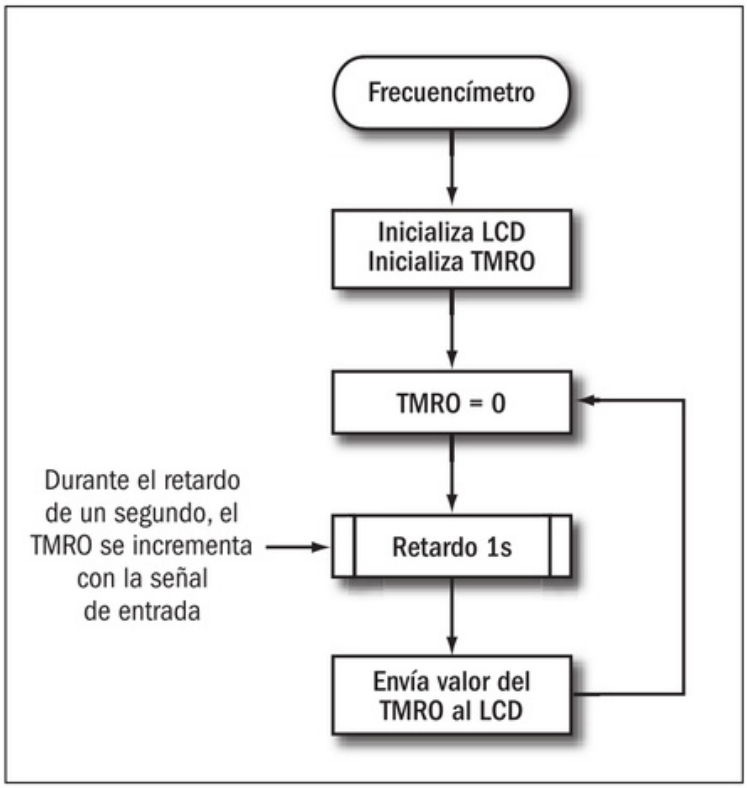


***Figura 6.** El método base para medir frecuencia con un microcontrolador PIC. Simplemente lee el valor del TMR0 después de un segundo.*

El método es sumamente sencillo, pero la desventaja es que el TMR0 del PIC16F84A es de sólo 8 bits, con lo cual únicamente podríamos medir frecuencias de hasta 255 Hz ya que si el TMR0 se desborda (cuenta más de 255) iniciará de 0 nuevamente. Para resolver este inconveniente podemos usar un registro auxiliar para que cada vez que el TMR0 se desborde, éste registro auxiliar se incremente y de esta forma podamos contar más allá de los 255 que limitan un registro de 8 bits. Veamos un ejemplo de un frecuencímetro que mida señales de hasta 10 KHz.

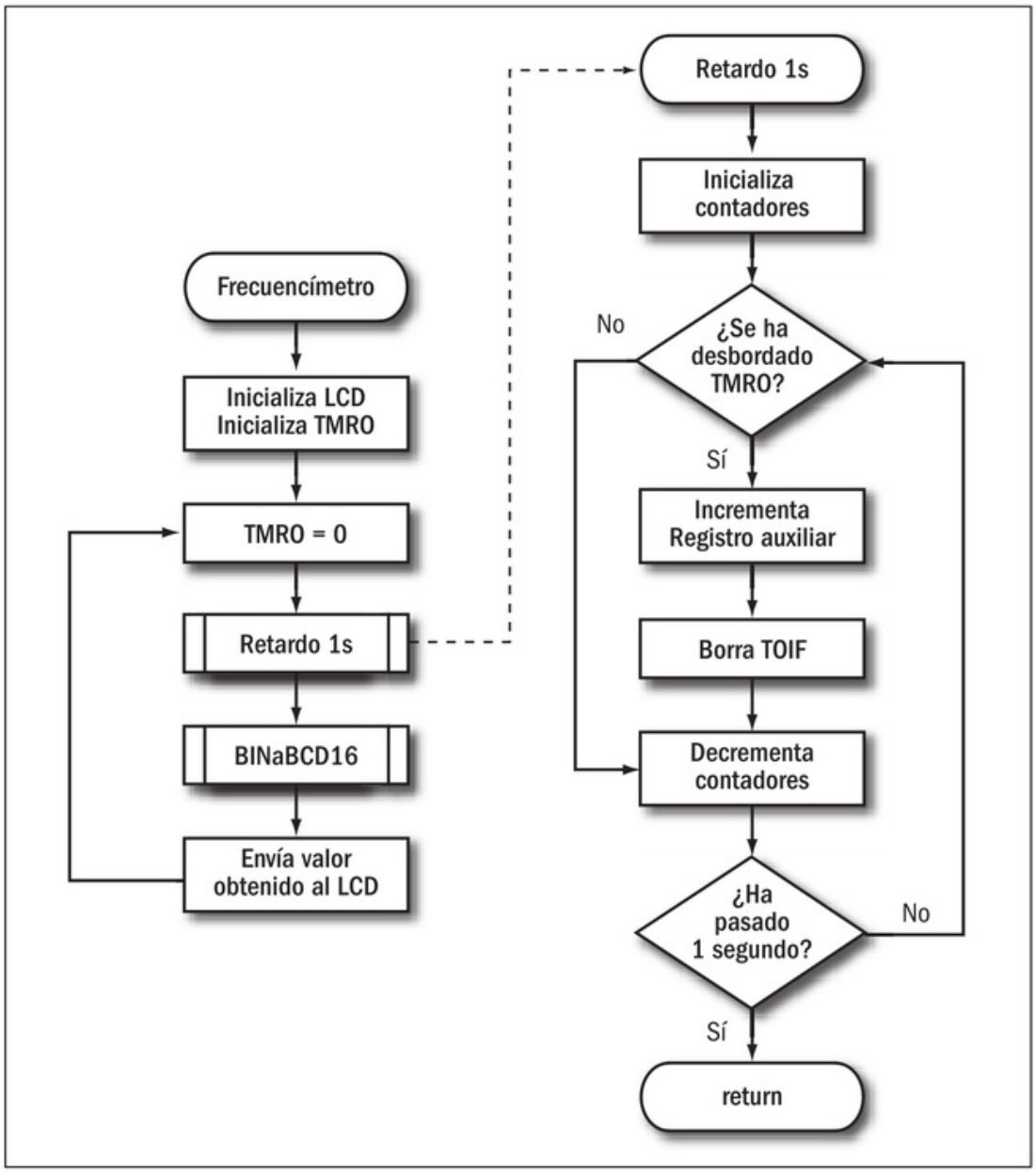


Figura 7. *El segundo método básico para medir frecuencias más allá de 255 Hz.*

Una librería más para convertir de binario a BCD

Para medir frecuencias más altas no sólo debemos poder contar los pulsos en el pin RA4/T0CKI, sino que debemos poder representar también esa cuenta en el display. Recordemos que la librería **BINABCD.INC** que estudiamos antes sólo puede convertir números de tres dígitos y hasta 255, ya que ese es el límite para un registro. Ahora necesitaremos convertir un número de más de 8 bits. Para ello utilizaremos dos registros, para poder tener un número de 16 bits con el cual podremos contar hasta 65535_{10} y de esa forma representar números en BCD mucho más allá de 255. Podemos descargar la librería **BINABCD16.INC** de **www.redusers.com** para utilizarla en nuestro frecuencímetro. Es conveniente que la abramos en MPLAB para estudiar su uso, el cual se indica con todo detalle en los comentarios de la misma librería.

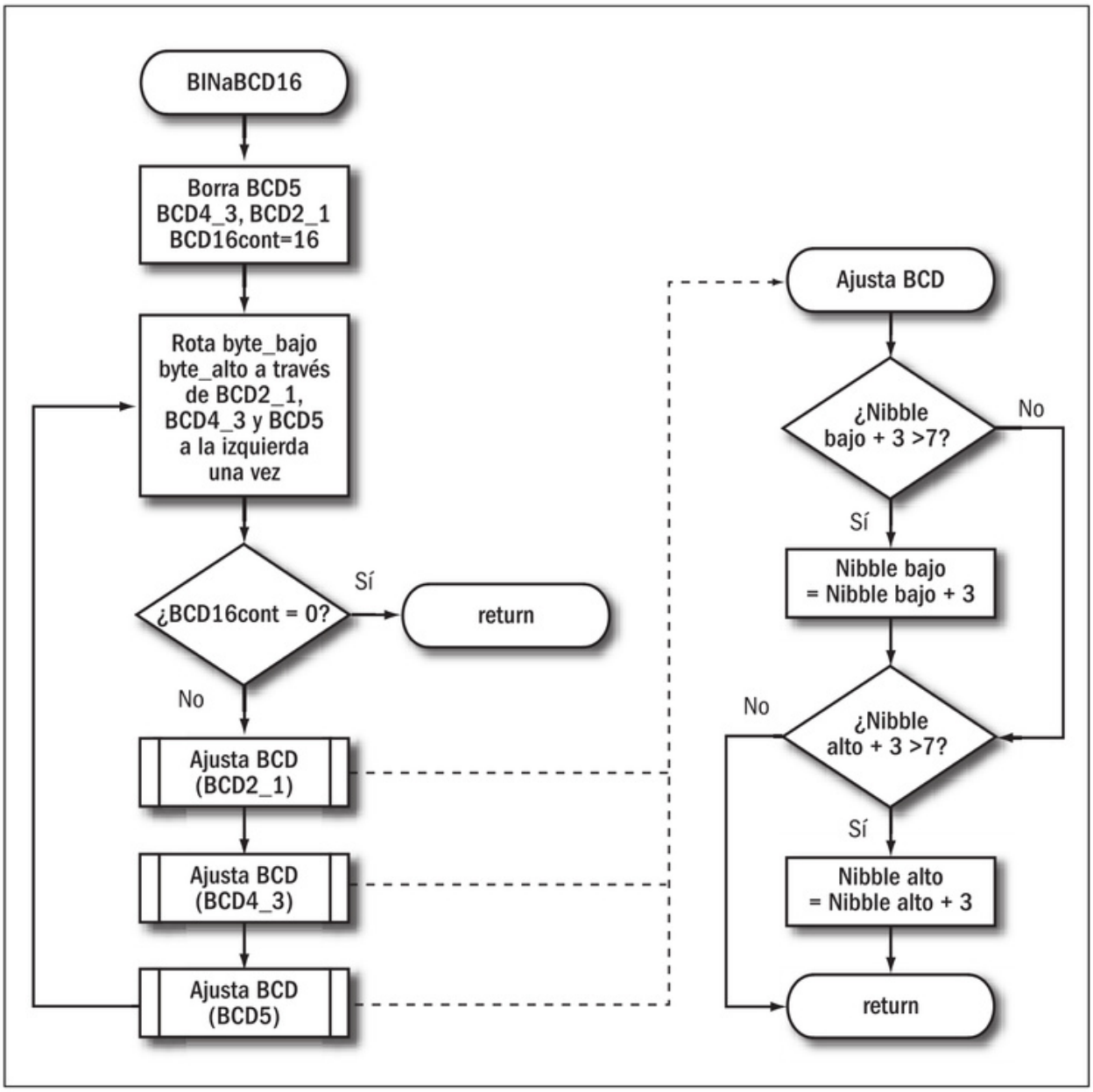


Figura 8. *Método de conversión de binario a BCD de 16 bits.*

De esta forma tenemos ya las bases para continuar con nuestro frecuencímetro, en el cual, además de la librería para convertir de binario a BCD de 16 bits, también usaremos la librería de manejo de LCD, ya sea la de 8 ó 4 bits.
El código fuente es el siguiente:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

CBLOCK 0x0C
```

```
      cont_1, cont_2, cont_3, LCDaux
      ENDC

      ORG             0x00

      call            LCD_inicializa ;Inicializa el LCD

      bsf             STATUS,RP0              ;Acceso al Banco 1
      movlw           b'00101000'
      movwf           OPTION_REG              ;TMR0 como contador, sin prescaler
      bcf             STATUS,RP0              ;Acceso al Banco 0

;Programa principal:

      movlw           b'10000101'
      call            LCD_comando             ;Envía el cursor a la posición 5
      movlw           'H'
      call            LCD_caracter
      movlw           'z'
      call            LCD_caracter            ;Coloca la leyenda "Hz" en el LCD

inicio
      bcf             INTCON, 2               ;Borra la bandera de
desbordamiemto del TMR0
      clrf            byte_alto               ;Borra el byte alto
      clrf            TMR0                    ;Borra el TMR0
      call            retardo_1s              ;Retardo de 1 segundo para
efectuar la medición

      movf            TMR0, W
      movwf           byte_bajo               ;Pasa el valor del TMR0 a W
```

III UN FRECUENCÍMETRO DE MICROCHIP

Si queremos construir un frecuencímetro muy preciso y que mida altas frecuencias, podemos descargar la nota de aplicación **AN592 Frequency Counter Using PIC16C5x** de **www.microchip.com**, donde se describe un método para hacerlo. Así podremos medir frecuencias desde 50 Hz hasta 50M Hz. El método es para los PIC16C5x, pero se puede adaptar fácilmente al PIC16F84A.

```
     call          BINaBCD16          ;Obtiene los dígitos de la
medición en BCD

     call          LCD_origen1        ;Envía el cursor del LCD al
inicio de la línea 1

     movf          BCD5, F
     btfss         STATUS, Z          ;¿Dígito 5 > 0?
     goto          over_range         ;Si, sobre-rango
     clrf          LCDaux             ;Registro auxiliar para eliminar
los ceros a la izquierda
LCD4
     swapf         BCD4_3, W
     andlw         b'00001111'        ;Obtiene el dígito número 4
     btfss         STATUS, Z          ;¿Es cero?
     goto          enviaLCD4          ;No, envíalo al LCD
     call          espacioLCD         ;Si, envía un espacio
     bsf           LCDaux, 0          ;Coloca una bandera de cuarto
dígito = 0
     goto          LCD3               ;Ve al siguiente dígito enviaLCD4
     call          numeroaLCD         ;Envía el dígito al LCD

LCD3
     movlw         b'00001111'        ;Obtiene el dígito número 3
     andwf         BCD4_3, W
     btfss         STATUS, Z          ;¿Es cero?
     goto          enviaLCD3          ;No, envíalo
     btfss         LCDaux, 0          ;Si, ¿El anterior fue cero?
     goto          enviaLCD3          ;No, envíalo
     call          espacioLCD         ;Si, envía un espacio
     bsf           LCDaux, 1          ;Coloca la bandera de tercer
```

DOS MÉTODOS PARA MEDIR FRECUENCIA

Para medir frecuencia con un PIC podemos utilizar dos métodos. Uno es el que ya vimos, contar el número de pulsos durante un segundo, lo que nos da la frecuencia directamente. El otro es medir el tiempo de un solo pulso de la señal de entrada y calcular su frecuencia con la fórmula **f=1/T**.

```
dígito = 0
     goto          LCD2                ;Ve al dígito 2
enviaLCD3
     movlw         b'00001111'
     andwf         BCD4_3, W
     call          numeroaLCD

LCD2
     swapf         BCD2_1, W           ;Obtiene el dígito número 2
     andlw         b'00001111'
     btfss         STATUS, Z           ;¿Es cero?
     goto          enviaLCD2           ;No, envíalo
     btfss         LCDaux, 1           ;Si, ¿El tercero fue cero?
     goto          enviaLCD2           ;No, envíalo
     btfss         LCDaux, 0           ;Si, el cuarto fue cero?
     goto          enviaLCD2           ;No, envíalo
     call          espacioLCD          ;Si, los dos anteriores y este

son cero, envía espacio
     goto          LCD1                ;Ve al dígito 1
enviaLCD2
     swapf         BCD2_1, W
     andlw         b'00001111'
     call          numeroaLCD

LCD1
     movlw         b'00001111'         ;Obtiene el dígito número 1
     andwf         BCD2_1, W
     call          numeroaLCD          ;Envíalo al LCD
     goto          inicio              ;Inicia otra medición

numeroaLCD                             ;Envía un dígito al LCD
     addlw         '0'                 ;Suma "cero" para obtener el
valor ASCII
     call          LCD_caracter        ;Envíalo al LCD
     return

espacioLCD                             ;Envía un espacio al LCD
     movlw         ' '
     call          LCD_caracter
```

```
     return

;Muestra una indicación de sobre-rango si la medición es mayor a 9999Hz:

over_range
     movlw          d'4'
     movwf          cont_1
     call           LCD_origen1
loop_ov                                   ;Envía cuatro símbolos > para
indicar sobre-rango
     movlw          '>'
     call           LCD_caracter
     decfsz         cont_1, F
     goto           loop_ov
     goto           inicio

;Retardo de un segundo para efectuar la medición, dentro del retardo se
verifica
;si se ha desbordado el TMR0, e incrementa byte_alto cada vez que sucede
para
;poder contar más allá de 255:

retardo_1s
     movlw          d'3'
     movwf          cont_1
loop1
     movlw          d'190'
     movwf          cont_2
loop2
     movlw          d'250'
     movwf          cont_3
loop3
     btfsc          INTCON, 2             ;¿Se ha desbordado el TMR0?
     goto $+2
     goto $+3
     incf           byte_alto, F          ;Si, Incrementa byte_alto y
     bcf            INTCON, 2             ;borra la bandera de
desbordamiento
     decfsz         cont_3, F             ;No, continúa
     goto           loop3
```

```
     decfsz         cont_2, F
     goto           loop2
     decfsz         cont_1, F
     goto           loop1
ret_aux                                  ;retardo auxiliar para ajustar el
tiempo a un segundo exacto
     movlw          d'25'
     movwf          cont_1
loop_aux
     nop
     btfsc          INTCON, 2
     goto $+2
     goto $+3
     incf           byte_alto, F
     bcf            INTCON, 2
     decfsz cont_1, F
     goto           loop_aux
     nop
     return
     #INCLUDE       <BINABCD16.INC>
     #INCLUDE       <LCD4BITS.INC>

     END
```

Si analizamos el código anterior, podemos apreciar que se dispone de varias rutinas. Veamos en detalle cómo actúa cada una de ellas para comprender mejor su funcionamiento: en primer lugar se inicializa el LCD, luego se debe configurar el TMR0 como contador, es decir, para que se incremente con la señal en T0CKI, y se asigna el prescaler al WDT, para que no tenga efecto en el TMR0.

A continuación, se envía la leyenda **Hz** al display, la cual quedará fija en él. Por último, simplemente se borra el TMR0 y se lanza el retardo de 1 segundo para contar los pulsos de la señal de entrada.

Es importante notar que el retardo usado es un tanto especial, ya que dentro de él se verifica constantemente si el TMR0 se ha desbordado, y de esta forma se incrementa el registro auxiliar para contar los pulsos con 16 bits. En este caso, el registro auxiliar es el registro **byte_alto**, que está definido en la subrutina **BINaBCD16**. La sección de verificación de desbordamiento está diseñada de manera que los dos caminos que puede tomar en la verificación toman el mismo tiempo, y así no se altera el tiempo del retardo.

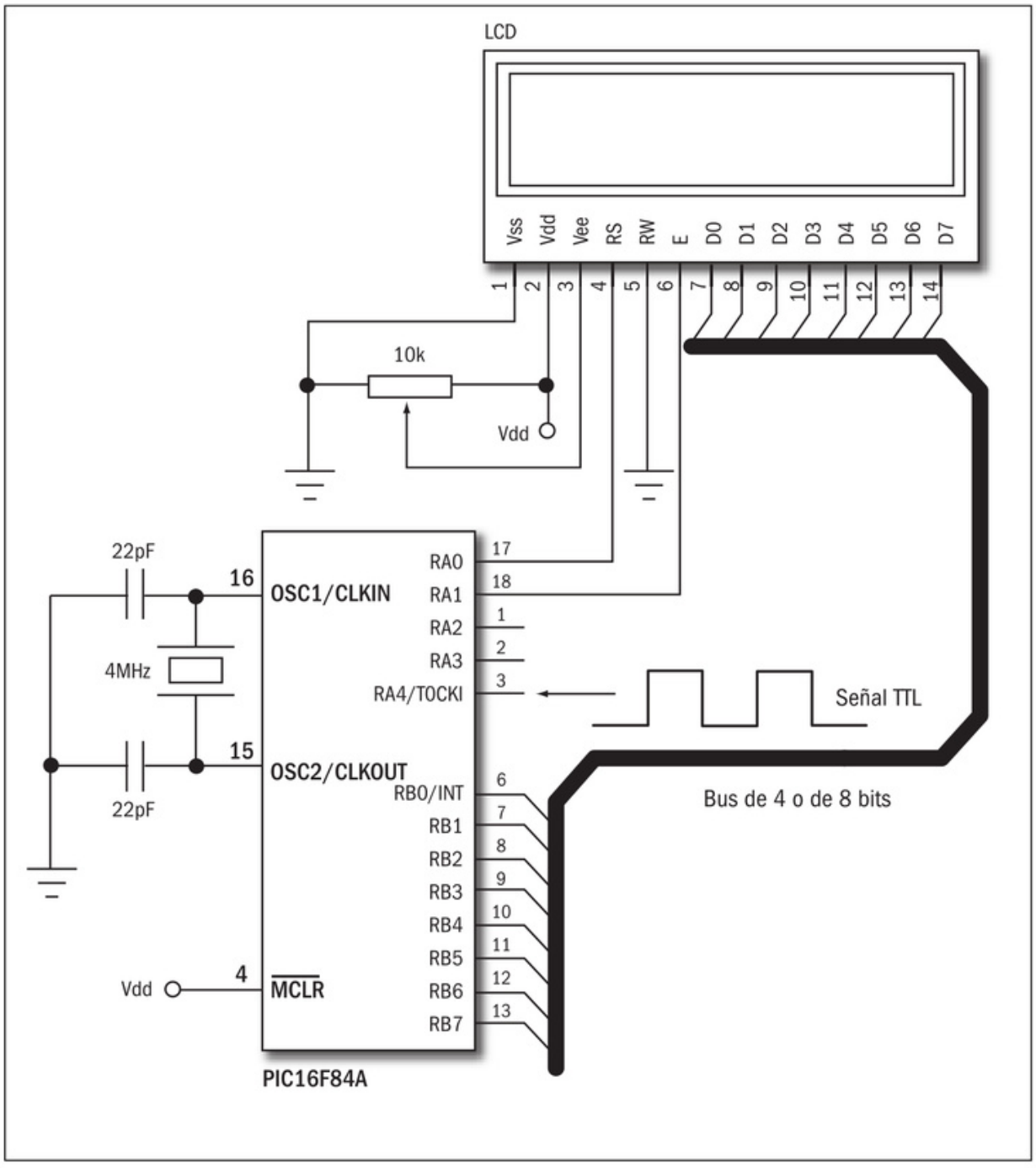


Figura 9. *El circuito usado para el frecuencímetro de 10 KHz con PIC16F84A.*

Podemos usar una conexión con el LCD de 4 ó de 8 bits a elección. Es importante notar que el frecuencímetro sólo puede aceptar señales TTL. De modo que si necesitamos medir señales diferentes, debemos colocar algún circuito que acondicione la señal para poder introducirla al PIC. El programa sólo permite medir señales de 0 Hz hasta 9999 Hz. Más allá de eso, nos indicará un sobrerango colocando signos **>>>>** en el display. Desde el sitio web de la editorial (**www.redusers.com**) podemos descargar los archivos **Frecuencimetro.asm** y **Frecuencimetro.hex** para su estudio y puesta en práctica.

EL TIMER 0 COMO TEMPORIZADOR

Si usamos el TMR0 de tal forma que se incremente con la señal interna de reloj, entonces lo hará a intervalos de tiempo regulares, por lo que en este modo el uso típico es como temporizador, es decir, para contar tiempo. Cuando el TMR0 está configurado para incrementarse con la señal interna, si tenemos un oscilador de 4 MHz, el TMR0 se incrementará con cada ciclo de máquina y, como ya sabemos, es de 1 microsegundo. De esta forma podemos calcular el tiempo que le tomará desbordarse, según el valor que le carguemos al inicio, con la ecuación:

$$\text{Tiempo} = \text{prescaler } (256 - \text{carga}) + 2$$

Donde:

Prescaler: es el valor asignado al prescaler
Carga: el valor que coloquemos en el TMR0

El tiempo en esta fórmula está dado en ciclos de máquina, por lo que debemos multiplicar por el tiempo del ciclo de máquina que usaremos. Para el caso de un oscilador de 4 MHz ya sabemos que el ciclo de máquina es de 1 microsegundo. Si el prescaler es asignado al WDT, entonces tomará el valor de 1 en la ecuación. Se suman dos para ajustar, ya que cuando cargamos un valor en el TMR0 el siguiente incremento se retarda 2 ciclos de máquina. Veamos un ejemplo. Vamos a calcular el tiempo que le tomará desbordarse al TMR0 si se le carga un valor inicial de 150 sin usar prescaler. De la ecuación anterior calculamos:

$$\text{Tiempo} = 1\ (256 - 150) + 2 = 108 \text{ ciclos}$$

Así, para un oscilador de 4 MHz, el tiempo medido con este ejemplo será de 108 microsegundos. El tiempo máximo que podemos obtener es cuando asignamos el prescaler en 1:256 y cargamos un 0 al TMR0, como vemos en la siguiente ecuación:

$$\text{Tiempo} = [256\ (256 - 0) + 2]\ 1\mu s = 65.538\ \mu s = 65.538 \text{ ms}$$

Como ejemplo del uso del TMR0 como temporizador podemos hacer ahora lo contrario al frecuencímetro que ya estudiamos. Ahora generaremos una señal cuadrada

en alguna salida de nuestro microcontrolador. De la ecuación para calcular podemos despejar **carga**, de modo que la fórmula quedará expresada de la siguiente forma:

$$\text{Carga} = 256 - \frac{\text{tiempo} - 2}{\text{prescaler}}$$

Supongamos que necesitamos generar una señal de 800 Hz a la salida del pin RA3, y con un ciclo activo del 50%, entonces el período de la señal será:

$$T = \frac{1}{f} = \frac{1}{800} = 1.25\text{ms}$$

Si el ciclo activo es del 50%, entonces cada semiciclo debe durar la mitad, es decir, 625 microsegundos. Debemos generar una señal que se mantenga en alto 625 microsegundos y en bajo otros 625 microsegundos. Lo primero que debemos observar es el valor adecuado del prescaler que debemos usar. Si empleamos un prescaler de 1, el tiempo máximo que podemos lograr es de 258 microsegundos, lo cual es insuficiente; con un prescaler de 2 será de 514 microsegundos, lo cual también quedará corto, así que debemos usar un prescaler de 4. Con esto calcularemos el valor de **carga** para el TMR0:

$$\text{carga} = 256 - \frac{625 - 2}{4} = 100.25$$

No podemos usar valores fraccionarios, por lo tanto usamos el valor más cercano siempre hacia arriba, que en este caso es 101 decimal. Ahora escribimos nuestro programa, que es el siguiente:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

ORG             0x00
```

```
     bsf          STATUS,RP0          ;Acceso al Banco 1
     clrf         TRISA
     movlw        b'10000001'
     movwf        OPTION_REG          ;Prescaler=1:4, TMR0 como
temporizador
     bcf          STATUS,RP0          ;Acceso al Banco 0

inicio
     bsf          PORTA, 3            ;Coloca RA0 en alto
     call         temporiza           ;Llama a subrutina para temporizar
     bcf          PORTA, 3            ;Coloca RA0 en estado bajo
     call         temporiza           ;Temporiza de nuevo
     goto         inicio              ;Inicia de nuevo

temporiza
     movlw        d'101'              ;El valor calculado
     movwf        TMR0                ;Se carga el TMR0 con él
     bcf          INTCON, TOIF        ;Borra la bandera de
desbordamiento
loop
     btfss        INTCON, TOIF        ;¿Se ha desbordado el TMR0?
     goto         loop                ;No, continua en el loop

     return                           ;Si, regresa

     END
```

Para poner a prueba nuestro programa, podemos usar el circuito que desarrollamos en la **Figura 10**. Además, podemos utilizar MPLAB SIM para comprobar los tiempos obtenidos con el **Stopwatch**, tal como hemos estudiado antes. Recordemos que para eso debemos ir al punto donde se invierte el valor de salida.

III MIDIENDO CON EL STOPWATCH

Para medir los tiempos mediante **Stopwatch** debemos ir al punto donde se invierte el valor de salida (una instrucción después del **bsf**), que es donde inicia un nuevo semiciclo de la señal de salida, y a partir de ahí medir el tiempo hasta el próximo cambio en la salida (después del **bcf**). Podemos colocar **Breakpoints** en esos puntos y presionar el botón **Run** de MPLAB SIM.

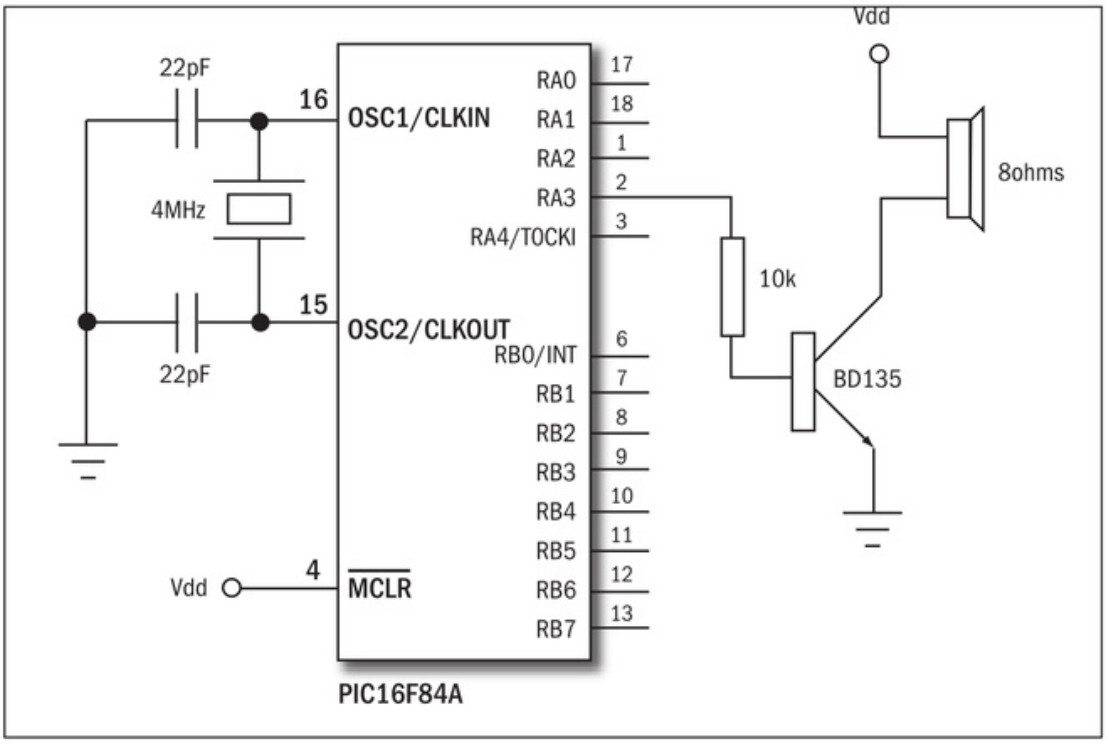


Figura 10. *Tenemos la posibilidad de escuchar la señal generada mediante un altavoz conectado a la salida RA3.*

Si medimos los tiempos en MPLAB SIM observaremos que no coinciden con lo esperado, realmente en alto nos da un tiempo de 631 microsegundos, y en bajo de 633 microsegundos. Esto se debe a que no hemos tenido en cuenta las instrucciones del programa que agregan un poco de tiempo antes de poder cambiar el valor de salida. Y además, el número de instrucciones para cada cambio no es el mismo. Si no hay problema con el tiempo podemos dejarlo así, pero si queremos lograr tiempos exactos debemos ajustarlos. Para eso podemos utilizar MPLAB SIM, si aumentamos el valor de carga del TMR0 hasta obtener tiempos menores al deseado. Luego podemos ajustar mediante instrucciones **nop** o **goto $+1** para lograr los tiempos exactos, tal como lo hemos hecho en el siguiente código ya ajustado:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

ORG             0x00

bsf             STATUS,RP0              ;Acceso al Banco 1
clrf            TRISA
movlw           b'10000001'
```

```
     movwf          OPTION_REG          ;Prescaler=1:4, TMR0 como
temporizador
     bcf            STATUS,RP0          ;Acceso al Banco 0

inicio
     bsf            PORTA, 3            ;Coloca RA0 en alto
     call           temporiza           ;Llama a subrutina para temporizar
     goto           $+1                 ;(Retardo de ajuste)
     bcf            PORTA, 3            ;Coloca RA0 en estado bajo
     call           temporiza           ;Temporiza de nuevo
     goto           inicio              ;Inicia de nuevo temporiza
     goto           $+1                 ;(Retardo de ajuste)
     goto           $+1                 ;(Retardo de ajuste)
     movlw          d'104'              ;(Valor ajustado)
     movwf          TMR0                ;Se carga el TMR0 con él
     bcf            INTCON, T0IF        ;Borra la bandera de
            desbordamiento
loop
     nop
     btfss          INTCON, T0IF        ;¿Se ha desbordado el TMR0?
     goto           loop                ;No, continua en el loop
     return                             ;Si, regresa

     END
```

En este caso los tiempos en alto y bajo son exactamente de 625 microsegundos, cada uno para lograr una frecuencia exacta de 800 Hz. De esta forma es como usaremos el TMR0 como temporizador cuando así lo necesitemos. Hay una característica muy importante que acompaña al TMR0, y es la interrupción por desbordamiento, lo cual le da aun mayor utilidad, pero de esto hablaremos en un capítulo posterior.

RESUMEN

En este capítulo hemos estudiado qué es y cómo se utiliza el temporizador/contador llamado Timer 0 del PIC16F84A, y sus aplicaciones típicas, como contador de eventos externos, o como temporizador para lograr medir tiempo. Además, construimos un frecuencímetro basado en el Timer 0 como contador. Aún nos falta estudiar la interrupción asociada al desbordamiento del TMR0, pero esto lo haremos más adelante

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Para qué sirve el Timer 0 del PIC16F84A?

2 ¿Qué es el prescaler?

3 ¿Cuáles son las dos formas con que puede incrementarse el TMR0

4 Si necesitamos que el TMR0 se incremente con la señal de reloj interna debemos colocar un _____ en T0CS.

5 Si queremos asignar el prescaler al TMR0 debemos poner un _____ en PSA.

6 Si necesitamos un prescaler de 1:16 en el TMR0 debemos poner los bits PS2, PS1 y PS0 con un valor de _____.

7 ¿Cuál es el bit que nos indica un desbordamiento en el TMR0 y en qué registro se encuentra?

8 Si usamos al TMR0 como temporizador con oscilador de 4 MHz, se incrementará en un tiempo de _____.

9 ¿Por qué debemos cargar al TMR0 con un valor inicial cuando lo usamos como temporizador?

10 El bit T0IF debemos borrarlo por _____.

PRÁCTICAS

1 El programa TMR0 contador.asm sólo cuenta las pulsaciones de entrada hasta 255. Modifíquelo para que ahora cuente más allá de 255.

2 Calcule el tiempo de desbordamiento que se obtendrá si se carga el TMR0 con un valor de 12 y con un prescaler de 1:32.

3 Calcule el valor que deberá cargar al TMR0 y qué prescaler será adecuado para temporizar 1.2ms (1200 microsegundos).

4 Escriba en MPLAB los dos programas de la sección El Timer 0 como temporizador y simúlelos para comprobar los tiempos en cada uno con el Stopwatch. Luego ensámblelos y grábelos en el PIC16F84A para comprobar su funcionamiento real.

5 Escriba un programa para obtener una señal en el pin RA3 de 800 Hz, pero ahora con un tiempo en alto de 700 microsegundos y en bajo de 550 microsegundos.

6 Diseñe un programa para obtener una señal en el pin RA3 de 1500 Hz exactos.

Capítulo 9

Otras funciones del PIC16F84A

Existen otras funciones en el PIC16F84A que nos pueden ser de utilidad para nuestros diseños. En este capítulo estudiaremos el direccionamiento indirecto, el Watch Dog Timer, el modo de bajo consumo o sleep, los resistores de Pull-up del Puerto B, la utilización de la memoria EEPROM para almacenar datos y el uso de macros

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

DIRECCIONAMIENTO INDIRECTO

Hasta ahora hemos manejado el direccionamiento de forma **directa**, es decir, en las propias instrucciones donde tenemos la dirección del registro de la memoria de datos sobre el cual vamos a trabajar.

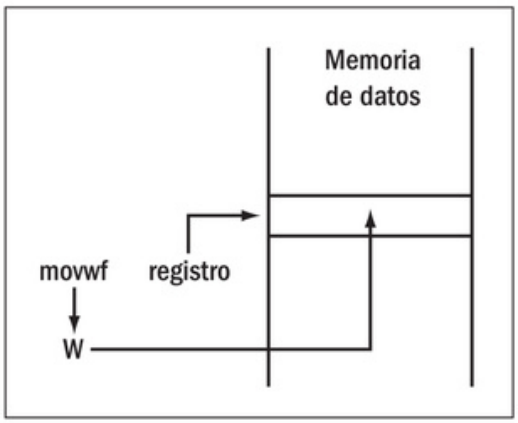


***Figura 1.** En el direccionamiento directo, el propio código contiene la dirección del registro a trabajar.*

Como veremos ahora, podemos direccionar los registros de la memoria de datos de forma **indirecta** si usamos los registros **INDF** y **FSR**.

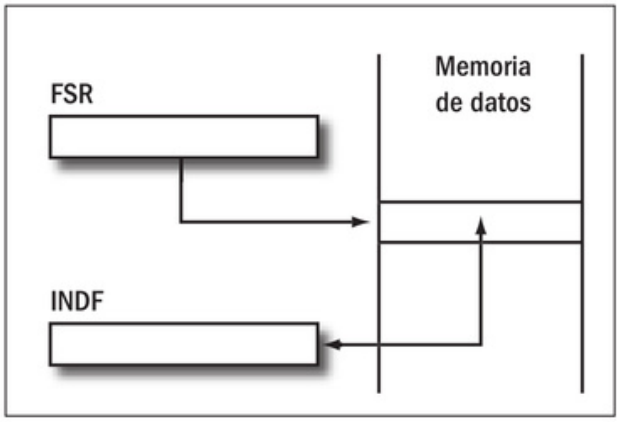


***Figura 2.** Los registros FSR e INDF permiten acceder a la memoria de datos de forma indirecta.*

El registro FSR (*File Select Register*), que se encuentra en la dirección 04h del banco 0 y se repite en la 84h del Banco 1, es realmente un **apuntador** que sirve para definir direcciones de memoria. El contenido del registro FSR es tomado, en verdad, como una dirección. El registro INDF es un registro **auxiliar**, en donde se reflejarán los datos a leer o a escribir con el método de direccionamiento indirecto. El registro INDF está en las direcciones 00h y 80h en ambos bancos respectivamente, aunque realmente no es un registro físico. Para comprender mejor su fincionamiento, veamos un ejemplo de cómo funciona el direccionamiento indirecto.

Supongamos que colocamos un dato 15d en el registro 0Ch y un 16d en el registro 0Dh. Ahora, si escribimos un 0Ch en el registro FSR, estamos apuntando a la dirección 0Ch y si leemos el contenido del registro INDF se reflejará el valor de esa dirección, que es 15d. Si incrementamos el registro FSR accederemos a la siguiente dirección 0Dh y en INDF leeremos 16d.

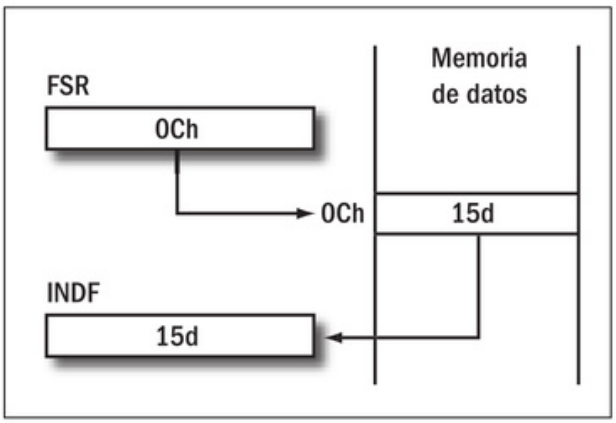


Figura 3. *Un ejemplo de cómo leer el contenido de un registro usando direccionamiento indirecto.*

Por supuesto, el direccionamiento indirecto no sólo permite la lectura, sino también la escritura. Para comprender mejor su funcionamiento, veamos un programa como ejemplo del uso del direccionamiento indirecto.

Supongamos que deseamos colocar el mismo valor en una serie de registros de la memoria de datos. El dato a escribir será 78h, por ejemplo, y lo escribiremos en los registros de la dirección 10h hasta la 20h:

```
     __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

     PROCESSOR 16F84A
     #INCLUDE <P16F84A.INC>

     ORG          0x00

inicio
     movlw        h'20'              ;Dirección última a escribir
     movwf        FSR                ;Se coloca en el apuntador FSR
loop
     movlw        h'78'              ;El dato a escribir
     movwf        INDF               ;Se envía a INDF
     decf         FSR, F             ;Decrementa la dirección
```

```
movlw       h'OF'               ;Dirección límite  - 1
subwf       FSR, W
btfss       STATUS, Z           ;¿Llegó al final?
goto        loop                ;No, continúa con la siguiente
goto        $                   ;Si terminó

END
```

De esta forma, podemos observar cómo se pueden escribir o leer datos de una serie de registros consecutivos mediante el direccionamiento indirecto, simplemente al incrementar o disminuir el contenido de FSR para ir accediendo a las direcciones apuntadas por él y leerlas o escribirlas mediante el registro INDF.

EL WATCH DOG TIMER

Como ya mencionamos antes, el PIC16F84A, en realidad, tiene dos timers: uno ya lo estudiamos en el **Capítulo 8** (el TMR0) pero, además de éste, también contiene otro llamado **Watch Dog Timer**, que se abrevia **WDT**. El nombre lo podríamos traducir como **temporizador de perro guardián**. Este temporizador tiene un propósito diferente del TMR0. En el caso del WDT, su tarea es vigilar al programa para que no se quede estancado en algún punto (de ahí la analogía con un perro guardián).

El WDT funciona mediante un oscilador RC interno y es totalmente independiente del oscilador principal, por lo que siempre está activo, sin importar si el oscilador principal está detenido. El período del WDT es de aproximadamente 18 ms, es decir, el tiempo que tarda en desbordarse si se está usando un prescaler de 1:1. Como la tarea del WDT es **vigilar** que el programa no se quede demasiado tiempo en un bucle en espera de un evento que no ocurre, o se estanque debido a un mal funcionamiento del programa, si se desborda provoca un reset al microcontrolador, obligándolo a reiniciar.

¡CUIDADO CON EL PERRO!

Hasta ahora hemos desactivado el WDT en nuestros programas mediante la directiva **__CONFIG**. Recordemos que, de manera predeterminada, el WDT está activado, por lo que si no lo desactivamos expresamente, puede provocar muchos dolores de cabeza, sobre todo a los programadores novatos, ya que los programas se reiniciarán constantemente sin razón aparente.

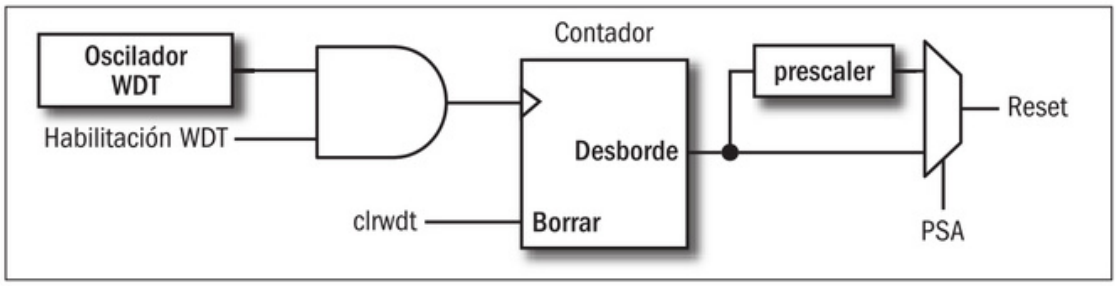


***Figura 4.** El WDT puede ser habilitado o no y tiene su propio oscilador.*

Habilitación y uso del WDT

Hasta ahora siempre habíamos tenido el WDT deshabilitado para que no afectara nuestros programas, pero si necesitamos usarlo debemos habilitarlo mediante la directiva **__CONFIG**. Por ejemplo, si la colocamos de esta forma:

```
__CONFIG _CP_OFF & _WDT_ON & _PWRTE_ON & _XT_OSC
```

Al colocar **WDT_ON** tendremos el WDT habilitado, por lo que al ejecutar el programa, el WDT estará funcionando. Para evitar que el WDT se desborde y provoque con ello el reset del sistema debemos usar la instrucción **clrwdt**, la cual pondrá a cero el contador del WDT y éste tendrá que contar de nuevo desde el principio.

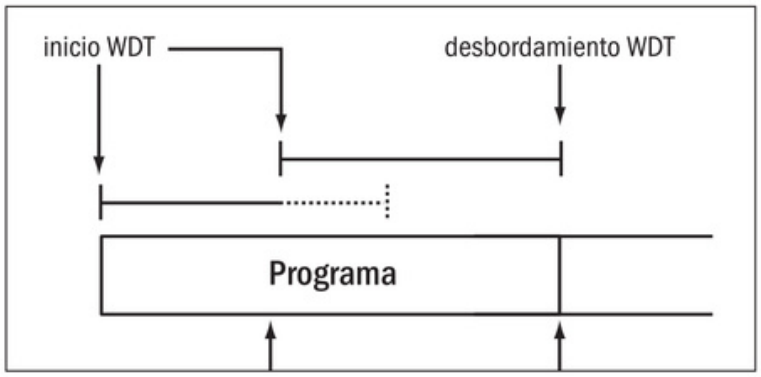


***Figura 5.** La instrucción clrwdt evita que el WDT se desborde y provoque un reset a todo el sistema.*

III ¿APROXIMADAMENTE 18 MS?

Hemos mencionado que el tiempo del WDT (sin prescaler) es de aproximadamente 18ms. Esto es porque el oscilador es del tipo RC y, como sabemos, este tipo de osciladores es algo inexacto debido a varios factores, pero principalmente por la temperatura y las variaciones en el voltaje de alimentación. Es por eso que el tiempo del WDT es aproximado y puede variar un poco.

Debemos colocar la instrucción **clrwdt** en lugares estratégicos de nuestro programa para evitar que el WDT se desborde mientras el funcionamiento sea normal, y así el WDT no tendrá efecto alguno. Sólo hasta que se produzca una situación no deseada, el WDT se desbordará y provocará el reset obligando al sistema a iniciar de nuevo todo el funcionamiento desde el principio.

Como vimos en el **Capítulo 8**, el prescaler puede asignarse ya sea al TMR0 o al WDT. Si asignamos el prescaler para que actúe sobre el WDT, podemos controlar el tiempo que tardará en actuar, desde 1:1 hasta 1:128 (ver la **Tabla 2** del **Capítulo 8**), lo cual nos da tiempos diferentes desde los 18 ms hasta aproximadamente 2.3 segundos con el prescaler más alto.

Debemos tener en cuenta que sólo se acostumbra utilizar el WDT en programas avanzados, donde se requiere mayor seguridad en el funcionamiento del los circuitos. En aplicaciones sencillas no es muy común emplearlo y debido a esto nos enfrentaremos pocas veces con un programa donde se use el WDT.

MODO DE BAJO CONSUMO (SLEEP)

En ocasiones, el microcontrolador ha cumplido con todas las tareas que debe ejecutar, o queda en espera de un acontecimiento durante un largo tiempo. En estas situaciones podemos hacer que nuestro microcontrolador entre en un modo llamado **sleep** o de **bajo consumo**, que ocasiona que el oscilador principal deje de funcionar, es decir que literalmente el microcontrolador se apaga. Para entrar en este modo sólo basta con ejecutar la instrucción **sleep** y sucederá lo siguiente:

- El oscilador principal del sistema se detiene.
- El TMR0 deja de funcionar.
- Los puertos mantienen el mismo estado en el que se encontraban.
- El consumo de potencia se reduce mucho al suceder todo esto.

{ } EL WDT COMO TERMÓMETRO

Ya mencionamos que uno de los factores que afectan la exactitud del WDT es la temperatura, por lo que el tiempo de él depende de ese factor. Si descargamos la nota de aplicación **AN828** del sitio web de **www.microchip.com**, encontraremos la descripción de una técnica para medir la temperatura con el WDT del PIC16F84A para poder construir un termómetro basado en él.

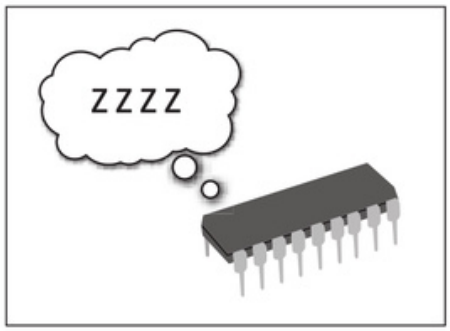


Figura 6. *La instrucción sleep literalmente pone el microcontrolador a "dormir".*

El objetivo de la instrucción **sleep** es mantener el microcontrolador en un estado donde se consume muy poca potencia. Esto es principalmente útil cuando se alimenta con baterías, o en caso que se quiera reducir al mínimo el consumo de potencia. Ahora, cuando colocamos el PIC en modo sleep o de bajo consumo, será necesario "despertarlo" de alguna forma para que se pueda continuar con la ejecución del programa. Existen varias formas de hacer esto:

- Al dar un reset al sistema por el pin Master clear.
- Por el desbordamiento del WDT.
- Por una interrupción (estudiaremos esto en el **Capítulo 10**).

Si damos un reset al sistema a través del pin Master clear, entonces el sistema reiniciará su funcionamiento desde el principio y hará salir el microcontrolador del modo de bajo consumo. Recordemos que el WDT tiene su propio oscilador independiente, por lo que, aunque se entre en modo de bajo consumo y el oscilador principal se detenga, el oscilador del WDT seguirá funcionando y cuando el WDT se desborde, provocará que el PIC salga del modo de bajo consumo y continúe la ejecución del programa en la siguiente instrucción después del sleep. Esto, claro, sólo si se ha habilitado el WDT. Podemos utilizar un par de bits del registro STATUS para verificar cuál fue la causa de un reset que son los bits TO' y PD'.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
IRP	RP1	RP0	**TO'**	**PD'**	Z	DC	C

Tabla 1. *Bits del registro STATUS para identificar los reinicios del sistema.*

De la **Tabla 1** hemos resaltado los bits que nos interesan ahora:

Bit 4 TO' (Timer Out): es el bit de desbordamiento del WDT:

TO'=0 cuando se ha desbordado el Watch dog timer
TO'=1 al conectar la alimentación al sistema (Vdd), o al ejecutar **clrwdt** o **sleep**

Bit 3 PD' (Power Down): es el bit de bajo consumo:

PD'=0 al ejecutar la instrucción **sleep** y entrar en modo de bajo consumo
PD'=1 al conectar la alimentación al sistema (Vdd), o al ejecutar **clrwdt**

Los dos bits comentados anteriormente son de sólo lectura, es decir, el usuario no puede escribir en ellos. El bit TO' nos servirá para detectar si un reset ha sido causado por el desbordamiento del WDT.

Ejemplo del uso del WDT y sleep

Veamos un programa como ejemplo del uso del WDT y el modo de bajo consumo. Es un programa simple que inicializa un contador y se pone en modo de bajo consumo con la instrucción **sleep**, hasta que el WDT se desborda provocando que continúe con la cuenta.

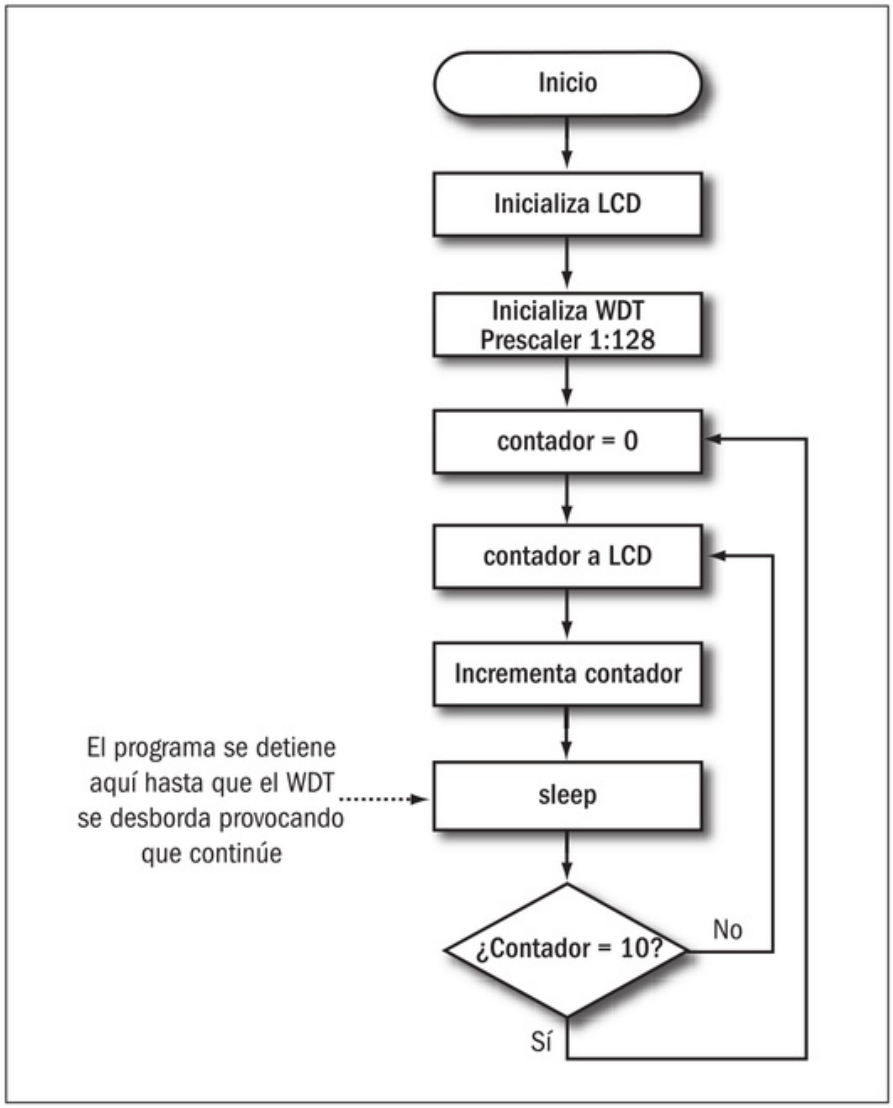


Figura 7. *Un programa de ejemplo de cómo se sale del modo sleep mediante el WDT.*

El código fuente de nuestro programa de ejemplo para comprender el uso del WDT y el modo de bajo consumo será, entonces, el siguiente:

```
      __CONFIG _CP_OFF & _WDT_ON & _PWRTE_ON & _XT_OSC   ;>>>WDT activado<<<

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      contador
      ENDC

      ORG            0x00

      call           LCD_inicializa
      bsf            STATUS, RP0
      movlw          b'10001111'
      movwf          OPTION_REG            ;Prescaler al WDT 1:128
      bcf            STATUS, RP0

inicio
      clrf           contador
loop
      call           LCD_origen1
      movf           contador, W
      addlw          '0'
      call           LCD_caracter          ;Muestra el número de la cuenta
      incf           contador, F           ;Incrementa la cuenta
      sleep                                ;Pasa al modo de bajo consumo,
hasta que el WDT se desborda
      movf           contador, W           ;Cuando sale del modo de bajo
consumo continúa aquí
      sublw          d'10'
      btfsc          STATUS, Z             ;¿Ha llegado a 10?
      goto           inicio                ;Si, reinicia el contador
      goto           loop                  ;No, continúa

      #INCLUDE       <LCD4BITS.INC>

      END
```

Se asigna el prescaler al WDT con un radio de 1:128, lo cual da unos 2.3 segundos para que se desborde, por lo tanto, el contador se incrementará en ese intervalo de tiempo. En el LCD se refleja la cuenta que sólo va de 0 a 9. Para probar este programa simplemente debemos tener el LCD conectado al PIC16F84A mediante un bus de 4 bits. Notemos cómo no hay ninguna subrutina de retardo, ya que el tiempo de la cuenta en este caso está controlado por el WDT.

RESISTORES DE PULL-UP DEL PUERTO B

Hasta ahora hemos usado resistores de **Pull-up externos** cada vez que necesitamos conectar un pulsador a alguna línea de entrada en nuestro microcontrolador.

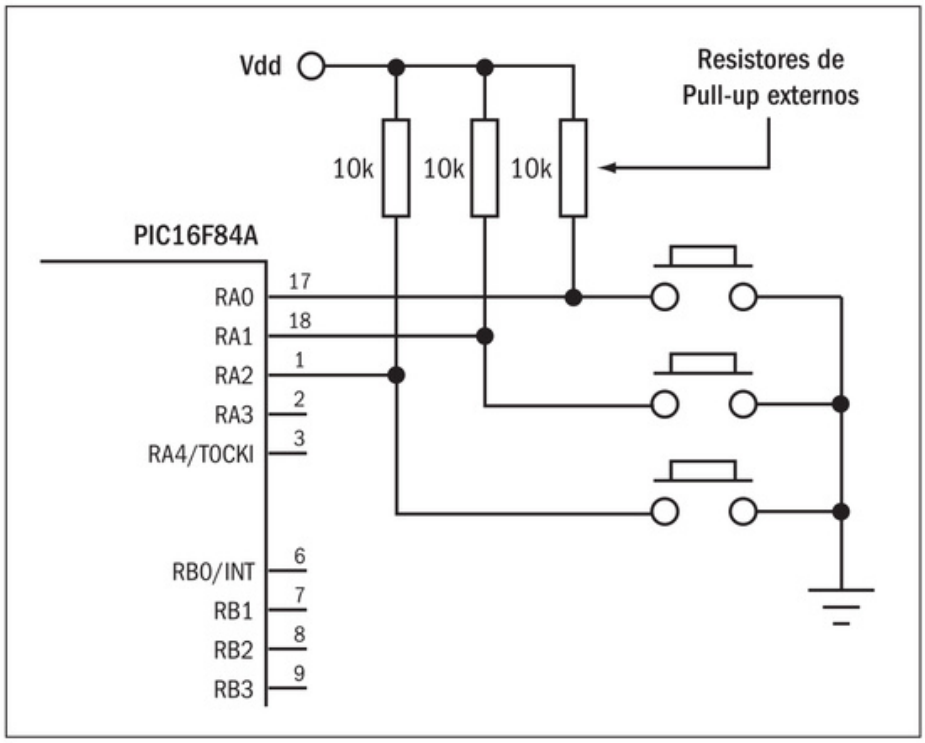


***Figura 8.** Hasta el momento hemos utilizado resistores externos para los pulsadores, sobre todo en las líneas del Puerto A.*

III WEAK PULL-UPS SÓLO PARA ENTRADAS

Los resistores de Pull-up internos del PIC16F84A se activan para no tener que colocar resistores externos. Estos resistores internos se activan sólo cuando el Puerto B o algunas de sus líneas son configurados como entradas. Cuando se configuran como salidas, los resistores de Pull-up se desactivan automáticamente.

Pero si colocamos pulsadores u otros elementos que requieran de resistores de Pull-up en el Puerto B, podemos emplear los que tiene internamente, conocidos como **Weak Pull-ups**. Esto significa que el Puerto B del PIC16F84A cuenta con resistores internos que podemos activar para no tener que usar los resistores externos. Para activar los resistores de Pull-up internos debemos hacerlo con el registro OPTION. tal como detallamos en la **Tabla 2**:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
RBPU'	INTDEG	T0CS	T0SE	PSA	PS2	PS1	PS0

***Tabla 2.** El bit RBPU' del registro OPTION habilita los Weak Pull-ups.*

De esta forma, podemos conectar pulsadores u otros elementos al Puerto B y habilitar los Pull-ups internos con la instrucción:

```
bcf  OPTION_REG, NOT_RBPU
```

Ya que se habilitan con un cero.

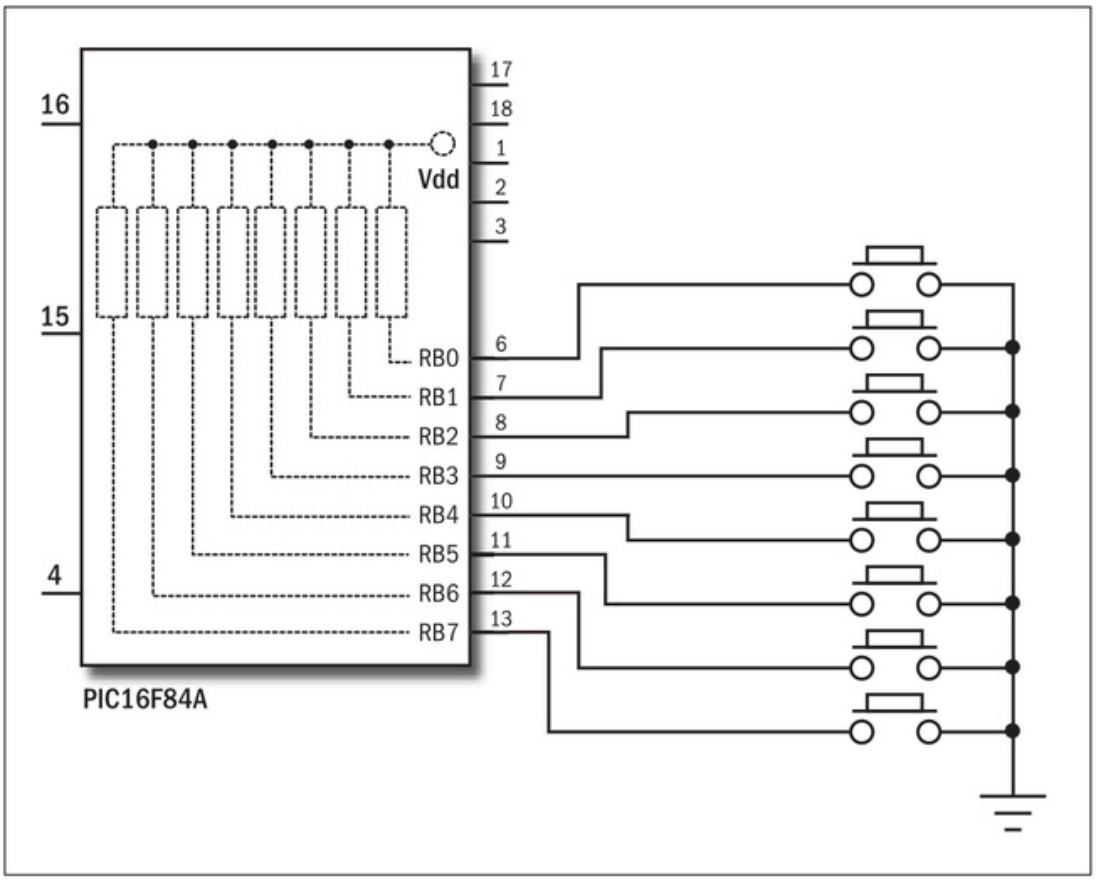


***Figura 9.** Al habilitar los Pull-ups del Puerto B nos ahorramos los resistores externos.*

LA MEMORIA EEPROM DE DATOS

Sabemos que podemos manejar y guardar datos, precisamente en la memoria de datos, pero si desconectamos la alimentación del circuito, estos datos se pierden, ya que la memoria de datos es del tipo RAM. Además, podemos guardar datos en la memoria de programa, pero éstos serán fijos y no podremos cambiarlos desde el propio programa. Si necesitamos guardar datos generados desde nuestro programa y que permanezcan incluso al quitar la alimentación del PIC, podemos usar una zona de memoria **EEPROM de datos** que tiene el PIC16F84A.

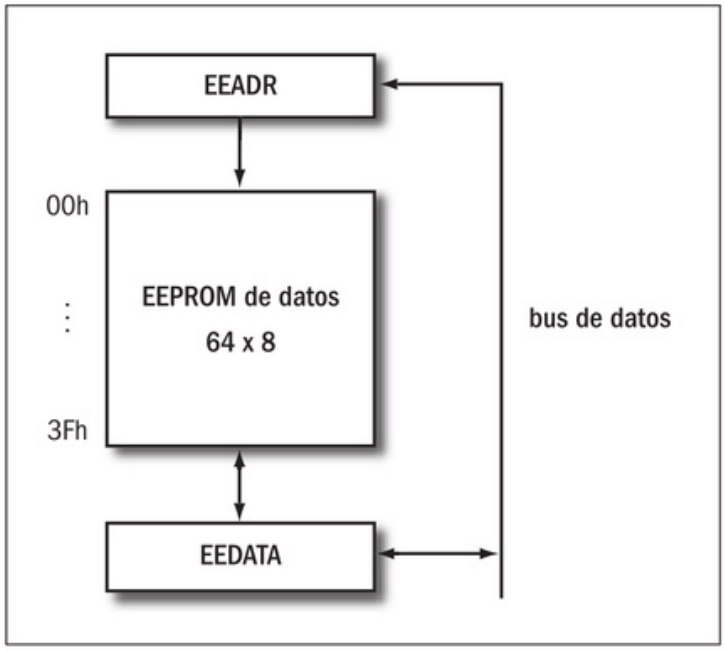


***Figura 10.** También podemos guardar datos en una memoria no volátil en el PIC16F84A.*

Esta memoria de tipo EEPROM se puede escribir y leer por el usuario y contiene **64 bytes**, es decir, 64 registros de 8 bits cada uno. De todos modos, debemos tener en cuenta que no se puede acceder directamente a la memoria EEPROM de datos, sino que para escribir o leer en ella usaremos algunos registros del área SFR en la memoria de datos del PIC16F84A.

{ } EVOLUCIÓN DE LA EEPROM

El tiempo de escritura en la memoria EEPROM de datos (y también en la memoria Flash del programa) marca una evolución en el PIC16F84A con respecto a su predecesor, el PIC16F84, en el cual los tiempos de escritura en estas memorias eran típicamente de 10 ms, contra los 4 ms del PIC16F84A. Es decir, el tiempo de escritura se ha reducido más de la mitad.

Registros relacionados con la EEPROM de datos

En particular, nos interesan cuatro registros relacionados con la memoria EEPROM de datos para poder realizar la lectura o escritura en ella.

Los registros EEDATA y EEADR

El registro **EEDATA**, que está en la dirección 08h del banco 1 de la memoria de datos, es donde colocaremos los datos a escribir o donde se reflejarán los datos a leer en la EEPROM de datos. El registro **EEADR**, que está en la dirección 09h de la memoria de datos, es donde colocaremos la dirección del dato a leer o a escribir de la memoria EEPROM.

Los registros EECON1 y EECON2

Estos registros sirven para controlar la escritura o lectura en la EEPROM de datos.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
-	-	-	EEIF	WRERR	WREN	WR	RD

***Tabla 3.** El registro EECON1.*

Bits 7 a 5: no implementados

Bit 4 EEIF (EEPROM Write operation Interrupt Flag Bit): bandera de finalización de escritura en la EEPROM:

EEIF=0 la operación de escritura no se ha completado o no ha iniciado
EEIF=1 la operación de escritura terminó, **debe borrarse por software**

Bit 3 WRERR (EEPROM Error Flag Bit): bit de error en la escritura:

WRERR=0 la escritura ha sido completada sin errores
WRERR=1 la escritura en EEPROM terminó prematuramente

Bit 2 WREN (EEPROM Write Enable Bit): habilita la escritura en la memoria EEPROM de datos:

WREN=0 no permite la escritura en la EEPROM de datos
WREN=1 permite la escritura en la EEPROM de datos

Bit 1 WR (Write Control Bit): bit de control de escritura en la EEPROM de datos:

WR=0 la escritura terminó
WR=1 se inicia un ciclo de escritura en EEPROM y se borra al finalizar

Bit 0 RD (**Read control Bit**): bit de control de lectura en la EEPROM de datos:

RD=0 no hay lectura de la EEPROM de datos
RD=1 se inicia un ciclo de lectura en la EEPROM de datos. Se borra automáticamente al finalizar.

El **EECON2** es un registro de control de escritura en la EEPROM de datos, y no está implementado físicamente, por lo que sólo podemos escribir en él. Si intentamos leerlo, devolverá ceros. El ciclo de escritura en la memoria EEPROM de datos tarda unos **4ms**. Como podemos ver, los bits WR y RD se pondrán a cero automáticamente al finalizar la escritura o lectura de datos respectivamente, sólo debemos poner un 1 en ellos cuando lo necesitemos. Al escribir un dato en la memoria EEPROM de datos, automáticamente se borrará el anterior y se escribirá el nuevo, no debemos preocuparnos por borrarlo nosotros (de hecho, no existe ningún comando de borrado). El bit EEIF no se elimina automáticamente, debemos borrarlo nosotros cuando sea necesario hacerlo.

Lectura de la memoria EEPROM de datos

Para leer un byte de la memoria EEPROM de datos sólo debemos seguir estos pasos:

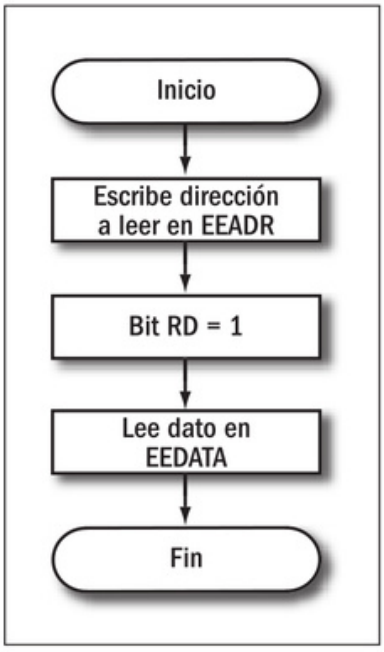


***Figura 11.** Proceso de lectura de la EEPROM de datos del PIC16F84A.*

El proceso de lectura es muy sencillo, sólo pondremos en el registro EEADR la dirección que vamos a leer de la memoria EEPROM de datos, que va desde 00h hasta 3Fh. Y luego colocamos el bit RD a 1 para iniciar la lectura. También es un proceso muy rápido, por lo que en el siguiente ciclo de reloj ya podemos ac-

ceder al dato leído mediante el registro EEDATA. Aquí permanecerá el valor leído hasta que se inicie una nueva lectura o escritura de otro byte.

Escritura en la memoria EEPROM de datos

El proceso de escritura es más complicado que el de lectura, ya que debemos seguir algunos lineamientos para asegurar que sea exitosa. Además, el proceso de escritura es más lento, por lo que debemos esperar un poco más para que se complete. El tiempo de escritura típico es de 4 ms por byte y el fabricante especifica un máximo de 8 ms.

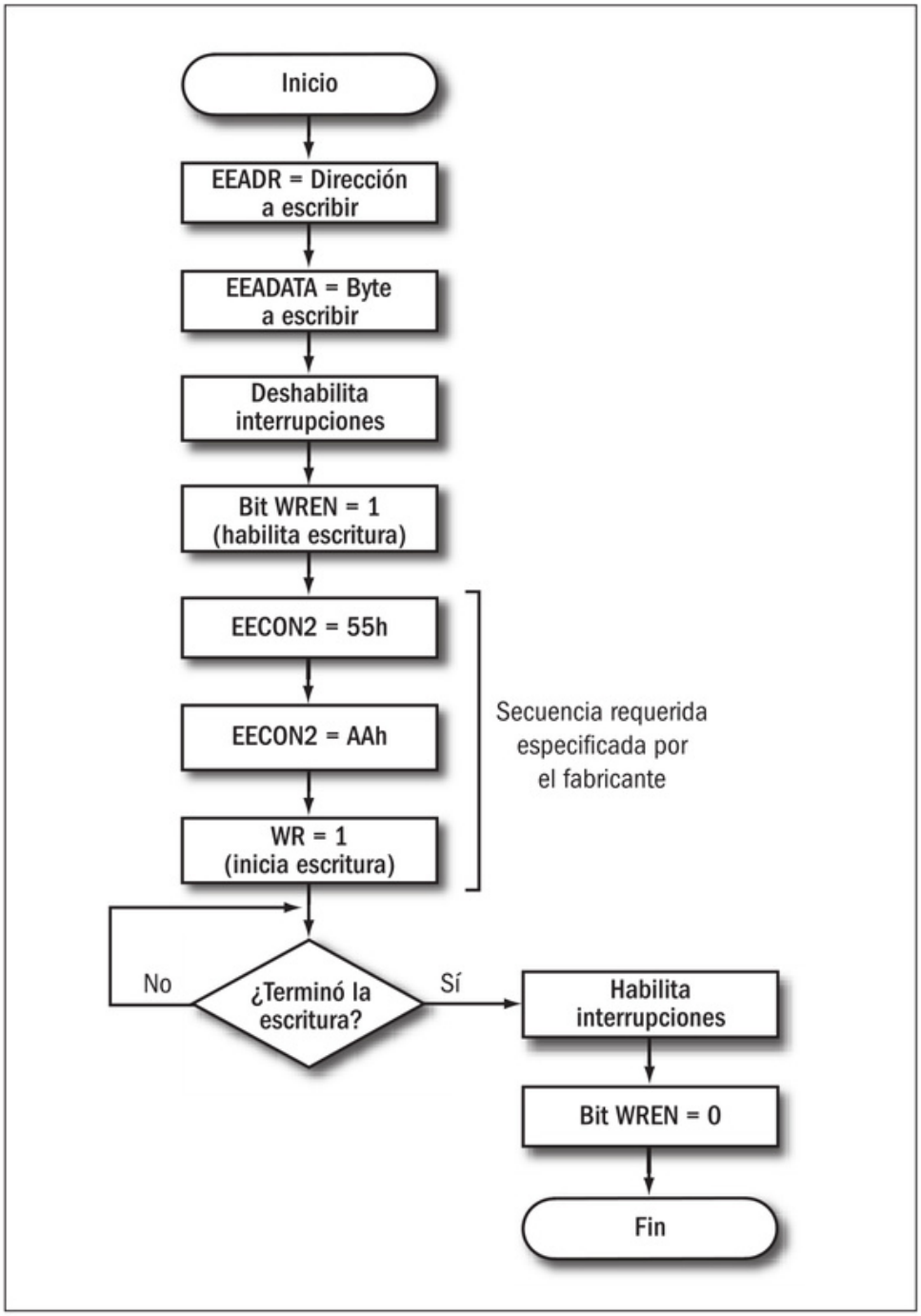


Figura 12. *Proceso de escritura de la EEPROM de datos.*

Como podemos apreciar en el diagrama de flujo de la **Figura 12**, debemos seguir algunos pasos para realizar la escritura de forma exitosa en la memoria EEPROM de datos. El fabricante especifica una secuencia de inicialización de escritura como la marcada en el diagrama, que es la siguiente:

```
     movlw          h'55'
     movwf          EECON2
     movlw          h'AA'
     movwf          EECON2
     bsf            EECON1, WR
```

Otra recomendación que debemos tener en cuenta es la de desactivar todas las interrupciones al momento de estar escribiendo un dato para asegurar que no haya ninguna interrupción que haga que la escritura no se complete.

Librería para manejo de EEPROM de datos

Como siempre, podemos escribir una librería para hacer más fácil y rápida la escritura y lectura de la memoria EEPROM de datos. En el sitio web de la editorial (**www.redusers.com**) podemos descargar la librería llamada **EEPROM.INC**, que contiene las subrutinas para lectura y escritura de datos en la EEPROM:

```
     CBLOCK
     guardaINTCON
     ENDC

;Subrutina para leer datos de la EEPROM de datos del PIC16F84A:

lee_EEPROM
     bcf            STATUS, RP0          ;Acceso al Banco 0
     movwf          EEADR                ;Dirección a leer
     bsf            STATUS, RP0          ;Acceso al Banco 1
     bsf            EECON1, RD           ;Inicia la lectura
EEPROM_leyendo
     btfsc          EECON1, RD           ;¿Ha terminado la lectura?
     goto           EEPROM_leyendo       ;No, espera
     bcf            STATUS, RP0          ;Si, Acceso al Banco 0
     movf           EEDATA, W            ;Pasa el dato leído a W
     return                              ;Regresa
```

```
;Subrutina para escribir en la memoria EEPROM de datos del PIC16F84A:

escribe_EEPROM
     bcf            STATUS, RPO           ;Acceso al Banco 0
     movwf          EEDATA                ;El dato a escribir
     movf           INTCON, W
     movwf          GuardaINTCON          ;Guarda el contenido de INTCON
     bsf            STATUS, RPO           ;Acceso al Banco 1
     bcf            INTCON, GIE           ;Deshabilita las interrupciones
     bsf            EECON1, WREN          ;Habilita la escritura en EEPROM

;Secuencia de inicialización especificada por el fabricante:

     movlw          h'55'
     movwf          EECON2
     movlw          h'AA'
     movwf          EECON2
     bsf            EECON1, WR            ;Inicia la escritura en EEPROM

EEPROM_escribiendo
     btfsc          EECON1, WR            ;¿Ha terminado la escritura?
     goto           EEPROM_escribiendo    ;No, espera
     bcf            EECON1, WREN          ;Si, Deshabilita la escritura en
EEPROM
     bcf            EECON1, EEIF          ;Borra la bandera de finalización
de escritura
     bcf            STATUS, RPO           ;Acceso al Banco 0
     movf           GuardaINTCON, W
     movwf          INTCON                ;Restaura el valor de INTCON
     return                               ;Regresa
```

La librería no es muy compleja y cumple con los requerimientos establecidos por Microchip para una correcta inicialización de escritura. Nos será de mucha utilidad en los programas donde hagamos uso de la memoria EEPROM de datos del PIC16F84A.

La directiva DE

La directiva **DE** (*Declare EEPROM Data Byte*) nos sirve para guardar datos iniciales en la memoria EEPROM de datos, es decir, para definir datos que se grabarán en la EEPROM al grabar el PIC. Su sintaxis es:

DE exp

exp son los datos a grabar. Debemos especificar, también, la dirección de inicio donde serán grabados los datos. El origen de la memoria EEPROM se especifica con la dirección 2100h, por ejemplo:

```
ORG             2100h
DE              h'77', h'12'
```

En este ejemplo se grabará 77 hexadecimal en la primera dirección de la memoria EEPROM y 12 hexadecimal en la segunda. También podemos definir caracteres ASCII, como en el siguiente ejemplo:

```
ORG             2100h
DE              "Mi programa V1.0"
```

De esta forma, podemos especificar datos que serán grabados en la EEPROM junto con el código máquina al momento de grabar el PIC. Como podemos apreciar, tenemos la posibilidad de guardar cadenas de caracteres para especificar versiones de programa, fechas, notas, o cualquier cosa que necesitemos. Los datos definidos con la directiva **DE** se pueden leer posteriormente con un grabador de PICs, tal como ya hemos visto. Si activamos la protección de código del PIC, el programa puede seguir leyendo y escribiendo en la EEPROM de datos, pero al intentar leer el dispositivo mediante un grabador, no se podrá leer la EEPROM de datos, ya que la protección de código lo impide.

La EEPROM en MPLAB

En MPLAB también podemos observar los datos que contiene la memoria EEPROM de datos. Si vamos al menú **View/EEPROM** aparecerá una ventana (Figura 13) donde se muestra el contenido de esta memoria.

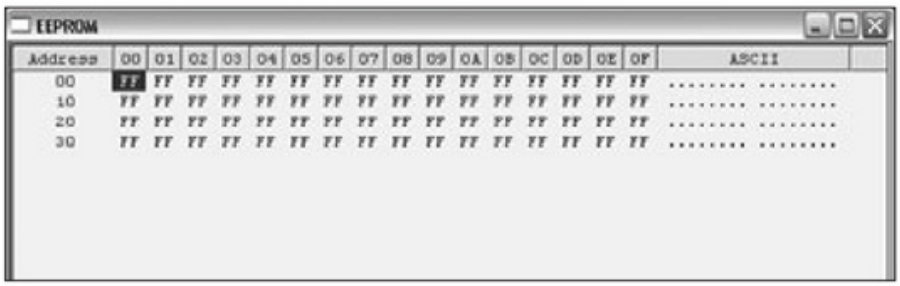


Figura 13. *Podemos observar el contenido de la memoria EEPROM de datos en MPLAB.*

Un ejemplo del uso de la directiva **DE** es el que desarrollamos a continuación, En este código fuente únicamente hemos definido una cadena de caracteres para que sean grabados en la memoria EEPROM de datos:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>

ORG             0x2100
DE              "Mi programa EEPROM V 1.0"

END
```

Como podemos apreciar en la **Figura 14**, al momento de ensamblar el programa, la cadena de caracteres aparece ya en la ventana EEPROM.

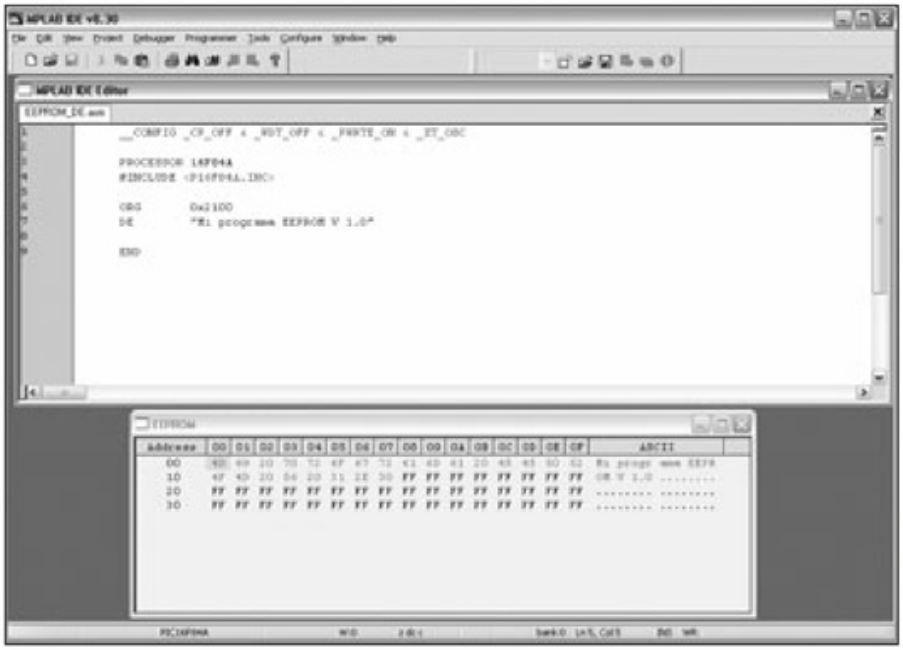


Figura 14. *La ventana EEPROM con los valores definidos por la directiva* ***DE****.*

La EEPROM en IC-Prog

Podemos comprobar que los datos asignados con la directiva **DE** se grabarán en la EEPROM al momento de grabar el PIC. Si abrimos IC-Prog y luego abrimos en él el archivo ***.hex** generado en el ensamblado de este ejemplo, aparecerán en la zona de datos EEPROM precisamente los datos que hemos definido con la directiva **DE** y, de esa forma, al grabar el PIC, también los grabará en la EEPROM de datos.

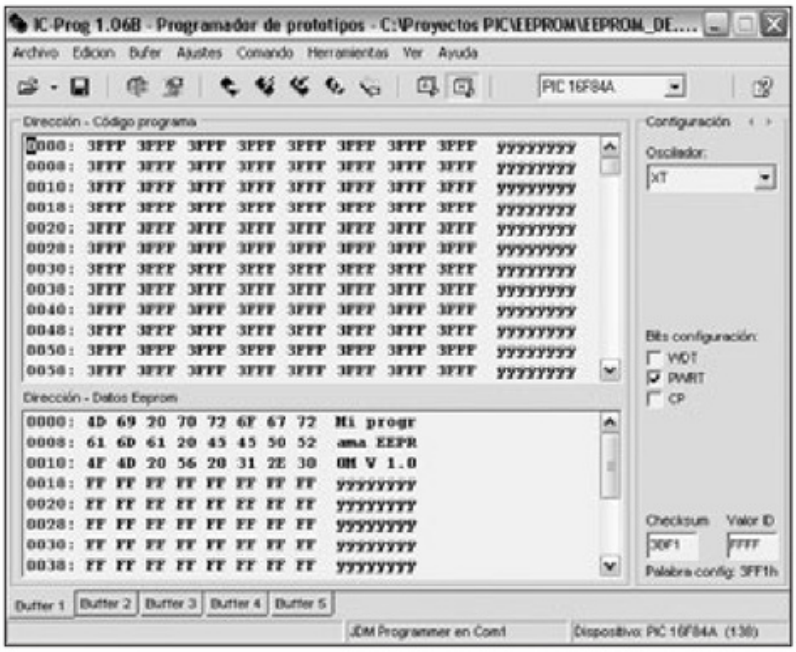


Figura 15. *IC-Prog grabará los datos en la EEPROM al momento de grabar el PIC.*

Podemos colocar datos en el mismo IC-Prog antes de la grabación, es decir, podemos escribir los valores que necesitemos en la propia ventana **Datos EEPROM** de IC-Prog, aunque es lo mismo que hacerlo con la directiva **DE**. De esta manera ya estamos en condiciones de grabar el PIC con datos incluidos en la memoria EEPROM de datos.

Contar el número de veces que se ha usado un circuito

Veamos un ejemplo sencillo del uso de la memoria EEPROM de datos. Simplemente la usaremos para almacenar un número, de tal forma que podamos contar el número de veces que un circuito ha sido encendido o ha sido reiniciado mediante un reset. En resumen, el número de veces que ha sido utilizado dicho circuito.

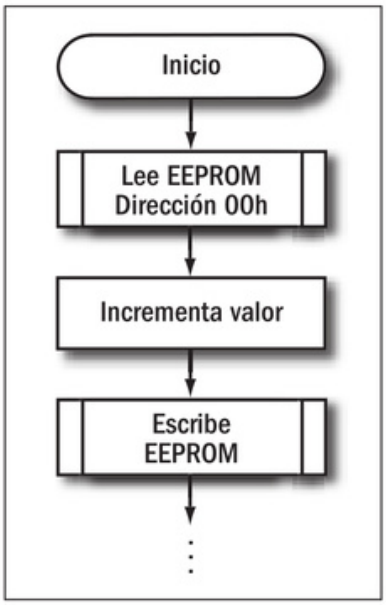


Figura 16. *Con este método podemos contar las veces que se ha empleado un circuito.*

De esta manera, podemos saber cuántas veces se ha usado un circuito. El código fuente sería algo similar a lo siguiente:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      ENDC

      ORG             0x2100              ;Primera dirección de la EEPROM
      DE              h'00'               ;Se grabará con 0

      ORG             00h

      clrw                                ;Borra W para leer la dirección 0
      call            lee_EEPROM          ;Lee la EEPROM
      addlw           d'1'                ;Se suma uno al valor leído
      clrf            EEADR               ;Borra EEADR para acceder a la
dirección 0
      call            escribe_EEPROM      ;Escribe el valor de W

      sleep                               ;Entra en modo de bajo consumo

      #INCLUDE        <EEPROM.INC>

      END
```

Como podemos apreciar, el código es bastante sencillo. Además, utilizaremos nuestra librería para leer y escribir en la memoria EEPROM de datos del PIC16F84A; simplemente al inicio del programa se lee el valor almacenado en la primera dirección de la EEPROM de datos, se incrementa ese valor, y se graba de nuevo, esto se realiza cada vez que el circuito es encendido o reiniciado mediante un reset. La instrucción final en este ejemplo es **sleep**, colocando al microcontrolador en modo de bajo consumo al terminar, pero podemos incorporar esta rutina en cualquiera de nuestros programas. El número de veces que sea empleado este circuito lo podemos obtener si colocamos el PIC en nuestro grabador y leemos el contenido de la memoria. Es importante recordar que esto sólo es posible si no hemos protegido el código con la opción CP (*Code Protect*).

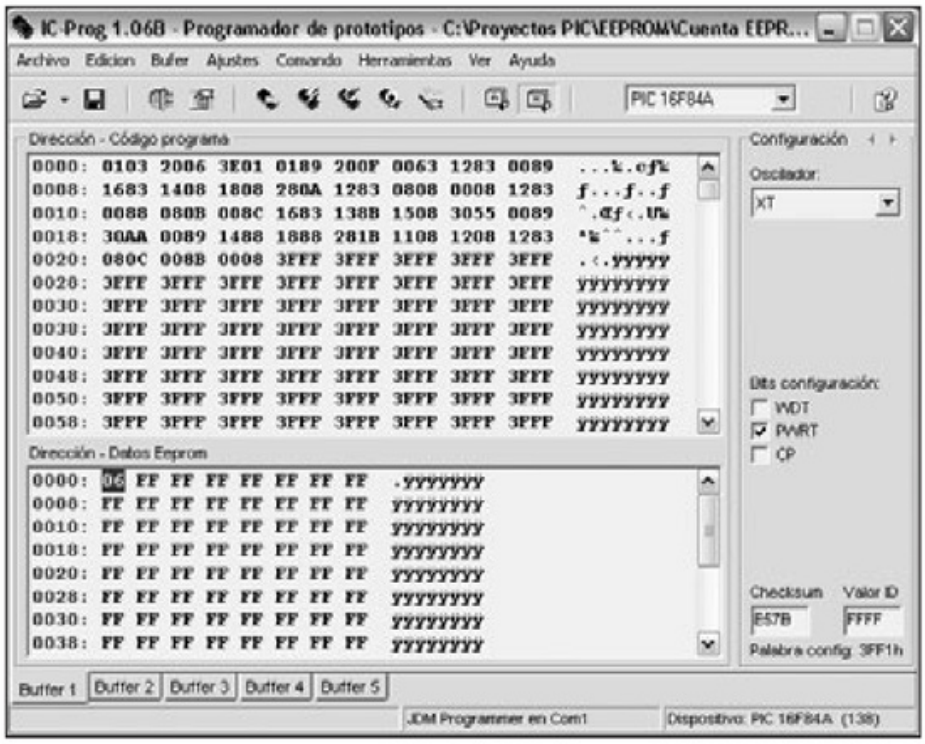


Figura 17. *Podemos leer el conteo en la EEPROM con nuestro grabador e IC-Prog.*

En la **Figura 17** podemos ver cómo al leer el programa del PIC16F84A también se lee el contenido de la EEPROM de datos, la cual nos marca el número de veces que el circuito fue usado. En este caso podemos observar el **06**. Recordemos que este valor está definido en hexadecimal.

MACROS

Las **macros** son tareas automatizadas, es decir, podemos definir una serie de instrucciones como una macro para utilizarlas en tareas rutinarias de forma más fácil y rápida. Recordemos el uso de la directiva **#DEFINE** que ya estudiamos antes, en donde podemos asignar una etiqueta a una instrucción, como por ejemplo:

```
#DEFINE banco1 bsf STATUS, RP0
```

De esta forma, hemos asignado una instrucción completa a una etiqueta. Así, cada vez que escribamos **banco1** en nuestro código fuente, el ensamblador lo sustituirá por la instrucción **bsf STATUS, RP0**. Con las macros podemos hacer algo muy parecido, pero con la ventaja de que las macros nos permiten definir un conjunto completo de instrucciones y directivas, y además admiten argumentos. Las macros deben definirse al principio del código fuente de la siguiente manera:

```
[etiqueta]  MACRO   arg, arg...
     [Instrucciones]
     ENDM
```

Donde **[etiqueta]** será el nombre de la macro, y con ella la invocaremos en el lugar donde sea necesario en nuestro programa; y **arg** son los argumentos de la macro si es que son necesarios. Luego se define el grupo de instrucciones y/o directivas que formarán la macro y, por último, se termina con **ENDM** (end macro) para indicar la finalización de su definición. Después, sólo será necesario invocar la macro mediante su nombre y argumentos en el lugar que necesitemos.

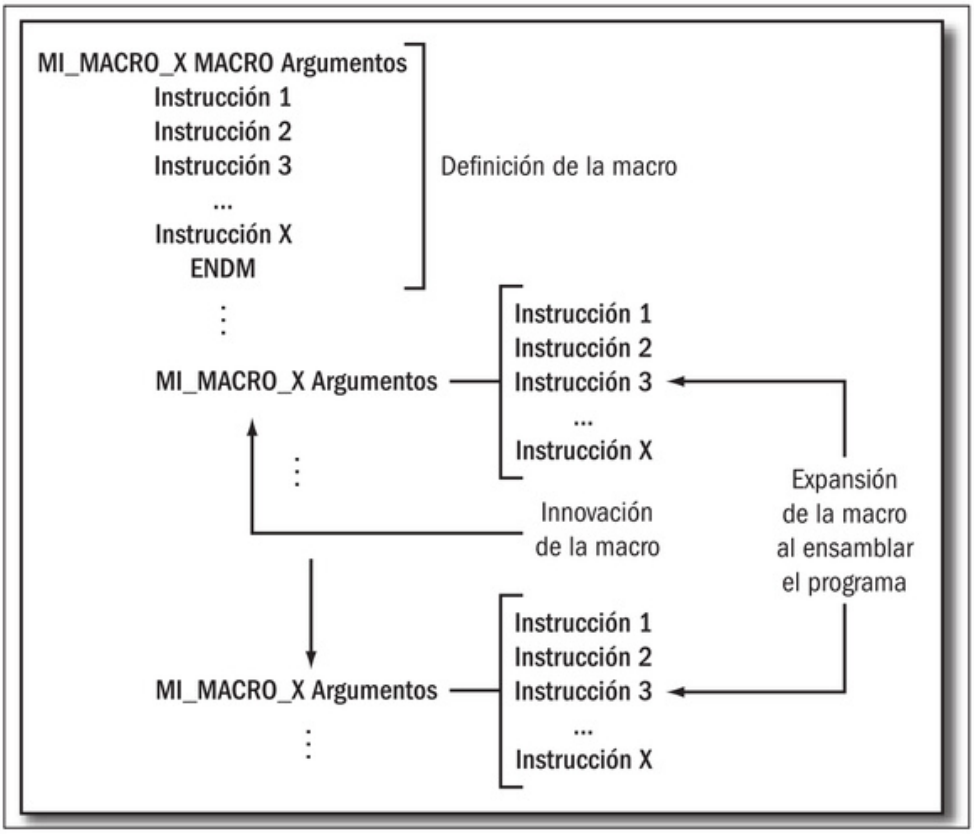


***Figura 18.** Estructura del uso de macros en un programa.*

Veamos un ejemplo sencillo de una macro para comprender bien su funcionamiento. Como sabemos, podemos pasar el contenido de un registro a otro, pero no podemos hacerlo directamente, sino que debemos usar dos instrucciones. Para ello, podemos hacer una sencilla macro y observar cómo se define y usa. Primero tenemos que definir nuestra macro de la siguiente forma:

```
MUEVE_FF          MACRO   reg1, reg2
     movf         reg1, W
```

```
        movwf       reg2
        ENDM
```

En primer lugar tenemos la etiqueta que nombra a la macro. En este ejemplo hemos colocado el nombre **MUEVE_FF**, para indicar que es una macro que moverá el contenido de un registro a otro. Lo indicamos en mayúsculas para diferenciar una macro de las instrucciones en minúsculas, pero también podemos escribirlo en minúsculas. Luego, colocamos la palabra **MACRO**, que le indica al ensamblador que estamos definiendo una macro, y luego vienen los argumentos necesarios, en este caso dos registros que usaremos (**reg** 1 y **reg2**).

Luego se colocan las instrucciones que formarán la macro, en este caso sólo dos. Observemos cómo los argumentos son asignados en las instrucciones. Esto es, cuando invoquemos la macro en algún lugar de nuestro programa deberemos especificar los nombres de los dos registros involucrados y serán insertados en el cuerpo de la macro tal como indican los argumentos. Finalmente, concluimos con **ENDM**. Esta definición debe colocarse al principio de nuestro código fuente o, al menos, antes de invocar la macro en el programa. Para invocarla, lo haremos de la siguiente forma:

```
        .
        .
MUEVE_FF    contador, auxiliar
        .
        .
        .
```

Al ensamblar se generará el siguiente código en el lugar donde se invocó a la macro:

```
        .
        .
        movf        contador, W
        movwf       auxiliar
        .
        .
        .
```

Y así hemos creado una macro para pasar el contenido de un registro a otro. En este caso el contenido del registro **contador** se moverá al registro **auxiliar**.

Por supuesto, en los argumentos podemos definir cualquier par de registros que necesitemos y, de esta forma, nuestra macro nos permitirá automatizar esta tarea de un modo muy sencillo, siempre que necesitemos hacerlo.

Veamos otro sencillo ejemplo de una macro, para cargar un valor al TMR0:

```
CARGA_TMR0          MACRO          carga
      movlw         carga
      movwf         TMR0
      ENDM
```

Así, podemos cargar el TMR0 con el valor definido por el argumento **carga**:

```
      .
      .
      CARGA_TMR0    d'65'
      .
      .
```

Y de este modo la macro cargará el valor 65 decimal al TMR0. Al ensamblar, éste será el código resultante de la macro invocada:

```
      movlw         d'65'
      movwf         TMR0
```

Los ejemplos anteriores son macros muy sencillas, pero nos sirven para comprender su funcionamiento. Las macros también nos permiten automatizar tareas complejas. Por eso, a continuación veremos otro ejemplo de un macro que nos puede resultar de utilidad en nuestros programas.

Macro para comparar dos registros

A continuación desarrollaremos un ejemplo de un macro para comparar dos registros y saltar a diferentes lugares según el resultado. En nuestro caso, compararemos dos registros cualesquiera y determinaremos si son iguales: si el primero es mayor al segundo, o si es menor. Dependiendo del resultado, la macro saltará a la dirección correspondiente del programa. De esta manera, definiremos nuestra macro tal como observamos en el código a continuación:

```
COMPARA_FF   MACRO   reg1, reg2, reg1_igual_reg2, reg1_mayor_reg2,
                     reg1_menor_reg2
     movf            reg2, W
     subwf           reg1, W
     btfsc           STATUS, Z
     goto            reg1_igual_reg2
     btfsc           STATUS, C
     goto            reg1_mayor_reg2
     goto            reg1_menor_reg2
     ENDM
```

En esta macro se hace la comparación de los registros definidos por los argumentos **reg1** y **reg2** mediante la resta **reg1-reg2** y observando el resultado en las banderas Z y C del registro STATUS. Si los registros son iguales, se saltará a la dirección definida por el argumento **reg1_igual_reg2**, pero si el primer registro es mayor, se saltará a **reg1_mayor_reg2** y, en caso de que sea menor, a **reg1_menor_reg2**.

Un ejemplo de invocación de esta macro es:

```
     COMPARA_FF contador1, contador2, iguales, mayor, menor
     .
     .
     .
iguales
     .
     .
mayor
     .
     .
menor
     .
     .
```

Mediante esta macro compararemos los valores contenidos en los registros **contador1** y **contador2**, y se saltará a las etiquetas **iguales**, **mayor** o **menor**, respectivamente, dependiendo de la comparación de los registros. Como sabemos, las etiquetas pueden estar en cualquier lugar del programa, incluso inmediatamente después de la invocación a la macro. De esta forma podemos usar sólo una comparación, por ejemplo, si sólo nos interesa saber si el **reg1** es mayor al **reg2**, invocaremos la macro de esta forma:

```
     COMPARA_FF contador1, contador2, no_mayor, mayor, no_mayor
no_mayor
     .
     .
mayor
     .
     .
     .
```

Si el contenido de **contador1** es mayor a **contador2**, saltará a la etiqueta **mayor**. Si no, sigue inmediatamente después de la invocación a la macro en la etiqueta **no_mayor**.

Librería de macros

Hemos colocado las tres macros estudiadas en una librería llamada **MACROS.INC** que podemos descargar de **www.redusers.com**. De esta manera, podemos usar la librería para invocar las macros en nuestros programas cuando lo necesitemos, sólo debemos recordar incluir la librería **MACROS.INC** al principio del código fuente, ya que de hacerlo al final del código, como con las librerías de subrutinas, al ensamblar nos dará errores.

Podemos escribir nuestras propias macros para definir tareas a las que recurrimos frecuentemente, por lo que es conveniente que ampliemos la librería de macros con todas las que nos puedan ser útiles. La ventaja de usar macros es que no generan código mientras no sean invocadas. La diferencia de usar macros en lugar de subrutinas es que en las primeras no hay salto hacia la subrutina, sino que se invocan en el mismo lugar donde son requeridas. Tampoco se usa instrucción de retorno, ya que el programa sigue inmediatamente después de la macro. Debemos recordar, también, que la diferencia con las subrutinas es que las macros se expanden cada vez que son invocadas, es decir, los códigos que generen las macros estarán repetidos en todos los lugares donde sean invocadas.

RESUMEN

En este capítulo hemos aprendido el uso de algunas funciones del PIC16F84A, algunas de ellas nos serán de gran utilidad en nuestros circuitos, como el uso de la memoria EEPROM, la cual es una memoria de usuario no volátil. Además, aprendimos a usar el WDT y el modo de bajo consumo, con lo cual podemos hacer circuitos que utilizarán poca energía. Otro punto importante que estudiamos fue el uso de macros, que nos permiten automatizar algunas tareas.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es el direccionamiento indirecto?

2 ¿Para qué sirve el registro FSR?

3 ¿Para qué sirve el registro INDF?

4 ¿Qué sucede cuando el WDT se desborda?

5 ¿Cuál es el período del WDT sin usar prescaler?

6 ¿Cuál es el período del WDT si le asignamos un prescaler de 1:32?

7 ¿Para qué sirven los resistores de Pull-up internos del Puerto B?

8 ¿La memoria EEPROM de datos es una memoria volátil o no volátil?

9 ¿Cuántos bytes tiene la EEPROM de datos del PIC16F84A?

10 ¿Qué es una macro?

PRÁCTICAS

1 Tome el dado que diseñamos en el Capítulo 6 y agregue una rutina para grabar en la memoria EEPROM de datos las veces que se usa el circuito, tal como lo vimos en el ejemplo de este capítulo. Además, el circuito debe bloquearse cuando el valor grabado en la EEPROM sea mayor a 14h (20 decimal). Al bloquearse, irá a una subrutina donde se encenderán todos los leds y luego entrará en modo de bajo consumo (sleep), del cual ya no saldrá.

2 Tome la librería BINABCD.INC y convierta la rutina para la conversión de binario a BCD en ella en una macro, que puede resultar útil para los programas en donde se usa la conversión una sola vez.

3 Tome la macro que generó en el punto anterior y agréguela a la librería de macros MACROS.INC

4 Diseñe y agregue a la librería de macros todas las macros que se le ocurran y que le puedan ser útiles en sus programas.

Interrupciones

Dentro de las funciones del PIC16F84A y los demás microcontroladores PIC existe una función muy especial: las interrupciones. En este capítulo estudiaremos qué son, cuántos tipos hay y cómo funcionan. Además, aprenderemos cuáles son sus posibles aplicaciones en el diseño de proyectos.

¿QUÉ SON LAS INTERRUPCIONES?

Una **interrupción** es, precisamente, una forma de "interrumpir" al microcontrolador, para que atienda algún proceso de manera inmediata. Cuando se genera o se solicita una interrupción, el microcontrolador deja lo que está haciendo y va directamente a atender el proceso indicado por la interrupción. Cuando termina ese proceso, el microcontrolador continúa lo que estaba haciendo antes, en el mismo punto en donde lo dejó. El PIC16F84A tiene cuatro mecanismos diferentes de interrupción:

- Interrupción externa en el pin RB0/INT (**INT**).
- Interrupción por cambio de estado en las líneas altas del Puerto B (**RBI**).
- Interrupción por desbordamiento del TMR0 (**T0I**).
- Interrupción por finalización de escritura en EEPROM (**EEI**).

Cada mecanismo de interrupción tiene principalmente dos bits de control, uno que sirve para habilitar la interrupción en cada caso, y otro para indicar que se ha generado el evento que pide la interrupción. Además de esto, existe un bit que habilita o deshabilita todas las interrupciones. Independientemente del tipo de interrupción que ocurra, el programa saltará a la dirección **04h** de la memoria de programa que, como ya sabemos, es el **vector de interrupción**, y ejecutará las instrucciones que comiencen en esa dirección hasta encontrar la instrucción **retfie**, que es la que hará regresar al punto del programa donde se generó la interrupción.

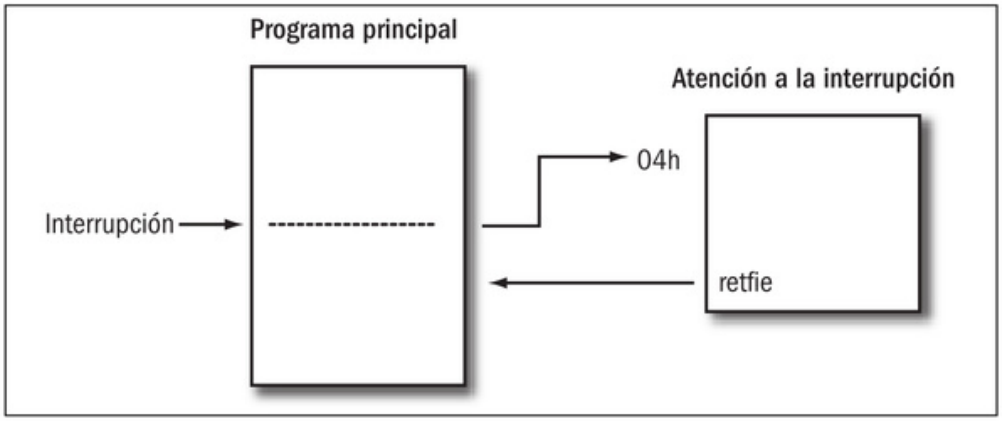


Figura 1. *Cuando se genere una interrupción, el programa saltará al vector de interrupción.*

La interrupción puede generarse en cualquier momento y en cualquier lugar del programa principal. El registro principal que controla las interrupciones es INTCON, que está en la dirección 0Bh y 8Bh del área SFR de la memoria de datos.

BIT 7	BIT 6	BIT 5	BIT4	BIT3	BIT 2	BIT 1	BIT 0
GIE	EEIE	TOIE	INTE	RBIE	TOIF	INTF	RBIF

Tabla 1. *El registro INTCON.*

Bit 7 GIE (*Global interrupt enable*): éste es el bit que habilita todas las interrupciones:

GIE=0 Todas las interrupciones están deshabilitadas
GIE=1 Todas las interrupciones están habilitadas

Bit 6 EEIE (*EEPROM write complete interrupt enable*): con este bit se habilita la interrupción por finalización de escritura en la memoria EEPROM de datos:

EEIE=0 Interrupción por finalización de escritura en EEPROM deshabilitada
EEIE=1 Interrupción por finalización de escritura en EEPROM habilitada

Bit 5 T0IE (*timer 0 interrupt enable bit*): bit de habilitación de la interrupción por desbordamiento del Timer 0:

T0IE=0 Interrupción por desbordamiento del TMR0 deshabilitada
T0IE=1 Interrupción por desbordamiento del TMR0 habilitada

Bit 4 INTE (*external INT enable bit*): habilitación de la interrupción externa INT:

INTE=0 Interrupción INT deshabilitada
INTE=1 Interrupción INT habilitada

Bit 3 RBIE (*RB port change interrupt enable*): habilitación de la interrupción RBI:

RBIE=0 Interrupción por cambio en las líneas del puerto B deshabilitada
RBIE=1 Interrupción por cambio en las líneas del puerto B habilitada

Bit 2 T0IF (*timer 0 overflow interrupt flag bit*): bandera de desbordamiento del Timer 0:

T0IF=0 El Timer 0 no se ha desbordado
T0IF=1 El Timer 0 se ha desbordado, si **T0IE=1** se produce la interrupción, debe borrarse por software

III ¿POR QUÉ INTERRUMPIR?

Los mecanismos de interrupción son muy importantes en los sistemas de microprocesador o microcontrolador, ya que permiten atender procesos de forma inmediata ante un evento determinado, sin tener que esperar a que el proceso actual termine. Prácticamente todos los microprocesadores y microcontroladores actuales tienen mecanismos de interrupción.

Bit 1 INTF (*external interrupt flag bit*): bandera de interrupción externa INT:

INTF=0 No se ha producido la interrupción INT
INTF=1 Se ha producido la interrupción INT, debe borrase por software

Bit 0 RBIF (*RB port change interrupt flag bit*): bandera de interrupción RBI:

RBIF=0 No se ha producido la interrupción RBI
RBIF=1 Se ha producido la interrupción RBI, debe borrarse por software

Como podemos apreciar, el bit GIE habilita o deshabilita todas las interrupciones. Cuando este bit está a 1, permite que ocurra cualquier interrupción. Cuando ocurre una interrupción, este bit se pone automáticamente a 0, para evitar que suceda otra interrupción mientras se atiende la que ya se produjo. Al terminar de atender la interrupción actual, mediante la instrucción de retorno **retfie**, este bit se pone automáticamente a 1 nuevamente para habilitar otra vez todas las interrupciones.

Los bit que finalizan con **IE** (*Interrupt Enable*), como **EEIE**, **T0IE**, **INTE**, **RBIE**, sirven para habilitar individualmente cada uno de los mecanismos de interrupción y los usaremos para habilitar sólo las interrupciones que vayamos a utilizar. Los bits terminados en **IF** (*interrupt Flag*), como **T0IF**, **INTF**, **RBIF**, son banderas que indican cuándo se ha producido la interrupción en cada caso. Si bien falta la bandera de interrupción por finalización de escritura en EEPROM, ésta se encuentra en el bit 4 del registro EECON1, y es el llamado **EEIF**, como ya hemos estudiado.

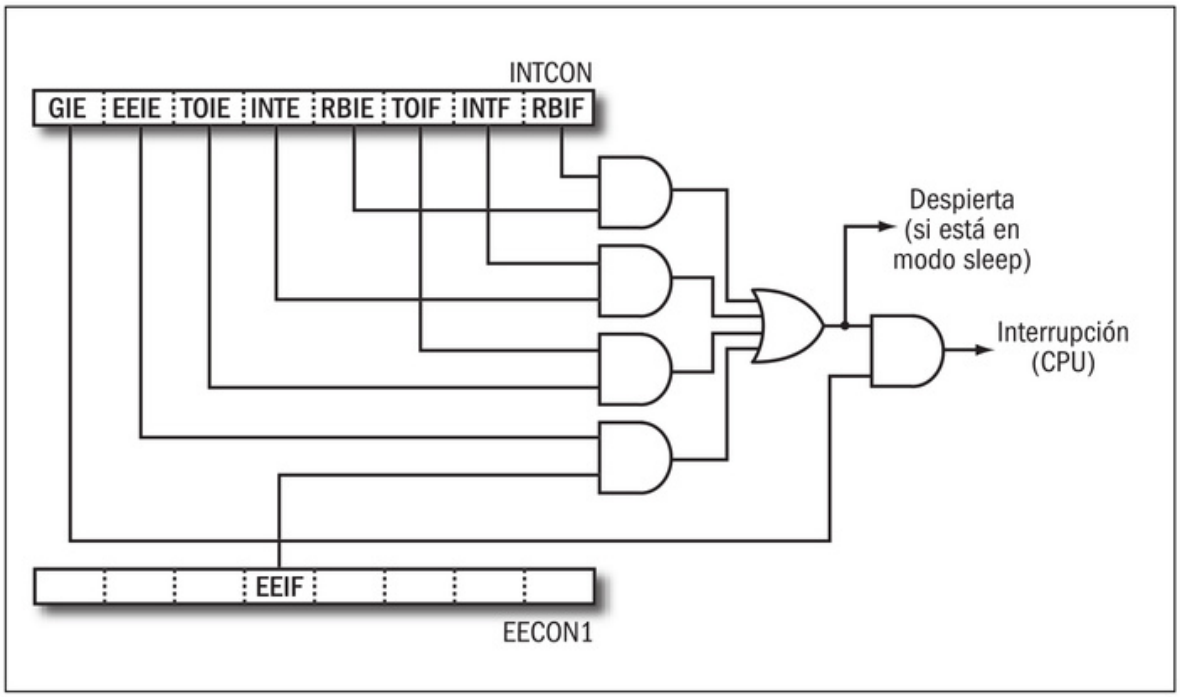


***Figura 2.** Lógica de las interrupciones del PIC16F84A.*

Mecanismo de funcionamiento de interrupciones

Cuando ocurre cualquier interrupción de cualquiera de los cuatro mecanismos de interrupción del PIC16F84A, sucede una serie de eventos que llevan a cabo la atención a la interrupción, cuyo proceso lo vemos representado en la **Figura 3**.

1. Se guarda el valor actual del contador de programa (PC) en la pila.
2. El bit GIE se pone a 0.
3. Se coloca el valor 04h en el contador de programa.
4. Comienza la ejecución del programa desde la dirección 04h (vector de interrupción).

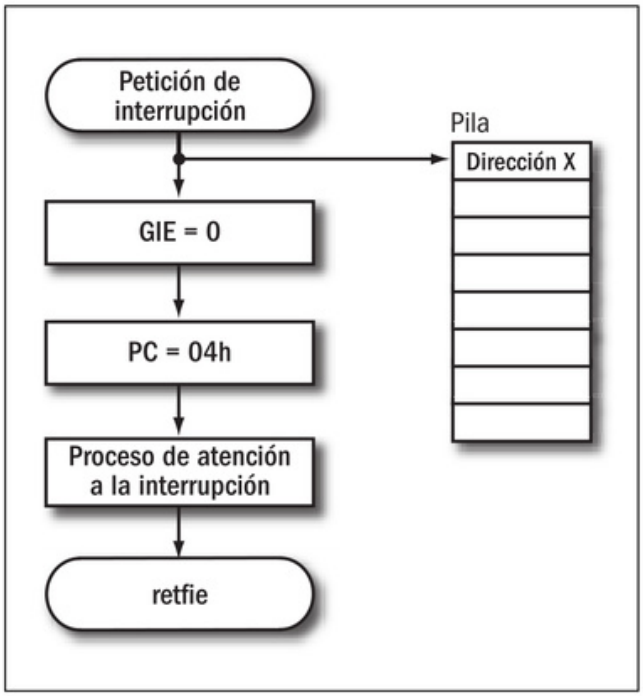


***Figura 3.** Mecanismo general de acción de una interrupción.*

Una vez terminada la rutina de atención a la interrupción, la instrucción **retfie** tomará el valor que se almacenó en la pila, regresando de esa forma al lugar del programa donde se generó la interrupción y el bit GIE se pone a 1 y así habilita, nuevamente, todas las interrupciones.

{} OTROS RETORNOS DE INTERRUPCIÓN

Para retornar de una interrupción al punto donde se generó, se debe usar la instrucción **retfie**, la cual automáticamente activa o pone a 1 el bit GIE del registro INTCON y habilita de nuevo todas las interrupciones. También es posible regresar de una interrupción con las instrucciones **return** o **retlw**, pero éstas no ponen a 1 el bit GIE.

INTERRUPCIÓN EXTERNA INT

El primer mecanismo de interrupción que estudiaremos aquí es la **interrupción externa**, que graficamos en el diagrama de la **Figura 4**, y que ocurrirá cuando en el pin RB0/INT se tenga un flanco de subida o bajada, a elección del diseñador.

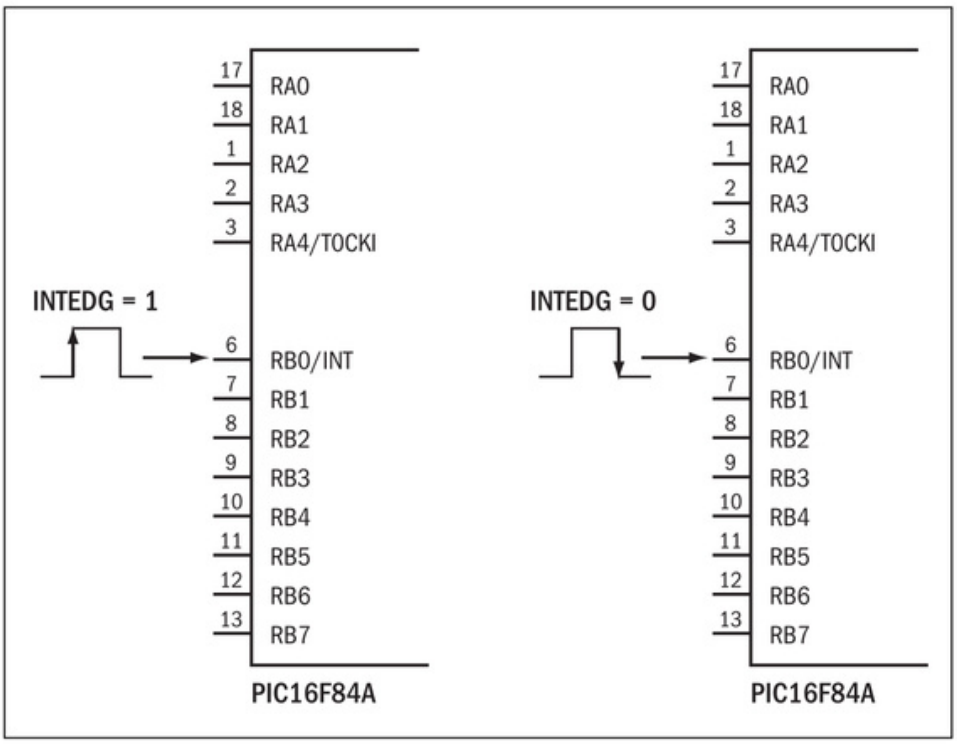


***Figura 4.** La interrupción INT se activa mediante un flanco en el pin RB0/INT.*

El flanco de la señal que producirá la interrupción se elige con el bit INTEDG del registro OPTION.

BIT 7	BIT 6	BIT 5	BIT4	BIT3	BIT 2	BIT 1	BIT 0
RBPU'	**INTEDG**	T0CS	T0SE	PSA	PS2	PS1	PS0

***Tabla 2.** EL bit INTEDG del registro OPTION.*

Bit 6 INTEDG (*interrupt edge select bit*): bit de selección de flanco en RB0/INT:

INTEDG=0 Flanco de bajada en RB0/INT
INTEDG=1 Flanco de subida en RB0/INT

Así, cuando se produzca un flanco de subida o bajada (según el valor del bit INTEDG) en el pin RB0/INT y si están habilitadas las interrupciones globales (**GIE=1**) y la interrupción externa (**INTE=1**), se producirá una interrupción y se pondrá en marcha el proceso que atenderá a dicha interrupción saltando a la dirección 04h.

Veamos un ejemplo del uso de esta interrupción para entender mejor su funcionamiento. Tomaremos como base el circuito de la **Figura 5**, en el cual hemos colocado un pulsador en el pin RB0/INT, además de tener conectado nuestro display LCD.

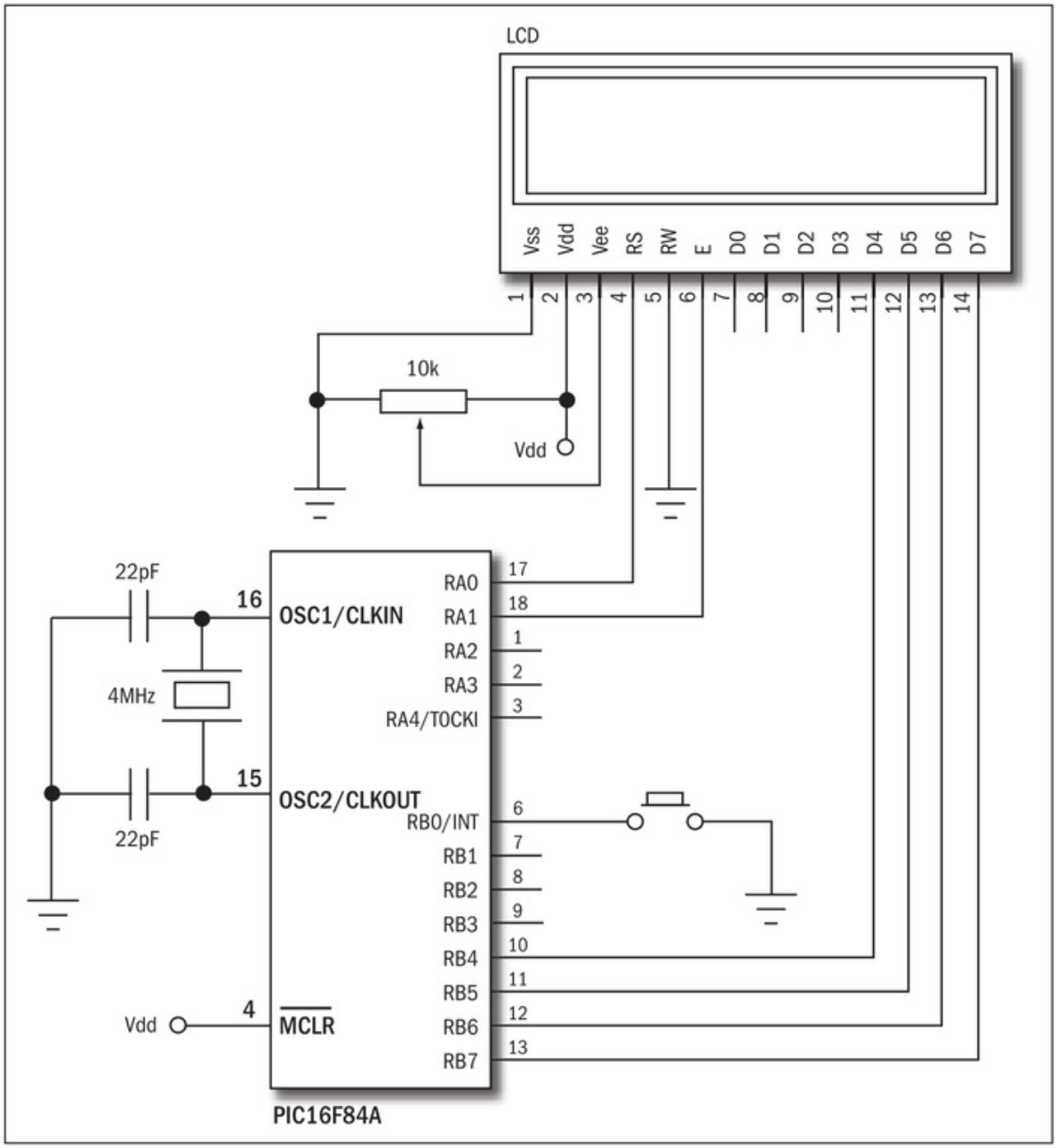


***Figura 5.** Circuito para comenzar a usar interrupciones externas en RB0/INT.*

El programa es muy sencillo, sólo se incrementará un contador cada vez que presionemos el pulsador y se produzca una interrupción. La cuenta se mostrará en el LCD:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
```

```
     #INCLUDE <P16F84A.INC>

     CBLOCK 0x0C
     contador
     ENDC

     ORG          00h                  ;Origen en la dirección 0
     goto         inicio               ;Salta el vector de interrupción

     ORG          04h                  ;Origen en la dirección 4
     goto         interrupcion         ;Ve a la rutina de interrupción

inicio
     call         LCD_inicializa
     bsf          STATUS, RP0          ;Acceso al banco 0
     bsf          TRISB, 0             ;RB0/INT se configura como
entrada
     bcf          OPTION_REG, NOT_RBPU ;Se activan los Pull-ups internos
     bcf          OPTION_REG, INTEDG   ;Se configura INT por flanco de
bajada
     bcf          STATUS, RP0          ;Acceso al banco 1
     clrf         contador             ;Inicializa el contador

     call         display              ;Se envía al LCD
     movlw        b'10010000'
     movwf        INTCON               ;Habilita INT y las
interrupciones globales
loop
     sleep                             ;Entra en sleep hasta que se
produce la interrupción
     goto         loop                 ;Se genera un bucle

display                                ;Subrutina para mostrar la cuenta
en el LCD
     call         borra_display
     movf         contador, W
     call         BINaBCD
     movf         BCDcentenas, W
     btfsc        STATUS, Z
     goto         decenas
```

```
     addlw          '0'
     call           LCD_caracter
     goto           muestra_decenas
decenas
     movf           BCDdecenas, W
     btfsc          STATUS, Z
     goto           unidades
muestra_decenas
     movf           BCDdecenas, W
     addlw          '0'
     call           LCD_caracter
unidades
     movf           BCDunidades, W
     addlw          '0'
     call           LCD_caracter
     return

;Rutina de atención a la interrupción:

interrupcion
     call           LCDretardo20ms      ;Retardo antirrebotes
     btfsc          PORTB, 0            ;¿Falso disparo?
     goto           termina_int         ;Si, regresa
     incf           contador, F         ;No, incrementa el contador
     call           display             ;Muéstralo en el LCD
termina_int
     bcf            INTCON, INTF        ;Borrael bit INTF
     retfie                             ;Regresa de la interrupción

     #INCLUDE       <LCD4BITS.INC>
     #INCLUDE       <BINABCD.INC>

     END
```

Si analizamos este código encontraremos cosas nuevas. En primer lugar se marca el origen en la dirección 00h y se salta a la etiqueta **inicio** para saltar el vector de interrupción. Luego se establece el origen en la dirección 04h, que es precisamente el vector de interrupción, siendo el lugar al cual saltará el programa al producirse la interrupción INT. En la dirección 04h se salta a la rutina que atenderá a la interrupción cuando ésta se produzca. Se activan los Pull-ups internos, ya que usaremos

RB0/INT como entrada, y se habilita la interrupción INT y las interrupciones globales (GIE) para permitir que la interrupción INT se lleve a cabo.

Luego, simplemente se coloca el microcontrolador en modo de bajo consumo para esperar a que se produzca la interrupción al presionar el pulsador. Cuando esto sucede, se la atiende incrementando el valor del contador y mostrándolo en el LCD. Es muy importante notar cómo, antes de que se termine de atender a la interrupción, se borra el bit INTF ya que si no lo hacemos, al salir de la interrupción, ésta se volverá a ejecutar inmediatamente porque si este bit queda a 1 el PIC asume que se ha producido de nuevo la interrupción.

Recordemos que las banderas de interrupción no se borran solas, sino que debemos eliminarlas por software. Finalmente, como podemos observar, con la instrucción **retfie**, regresa de la interrupción y entra en modo de bajo consumo de nuevo hasta que se produce nuevamente la interrupción.

Manejo adecuado de las interrupciones

Una interrupción puede ocurrir en cualquier momento y en cualquier parte del programa, por lo que durante el proceso de atención a la interrupción pueden alterarse los contenidos de los registros importantes con los que se está trabajando en el momento de producirse la interrupción, sobre todo del registro W y del registro STATUS. Si se cambia su contenido durante la atención a una interrupción, al regresar de ella, el programa no funcionará correctamente, por lo que cuando ocurre una interrupción es conveniente guardar sus valores para luego restaurarlos antes de regresar de la interrupción.

Para guardar el contenido del registro W basta con definir un registro auxiliar para este fin, por ejemplo, un registro llamado **guardaW**:

```
movwf       guardaW
```

III ¿GUARDAR O NO GUARDAR?

Guardar el registro W, STATUS y otros registros de la memoria de datos al producirse una interrupción se hace sólo cuando realmente es necesario, es decir, cuando el programa debe seguir atendiendo otros procesos después de la interrupción, momento en el que se necesita no alterar los valores de estos registros.

```
Y con esto queda guardado el contenido del registro W
```

Pero, sin embargo, para el registro STATUS es algo más complicado. Veamos qué sucede si lo intentamos guardar de la forma que muestra el siguiente código:

```
      movf          STATUS, W
      movwf         guardaSTATUS

Es incorrecto...
```

La instrucción **movf** puede alterar el valor de la bandera de cero (Z) y corromper, así, el contenido del registro STATUS. Para resolver este inconveniente, Microchip recomienda hacerlo de la siguiente forma:

```
      swapf         STATUS, W
      movwf         guardaSTATUS
```

De esta manera no se corrompe el contenido del registro STATUS, ya que la instrucción **swapf** no afecta ninguna bandera y **movwf** tampoco, por lo que el registro STATUS se guarda correctamente. También deben guardarse los contenidos de otros registros que puedan ser cambiados por la rutina de atención a la interrupción, aunque se recomienda no usar los mismos registros en el programa principal y en las rutinas de interrupciones.

Recuperar el valor de STATUS se hace también mediante una instrucción **swapf**, que invierte nuevamente los nibbles en él y lo deja como estaba antes de la interrupción. La rutina completa del proceso sería:

```
      movwf         guardaW             ;W queda guardado
      swapf         STATUS, W
      movwf         guardaSTATUS        ;STATUS queda guardado
      movf          regA, W             ;(Esto es opcional si necesitamos
      movwf         guardaRegA          ; guardar otro registro)

(Aquí la subrutina de atención a la interrupción)

      swapf         guardaSTATUS, W
```

```
movwf       STATUS              ;Restaura el contenido de STATUS
swapf       guardaW, F
swapf       guardaW, W          ;Restaura el contenido de W
```

Y así se guardan y restauran los registros importantes durante una interrupción. En el proceso de restauración tampoco se usa la instrucción **movf**.

Además de guardar los registros importantes, también debemos tener algún mecanismo que nos permita identificar el tipo de interrupción. Todas las interrupciones ocasionan que el programa salte al vector de interrupción, es decir, a la dirección 04h, por lo que si estamos usando más de una interrupción en nuestro programa debemos identificar cuál fue la interrupción que ocurrió y así ir al proceso de atención adecuado. Para lograrlo sólo basta con verificar los bits o banderas de interrupción, es decir, el estado de los bits INTF, RBIF, T0IF, o EEIF, según el caso. Por ejemplo:

```
btfsc       INTCON, INTF
goto        interrupcionINT
btfsc       INTCON, RBIF
goto        interrupcionRBI
```

Así nos asegurarnos de que se ejecutará la atención correcta para cada interrupción. Por supuesto, esto dependerá del número de interrupciones y de cuáles sean las que estamos usando: si usamos sólo una, esto no será necesario. De esta forma, se manejan de modo correcto las interrupciones sin que el programa haga cosas que no debe o sin que se salga de control por datos corruptos.

INTERRUPCIÓN RBI

La **interrupción RBI** tiene que ver con un cambio de estado en los pines altos (RB4 a RB7) del puerto B. Para ello hay que colocar el bit RBIE a 1 al igual que el bit GIE. Esta interrupción se llevará a cabo en cualquier cambio en el estado de las cuatro líneas más altas del Puerto B, es decir, cuando se pasa de 1 a 0 ó de 0 a 1. Veamos un ejemplo sencillo de estas interrupciones. Tomemos como base el circuito de la **Figura 6** a partir del cual diseñaremos un programa que lea los pulsadores conectados a las líneas altas mediante interrupciones RBI. En el display se mostrará el nombre de la línea del pulsador presionado.

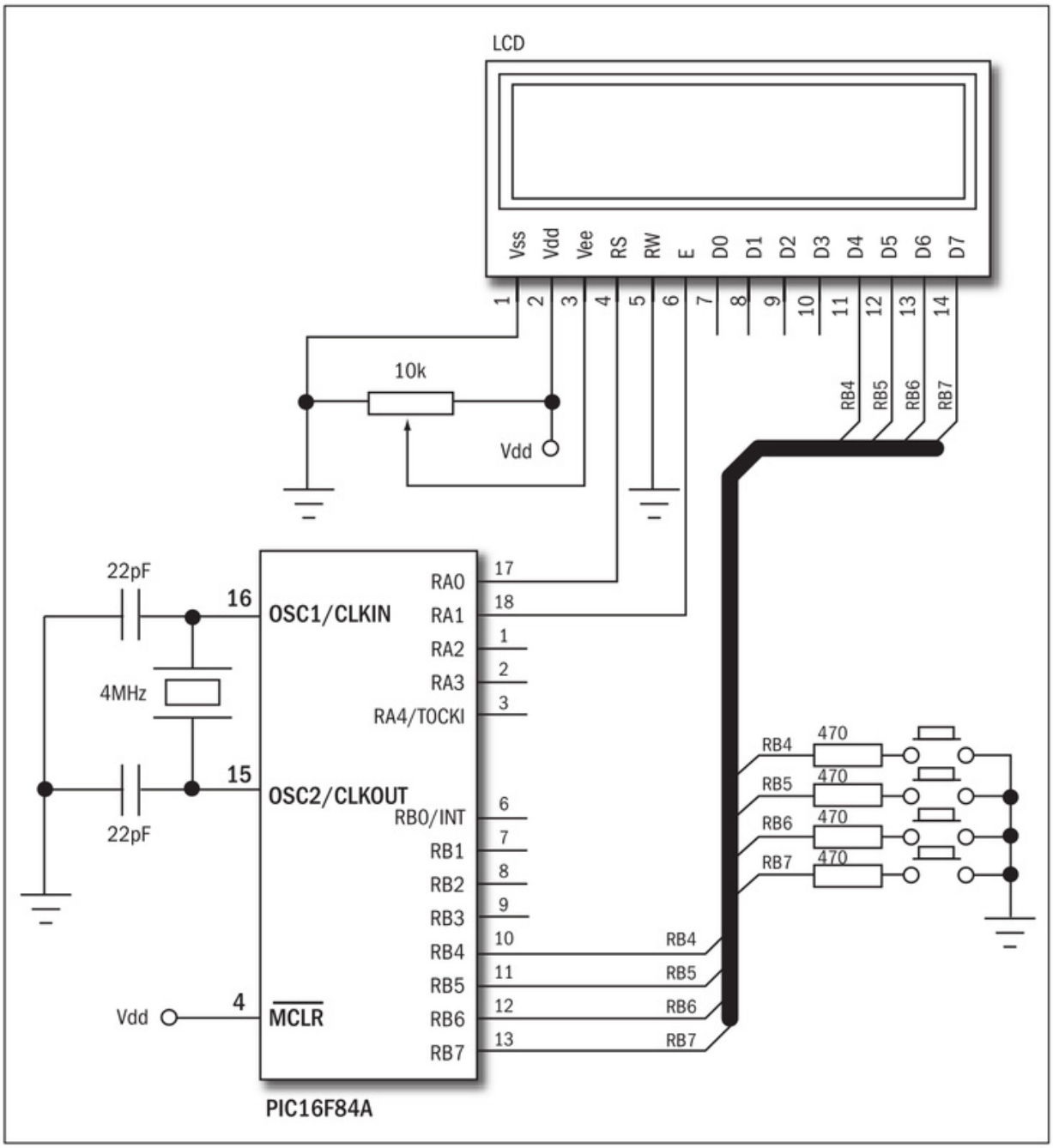


Figura 6. *Circuito para comenzar a usar interrupciones RBI.*

Podemos apreciar un detalle curioso en este circuito. Como necesitamos introducir los cambios de estado en las líneas altas del puerto B del PIC16F84A para producir las interrupciones, pero al mismo tiempo necesitamos usar el LCD, emplearemos un pequeño truco para conectar ambas cosas. Recordemos que para enviar datos o comandos al LCD debemos configurar las líneas como salida, pero en el tiempo que no enviamos nada al LCD podemos usarlas para otros fines, en este caso, como entradas para los pulsadores. Entonces, debemos colocar los resistores de 470 Ohms en serie con los pulsadores, para evitar un cortocircuito en las líneas cuando se está escribiendo en el LCD, y de esta forma podremos usar el LCD y los pulsadores al mismo tiempo.

El código fuente será el siguiente:

```
     __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

     PROCESSOR PIC16F84A
     #INCLUDE <P16F84A.INC>

     CBLOCK 0x0C
     ENDC

     ORG            00h
     call           LCD_inicializa
     goto           inicio

     ORG            04h
     goto           intRBI

inicio
     bsf            STATUS, RP0           ;Acceso al banco 1
     bcf            OPTION_REG, NOT_RBPU  ;Activa Pull-ups internos
     movlw          b'11110000'
     movwf          TRISB                 ;Líneas altas como entrada
     bsf            INTCON, RBIE          ;Habilita interrupción RBI
     bsf            INTCON, GIE           ;Habilita interrupciones globales
     bcf            STATUS, RP0           ;Acceso al banco 0
loop
     sleep                                ;Espera hasta la interrupción
     goto           loop

intRBI                                    ;Subrutina de interrupción
     call           LCDretardo20ms        ;Retardo de LCD4BITS.INC como
anti-rebotes
     btfss          PORTB, 4              ;¿Se presiono RB4?
     goto           LCD_RB4               ;Si, ve a la rutina adecuada
     btfss          PORTB, 5              ;No, ¿Se presionó RB5?
     goto           LCD_RB5               ;      .
     btfss          PORTB, 6              ;      .
     goto           LCD_RB6               ;      .
     btfss          PORTB, 7
     goto           LCD_RB7
```

```
     goto          fin_int              ;Es falso disparo, termina
LCD_RB4
     movlw         mensaje1
     call          LCD_mensaje          ;Coloca mensaje de RB4
espera4                                 ;Espera que se deje de pulsar
     btfss         PORTB, 4
     goto          espera4
     goto          fin_int              ;Termina interrupción
LCD_RB5                                 ;      .
     movlw         mensaje2             ;      .
     call          LCD_mensaje          ;      .
espera5
     btfss         PORTB, 5
     goto          espera5
     goto          fin_int
LCD_RB6
     movlw         mensaje3
     call          LCD_mensaje
espera6
     btfss         PORTB, 6
     goto          espera6
     goto          fin_int
LCD_RB7
     movlw         mensaje4
     call          LCD_mensaje
espera7
     btfss         PORTB, 7
     goto          espera7
fin_int
     call          borra_display        ;Borra la pantalla
     bcf           INTCON, RBIF         ;Borra bandera de interrupción
```

III INTERRUPCIONES Y SUBRUTINAS

El proceso de atención a una interrupción es muy parecido al de las subrutinas: cuando se genera una interrupción es como si se llamara a la subrutina que comienza en la dirección 04h, el vector de interrupción. Para finalizar la atención de esta interrupción se debe regresar al punto donde fue "llamada" mediante la instrucción **retfie**.

```
     retfie                              ;Regresa de la interrupción

mensajes                                 ;Mensajes a mostrar
     addwf          PCL, F
mensaje1
     DT             "Pulsado: RB4", 00h
mensaje2
     DT             "Pulsado: RB5", 00h
mensaje3
     DT             "Pulsado: RB6", 00h
mensaje4
     DT             "Pulsado: RB7", 00h

     #INCLUDE       <LCD4BITS.INC>
     #INCLUDE       <LCDMENSAJES.INC>
     END
```

Así podemos apreciar el funcionamiento de las interrupciones RBI, que nos resulta de mucha utilidad cuando se requiera lectura de pulsadores o teclados. Es importante notar cómo con las interrupciones nos olvidamos por completo de la técnica de **muestreo** para leer los pulsadores, ya que la lectura de éstos se genera de forma inmediata al producirse una interrupción. Esto tiene muchas ventajas porque no debemos detener el programa para muestrear las líneas de los pulsadores y el programa puede atender otros procesos. Podemos descargar el código fuente y el código máquina de este ejemplo (**RBI.asm** y **RBI.hex**) del sitio **www.redusers.com** para su estudio y comprobación.

TECLADOS

En el **Capítulo 7** estudiamos los displays LED y LCD, para la comunicación del PIC hacia el usuario, pero también es fundamental la comunicación del usuario al PIC. Para ello hemos estudiado el uso de pulsadores, pero en ocasiones necesitaremos más de unos cuantos de ellos, lo cual conforma un **teclado**. Ahora estudiaremos la estructura, la conexión y la utilización de teclados con el PIC16F84A.

Figura 7. *Los teclados son dispositivos de entrada en los sistemas digitales.*

Si quisiéramos conectar varios pulsadores, uno en cada línea de entrada de nuestro microcontrolador, podríamos tener una cantidad muy limitada, pero si necesitamos conectar muchos pulsadores (un teclado) usando pocas líneas del microcontrolador podemos recurrir a la técnica de los **teclados matriciales**.

Un teclado matricial es una disposición de pulsadores en filas y columnas, de tal forma que permite la conexión al microcontrolador usando pocas líneas de entrada o salida. En la **Figura 8** tenemos un ejemplo de la estructura de un teclado matricial de 4x4.

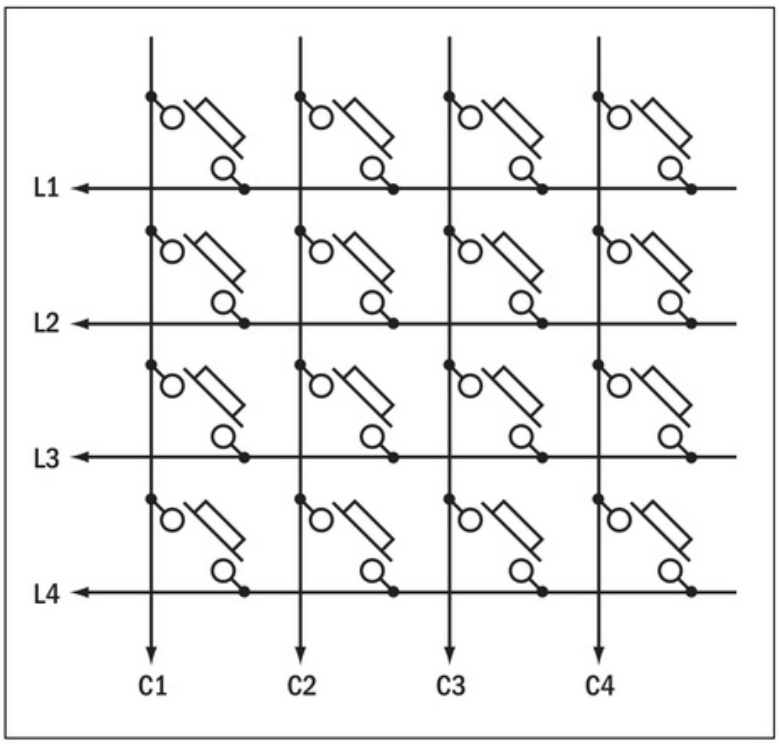


Figura 8. *Una matriz de 4x4 pulsadores forma un teclado de 16 teclas.*

De esta manera, mediante 8 líneas, se puede leer una matriz de 16 pulsadores y por eso se lo llama de 4x4. El teclado está organizado en cuatro filas y cuatro columnas, y de esta forma se conectará a un puerto de 8 líneas del microcontrolador. En nuestro caso, como el PIC16F84A cuenta con un solo puerto de 8 bits, conectaremos nuestro teclado al Puerto B, como muestra la **Figura 9**.

III LA IMPORTANCIA DE LOS TECLADOS

Los teclados son dispositivos de entrada muy útiles en los sistemas actuales. Basta con observar cuántos aparatos electrónicos modernos cuentan con uno de ellos, por ejemplo, las computadoras, los teléfonos, los celulares, un equipo de sonido, prácticamente todos los aparatos digitales cuentan con alguna especie de teclado para controlar sus funciones.

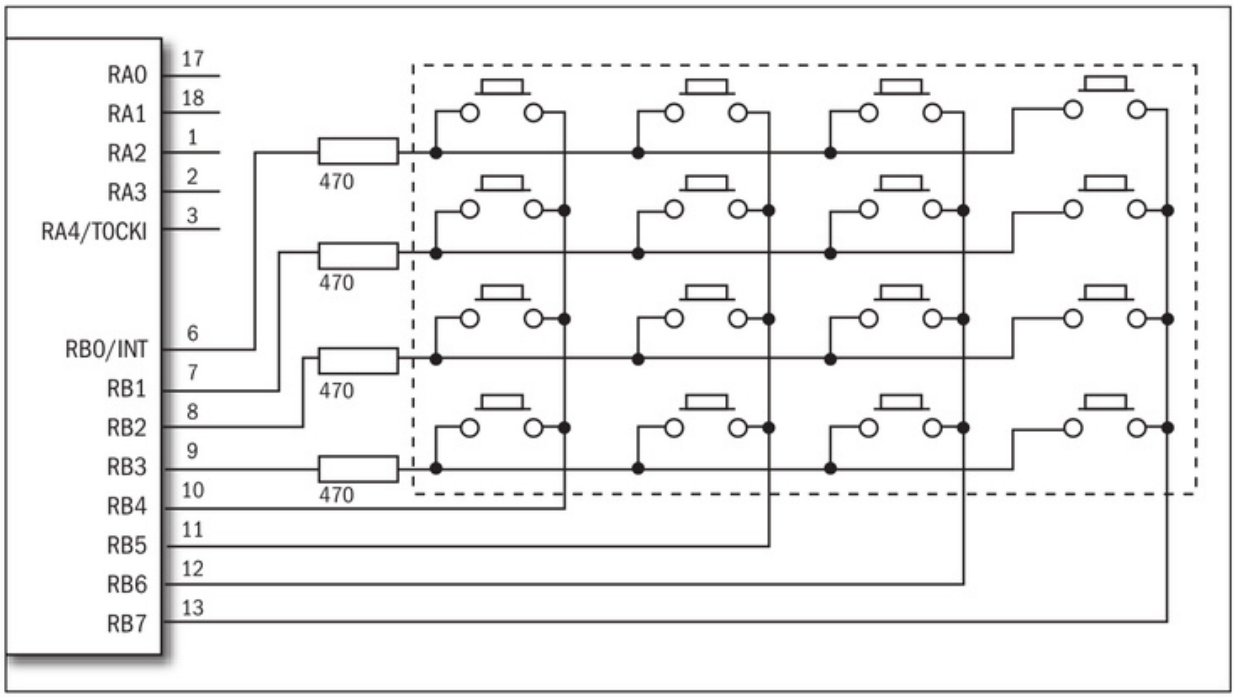


Figura 9. *Forma típica de conexión de un teclado de 4x4 al PIC16F84A.*

Para comprender mejor su funcionamiento, veamos cuál es la forma de leer este teclado. En primer lugar, es importante la configuración del puerto donde se conecta el teclado, en este caso, el Puerto B. Las líneas bajas (RB0 a RB3) se configuran como salidas, mientras que las líneas altas (RB4 a RB7) se configuran como entradas. De este modo, al presionar un pulsador, el estado que haya en la fila donde esté conectado se reflejará en la columna, es decir, en la parte alta, dependiendo del pulsador que se presione.

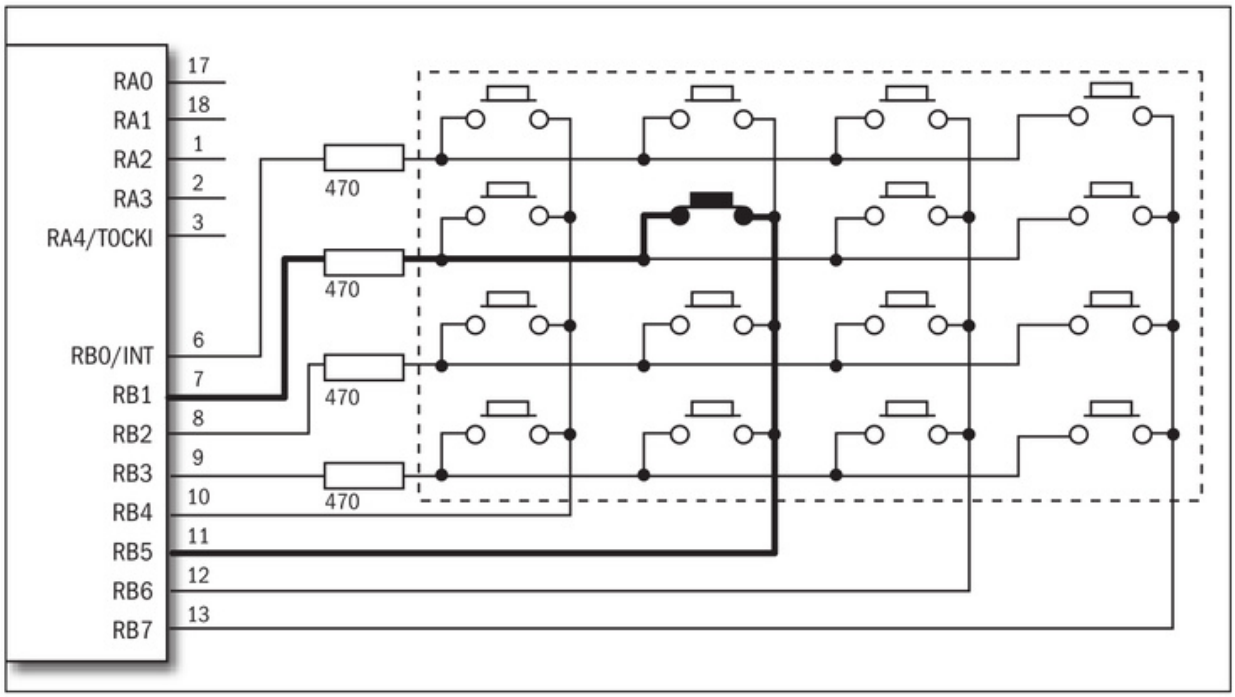


Figura 10. *Ejemplo de un pulsador accionado en el teclado y su efecto.*

Así se puede leer la coordenada del pulsador que fue presionado mediante la lectura de las filas y las columnas que conforman la matriz para obtener su valor. Veamos cómo funciona la rutina para determinar cuál del los pulsadores se ha presionado.

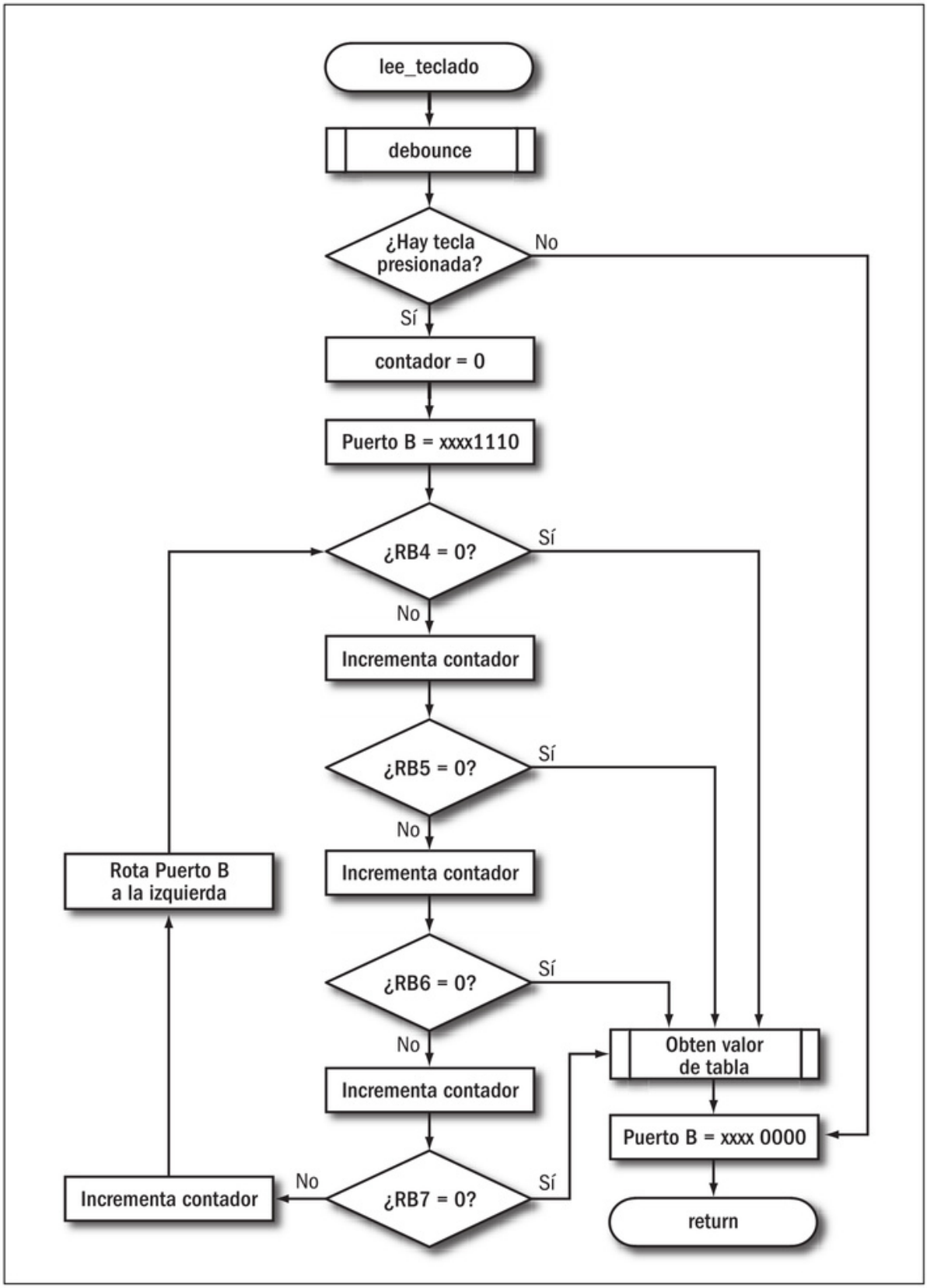


Figura 11. *Funcionamiento de la subrutina para leer el teclado.*

Primero se coloca un 0 en la salida RB0 y 1 en las líneas RB1, RB2 y RB3. De esta forma, podemos analizar las entradas una a una para ver si en alguna de ellas se refleja el 0 de la primer fila. Esto se hace ordenadamente verificando RA4, RA5, RA6 y RA7, mientras se incrementa un contador, si en ninguna de ellas hay un 0, significa que en esa fila no hay ninguna tecla presionada. Luego se rota el valor del Puerto B para colocar el 0 en la siguiente fila, se hace la verificación de nuevo para ver si en esa fila hay algún pulsador presionado, y así hasta llegar al final. Cuando se ha detectado cuál es el pulsador presionado, se toma el valor del contador y éste tendrá el número de la tecla presionada, que va de 0 a 15.

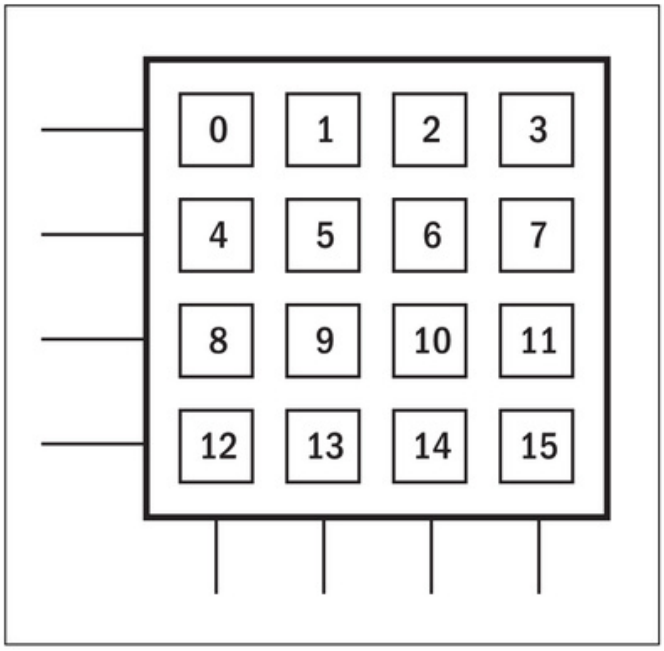


***Figura 12.** Numeración de las teclas para su reconocimiento mediante la rutina de lectura de teclado.*

Como podemos apreciar en la **Figura 12**, se obtendrá un número según la tecla presionada. Por ejemplo, si presionamos la última tecla, la de la fila más baja y la columna del extremo derecho, obtendremos en nuestro contador el número **15**. En algunos teclados se puede tener un orden de números distinto. Para ello tenemos que recurrir a una tabla de datos para obtener el valor real de la tecla pulsada según el teclado utilizado.

LCD EN BITS BAJOS

Cuando usamos pulsadores controlados mediante interrupciones RBI debemos conectarlos a la parte alta del Puerto B, donde ya está conectado el LCD. Si no vamos a usar la parte baja del puerto, podemos conectar ahí el display para que no comparta líneas con los pulsadores pero, por supuesto, tendríamos que modificar la librería de manejo del LCD.

Para fabricar nuestros teclados podemos utilizar microswitches y conectarlos en la matriz, y obtener uno como el de la **Figura 13**. En las tiendas del ramo también podemos comprar un teclado, generalmente de 16 teclas ó 4x4, listo para conectarlo al PIC y utilizarlo, como el que vemos en la **Figura 14**.

Figura 13. *Un teclado de 4x4 construido con* ***microswitches****.*

Figura 14. *Un teclado 4x4 listo para conectarlo al microcontrolador.*

Librería para manejo de teclados

Como siempre, podemos escribir nuestra librería para manejar de forma fácil y rápida los proyectos que involucren un teclado. En **www.redusers.com** tenemos la posibilidad de descargar la librería **TECLADO.INC**, donde tenemos las subrutinas para la lectura y la decodificación de teclados matriciales. El código es el siguiente:

```
     CBLOCK
     tecla
     ENDC

;Tabla de datos para decodificar el valor obtenido por la subrutina
lee_teclado
;en donde se obtiene el valor decimal del orden real del teclado, se coloca al
;principio para evitar sobrepasar los primeros 256 bytes:

teclado_tabla
     addwf   PCL, F
     DT            .1, .2, .3, .10      ;Primera fila
     DT            .4, .5, .6, .11      ;Segunda fila
     DT            .7, .8, .9, .12      ;Tercer fila
```

```
     DT              .14, .0, .15, .13     ;Cuarta fila
fin_tabla

;Si se superan los primeros 256 bytes, aparecerá un error al ensamblar:

     IF (fin_tabla > 0xFF)
            ERROR "La tabla supera los primeros 256 bytes de la memoria de
                   programa"
     ENDIF

;Inicializa el teclado configurando adecuadamente el Puerto B
;y activando los Pull-ups internos:

teclado_inicializa
     bsf             STATUS, RP0
     bcf             OPTION_REG,NOT_RBPU   ;Activa los pull-ups internos
     movlw           b'11110000'
     movwf           TRISB                 ;RB0 a RB3 salidas, RB4 a RB7
entradas
     bcf             STATUS, RP0
     call            teclado_pulsado
     clrf            PORTB
     return

;Subrutina para leer el teclado y obtener el valor
;numérico de la tecla pulsada:

lee_teclado
     call            debounce              ;Llama a subrutina anti-rebotes
     movf            PORTB, W              ;lee Puerto B
     sublw           b'11110000'
     btfsc           STATUS, Z             ;¿hay alguna tecla presionada?
     goto            falso                 ;No, falso disparo, sale
     clrf            tecla                 ;Si, inicializa "tecla"
     bsf             STATUS, C
     movlw           b'00001110'           ;Colocará 0 en la primera fila
lee_columna
     movwf   PORTB
     btfss   PORTB, 4                      ;¿Primer pulsador presionado?
     goto            fin_lectura           ;Si, finaliza lectura
```

```
     incf          tecla, F             ;No, incrementa tecla y sigue
     btfss         PORTB, 5             ;¿Segundo pulsador presionado?
     goto          fin_lectura          ;Si, finaliza lectura
     incf          tecla, F             ;No, Incrementa tecla y sigue
     btfss         PORTB, 6             ;      .
     goto          fin_lectura          ;      .
     incf          tecla, F             ;      .
     btfss         PORTB, 7
     goto          fin_lectura
     incf          tecla, F
     rlf           PORTB, W             ;No, rota Puerto B
     goto          lee_columna          ;Ve a leer siguiente columna
fin_lectura
     movf          tecla, W
     call          teclado_tabla        ;Obtén el valor correcto de la
tabla
     movwf         tecla                ;Colócalo en "tecla"
falso
     clrf          PORTB                ;Borra Puerto B
     return

;Subrutina para verificar si el teclado sigue pulsado y no salir hasta
;que no haya ninguna tecla pulsada en él:

teclado_pulsado
     clrf          PORTB                ;Borra Puerto B
teclado_espera
     call          debounce
     movf          PORTB, W             ;lee Puerto B, si no hay tecla
pulsada
     sublw         b'11110000'          ; el Puerto B debe tener 11110000
     btfss         STATUS, Z            ;¿Es cero?
     goto          teclado_espera       ;No, hay tecla pulsada, espera
     return                             ;Si, no hay tecla pulsada
regresa

;Se incluye la librería anti-rebotes, es importante tomarlo en cuenta
;para no agregarla en el programa principal:

     #INCLUDE      <DEBOUNCE.INC>
```

En esta librería tenemos tres subrutinas principales:

- **teclado_inicializa**: configura correctamente el Puerto B para el funcionamiento del teclado y activa sus *weak pull-ups*. De esta forma, se inicializa el uso del teclado. Además, verifica si hay alguna tecla pulsada y espera a que se deje de pulsar.
- **lee_teclado**: verifica si se ha pulsado alguna tecla y obtiene el valor de la tecla pulsada. El valor se obtiene de la tabla que está al inicio (**teclado_tabla**), donde se colocan los valores adecuados al teclado conectado. En la librería se obtienen los valores que corresponden a un teclado como el de la **Figura 14**, siendo la tecla de asterisco (*) la 14 y el numeral (#) la 15 por si se necesita agregar la E y F para formar un teclado hexadecimal. Si se necesitan valores o caracteres diferentes para las teclas, habrá que cambiar la tabla de la librería.
- **teclado_pulsado**: subrutina para verificar si hay alguna tecla pulsada. Sirve para detener el programa hasta que se libere la tecla pulsada y así evitar que se repita constantemente la lectura mientras la tecla permanece activada.

Con esta librería podemos leer un teclado, ya sea con la técnica de muestreo, o mediante interrupciones RBI. Veamos un ejemplo de ello para lo que tomaremos el circuito de la **Figura 15**. A partir de ahora colocaremos los teclados con un símbolo.

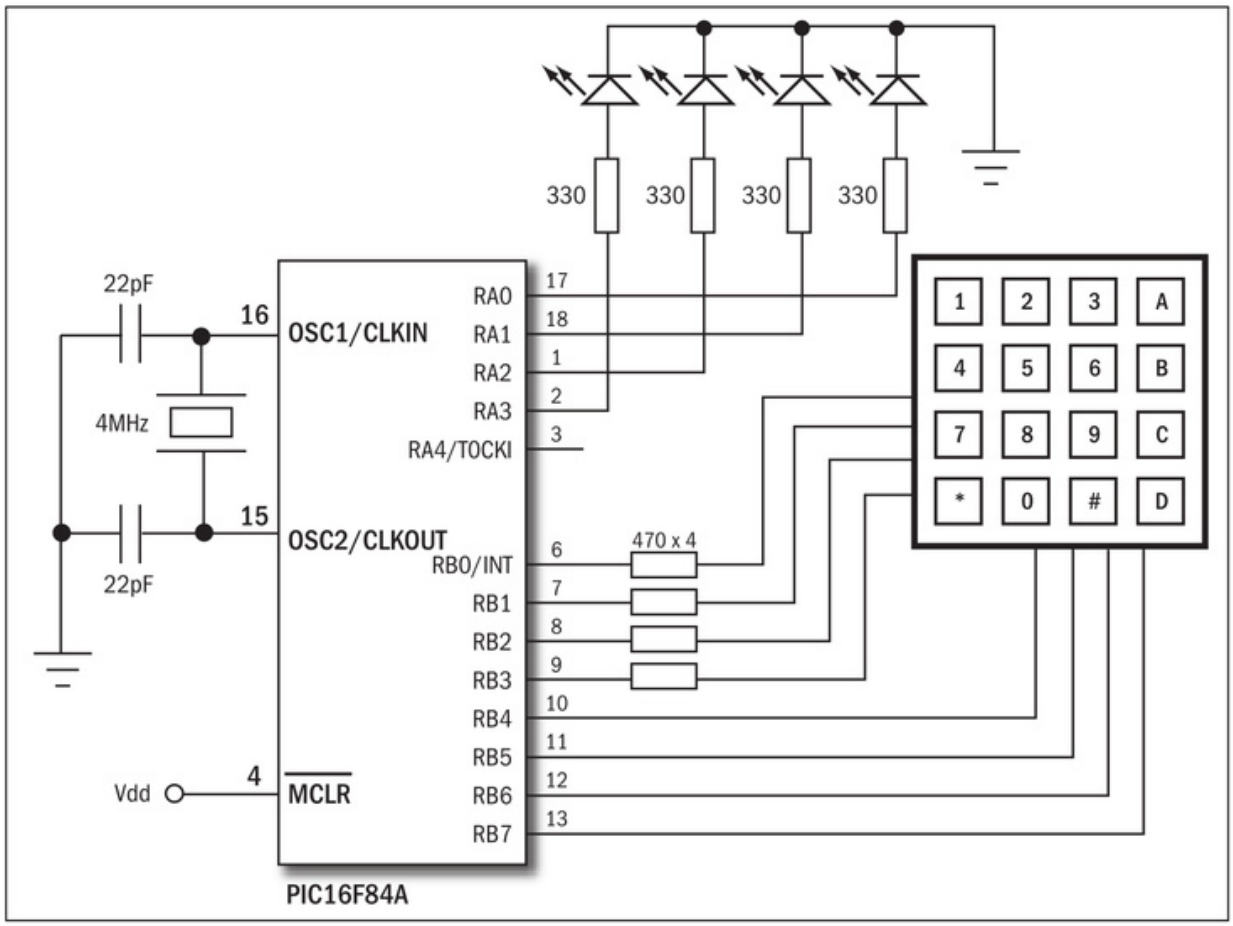


Figura 15. *Circuito para comenzar a usar teclados de 4x4.*

El ejercicio que proponemos es muy sencillo. Utilizando la librería para manejo de teclado diseñaremos un programa que simplemente muestre en los leds conectados al Puerto A, el número de la tecla que se presione:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      ENDC

      ORG             00h
inicio
      bsf             STATUS, RP0
      clrf            TRISA
      bcf             STATUS, RP0
      call            teclado_inicializa
loop
      call            lee_teclado
      movf            tecla, W
      movwf           PORTA
      goto            loop

      #INCLUDE        <TECLADO.INC>

      END
```

Como podemos apreciar, el código es bastante sencillo al usar nuestra librería. En este ejemplo tenemos un bucle que lee constantemente el teclado y refleja el número de la tecla pulsada en los 4 primeros bits del Puerto A. Pero para hacer más eficientes nuestros programas podemos hacer uso de la interrupción RBI para leer el teclado sólo cuando se presione alguna de sus teclas. Veamos cómo se hace:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>
```

```
      CBLOCK 0x0C
      ENDC

      ORG             00h
      goto            inicio

      ORG             04h
      call            lee_teclado
      movf            tecla, W
      movwf           PORTA
      call            teclado_pulsado
      bcf             INTCON, RBIF
      retfie

inicio
      call            teclado_inicializa
      bsf             STATUS, RP0
      movlw           b'10001000'
      movwf           INTCON                  ;Activa GIE y RBIE
      clrf            TRISA
      bcf             STATUS, RP0
      clrf            PORTB
      clrf            PORTA

principal
      sleep
      goto            principal

      #INCLUDE        <TECLADO.INC>

      END
```

III ¿Y LOS RESISTORES?

Los resistores de 470 ohms conectados con el teclado tienen una doble función. Por una parte evitan que se genere un cortocircuito entre dos líneas de salida, si se presiona más de una tecla, y por otro para permitir la conexión del LCD en el mismo puerto, también para evitar un cortocircuito cuando se presiona una tecla y se está escribiendo al mismo tiempo en el LCD

Las ventajas de usar interrupciones son evidentes: sólo se accede a la lectura del teclado cuando se requiere. Mientras no se presione ninguna tecla, se puede atender a otros procesos, o entrar en modo de bajo consumo y así ahorrar energía.

Teclado y display LCD en el Puerto B

Cuando necesitamos mostrar datos a la salida en un display LCD y a su vez leer datos desde un teclado, podemos conectar ambas cosas en el Puerto B, tal como lo muestra la **Figura 16**. Como el bus de datos del display se pone en alta impedancia mientras se encuentra deshabilitado (**E = 0**), podemos compartir las líneas del Puerto B entre el display y el teclado perfectamente.

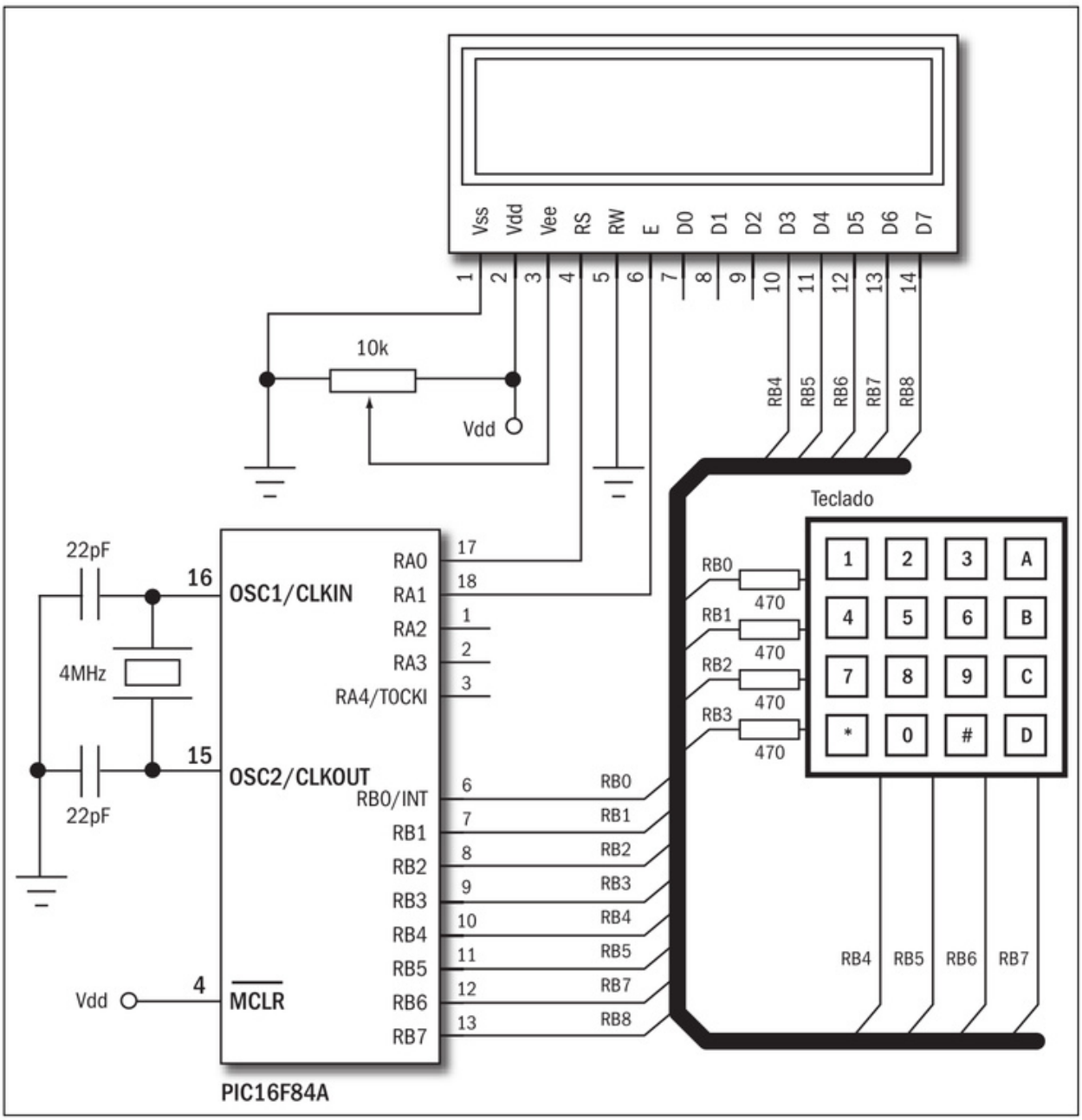


Figura 16. *Circuito para compartir un teclado y display LCD en el Puerto B del PIC16F84A.*

CERRADURA ELECTRÓNICA

La aplicación por excelencia que se desarrolla para observar el funcionamiento del teclado y el display LCD juntos es una **cerradura electrónica**. Para eso, primero necesitamos definir el circuito, que incluye el teclado y el LCD. Además de ello usaremos el pin RA3 como salida de control para el mecanismo de apertura de una puerta u otro elemento que necesitemos proteger con nuestra cerradura electrónica con PIC16F84A.

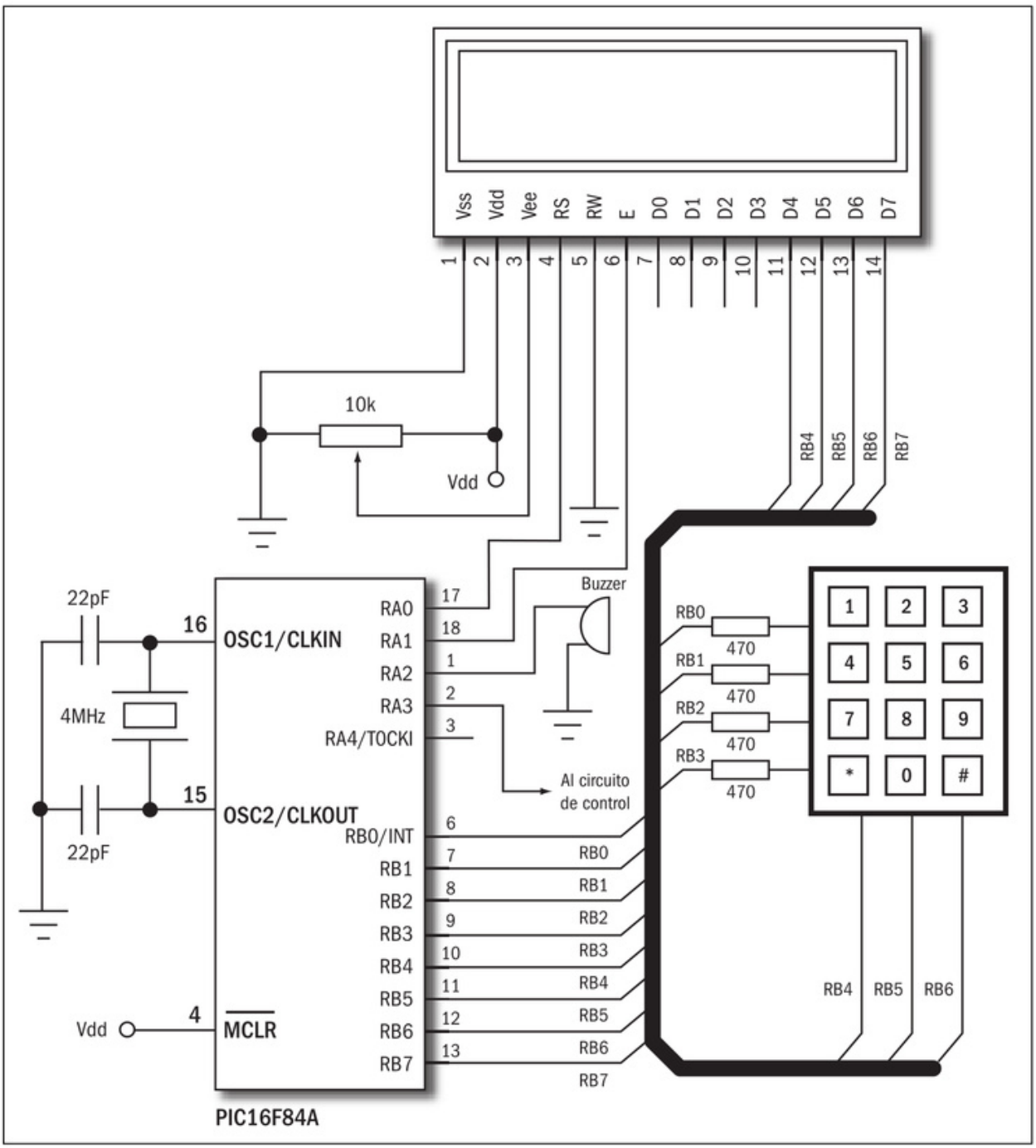


Figura 17. *Circuito para nuestra cerradura electrónica con microcontrolador PIC.*

El circuito deberá funcionar de la siguiente manera: al encenderlo, la pantalla del display estará vacía y el microcontrolador entrará en modo de bajo consumo. Al presionar por primera vez una de las teclas, comenzará a recibir la clave, que en nuestro caso será de 4 dígitos. En este caso usaremos un teclado decimal o de **12 teclas**, tal como vemos en la **Figura 17**, ya que la clave a ingresar constará de 4 dígitos decimales. Una vez que se han presionado 4 teclas, el programa deberá comparar la clave introducida con la clave almacenada para ver si es igual. En caso de serlo, se activará (se pondrá a 1) el pin RA3 por 5 segundos, permitiendo la apertura de la puerta que protege y colocando el mensaje **Clave correcta** en el display. En caso de no ser la clave correcta, aparecerá en el display la indicación **Clave incorrecta** durante tres segundos y, por supuesto, el pin RA3 no se activará. En cualquiera de los dos casos, después de transcurridos los tres o cinco segundos respectivamente, el sistema estará listo para recibir nuevamente una clave en el teclado.

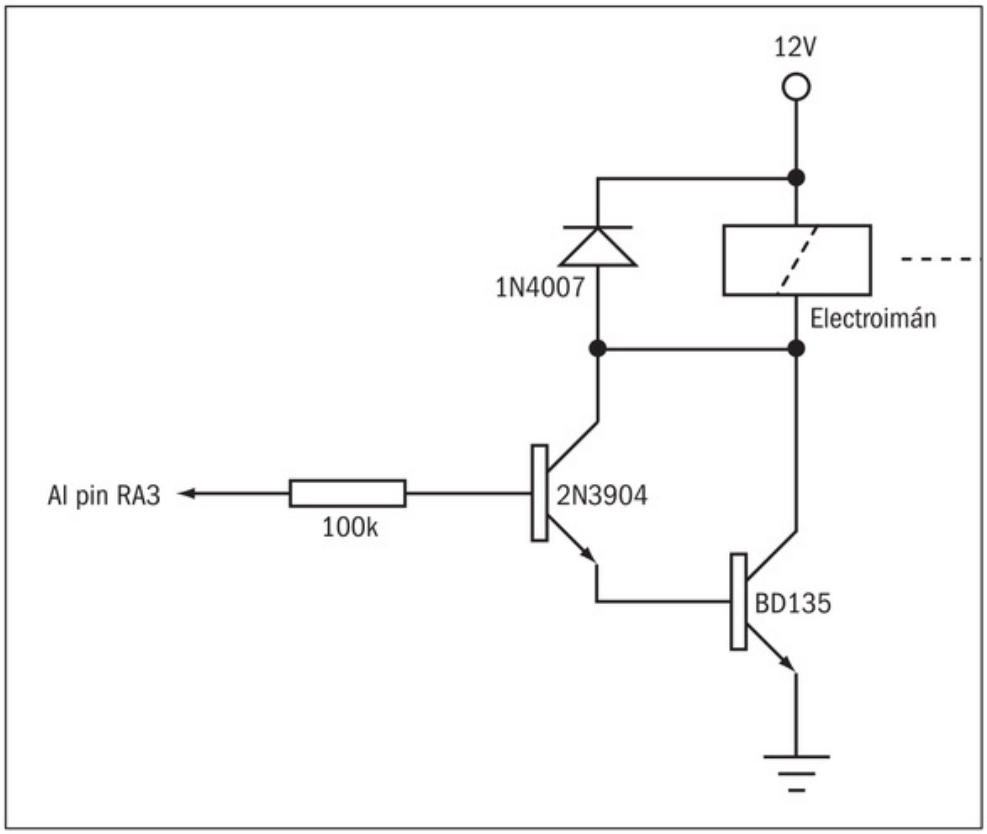


***Figura 18.** Posible circuito de control para una cerradura de puerta eléctrica con electroimán.*

El circuito de control será aquél que abra la puerta mediante una cerradura eléctrica, que generalmente es un electroimán que deberá activarse de forma adecuada. Puede conectarse a la salida del PIC, como se muestra en la **Figura 18**, aunque por supuesto el circuito dependerá de lo que tengamos exactamente, o de la aplicación que vayamos a asignarle. En cualquier caso, nuestro código para la cerradura electrónica puede ser el que detallamos a continuación:

```
     __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

     PROCESSOR 16F84A
     #INCLUDE <P16F84A.INC>

     CBLOCK 0x0C
     cont_clave, cont_ret                ;Un par de contadores auxiliares
     ENDC

     ORG            00h
     goto           inicio
     ORG            04h
     goto           cerradura

clave_secreta
     addwf          PCL, F
     DT             .7, .2, .8, .0       ;Esta es la clave de la cerradura

mensajes
     addwf          PCL, F
mens_incorrecta
     DT             "Clave incorrecta", 00h
mens_correcta
     DT             "Clave correcta", 00h

inicio
     call           LCD_inicializa
     call           teclado_inicializa
     bsf            STATUS, RP0
     bcf            TRISA, 3             ;Pin RA3 como salida
     bcf            STATUS, RP0
```

III MENOS COLUMNAS EN EL TECLADO

La librería para el teclado 4x4 también es capaz de manejar teclados con menos columnas. Por ejemplo, se puede manejar perfectamente un teclado 4x3. Esto es por que la columna que falta no generará ningún efecto, tal como si estuviera, pero nunca fueran presionadas sus teclas. El pin de entrada de la columna faltante deberá dejarse libre.

```
     movlw          b'10001000'
     bcf            PORTA, 3            ;Deshabilita la salida
     movwf          INTCON              ;Habilita interrupciones

duerme
     sleep                              ;Queda en bajo consumo
     goto           duerme

cerradura
     movlw          0x20                ;Inicializa la dirección donde
     movwf          FSR                 ; se almacenará la clave tecleada
lee_clave                               ;Rutina para leer la clave
tecleada
     call           lee_teclado         ;Lee   la tecla pulsada

     movf           tecla, W
     sublw          d'14'
     btfsc          STATUS, Z           ;¿Es la 14 (asterisco)?
     goto           cancelar            ;Si, cancela todo

     movf           tecla, W
     sublw          d'9'
     btfss          STATUS, C           ;¿Es mayor a 9?
     goto           tecla_no_valida     ;Si, ningún efecto

     movf           tecla, W            ;Si es dígito decimal
            léelo
     movwf          INDF                ;Almacénalo
     incf           FSR, F              ;Incrementa puntero FSR
     movlw          '*'
     call           LCD_caracter        ;Envía "*" al display
     movf           FSR, W
     sublw          h'24'
     btfsc          STATUS, Z           ;¿Se han leído 4 dígitos?
     goto           fin_clave           ;Si, termina la lectura
tecla_no_valida
     call           teclado_pulsado     ;Si el teclado esta pulsado
espera
no_pulsado
     call           debounce
```

```
     movf        PORTB, W
     sublw       b'11110000'
     btfsc       STATUS, Z          ;¿Se ha pulsado otra tecla?
     goto        no_pulsado         ;No, espera a que se pulse
     goto        lee_clave          ;Si, lee el dígito pulsado

fin_clave                           ;compara la clave leída
     movlw       0x20               ;Inicializa puntero para
     movwf       FSR                ; leer la clave almacenada
     clrf        cont_clave         ;Contador para leer la clave de
la tabla
compara_clave
     movf        cont_clave, W
     call        clave_secreta      ;Obtén dígito de la tabla
     subwf       INDF, W            ;Obtén dígito tecleado y restalos
     btfss       STATUS, Z          ;¿Son iguales?
     goto        clave_incorrecta   ;No, la clave tecleada es
incorrecta
     incf        FSR, F             ;Si, continua comparando
     incf        cont_clave, F      ;Incrementa para acceder al
siguiente dígito
     movf        FSR, W             ;Incrementa puntero para leer
siguiente dígito
     sublw       h'24'
     btfss       STATUS, Z          ;¿Se han comparado 4 dígitos?
     goto        compara_clave      ;No, continua comparando
clave_correcta                      ;Si, la clave es correcta
     bsf         PORTA, 3           ;Activa la salida RA3
     movlw       mens_correcta
     call        LCD_mensaje        ;Coloca mensaje en el display
     call        retardo_5s         ;Espera 5 segundos
     bcf         PORTA, 3           ;desactiva RA3
     goto        cancelar           ;termina
clave_incorrecta
     movlw       mens_incorrecta
     call        LCD_mensaje        ;Coloca mensaje en el display
     call        retardo_3s         ;Espera 3 segundos
cancelar                            ;Termina
     call        borra_display      ;Borra el display
fin_int
```

```
     call          teclado_pulsado       ;Si el teclado esta pulsado
espera
     bcf           INTCON, RBIF          ;Limpia bandera de interrupción
     retfie                              ;Termina interrupción

retardo_5s                               ;Retardos
     movlw         d'163'
     movwf         cont_ret
     goto          ret_loop
retardo_3s
     movlw         d'98'
     movwf         cont_ret
ret_loop
     call          debounce              ;Se aprovecha el retardo debounce
     decfsz        cont_ret, F
     goto          ret_loop
     return

;Se agregan las librerías necesarias:

     #INCLUDE      <TECLADO.INC>
     #INCLUDE      <LCD4BITS.INC>
     #INCLUDE      <LCDMENSAJES.INC>

     END
```

El diagrama de flujo de nuestro ejemplo de la cerradura electrónica con PIC16F84A lo podemos observar en la **Figura 19**. Con el diagrama y los comentarios del propio código fuente no debemos tener mayor problema para comprender el funcionamiento del programa, aunque a continuación comentaremos algunos puntos importantes de su funcionamiento.

APLICACIONES DE LA CERRADURA

La cerradura electrónica de este capítulo puede tener multitud de aplicaciones: puede ser una cerradura clásica que permite abrir la puerta de una habitación o la puerta de una caja fuerte. Con algunas modificaciones del programa, podemos usarla para permitir el encendido de algún aparato electrónico o eléctrico. Sólo es cuestión de usar nuestra imaginación.

La clave secreta está grabada en la memoria de programa, en la tabla llamada **clave_secreta**, por lo que es fija y no puede modificarse a menos que se grabe de nuevo el programa en el PIC. La rutina de comparación de la clave tecleada compara los dígitos introducidos por el teclado con la clave de esta tabla.

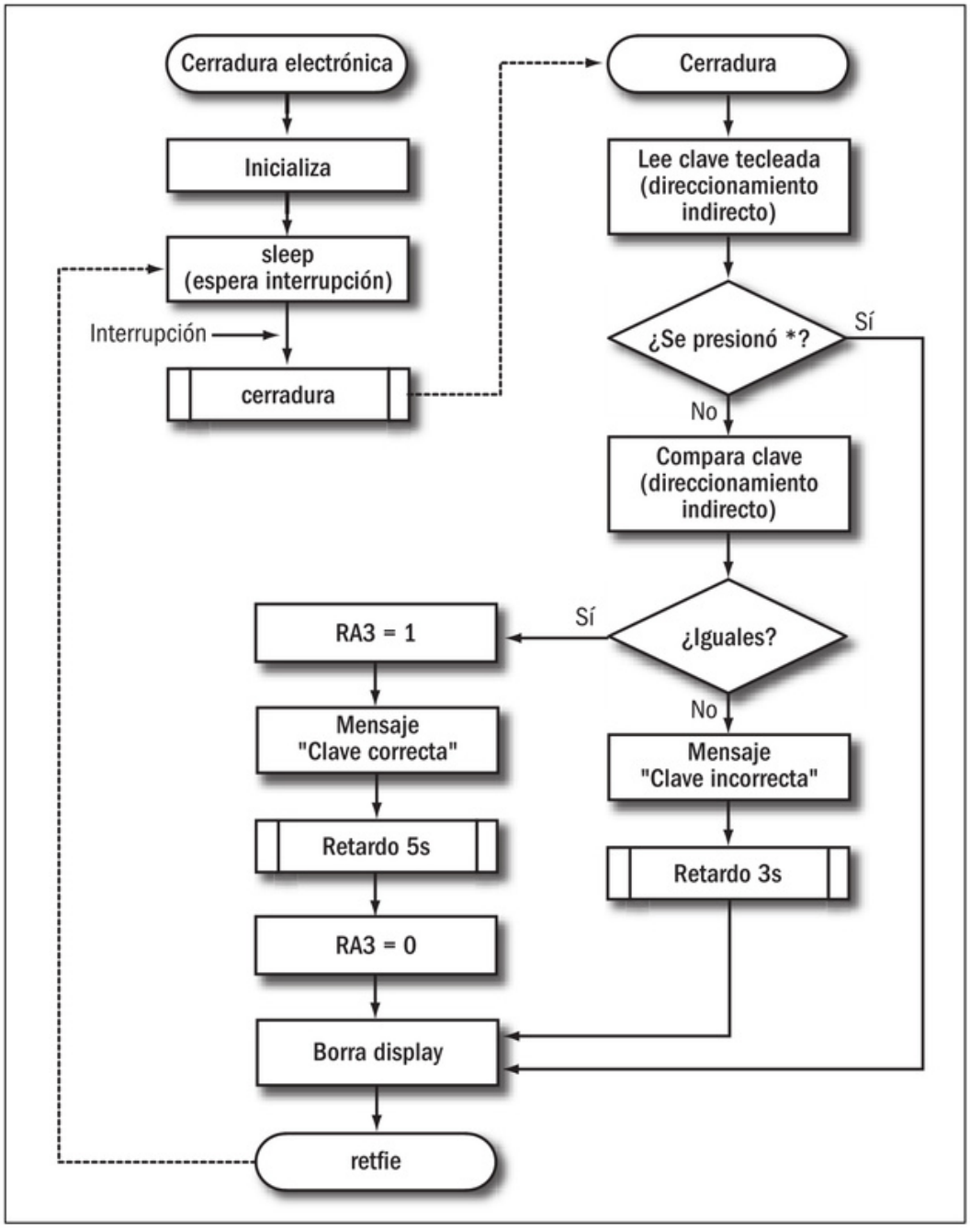


***Figura 19.** Detalle del funcionamiento del programa para la cerradura electrónica.*

El microcontrolador siempre estará en modo de bajo consumo, hasta que se presione cualquiera de las teclas, se produzca la interrupción y se comience la ejecución de la subrutina de atención, que leerá la clave tecleada, la comparará y actuará conforme se necesite si la clave es correcta o no. Para la lectura de la clave tecleada y su comparación con la clave almacenada, se utiliza direccionamiento indirecto. De esta manera,

la clave tecleada se almacenará a partir de la dirección 20h de la memoria de datos. Esto es para evitar sobrescribir los registros utilizados en el programa o en las librerías.

Al contar con un teclado decimal conformado por los dígitos del 0 al 9 y los símbolos * y #, tal como se muestra en el diagrama del circuito, debemos colocar un teclado de 4x3. En caso de tener uno de 4x4 (como el de la **Figura 14**), podemos también usarlo, ya que las teclas correspondientes a letras y el símbolo # son ignoradas en el programa y no generarán ningún efecto si son presionadas. La tecla de asterisco se usa para cancelar la introducción de dígitos en cualquier momento entre la pulsación del primer al tercer dígito, por lo tanto, se puede llamar a esta tecla **Cancelar**.

Al introducir una clave errónea, aparecerá el mensaje **Clave incorrecta** en el display y se mantendrá durante tres segundos, en los cuales no se podrá teclear otra clave hasta que el mensaje desaparezca de la pantalla. Si la clave introducida es correcta, se activará el pin RA3 durante 5 segundos, tiempo suficiente para abrir la puerta. Después de este tiempo, el pin RA3 se desactivará nuevamente. En el diagrama se muestra un zumbador (*buzzer*) que es opcional, pero nos servirá si queremos introducir en el programa algún tipo de alarma cuando se teclea una clave incorrecta o algo por el estilo.

Con lo visto aquí tenemos un claro ejemplo del uso de un teclado en nuestros proyectos. Por supuesto podemos introducir todas las mejoras que consideremos para la aplicación exacta que tengamos en mente. En el sitio **www.redusers.com** podemos descargar los archivos **Cerradura.asm** y **Cerradura.hex** para su comprobación en el circuito.

INTERRUPCIÓN POR DESBORDAMIENTO DEL TIMER 0

Ya estudiamos antes el uso del Timer 0 del PIC16F84A, ahora veremos cómo se puede tener una interrupción cada vez que el Timer 0 se desborda, es decir, cuando alcanza su máximo valor que es 11111111, se incrementa una vez más pasando a 00000000 y provoca que se active el bit T0IF. Además, veremos que si los bits GIE y T0IE están a 1, se generará una interrupción, llamada **T0I**.

Latencia de interrupción

La interrupción por desbordamiento del Timer 0 nos puede resultar de mucha utilidad en la medición y control de tiempos. Como vimos en el **Capítulo 8**, podemos calcular tiempos de desbordamiento del timer 0 con la fórmula:

$$\text{Tiempo} = \text{prescaler}\ (256 - \text{carga}) + 2$$

Pero cuando usamos interrupciones, hay que agregar además el tiempo de latencia, que es el momento en el que se atiende a la interrupción cuando ésta se genera. Es decir, cuando ocurre una interrupción, la ejecución o salto hacia el vector de interrupción lleva un poco de tiempo. Para las interrupciones internas, principalmente para la interrupción por desbordamiento del TMR0, esta latencia es siempre fija y tiene un valor de **3 ciclos de máquina**. Esto significa que desde que se genera la interrupción T0I hasta que comienza a ejecutarse la instrucción de la dirección 04h o el vector de interrupción, pasarán tres ciclos de máquina, por lo que la fórmula para calcular el tiempo de desbordamiento del Timer al usar la interrupción será:

$$\text{Tiempo} = \text{prescaler}\ (256 - \text{carga}) + 5$$

agregando los tres ciclos de máquina de la latencia. Si despejamos el valor de la carga, obtendremos la siguiente fórmula:

$$\text{carga} = 256 - \frac{\text{Tiempo} - 5}{\text{prescaler}}$$

El tiempo siempre debe darse en microsegundos.

Hemos estudiado cómo obtener temporizaciones exactas con el Timer 0. Esta vez haremos lo mismo, pero empleando la interrupción lo cual, como nos hemos dado cuenta, tiene muchas ventajas. Un ejemplo es el siguiente programa que genera una onda cuadrada de 4 KHz en el pin RA3 del PIC16F84A. Esta vez se realizará mediante la interrupción T0I. Necesitaremos en esta ocasión un período de 250 microsegundos y, por lo tanto, un tiempo en alto de 125 y en bajo también, 125 microsegundos. Al calcular con nuestra nueva ecuación obtendremos:

$$\text{carga} = 256 - \frac{125 - 5}{1} = 136$$

De la misma forma debemos escribir el programa en MPLAB y con el simulador probar hasta lograr ajustar los tiempos exactos tomando en cuenta las instrucciones extra que deben ejecutarse. En este caso hemos ajustado a 139 el valor de carga del Timer 0:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      ORG            00h
      goto           inicio

      ORG            04h
      goto           int_TOI

inicio
      bsf            STATUS, RP0
      clrf           TRISA
      movlw          b'00001000'
      movwf          OPTION_REG            ;Sin prescaler, TMR0 como
temporizador
      bsf            INTCON, GIE           ;Activa interrupciones globales
      bsf            INTCON, TOIE          ;Activa la interrupción TOI
      bcf            STATUS, RP0
      bcf            PORTA, 3
      movlw          d'154'
      movwf          TMR0

;Se genera un bucle para esperar la interrupción, no se puede
;poner en modo de bajo consumo, ya que eso detendría el Timer 0:

bucle
      goto           bucle

int_TOI                                    ;Subrutina de interrupción
      movlw          d'139'
      movwf          TMR0                  ;Carga TMR0
      btfss          PORTA, 3
      goto           salida_a_uno
      nop                                  ;Retardo de ajuste
```

```
salida_a_cero
     bcf            PORTA, 3
     bcf            INTCON, TOIF
     retfie
salida_a_uno
     bsf            PORTA, 3
     bcf            INTCON, TOIF
     retfie

     END
```

Y entonces podemos lograr también la misma tarea de temporización si usamos la interrupción por desbordamiento del Timer 0.

Tiempos largos

En ocasiones necesitaremos temporizar o medir tiempos más largos. Como sabemos, el límite del timer 0, si usamos un oscilador de 4 MHz y el prescaler más alto, sería:

> Tiempo = prescaler (256 - carga) + 5 = 256 (256 - 0) + 5 = 65.541 ms

Por lo que si necesitamos contar tiempos más largos debemos usar un registro auxiliar que nos permita contar las veces que se ha generado la interrupción y así lograr nuestro objetivo. A continuación veremos un ejemplo.

RELOJ DIGITAL BÁSICO

Supongamos que deseamos diseñar un reloj, el cual indicará las horas, los minutos y los segundos en el display LCD. Entonces, necesitaremos contar con segundos exactos, y a partir de ellos medir los minutos y las horas. Con el máximo prescaler del TMR0 sólo podemos alcanzar un tiempo de unos 65 ms por lo que, para poder medir un segundo, deberemos utilizar un registro auxiliar como contador. Podemos hacer un cálculo para obtener un tiempo lo más cercano a un segundo. Por ejemplo, si calculamos un tiempo de 50 ms para el Timer 0 y luego lo multiplicamos por 20, obtendremos un tiempo de 1 segundo. De esta manera, calcularemos el tiempo de desbordamiento para 50 ms de la siguiente forma:

$$carga = 256 - \frac{Tiempo - 5}{prescaler} = 256 - \frac{50000 - 5}{256} = 60.7$$

Pero claro, debemos cargar el Timer 0 con un valor entero, así que tomaremos el valor más cercano hacia arriba, que es 61. Con esto recalculamos el tiempo:

$$Tiempo = prescaler\ (256 - carga) + 5 = 256\ (256 - 61) + 5 = 49.925\ ms$$

Un valor muy cercano pero no exacto de 50 ms. Para llegar a 50 ms necesitaremos 75 microsegundos más, lo cual conseguiremos con instrucciones o con un retardo. Una vez ajustado el tiempo exacto a 50 ms, sólo nos resta cargar el contador auxiliar con un valor de 20, y cuando se han contado 20 interrupciones de 50 ms cada una, tendremos un segundo exacto. Podemos utilizar la función **Stopwatch** para ajustar los tiempos exactos de nuestro programa y así asegurar una gran exactitud de nuestro reloj o cualquier otro proyecto que involucre medición de tiempos muy exactos.

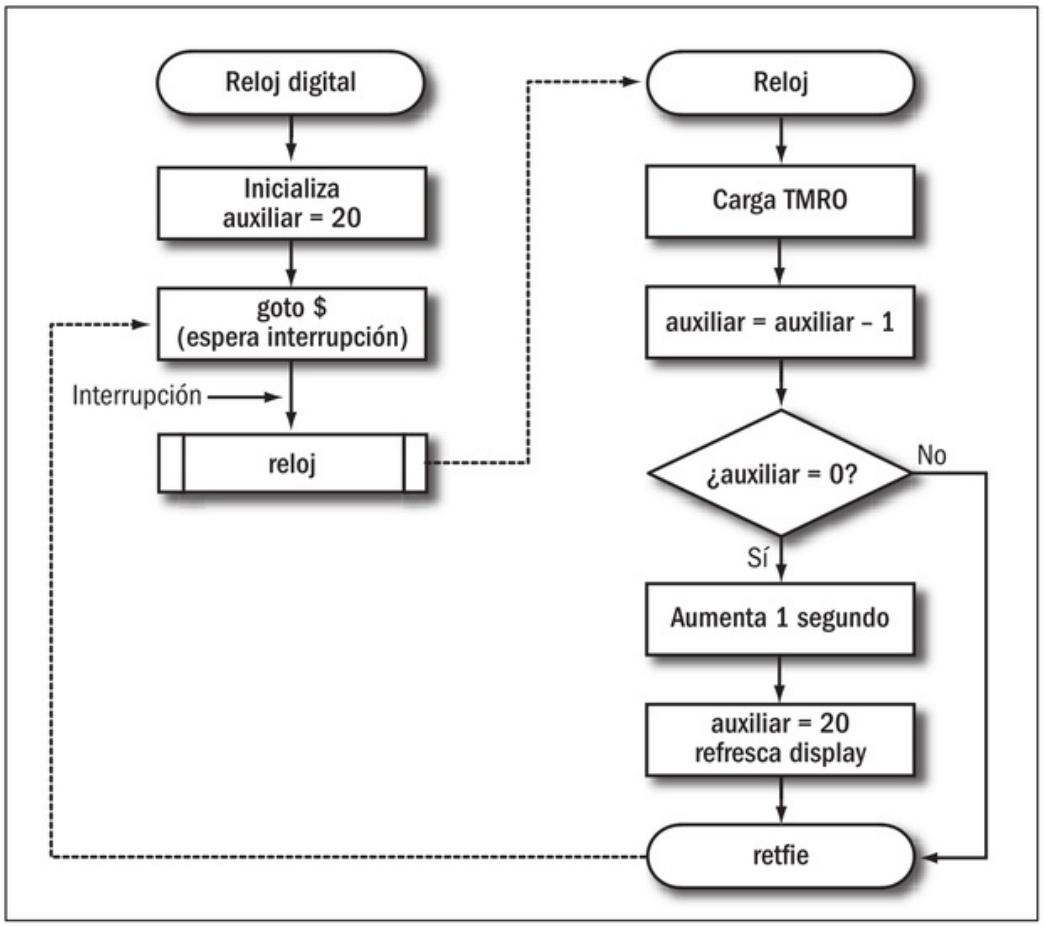


Figura 20. *Detalle del funcionamiento del programa para el reloj básico.*

De esta forma escribimos nuestro código fuente que quedará similar a esto:

```
      __CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

      PROCESSOR 16F84A
      #INCLUDE <P16F84A.INC>

      CBLOCK 0x0C
      horas, minutos, segundos, auxiliar, cont_ret
      ENDC

      ORG             00h
      goto            inicio

      ORG             04h
      goto            int_TOI

inicio
      movlw           d'61'
      movwf           TMR0                    ;Carga el Timer al arranque

      bsf             STATUS, RP0
      movlw           b'00000111'
      movwf           OPTION_REG              ;prescaler 1:256, TMR0 como
temporizador
      bsf             INTCON, GIE             ;Activa interrupciones globales
      bsf             INTCON, TOIE            ;Activa la interrupción TOI
      bcf             STATUS, RP0

      call            LCD_inicializa
      movlw           d'20'
      movwf           auxiliar                ;Inicializa el contador auxiliar
      movlw           b'10000110'
      call            LCD_comando
      movlw           ':'
      call            LCD_caracter            ;Coloca los dos puntos para
separar
      movlw           b'10001001'
      call            LCD_comando
      movlw           ':'
```

```
     call          LCD_caracter          ;Otros dos puntos
     clrf          horas                 ;Inicia todo en cero...
     clrf          minutos
     clrf          segundos
     call          LCDsegundos
     call          LCDminutos
     call          LCDhoras

espera
     goto          $                     ;Pausa para esperar la
interrupción

;Subrutina de atención a la interrupción:

int_TOI
     call          retardo               ;Retardo de ajuste
     movlw         d'61'
     movwf         TMR0                  ;Se recarga el TMR0
     decfsz        auxiliar, F           ;¿Auxiliar = 0?
     goto          fin_int               ;No, sale de la interrupción
     call          reloj                 ;Si, incrementa un segundo
     movlw         d'20'
     movwf         auxiliar              ;Recarga el registro auxiliar
fin_int
     bcf           INTCON, TOIF          ;Borra la bandera de interrupción
     retfie                              ;Terminó la interrupción

;Subrutina para aumentar un segundo en el reloj:

reloj
     incf          segundos, F           ;Incrementa los segundos
```

¿QUÉ TAN EXACTO?

Si calculamos debidamente las rutinas de temporización, podemos obtener relojes y demás aplicaciones que requieran de medición de tiempo, sumamente exactas. Tan exactas que pueden competir con los más precisos relojes comerciales. Esto, por supuesto, si usamos un cristal como oscilador, ya que las temporizaciones dejarán de ser exactas si empleamos un oscilador RC.

```
     movlw  d'60'
     subwf  segundos, W

     btfss          STATUS, Z            ;¿Segundos = 60?
     goto           fin_segundos         ;No, salta
     clrf           segundos             ;Si, borra segundos y continúa
     call           LCDsegundos          ;Coloca los segundos en el display

     incf           minutos, F           ;Incrementa los minutos
     movlw          d'60'
     subwf          minutos, W
     btfss          STATUS, Z            ;¿Minutos = 60?
     goto           fin_minutos          ;No, salta
     clrf           minutos              ;Si, borra minutos y continúa
     call           LCDminutos           ;Coloca los minutos en el display

     incf           horas, F             ;Incrementa las horas
     movlw          d'24'
     subwf          horas, W
     btfss          STATUS, Z            ;¿Horas =24?
     goto           fin_horas            ;No, salta
     clrf           horas                ;Si, borra horas

fin_horas
     call           LCDhoras             ;Coloca las horas en el display
     goto           fin_reloj
fin_minutos
     call           LCDminutos
     goto           fin_reloj
fin_segundos
     call           LCDsegundos
fin_reloj
     return

;Subrutinas para enviar los datos al display:

LCDsegundos
     movlw  b'10001010'
     call           LCD_comando          ;Coloca el cursor en la posición
correcta
```

```
     movf          segundos, W
     call          BINaBCD
     call          envia_digitos
     return

LCDminutos
     movlw         b'10000111'
     call          LCD_comando          ;Coloca el cursor en la posición
correcta
     movf          minutos, W
     call          BINaBCD
     call          envia_digitos
     return

LCDhoras
     movlw         b'10000100'
     call          LCD_comando          ;Coloca el cursor en la posición
correcta
     movf          horas, W
     call          BINaBCD
     movf          BCDdecenas, W
     btfss         STATUS, Z            ;¿Decenas = 0?
     goto          no_cero              ;No, muéstralas
     movlw         b'01110000'          ;Si, no las muestres
     movwf         BCDdecenas           ;Envía el valor correcto para
obtener un espacio
no_cero
     call          envia_digitos
     return

envia_digitos
```

III PRIORIDAD DE INTERRUPCIONES

Si estamos utilizando dos o más mecanismos de interrupción en un programa, debemos verificar las banderas de interrupción para determinar cuál de ellas se produjo. La bandera de interrupción que se verifica primero es la que tendrá mayor prioridad, ya que es la que se atenderá más rápidamente al producirse, y la última tendrá la menor prioridad.

```
     movf          BCDdecenas, W
     addlw         '0'
envia_digito
     call          LCD_caracter
     movf          BCDunidades, W
     addlw         '0'
     call          LCD_caracter

     return

;Retardo de ajuste para obtener un tiempo de 50ms
;exactos con cada interrupción:

retardo                                  ;Retardo 72 microsegundos
     movlw  d'16'
     movwf  cont_ret
loop1
     nop
     decfsz cont_ret, F
     goto          loop1
     goto          $+1
     nop
     return

     #INCLUDE      <LCD4BITS.INC>
     #INCLUDE      <BINABCD.INC>

     END
```

El programa se encargará de temporizar el reloj con una gran exactitud. Observemos cómo se espera la siguiente interrupción cada vez mediante un bucle (**goto $**), ya que nuevamente no podemos colocar el PIC en modo de bajo consumo, porque eso detendría el Timer 0. Desde el momento en que se recarga el Timer 0 con el valor de 61 hasta que se genera la siguiente interrupción, pasarán exactamente 50 ms, y con esta base de tiempo hemos construido nuestro reloj con el PIC16F84A.

Un detalle importante a notar es cómo el reloj diseñado empieza desde 00:00:00 al momento de encenderlo o aplicar un reset. Para ello, hacen falta las rutinas de ajuste de horas y minutos para ponerlo a tiempo, que no se han incluido, ya que es sólo cuestión de programarlas y agregarlas al código fuente. Con lo que hemos

estudiado hasta aquí no debemos tener mayores problemas para hacerlo, por lo que queda como un buen ejercicio de programación para completar el reloj para que sea 100% funcional.

INTERRUPCIÓN POR FINALIZACIÓN DE ESCRITURA EN EEPROM

Hemos estudiado ya la escritura y la lectura en la memoria EEPROM de datos del PIC16F84A y, como sabemos, el proceso de escritura tarda algún tiempo en completarse (normalmente, 4 ms). En la librería para escribir en la EEPROM se genera un bucle que lee el bit EEIF hasta que indica que la escritura ha finalizado, y de esa manera sabremos que se ha escrito el dato. Ahora, si activamos el bit GIE y el bit EEIE, tendremos una interrupción cuando la escritura de un byte en la EEPROM de datos haya finalizado y se active el bit EEIF. Esto puede ser útil en los programas donde se escriba en ella. Para no esperar a que termine, simplemente activamos la interrupción EEI y así podemos atender otros procesos mientras se escribe el dato, y al finalizar la interrupción nos indicará que así fue.

... RESUMEN

En este capítulo hemos estudiado los cuatro mecanismos de interrupción con los que cuenta el PIC16F84A. Son sumamente útiles en muchos proyectos. Por eso, hemos visto algunas aplicaciones, tal como las temporizaciones exactas, usando la interrupción del Timer 0, o el uso de teclados, que es una aplicación típica de las interrupciones RBI. El uso de interrupciones nos permite el diseño de poderosas y eficientes aplicaciones de forma fácil.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es una interrupción?

2 ¿Cuántos mecanismos de interrupción tiene el PIC16F84A?

3 ¿Cuáles son los mecanismos de interrupción del PIC16F84A?

4 ¿A qué dirección salta el programa cuando se genera una interrupción?

5 ¿Para qué sirve el bit GIE del registro INTCON?

6 ¿En qué pin se genera la interrupción INT con un flanco de una señal?

7 ¿Por qué se deben borrar siempre las banderas de interrupción?

8 ¿Por qué los teclados se llaman "matriciales"?

9 ¿Qué es la latencia de interrupción?

10 ¿Por qué no se pone en modo sleep el microcontrolador cuando se usa la interrupción T0I?

PRÁCTICAS

1 Escriba el programa de la sección Interrupción externa INT en MPLAB, ensámblelo y grábelo en el microcontrolador para comprobar su funcionamiento.

2 Modifique el programa de la cerradura electrónica de tal forma que ahora la clave se guarde en la memoria EEPROM de datos. De esa manera, se permitirá cambiar la clave al usuario. Al presionar la tecla # se pedirá la clave actual y, si es correcta, se pedirá la clave nueva, la cual una vez tecleada se guardará en la EEPROM sustituyendo a la anterior.

3 Agregue el modo de 12 horas al reloj digital, de tal forma que mediante un pulsador conectado en algún pin libre del Puerto A se cambie entre los dos modos al presionarlo.

4 Agregue las rutinas necesarias para el ajuste de las horas y los minutos del reloj digital. Mediante un pulsador se entrará en modo de ajuste al presionarlo por 2 segundos (las horas comenzarán a parpadear en la pantalla) y mediante otro pulsador se incrementarán. Al presionar de nuevo el pin de ajuste se pasa a los minutos para ser ajustados. Al presionar por última vez el pulsador de ajuste, el reloj funcionará normalmente con el tiempo ajustado.

Capítulo 11

Comunicación en bus I2C

En este capítulo hablaremos del uso de un protocolo de comunicación serial llamado I2C, que pude ser de gran utilidad para usar dispositivos muy variados, con los cuales podemos extender las capacidades de nuestros microcontroladores, ya sea mediante memorias, expansores de puertos, sensores, y demás circuitos integrados que se comunicarán con nuestro PIC.

SERVICIO DE ATENCIÓN AL LECTOR: lectores@redusers.com

EL BUS I2C

La comunicación en serie entre diferentes dispositivos electrónicos es muy utilizada en la actualidad por sus ventajas sobre los métodos paralelos. **Philips** es el desarrollador de un protocolo de comunicación serial llamado **I^2C** o **IIC** (**Inter IC**), el cual permite la comunicación serial entre dispositivos con el uso de tan sólo dos líneas de comunicación. Hoy en día, existen muchos circuitos integrados que funcionan bajo el protocolo I2C, y que pueden resultarnos sumamente útiles para algunas aplicaciones con microcontroladores PIC. Es por eso que en esta sección estudiaremos el funcionamiento y la aplicación del bus I2C con el microcontrolador PIC16F84A.

Conexión de dispositivos I2C

La conexión de dispositivos I2C, como ya mencionamos, se realiza con únicamente dos líneas de comunicación. A través de éstas se llevará a cabo la comunicación serial entre los dispositivos o circuitos integrados conectados al bus.

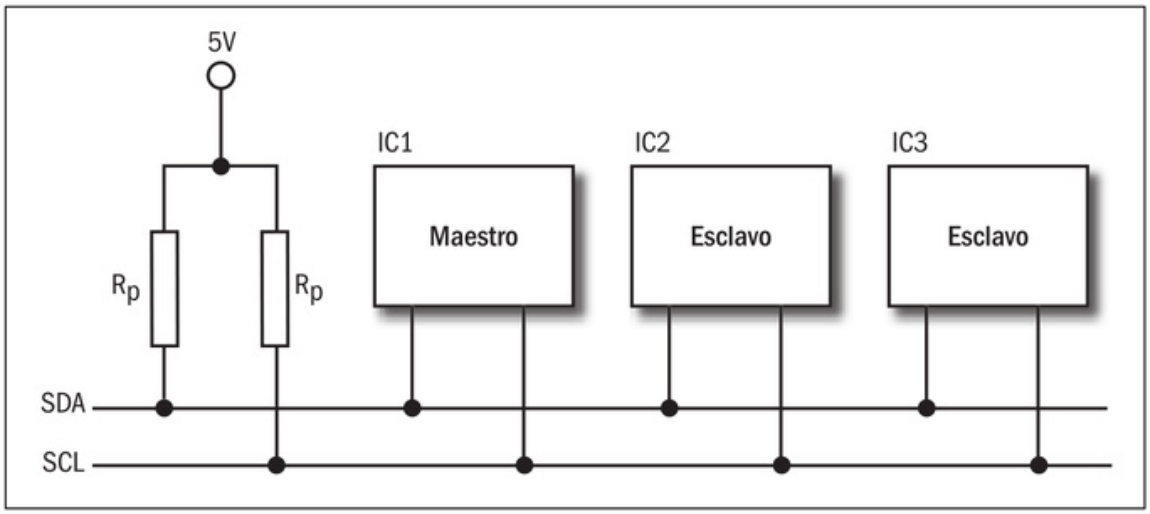


Figura 1. *Estructura de la conexión de un bus I2C.*

Existen dos líneas que forman el bus: una, llamada **SDA** (*Serial DAta line*), y otra, **SCL** (*Serial CLock line*). La línea SDA es por la que se transmitirán los datos y la

EL NOMBRE I2C

El bus del que estamos hablando es llamado **I^2C**, que significa **Inter integrated circuit**, o en español podríamos traducirlo como **inter circuito integrado**. Esto es debido al propósito de este bus de servir de comunicación entre dos o más circuitos integrados. Aunque el superíndice indica la doble I (I al cuadrado) es también común sólo escribirlo como I2C.

línea SCL es la encargada de sincronizar las transferencias y es la señal de reloj. Las líneas de comunicación SDA y SCL son del tipo **drenador abierto**, esto significa que los dispositivos I2C conectados al bus tienen una configuración interna de colector o drenador abierto, por lo cual sólo pueden llevar las salidas a nivel bajo, pero no a nivel alto, y es por eso que se usan los resistores de pull-up ($\mathbf{R_p}$) para mantener las líneas a nivel lógico alto.

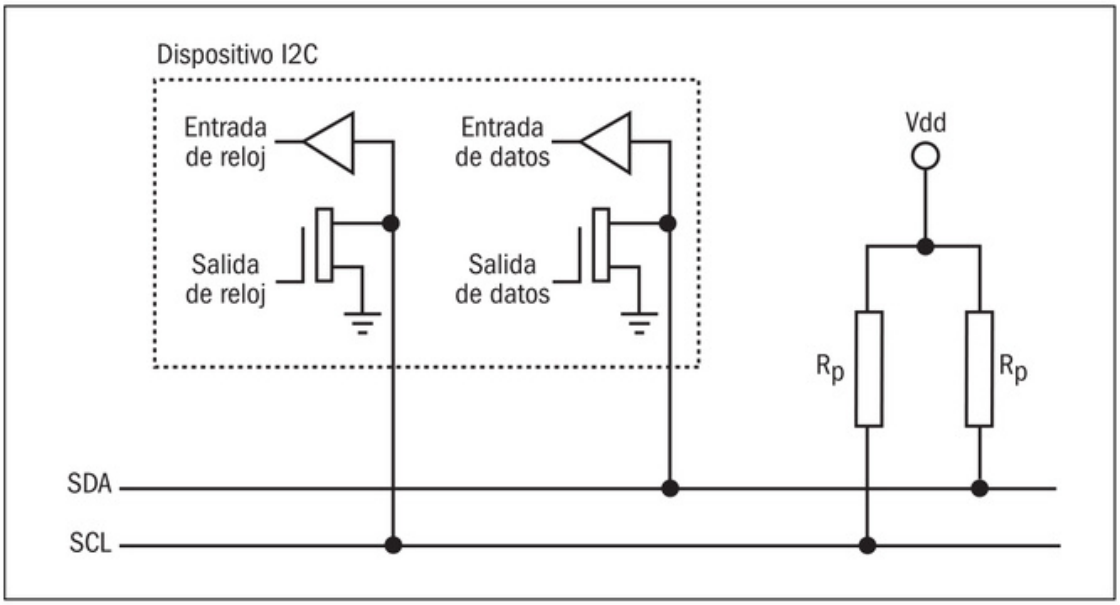


***Figura 2.** Configuración interna de entrada y salida de datos en un dispositivo I2C.*

De esta forma, cuando el bus esté inactivo estará siempre en un nivel alto, ya que los dispositivos se encontrarán en modo de alta impedancia y los resistores de pull-up obligarán al bus a permanecer en nivel alto. El número de dispositivos conectados al bus en principio no tiene límite, pero realmente éste estará dado por la capacitancia del bus al conectar dispositivos en él, que no debe superar los 400 pf. El valor de los resistores de pull-up no es muy crítico, puede usarse desde 1 kohm a 20 kohms. A un valor más bajo de los resistores disminuye la sensibilidad al ruido y mejora los tiempos de los flancos de bajada y subida, pero también aumentará el consumo del bus. Los valores recomendados están entre 1.5 kohms y 10 kohms.

El sistema está basado en la configuración **maestro/esclavo** o *master/slave*. El maestro es el encargado de generar la señal de reloj, y es quien inicia y detiene la transmisión de datos en el bus. Los esclavos sólo pueden transmitir hacia el maestro, o recibir datos del maestro, no pueden comunicarse entre sí. Los esclavos siempre están atentos, "escuchando" el bus, para cuando el maestro necesite transmitir o leer datos en alguno de ellos (lo indica mediante la dirección del esclavo), éste responderá. Cada esclavo tendrá, por lo tanto, una única dirección en el bus y se identificará con ella. El bus puede tener más de un maestro en lo que se llama configuración **multi-maestro**, pero esto no es muy común.

El protocolo I2C

Veamos cómo se transmite la información a través del bus I2C. En nuestro caso utilizaremos el microcontrolador como dispositivo maestro y, emulando el protocolo, se comunicará con dispositivos I2C para enviar o recibir datos de ellos. Por supuesto, la transmisión debe seguir un orden o protocolo para llevarse a cabo correctamente.

Condiciones START y STOP

Cuando no hay transmisión alguna en el bus, las líneas SDA y SCL estarán en estado alto, debido precisamente a los resistores de pull-up. Esta condición se llama **bus libre**. Cuando el maestro, que es el único que puede iniciar una comunicación, necesita hacerlo, lo lleva a cabo mediante una condición llamada **START**, o condición de **inicio**, en la cual el maestro lleva la línea SDA al estado bajo mientras la SCL permanece en alto, y luego lleva a SCL al estado bajo también. Esta condición de inicio hará que todos los dispositivos esclavos estén atentos ya que se ha iniciado una transferencia de datos.

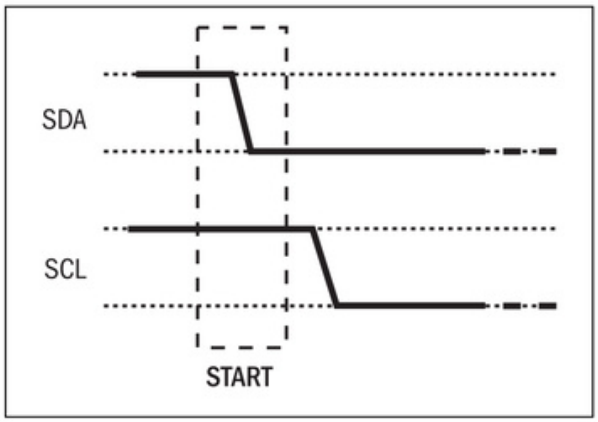


***Figura 3.** La condición START iniciará una comunicación en el bus I2C.*

Una vez que finaliza la transferencia de datos, el maestro enviará una condición de **STOP** al bus. La condición de STOP o **fin** es contraria a la de START. Primero, el maestro libera la línea SCL para que quede a nivel alto, y luego la línea SDA, quedando ambas a uno, es decir, nuevamente en el estado de bus libre.

III VENTAJAS DEL I2C

Las ventajas de usar el protocolo del bus I2C son evidentes. Por ejemplo, se tienen sólo dos líneas de comunicación, lo cual representa una gran ventaja frente a los buses paralelos que requieren muchas líneas o cables. Además, esto hace que los sistemas que lo utilizan sean más eficientes, pequeños, baratos, y más inmunes a las interferencias electromagnéticas.

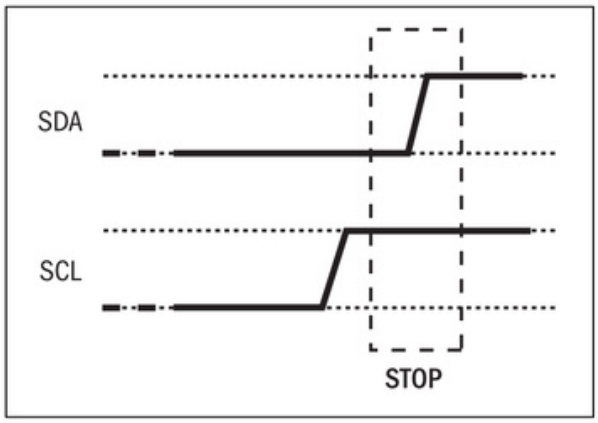


Figura 4. *La condición STOP terminará una comunicación en el bus I2C.*

Y de esta forma se inicia y se finaliza una transmisión de datos a través del bus. Después de la condición de inicio o START se enviarán o recibirán uno a uno los bits de datos en serie, según sea el caso.

Envío de datos

Una vez establecida la condición de inicio o START por parte del maestro, se iniciará la comunicación por bytes o grupos de 8 bits. Para la transmisión de datos, la línea SDA no puede cambiar de estado mientras SCL esté en alto, ya que esto sería interpretado como una condición de START o de STOP, por lo que, para transmitir un bit la línea SDA, sólo debe cambiar mientras la línea SCL esté en estado bajo.

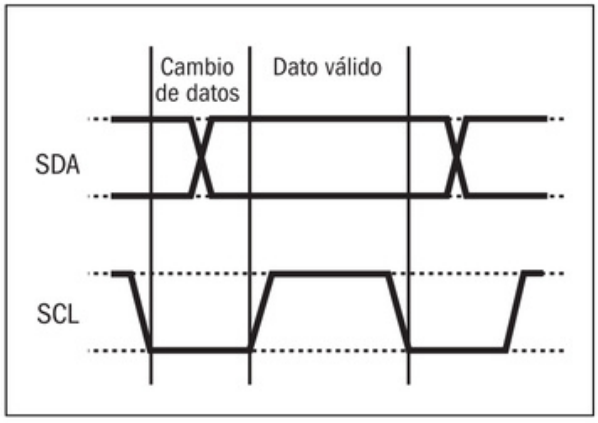


Figura 5. *El cambio de estado en SDA sólo debe ocurrir cuando SCL está en estado bajo.*

Para la transmisión de un byte en el bus se debe enviar primero el bit de más peso o MSB y terminar con el de menos peso o LSB. El primer byte de la comunicación siempre es del maestro hacia el bus y debe contener la dirección del esclavo al cual se quiere acceder en los 7 bits altos, y el bit más bajo es destinado a indicar si se realizará lectura o escritura en el esclavo definido por la dirección anterior. Este bit es llamado **R/W**. Se transmite un bit en cada pulso de reloj de SCL.

Si el bit **R/W = 0** el esclavo recibirá datos del maestro
Si el bit **R/W = 1** el esclavo enviará datos al maestro

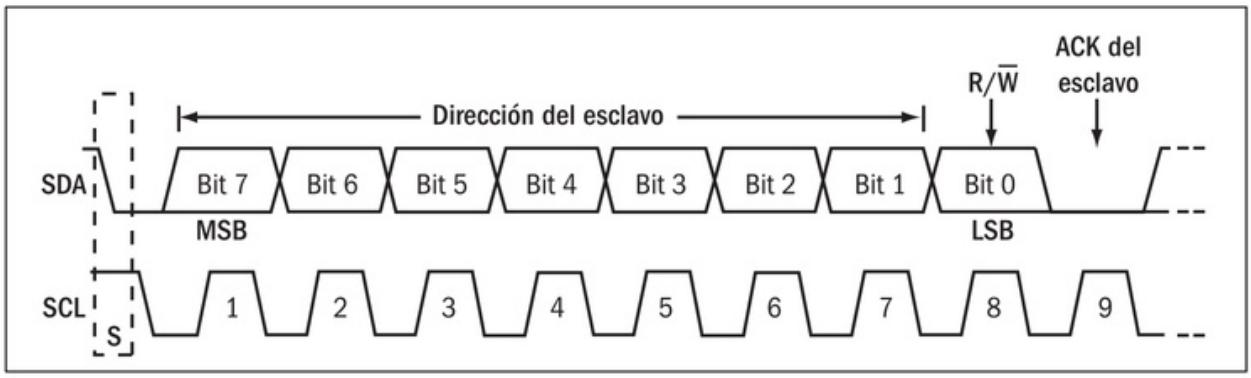


***Figura 6.** Formato para la transmisión del primer byte a través del bus I2C.*

Una vez que se ha transmitido este byte con la dirección y el bit R/W, el dispositivo esclavo que tiene la dirección correcta enviará un bit de reconocimiento o *acknowledgement* (se puede abreviar **ACK**) en el noveno pulso de reloj, indicando al maestro que está listo para el envío o recepción de datos. El bit de reconocimiento es obligatorio para completar cada transmisión de un byte, excepto en el último byte leído de un esclavo, y quien enviará este bit será quien reciba los datos, ya sea algún esclavo o el propio maestro. En ambos casos, el maestro es quien generará el pulso de reloj para el bit ACK. También se puede enviar un bit de reconocimiento negado (**NOT ACK** o **NACK**) es decir, que equivale a un estado alto en el bit de reconocimiento.

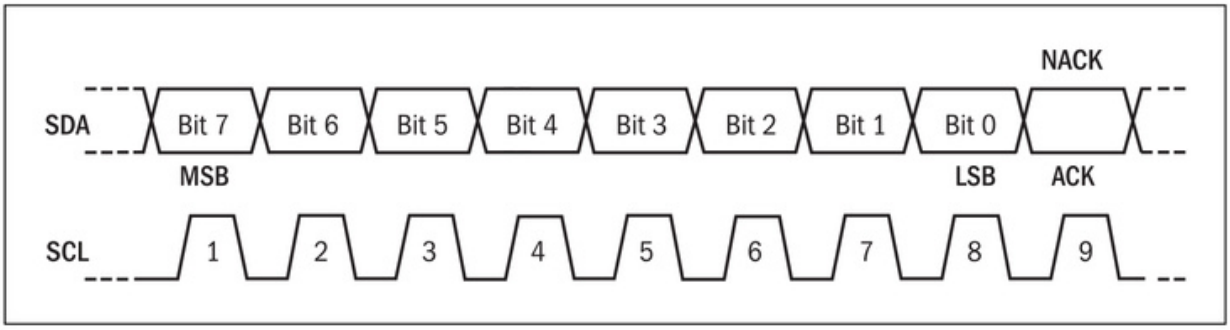


***Figura 7.** Formato para la transmisión o recepción de un byte a través del bus I2C. La dirección depende del bit R/W del primer byte.*

Si uno de los esclavos no puede recibir o transmitir datos debido a que se encuentra ocupado en procesos internos, entonces puede mantener la línea SCL en bajo, obligando al maestro a entrar en modo de espera hasta que el esclavo libera la línea y la transmisión puede llevarse a cabo.

Transferencia de maestro a esclavo

Una de las principales tareas es cuando el maestro envía datos hacia uno de los esclavos, es decir, cuando simplemente el maestro inicia la transferencia. En el primer

byte indica la dirección del esclavo hacia el cual enviará datos, el esclavo responde con el bit ACK y entonces se le envía uno o más bytes consecutivamente. Al finalizar, el maestro enviará la condición STOP para terminar la transmisión.

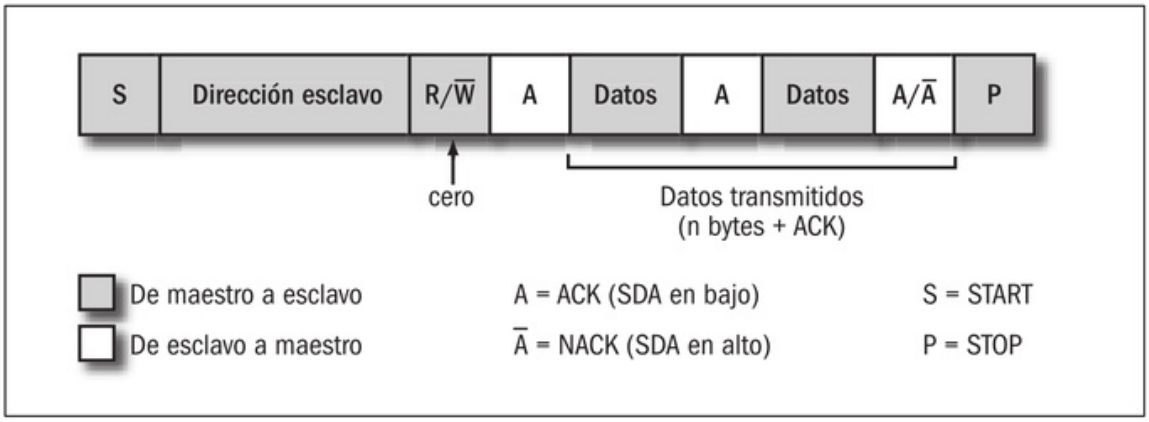


***Figura 8.** Transmisión de datos maestro-esclavo a través del bus I2C.*

Transferencia de esclavo a maestro

También los esclavos podrán enviar datos al maestro cuando éste así lo requiera. En este caso, el bit R/W deberá ser 1. El primer bit ACK debe enviarlo el esclavo para indicarle al maestro que está listo para el envío, entonces comenzará el envío de datos y los siguientes bits ACK los deberá enviar en este caso el maestro para indicarle al esclavo que ha recibido cada byte.

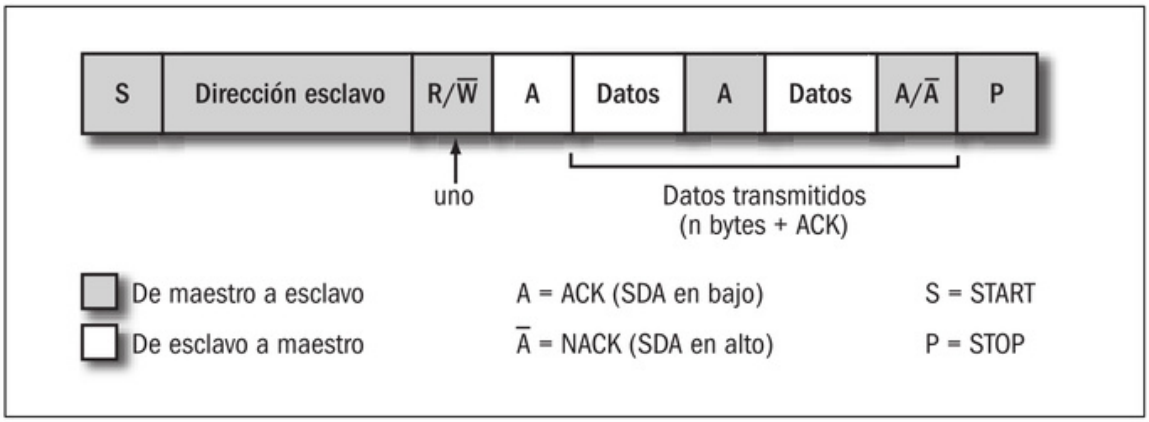


***Figura 9.** Transmisión de datos esclavo-maestro a través del bus I2C.*

NO OLVIDAR PULL-UPS

En el bus I2C se usan los resistores de pull-up para mantener el estado del bus en alto, ya que los dispositivos conectados tienen configuración de colector o drenador abierto, y sólo pueden enviar ceros y no unos. Las primeras veces que se usa el bus I2C es muy común que nos olvidemos de colocar los resistores, y por lo tanto el bus no funciona.

En la transferencia del esclavo hacia el maestro, éste no debe enviar el bit ACK en el último byte, sino que en él se debe enviar un bit NACK, para que el esclavo deje libre el bus y el maestro pueda enviar la condición de STOP. Si se envía el bit ACK en el último byte, el esclavo puede no liberar el bus ya que asumirá que el envío no ha terminado, y por lo tanto el maestro no podrá enviar la condición de STOP si el bus no está libre.

Transferencia mixta

También puede haber un cambio en la dirección de transferencia. Si el maestro necesita seguir comunicándose con el mismo esclavo o con otro, en lugar de generar la condición STOP puede enviar una nueva condición de START, llamada **START repetido** o **Sr**, y con ello iniciará una nueva transferencia de datos, en la que el bit R/W puede cambiar la dirección de transferencia.

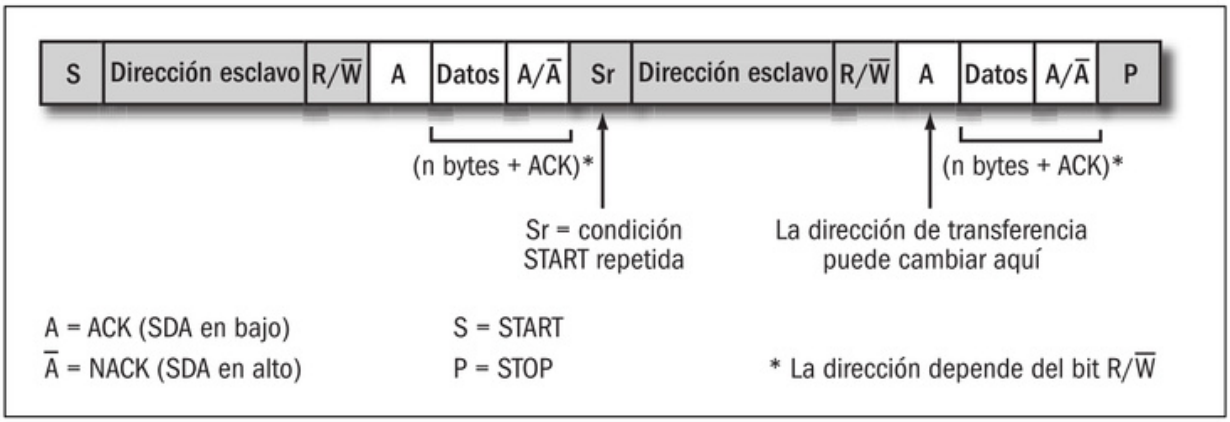


***Figura 10.** Mediante la condición Sr se inicia una nueva transferencia de datos.*

Temporizaciones

El bus I2C tiene tres velocidades de trabajo:

- **Modo estándar**: es el modo más utilizado, con una velocidad aproximada de 100 kb/s. Es la velocidad que usaremos nosotros para trabajar con el PIC16F84A y los dispositivos I2C.

¿NOT ACK O NO ACK?

En la transmisión de un esclavo hacia el maestro se debe enviar un **NACK** o **ACK negado** en el último byte. Esto es para que el esclavo deje libre el bus. Debemos observar que técnicamente no es lo mismo enviar un NACK que **no enviar ACK**, esto último puede suceder cuando un esclavo no recibe bien los datos del maestro por algún problema y no envía el bit ACK.

- **Modo rápido**: en este modo la velocidad de transferencia está en unos 400 kb/s.
- **Modo de alta velocidad**: es el modo más rápido que puede alcanzar un bus I2C. Está en el rango de aproximadamente 3.4 Mb/s.

Como nosotros usaremos el modo estándar, la frecuencia máxima de reloj en la línea SCL será de 100 KHz. Los tiempos para las señales en el bus son los que vemos representados en el diagrama de la **Figura 11**.

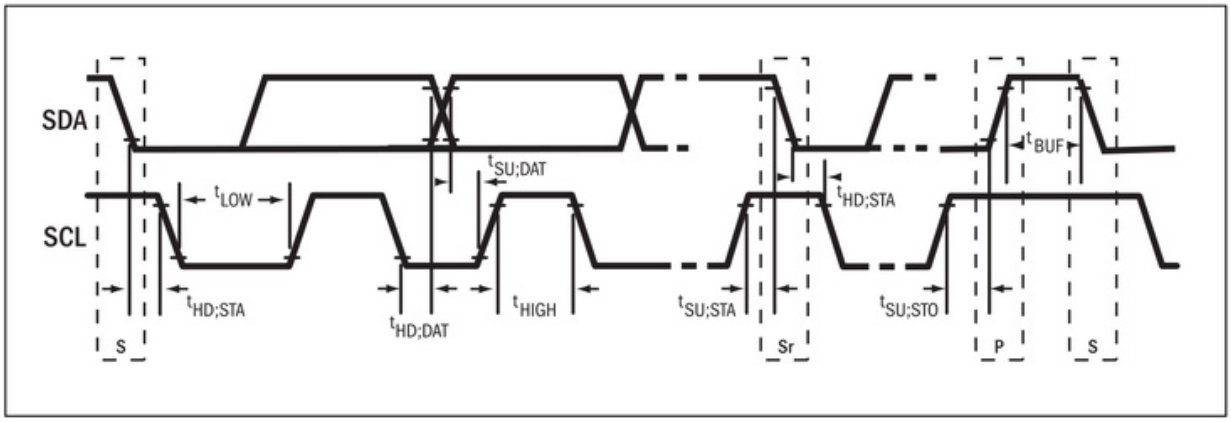


***Figura 11.** Tiempos definidos para el modo estándar en las señales del bus I2C.*

Debemos tener en cuenta estos tiempos mínimos para que las transmisiones sean exitosas en el bus. En la **Tabla 1** encontraremos la explicación de cada uno de los tiempos marcados en la **Figura 11**.

PARÁMETRO	SÍMBOLO	TIEMPO MÍNIMO	UNIDADES
Condición de espera para la condición START o START repetido. Tiempo entre el flanco de bajada en SCL y el flanco de bajada en SDA.	tHD;STA	4	Microsegundos
Tiempo en nivel bajo de la señal de reloj en SCL.	tLOW	4.7	Microsegundos
Tiempo de mantenimiento de dato. Tiempo entre el flanco de bajada en SCL y el cambio de dato en SDA.	tHD;DAT	5	Microsegundos
Tiempo de colocación de dato. Tiempo entre el cambio de dato en SDA y el flanco de subida en SCL.	tSU;DAT	250	Nanosegundos
Tiempo en nivel alto de la señal de reloj en SCL.	tHIGH	4	Microsegundos
Tiempo de inicio de la condición de START repetido. Tiempo en que debe estar libre el bus antes de la condición de inicio repetido.	tSU;STA	4.7	Microsegundos
Tiempo de inicio de la condición STOP. Tiempo entre el flanco de subida en SDA y el flanco de subida en SCL.	tSU;STO	4	Microsegundos
Tiempo que debe estar el bus libre entre una condición de STOP y una START.	tBUF	4.7	Microsegundos

***Tabla 1.** Tiempos mínimos de las señales del modo estándar del bus I2C.*

La conexión de los dispositivos I2C al PIC16F84A la realizaremos en las líneas RA3 y RA4, siendo la línea RA3 el reloj SCL, y RA4 el bus de datos SDA.

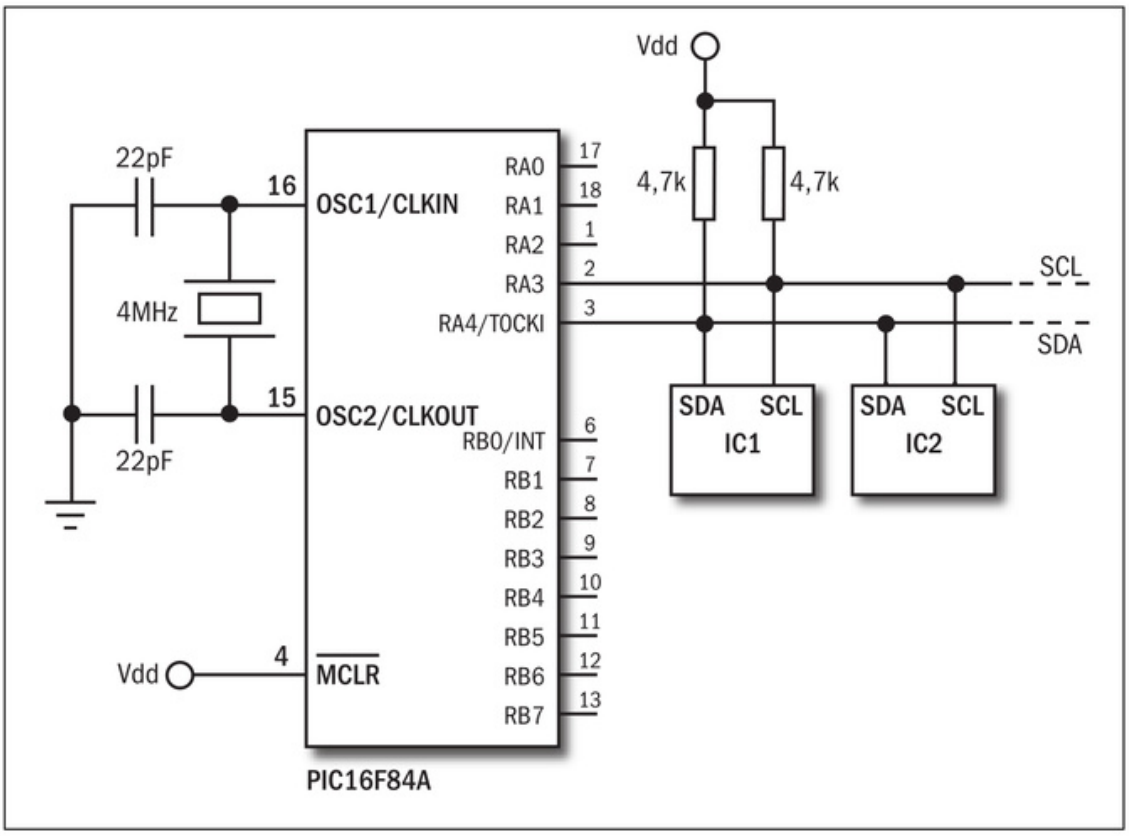


***Figura 12.** Ejemplo de conexión de dos dispositivos I2C al PIC16F84A.*

El microcontrolador actuará como maestro, controlando la comunicación en el bus entre él y los dispositivos I2C conectados, para enviar o recibir datos de éstos.

Librería I2C

Ahora que ya conocemos el protocolo, la conexión y la temporización de la comunicación del bus I2C, podemos escribir una librería para las comunicaciones I2C con el PIC16F84A. A continuación, incluimos la librería **I2C.INC** que podemos descargar de **www.redusers.com** para su utilización y su estudio:

I2C Y LOS APARATOS DE CONSUMO

El bus I2C se plantea inicialmente para ser utilizado en equipos electrónicos de consumo masivo, como televisores, sistemas de video, audio, etcétera. Es por ello que podemos encontrar muchos circuitos integrados I2C que realizan funciones propias de estos aparatos, como procesadores de video, ecualizadores, controles de volumen, amplificadores, etcétera.

```
      CBLOCK
      I2C_byte, cuenta_bits, ACKflag
      ENDC

      #DEFINE SCL            PORTA, 3
      #DEFINE SDA            PORTA, 4
      #DEFINE banco0  bcf    STATUS, RP0
      #DEFINE banco1  bsf    STATUS, RP0

SCL_alto            MACRO
      banco1
      bsf           SCL                  ;Libera la línea SCL para ponerla
      en alto
      banco0
      ENDM

SCL_bajo            MACRO
      banco1
      bcf           SCL                  ;SCL como salida
      banco0
      bcf           SCL                  ;SCL a nivel bajo
      ENDM
SDA_bajo            MACRO
      banco1
      bcf           SDA                  ;SDA como salida
      banco0
      bcf           SDA                  ;SDA a nivel bajo
      ENDM
;Subrutina para enviar una condición de inicio (START) al bus I2C:
I2C_START
      call          bus_libre            ;Verifica si el bus esta libre
      banco1
      bsf           SCL                  ;SDL como entrada, libera la
línea SCL
      bcf           SDA                  ;SDA como salida
      banco0
      bsf           SDA                  ;SDA en alto
      goto          $+1                  ;Espera (ajusta tiempo tBUF)
      goto          $+1
      bcf           SDA                  ;SDA a nivel bajo
```

```
     SCL_bajo                          ;SCL a nivel bajo
     return
;Subrutina para enviar una condición de fin (STOP) al bus I2C:
I2C_STOP
     SDA_bajo                          ;SDA a nivel bajo
     SCL_alto                          ;SCL a nivel bajo
     goto          $+1                 ;Espera (ajusta tiempo tSU;STO)
     bsf           SDA                 ;SDA a uno
     return
;Subrutina para enviar un byte a través del bus I2C:
I2C_enviaByte
     movwf         I2C_byte            ;Obtiene el byte a enviar
     movlw         d'8'
     movwf         cuenta_bits         ;Contador para enviar los 8 bits
     banco1
     bcf           SDA                 ;SDA como salida
     banco0
I2C_enviaBit
     btfss         I2C_byte, 7         ;¿Bit a enviar = 1?
     goto          I2C_cero            ;No, envía un cero
     bsf           SDA                 ;Si, envía un uno
     goto          SCL_reloj
I2C_cero
     bcf           SDA                 ;SDA a nivel bajo
SCL_reloj
     SCL_alto                          ;SCL en alto para el puso de reloj
     goto          $+1                 ;Espera (ajusta tiempo tHIGH)
     SCL_bajo                          ;SCL en bajo, termina el pulso de
reloj
     rlf           I2C_byte, F         ;Rota el byte para enviar el
siguiente bit
     decfsz        cuenta_bits, F      ;¿Fue el último bit?
     goto          I2C_enviaBit        ;No, continua enviando bits
     banco1                            ;Si, da el pulso 9 para recibir
el ACK
     bsf           SDA                 ;SDA como entrada
     bsf           SCL                 ;SCL como entrada, SCL a uno
     banco0
     goto          $+1                 ;Espera (ajusta tiempo tHIGH)
     SCL_bajo                          ;SCLA a nivel bajo
```

```
     return                              ;Terminó el envío

;Subrutina para leer un byte desde un esclavo en el bus I2C:

I2C_leeByte
     banco1
     bsf            SDA                 ;SDA como entrada
     banco0
     movlw          d'8'
     movwf          cuenta_bits         ;Cuenta los bits a leer
I2C_recibeBit
     SCL_alto                           ;SCL en alto
     nop                                ;Espera 1us
     btfss          SDA                 ;¿Bit recibido = 1?
     goto           lee_cero            ;No, recibe un cero
     bsf            STATUS, C           ;Si, recibe un uno en carry
     goto           rota
lee_cero
     bcf            STATUS, C           ;Coloca un cero en carry
rota
     rlf            I2C_byte, F         ;Rota para recibir el siguiente
bit
     SCL_bajo                           ;SCL a nivel bajo
     decfsz         cuenta_bits, F      ;¿Bits recibidos = 8?
     goto           I2C_recibeBit       ;No, continúa recibiendo bits
     btfsc          ACKflag, 0          ;¿Es el último byte a leer?
     goto           I2C_NACK            ;Si, no enviar ACK
     SDA_bajo                           ;No, envía ACK
I2C_NACK
     SCL_alto                           ;Inicio del pulso de reloj para ACK
     goto           $+1                 ;Espera (tiempo tHIGH)
     SCL_bajo                           ;Fin del pulso de reloj para ACK
     movf           I2C_byte, W         ;Pasa el byte leído a W
     return
;Subrutina para verificar si el bus esta libre leyendo si SDA esta en alto:
     bus_libre
     banco1
     bsf            SDA                 ;SDA como entrada
banco0
     btfss          SDA                 ;¿SDA = 1?
```

```
goto          bus_libre      ;No, bus ocupado, espera
return                       ;Si, bus libre
```

Podemos observar cómo, con el PIC16F84A, se emula el protocolo I2C para poder comunicarnos con dispositivos integrados, para poder diseñar circuitos muy versátiles.

En la librería tenemos las subrutinas necesarias para establecer la comunicación. La subrutina **I2C_START** genera una condición de START en el bus para comenzar la comunicación. La subrutina **I2C_enviaByte** sirve para enviar un byte a alguno de los esclavos del bus. La subrutina **I2C_leeByte** leerá un byte enviado por alguno de los esclavos hacia el microcontrolador, mientras que la subrutina **I2C_STOP** generará la condición de STOP para finalizar la recepción o el envío de información en el bus.

Es importante notar cómo, la mayoría de los tiempos requeridos, los dan las propias instrucciones, por lo tanto, no es necesario usar retardos, excepto en algunos casos, para ajustar y cumplir con los tiempos mínimos que marca el protocolo para el modo estándar. Hemos utilizado algunas macros para facilitar la escritura y la lectura de la librería, no hemos empleado subrutinas para evitar los tiempos de los saltos de la instrucción **call** y **return**, lo cual le quitaría un poco de velocidad a la transmisión. Con esta librería se puede transmitir muy cerca de la velocidad máxima del modo estándar del bus I2C.

Es muy importante tener en cuenta ciertos detalles del uso de esta librería: por un lado, la subrutina **I2C_START** permite enviar las condiciones de START y de START repetido (Sr). Y para la lectura de bytes desde alguno de los esclavos se debe usar el registro ACKflag para indicar si el byte que está por leerse es el último o no. Recordemos que en el último byte leído desde alguno de los esclavos se debe enviar el bit de reconocimiento negado (NACK), por lo que el bit 0 del registro ACKflag se usa para indicar con un 0 en él si no es el último byte a leer, y con un 1 si es el último byte, para que la librería genere el bit NACK en el último byte a leer de alguno de los esclavos del bus.

EXPANSOR DE PUERTOS I2C PCF8574

Un sencillo ejemplo de un circuito integrado que podemos utilizar mediante la conexión en bus I2C es un **PCF8574**, el cual es un expansor de puerto de 8 bits. Es decir, con este circuito integrado podemos colocar un puerto más de entada/salida a nuestro PIC, el cual se comunicará a través del bus I2C.

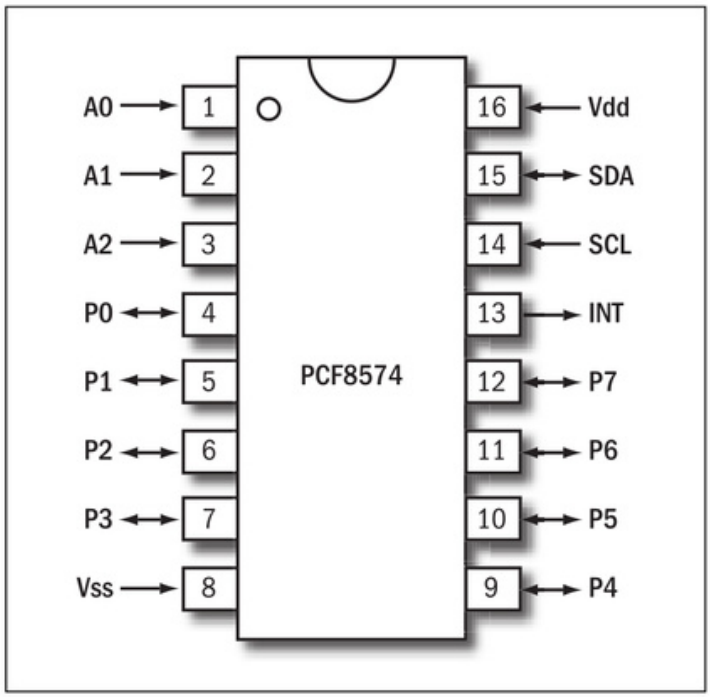


Figura 13. *El PCF8574 contiene un puerto de entrada/salida de 8 líneas (P0 a P7).*

Las características de este circuito integrado son:

- Expansor remoto de 8 bits (líneas) de entrada o salida
- Opera con voltaje de alimentación de 2.5 a 6 V
- Salida de interrupción de drenador abierto
- Compatible con la mayoría de los microcontroladores

Cuando los puertos de nuestro PIC son insuficientes, o simplemente queremos agregar más, podemos recurrir a expansores como éstos, para aumentar las líneas de entrada y/o salida. En un solo bus I2C podemos conectar hasta 8 circuitos integrados PCF8574 o hasta 16 si lo combinamos con el PCF8574A (como veremos en la siguiente sección), y de esta forma podemos tener hasta 128 líneas de entrada/salida disponibles para nuestro PIC.

Dirección como esclavo del PCF8574

Como todo circuito que funcione bajo el protocolo de comunicación I2C, el PCF8574 debe identificarse mediante una dirección específica para poder acceder a él para leer o escribir. Los 3 bits más bajos de esta dirección están dados por los pines de entrada A0, A1 y A2, del circuito integrado, los cuales podemos configurar según necesitemos. Los 4 bits más altos de la dirección son fijos y están dados por el fabricante. En el caso del PCF8574, tiene asignados los bits 0100, mientras que para el PCF8574A la parte fija es 0111.

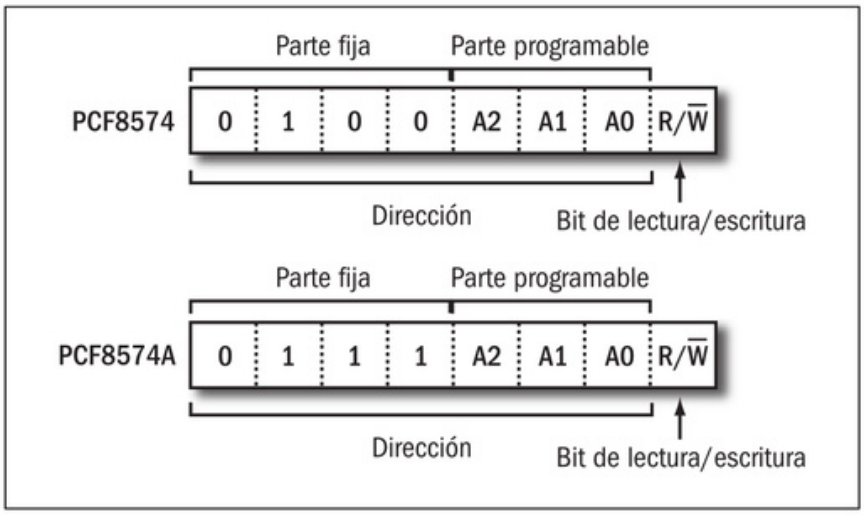


Figura 14. *Especificación de dirección como esclavo para el PCF8574 y el PCF8574A.*

De esta forma, si conectamos las terminales A0, A1 y A2 del PCF8574 a tierra, la dirección de éste en el bus será 0100000x, siendo **x** el bit que define si vamos a escribir o leer en él. Si conectamos las terminales A0, A1 y A2 a Vdd, entonces la dirección será 0100111x. Como podemos apreciar, la diferencia entre el PCF8574 y el PCF8574A es precisamente la dirección que tienen como esclavos en el bus. De esta forma, se pueden conectar hasta 16 expansores en un solo bus I2C.

El PCF8574 dispone de un puerto de 8 líneas que pueden ser usadas tanto como entrada o como salida de datos, y no necesitan configurarse. Cuando se usan como puerto de salida, simplemente hay que enviar el dato al PCF8574 a través del bus I2C, y éste lo colocará automáticamente en el puerto. Para usar el puerto como entrada simplemente hay que leer el puerto a través del bus I2C.

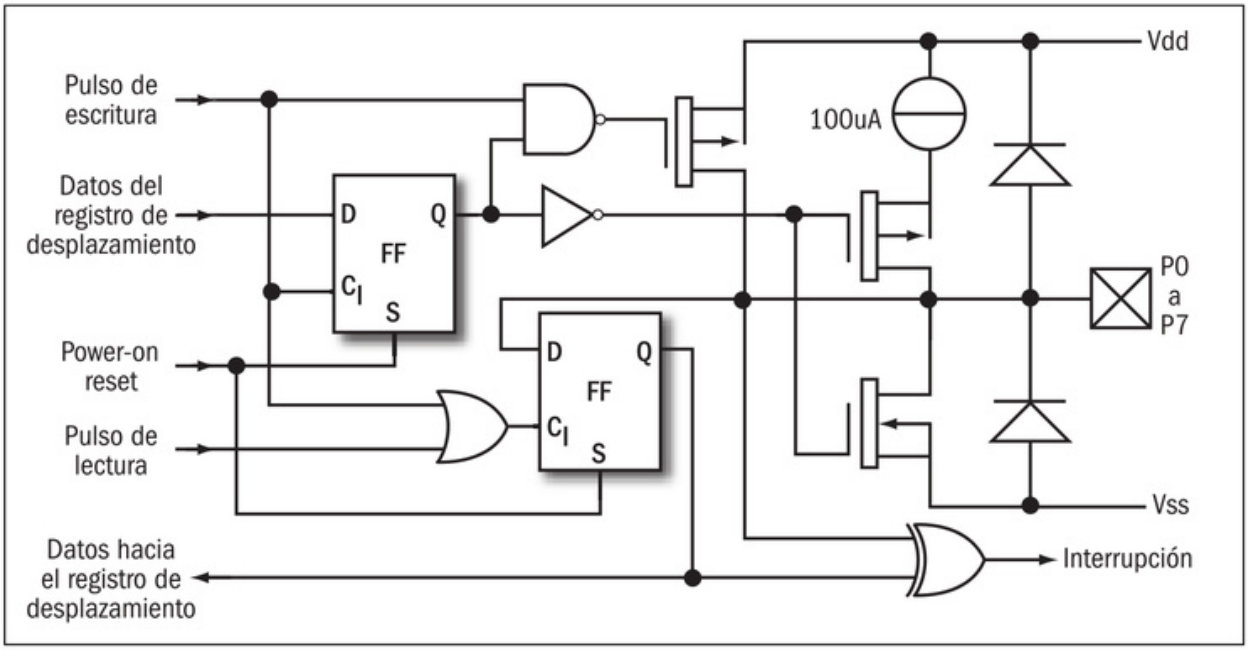


Figura 15. *Configuración interna de cada uno de los pines del puerto del PCF8574.*

PCF8574 como puerto de salida

Para utilizar el PCF8574 como puerto de salida simplemente debemos enviarle los datos que necesitemos obtener a la salida y éstos se reflejarán automáticamente en el puerto del PCF8574. Debemos tener en cuenta que la corriente máxima que cada línea del puerto puede entregar es de 25 mA en estado bajo, es decir, cuando la corriente entra al puerto, pero la salida de corriente del puerto es de sólo 300 uA, así que no debemos sobrepasar ese límite. El puerto puede manejar directamente leds, pero únicamente en estado bajo, no en estado alto.

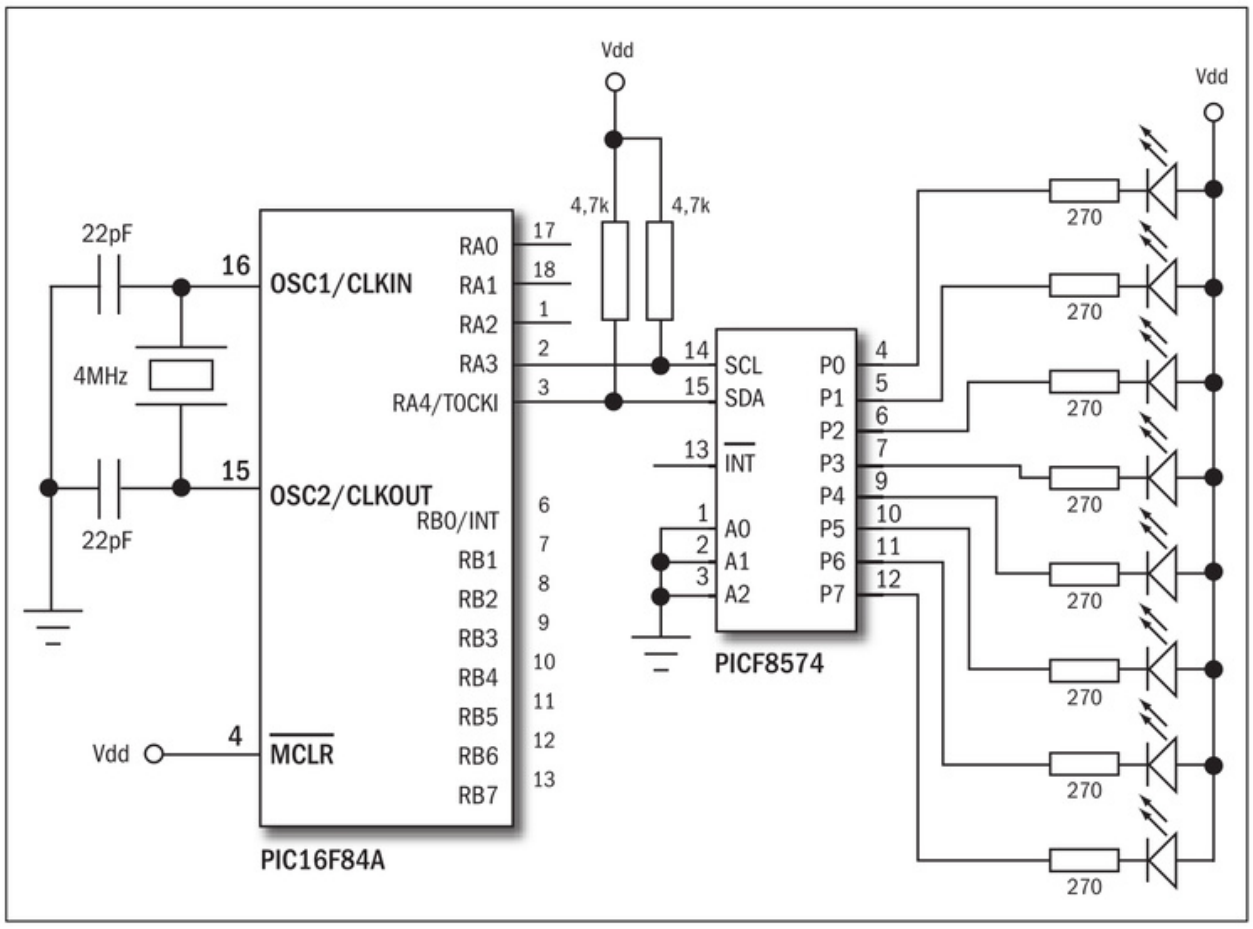


***Figura 16.** Circuito de ejemplo para uso del PCF8574 como puerto de salida.*

Veamos un ejemplo del uso del PCF8574 como puerto de salida. Simplemente haremos un contador binario que se mostrará a la salida del puerto:

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>
```

```
      CBLOCK 0x0C
      salida
      contador1
      contador2
      ENDC

      ORG           00h
      movlw         b'11111111'
      movwf         salida                ;Inicializa el registro de salida

inicio
      call          I2C_START             ;Condición de START
      movlw         b'01000000'           ;Dirección PCF8574 + bit escritura
      call          I2C_enviaByte
      movf          salida, W             ;Pasa el valor de salida a W
      call          I2C_enviaByte         ;Envíalo al PCF8574
      call          I2C_STOP              ;Condición de STOP
      call          retardo               ;Espera
      decf          salida, F             ;Decrementa salida
      goto          inicio                ;Bucle

retardo                                   ;Retardo aprox. 500ms
      movlw         d'244'
      movwf         contador1
loop1
      movlw         d'0'
      movwf         contador2
loop2
      goto          $+1
      goto          $+1
      nop
```

III INTERRUPCIÓN DE VARIOS PCF8574

Si tenemos varios PCF8574 conectados a nuestro microcontrolador y necesitamos usarlos como entradas de datos, al menos a más de uno, entonces deberemos usar la salida de interrupción en ellos. Dado que la salida de interrupción del PCF8574 es de drenador abierto, podemos unirlas formando una AND cableada para conectarlas al PIC.

```
        decfsz      contador2, F
        goto        loop2
        decfsz      contador1, F
        goto        loop1
        return

        #INCLUDE    <I2C.INC>           ;Se incluye librería I2C

        END
```

Si analizamos en detalle el código anterior podemos observar cómo, simplemente, se envía el byte que será la salida al PCF8574 mediante el bus I2C, y éste reflejará el dato enviado en el puerto. Notemos también cómo no podemos encender los leds a la salida del puerto enviándoles un 1, ya que la corriente de salida no es suficiente. En su lugar, encenderemos los leds con un 0, por lo que la salida se inicializa en 11111111, y en lugar de un incremento se hace un decremento, para obtener a la salida un contador ascendente. Como podemos ver, el manejo del PCF8574 como salida es muy sencillo.

PCF8574 como puerto de entrada

Podemos usar también de forma simple el puerto del PCF8574 como entradas de datos, simplemente leyéndolas. La fuente de corriente de 100 microamperes en cada una de las líneas (**Figura 15**) nos permite usar el puerto como entrada sin necesidad de colocar resistores de pull-up externos. Si hemos usado previamente el puerto como salida de datos, es conveniente primero enviar unos a las líneas que vayan a ser usadas como entradas, para que funcionen correctamente. Al encender la alimentación, todas las líneas del PCF8574 estarán en estado alto, lo cual las configura como entradas.

Salida de interrupción

El PCF8574 dispone de una salida de interrupción que nos puede ser útil para generar una interrupción en el microcontrolador cuando los datos en el puerto cambien. La salida de interrupción es de drenador abierto, por lo tanto, debemos usar resistores de pull-up en ella. Cuando se genera un cambio de estado en las líneas de entrada, esta línea de interrupción envía un flanco de bajada para disparar la interrupción en el PIC. De esta forma, el PIC sabrá que hay nuevos datos sin tener que leer el PCF8574 a través del bus I2C. La salida de interrupción regresará al estado alto cuando el puerto regrese al estado anterior al cambio, o bien cuando se realice una lectura o escritura en el PCF8574 a través del bus I2C.

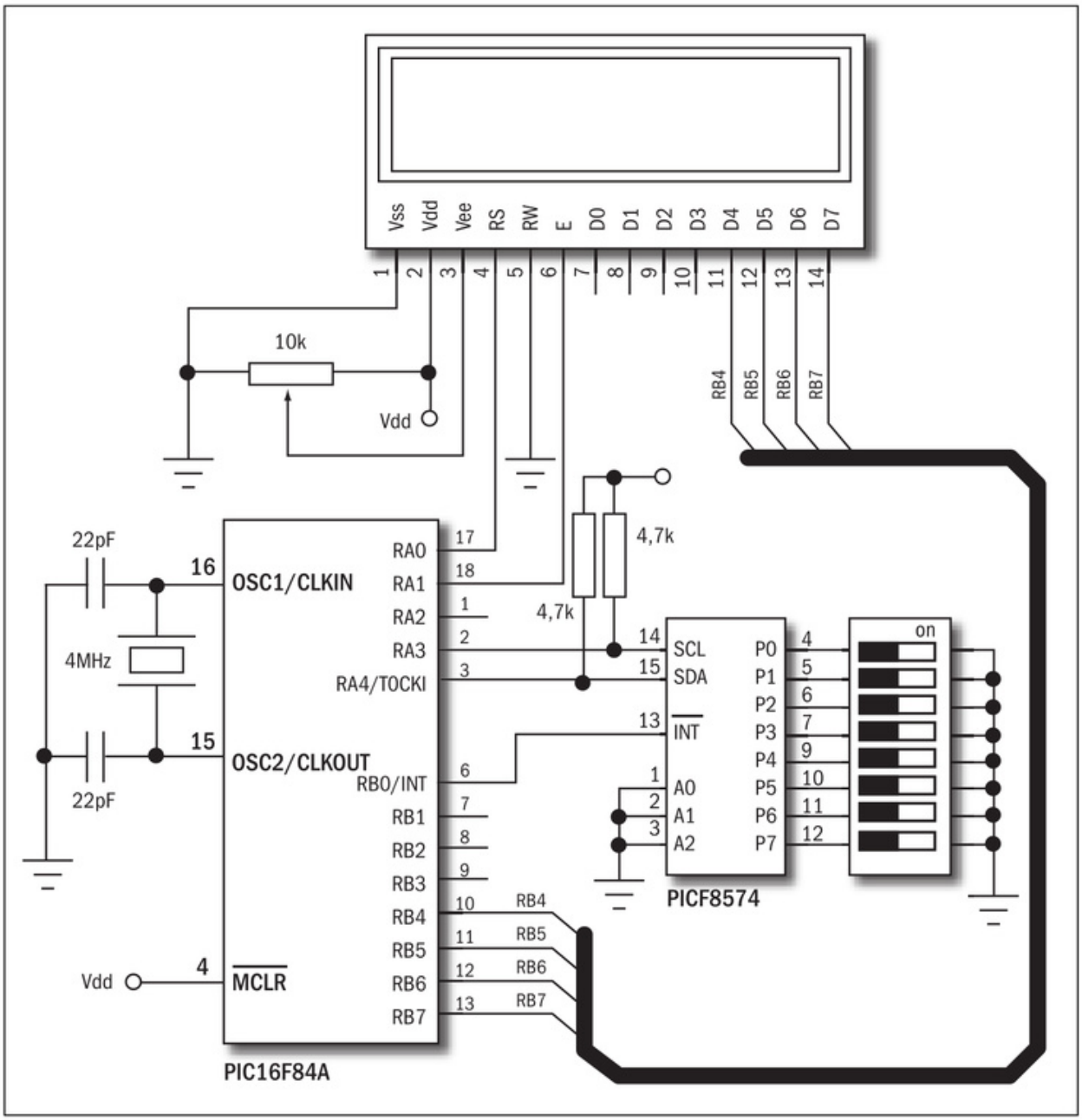


Figura 17. *Circuito de ejemplo para el uso de las líneas del PCF8574 como entradas.*

Veamos un ejemplo del PCF8574 como entrada de datos. Conectaremos un dip-switch al puerto del PCF8574 para enviar un dato, el cual será leído por el PIC y nos dará su valor decimal en el LCD. Usaremos la salida de interrupción para generar una interrupción INT y saber cuándo los datos de entrada cambiaron.

```
__CONFIG _CP_OFF & _WDT_OFF & _PWRTE_ON & _XT_OSC

PROCESSOR 16F84A
#INCLUDE <P16F84A.INC>
```

```
     CBLOCK 0x0C
     ENDC

     ORG            00h
     goto           inicializa
     ORG            04h
     goto           interrupcion

inicializa
     call           LCD_inicializa ;Inicializa el LCD

     call           interrupcion          ;Muestra el valor que hay al
principio

     bsf            STATUS, RP0
     bsf            TRISB, 0              ;RB0 como entrada
     movlw          b'00000000'
     movwf          OPTION_REG            ;Activa pull-ups, y INT con
flanco de bajada
     bcf            STATUS, RP0
     movlw          b'10010000'
     movwf          INTCON                ;Habilita INT y GIE

inicio
     sleep                                ;Espera la interrupción
     goto           inicio

interrupcion
     call           I2C_START             ;START
     movlw          b'01000001'           ;Dirección PCF8574 + bit lectura
     call           I2C_enviaByte
     bsf            ACKflag, 0            ;Es el último byte a leer
     call           I2C_leeByte           ;Lee el puerto
     call           I2C_STOP              ;STOP

     call           BINaBCD               ;Convierte el valor leído a BCD
     call           borra_display         ;Borra el LCD
     movf           BCDcentenas, W
     addlw          '0'
     call           LCD_caracter
```

```
        movf        BCDdecenas, W
        addlw       '0'
        call        LCD_caracter
        movf        BCDunidades, W
        addlw       '0'
        call        LCD_caracter
        bcf         INTCON, INTF
        retfie

        #INCLUDE    <I2C.INC>
        #INCLUDE    <LCD4BITS.INC>
        #INCLUDE    <BINABCD.INC>

        END
```

Como podemos apreciar es bastante fácil el uso del PCF8574 como puerto de entrada. La interrupción le notifica al PIC16F84A cuando ha cambiado la entrada, y de esa forma se lee el nuevo dato cada vez que así ocurre. No es necesario inicializar el puerto enviando unos, ya que, como mencionamos antes, al encender el circuito automáticamente se configura así. Al leer el estado del puerto del PCF8574 se borra la salida de interrupción, por lo que el circuito siempre estará atento ante cualquier cambio en las líneas de entrada. No utilizamos pull-ups externos ni en el puerto del PCF8574, ya que los tiene internos, ni en la salida de interrupción, ya que activamos los pull-ups internos del puerto B del PIC16F84A. La principal ventaja del PCF8574 es proporcionar un puerto de 8 bits extra, de modo que podemos aprovechar esta característica para conectar nuestro LCD, un teclado, un display de 7 segmentos, o cualquier otro circuito de entrada o salida. De esta forma tenemos la posibildad de construir un sistema muy complejo al aumentar la capacidad de líneas de entrada/salida del microcontrolador.

DS1621 TERMÓMETRO Y TERMOSTATO EN BUS I2C

Como otro ejemplo del bus I2C, tomaremos el circuito integrado **DS1621**, que es un sensor de temperatura y termostato. Sus características son:

- No necesita de componentes externos
- Mide temperatura en un rango de -55 a 125 grados centígrados

- La temperatura se mide en 9 bits (dos bytes)
- Puede alimentarse con voltaje de 2.7 a 5.5 V
- La conversión de temperatura se hace en menos de un segundo
- Los límites del termostato son no volátiles y se pueden definir por el usuario
- Utiliza la técnica delta-sigma para la conversión A/D

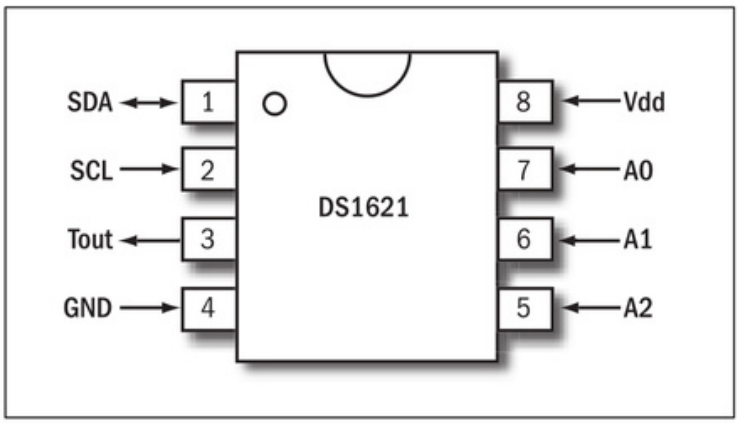


***Figura 18.** El DS1621 es un circuito integrado de tan solo 8 pines con un sensor de temperatura dentro de él.*

La función de los pines del DS1621 es:

SDA: línea de salida/entrada datos SDA del bus I2C
SCL: línea de entrada de la señal de reloj SCL del bus I2C
Tout: señal de salida de la alarma del termostato
GND: conexión de tierra
Vdd: conexión del voltaje de alimentación del circuito
A0, **A1**, **A2**: entradas de dirección del circuito integrado

El DS1621 tiene integrado el sensor de temperatura y debido a esto no necesita ningún componente externo para funcionar. Al conectarlo a nuestro microcontrolador con un bus I2C, podemos obtener la lectura de la temperatura a la que se encuentra, además de tener un termostato configurable.

La temperatura es convertida a valores digitales automáticamente en el DS1621. La lectura de los valores de temperatura se hace a través de dos registros de 8 bits (2 bytes), siendo el byte más alto o MSB la lectura del valor, y el byte bajo o LSB la parte decimal, ya sea 0 o 0.5 grados. Es decir, un valor de 00000000 en el LSB representa un x.0, y un 10000000 representará un x.5, ya que la resolución típica del DS1621 es de 0.5 grados centígrados. Las temperaturas negativas están dadas en complemento a dos, así que habrá que convertirlas después de leerlas. En la **Tabla 2** tenemos varios ejemplos de diferentes de lecturas posibles obtenidas del DS1621.

TEMPERATURA	SALIDA DIGITAL
+125 °C	01111101 00000000
+25 °C	00011001 00000000
+0.5 °C	00000000 10000000
0 °C	00000000 00000000
-0.5 °C	11111111 10000000
-25 °C	11100111 00000000
-55 °C	11001001 00000000

***Tabla 2.** Ejemplos de salidas de temperatura.*

Por ejemplo, una lectura de MSB=00001100 y LSB=10000000 representará una temperatura de 12.5 grados. Cuando se transmite el valor de la temperatura del DS1621 al PIC, primero se lee el MSB y luego el LSB. En el LSB sólo puede cambiar el bit más significativo, los 7 restantes siempre serán cero.

Control del termostato

El termostato del DS1621 puede programarse con los valores máximo y mínimo que el usuario necesite. Existen dos registros para esta tarea. Por un lado, el registro **TH** contiene el valor de la temperatura máxima, al alcanzarse esta temperatura, la salida **Tout** del DS1621 se activará e indicará que se ha alcanzado o sobrepasado. La salida Tout puede utilizarse para activar una alarma, para activar algún mecanismo o circuito (por ejemplo, un ventilador) y se puede programar para que se active en alto o bajo, es decir, para que cuando se alcance o sobrepase la temperatura TH la salida Tout se ponga en nivel alto o en bajo, según requiera el usuario. Por otro lado, el registro **TL** almacenará la temperatura mínima para el termostato. Cuando se ha rebasado la temperatura TH la salida Tout se mantendrá activa hasta que la temperatura alcance o baje de la temperatura definida en TL.

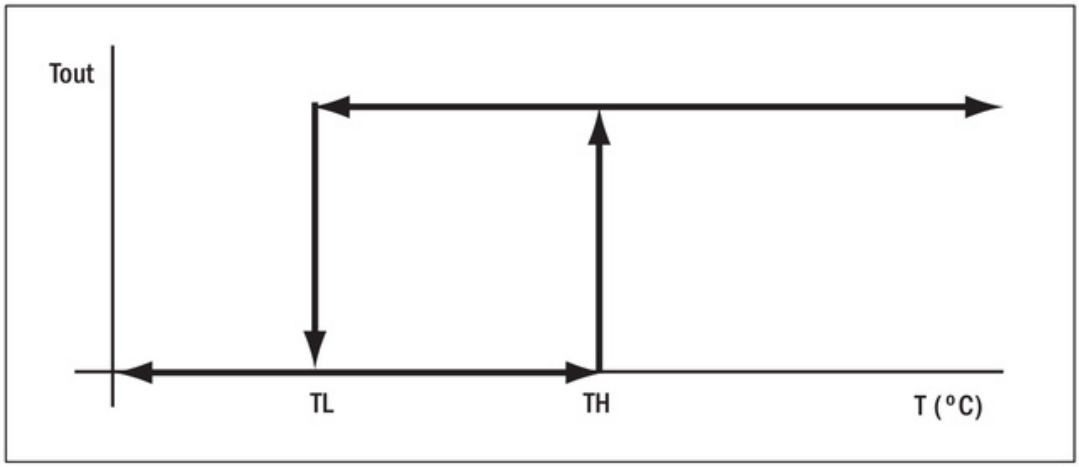


***Figura 19.** Gráfico del funcionamiento del termostato del DS1621 con salida Tout configurada en alto.*

El registro de configuración

El DS1621 contiene un registro de configuración en donde podemos obtener información de la operación del circuito y configurar los parámetros de funcionamiento.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
DONE	THF	TLF	NVB	x	x	POL	1SHOT

***Tabla 3.** El registro de configuración del DS1621 (ACh).*

DONE: bit de conversión A/D completa. Si este bit está a 0, la conversión de temperatura dentro del DS1621 está en progreso. Si está a 1, la conversión ha terminado. Podemos usar este bit como indicación de si se ha terminado una conversión para poder leerla, aunque si sabemos que la conversión dura un tiempo máximo de 750 ms, podemos esperar ese tiempo y luego leerla sin necesidad de monitorear este bit.
THF: bandera de temperatura máxima. Indica si se ha sobrepasado la temperatura máxima definida en TH. Una vez activado este bit permanecerá así, hasta que se borre por software o se desconecte la alimentación del circuito.
TLF: bandera de temperatura mínima. Indica si se ha alcanzado o descendido de la temperatura mínima establecida en TL. Una vez activado este bit permanecerá así, hasta que se borre por software o se desconecte la alimentación del circuito.
NVB: bandera de escritura en EEPROM. Algunos bits o registros del DS1621 son no volátiles, es decir, son del tipo EEPROM. Cuando se escribe en un bit o registro de este tipo se debe esperar un tiempo máximo de 10 ms para que la escritura se complete. Este bit indica con un 1 si una operación de escritura está en progreso, y con un 0 si ha terminado o no hay escritura en progreso.
x: reservado.
POL: este bit configura el modo de salida del pin Tout. Si está a 1, la salida será positiva o con nivel alto, es decir, al rebasarse la temperatura definida en TH se pondrá a 1. Si es 0, entonces al rebasarse la temperatura TH la salida en Tout se pondrá a 0.
1SHOT: configuración de modo. Si está en 1, el DS1621 estará en modo **one shot**, es decir que cuando se envíe un comando de inicio de conversión sólo se llevará a cabo una conversión y el DS1621 entrará en modo de bajo consumo al terminar. Si está en 0, al enviar un comando de inicio de conversión, el DS1621 realizará conversiones de temperatura en forma continua.

Dirección como esclavo del DS1621

Como hemos estudiado, cada dispositivo esclavo en el bus I2C debe tener una dirección para poder acceder a él, ya sea para leer o escribir. La dirección para el DS1621 está definida mediante 7 bits, los 4 bits más altos son fijos, y están dados por el fabricante (en este caso son 1001). Los tres últimos bits se definen mediante las entradas A0, A1 y A2 en los pines del circuito integrado.

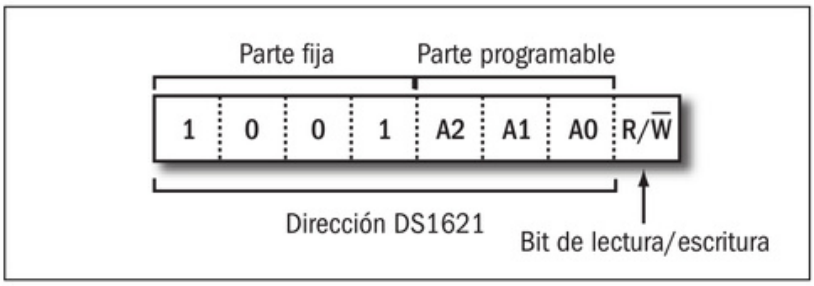


***Figura 20.** Configuración de la dirección del DS1621 en el bus I2C.*

Al colocar todas las posibles combinaciones en las terminales de dirección, podemos conectar 8 dispositivos DS1621, cada uno con una dirección diferente.

Comandos del DS1621

Para acceder a la configuración, leer la temperatura y demás funciones del DS1621, debemos hacerlo enviando comandos para cada operación. Los comandos principales usados en el DS1621 son:

AAh (lectura de temperatura): mediante este comando se lee la temperatura desde el DS1621. La lectura se obtendrá en dos bytes, como ya mencionamos antes, y se leerá la última conversión realizada por el DS1621.
A1h (acceso a TH): mediante el comando **A1h** se accede para leer o escribir en el registro TH, el cual define la temperatura máxima para el termostato. Al enviar este comando se escribirán o leerán dos bytes.
A2h (acceso a TL): se accede al registro TL para escribir o leer en él. Este registro define la temperatura mínima para el termostato, se leerán o escribirán dos bytes.
ACh (acceso al registro de configuración): mediante este comando se escribirá o leerá el registro de configuración del DS1621.
EEh (inicia conversión): mediante el comando **EEh** se iniciará la conversión de temperatura en el DS1621. En modo one shot sólo se llevará a cabo una conversión, en modo continuo se iniciará una serie de conversiones continuas. Es necesario enviar este comando para iniciar las conversiones en cualquiera de los dos modos.
22H (detiene conversión): este comando detiene la conversión de temperatura en el DS1621. Una vez enviado, se completará la conversión que se esté procesando en ese momento, y al finalizar entrará en modo de bajo consumo. Este comando sólo se usa en modo continuo para detenerlo.

Diseño de un termómetro con DS1621

Construiremos un termómetro digital con el circuito DS1621. En la **Figura 21** tenemos el diagrama con la conexión del DS1621 al PIC16F84A y en la salida Tout tenemos un led de color rojo que indica que se ha sobrepasado la temperatura del termostato.

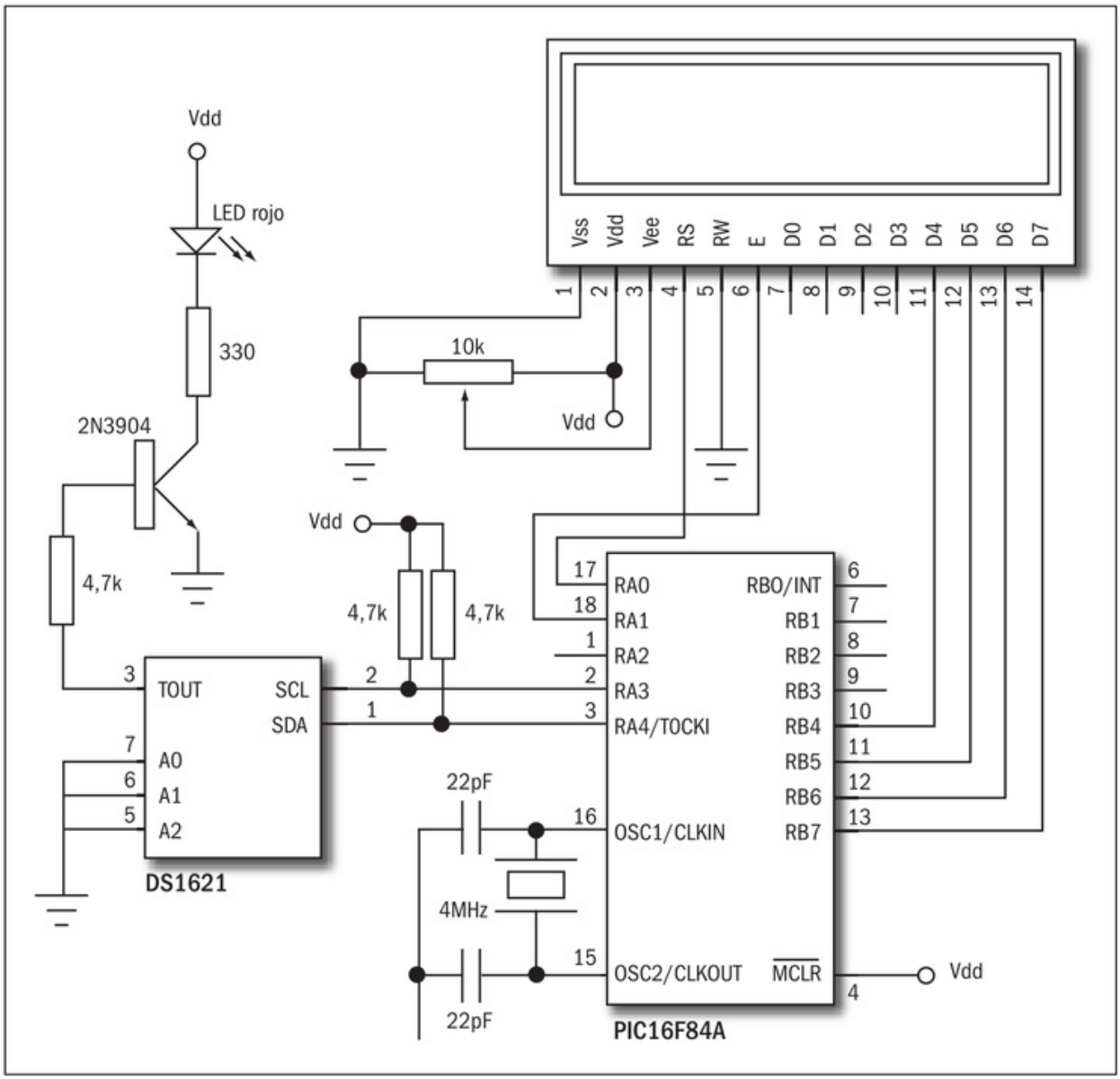


Figura 21. *Circuito completo para nuestro termómetro y termostato digital.*

Realmente, el circuito es muy sencillo, teniendo en cuenta que la conexión entre el PIC y el DS1621 se realiza con tan solo 2 líneas. Con este circuito mediremos la temperatura y la mostraremos en el display. Es importante considerar que, como el sensor de temperatura está dentro del DS1621, la temperatura medida será la temperatura a la cual esté expuesto el circuito integrado.

Inicialización del DS1621

Para inicializar el funcionamiento del DS1621, debemos enviar la palabra de configuración y escribir los bits POL y 1SHOT para configurar como se necesite. En nuestro caso configuraremos **POL = 1** para obtener una salida en alto en el pin Tout cuando se supere la temperatura TH. El bit 1SHOT lo configuraremos en 1 para obtener funcionamiento en modo one shot. Como hemos conectado las líneas A0, A1 y A2 a tierra, la dirección del DS1621 en el bus será entonces 1001000x.

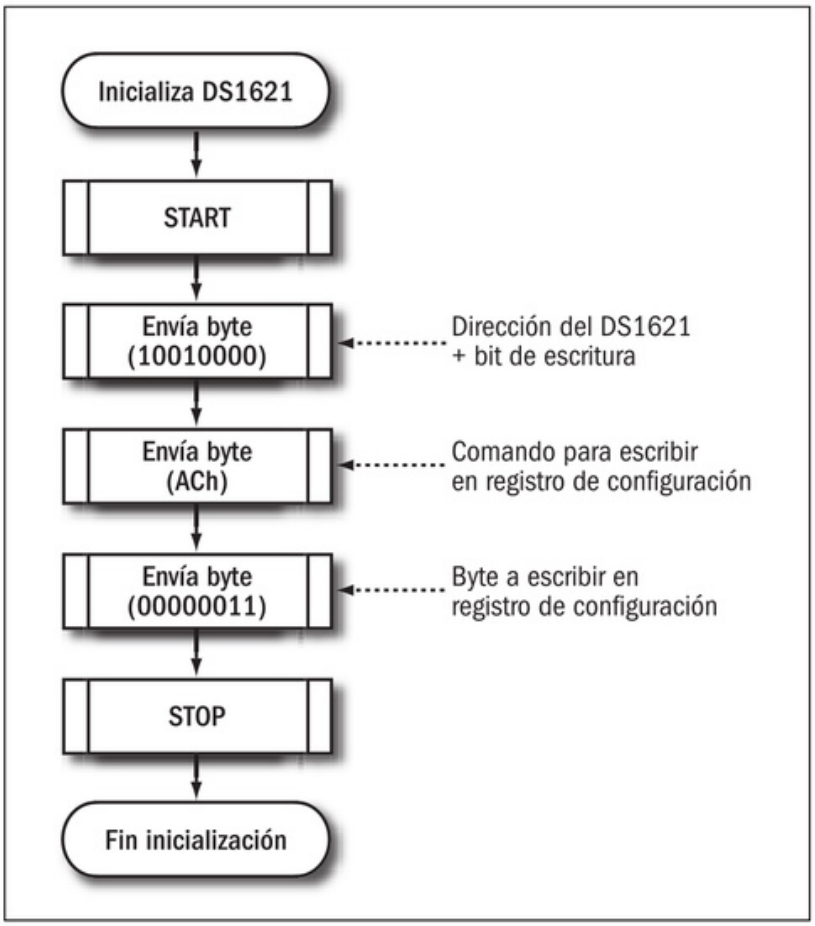


***Figura 22.** Proceso para escribir en el registro de configuración del DS1621.*

La escritura en el registro de configuración precisa el envío de tres bytes: primero se envía el byte con la dirección del DS1621 y el bit R/W a 0 para indicar que se va a escribir. Luego se envía el comando **ACh** para indicarle al DS1621 que se va a escribir en el registro de configuración. Y por último se envía el byte a escribir, el cual configura los dos bits más bajos en 1. Como algunos bits de este registro son no volátiles, hay que esperar al menos 10 ms para que se escriban antes de intentar enviar más datos al DS1621.

Si vamos a usar al DS1621 como termostato, debemos escribir los valores deseados en los registros TH y TL, para que queden configurados los límites de temperatura.

III OTROS MICROCONTROLADORES PIC

Aunque el PIC16F84A puede ser sumamente útil en multitud de aplicaciones, por supuesto que no es el único microcontrolador de la familia de los PIC. Existe una gran variedad con diferentes características y funciones. Algunos PIC de gama baja y media muy populares en la actualidad son: PIC12F629, PIC12F675, PIC16F628A, PIC16F88, PIC16F887.

Estos registros son no volátiles, por lo que sólo necesitamos escribirlos una sola vez y su valor se retendrá, aun si quitamos la alimentación del circuito. En el programa que diseñaremos habrá una rutina para leer el valor de los registros TH y TL, y compararlos con los valores que necesitamos. Si los valores son correctos no se escribe en ellos, de esta forma sólo se escribirán una vez.

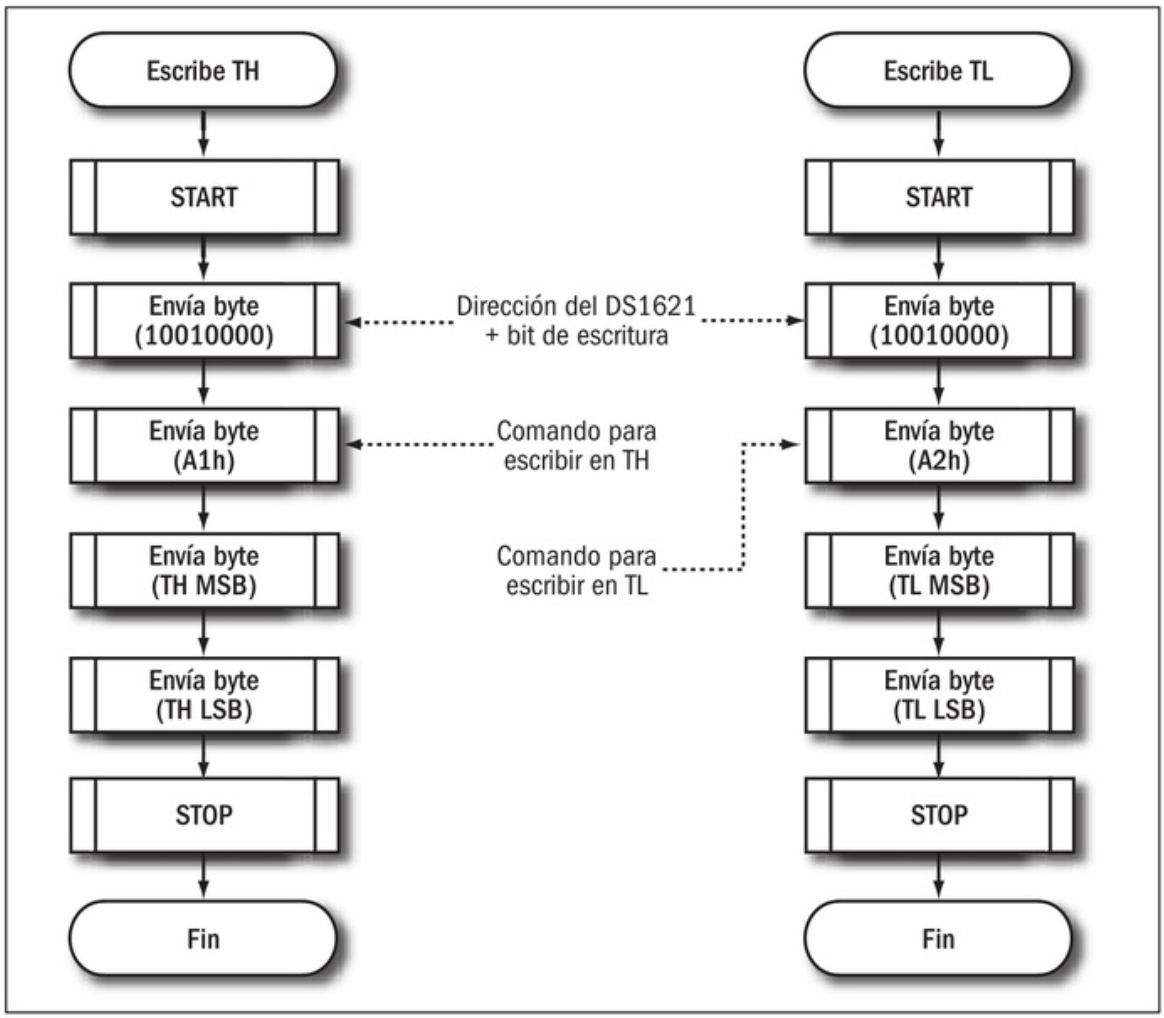


***Figura 23.** Procesos de escritura en TH y TL para la configuración de la histéresis del termostato.*

La escritura en TH y TL también dura un tiempo máximo de 10 ms, por lo que debemos esperar a que finalice la escritura o verificar la bandera NVB hasta que se ponga a 0. En nuestro caso usaremos un retardo para esperar 10 ms y asegurar que la escritura ha terminado. En este ejemplo tomaremos un valor de 30 grados para TH y 25 grados para TL.

Una vez escritos los registros TH, TL y la configuración, estamos listos para comenzar a leer la temperatura. Como configuramos en modo one shot, debemos iniciar la conversión cada vez que necesitemos obtener una nueva lectura. En este punto es donde inicia la lectura de temperatura para luego mostrarla en el display.

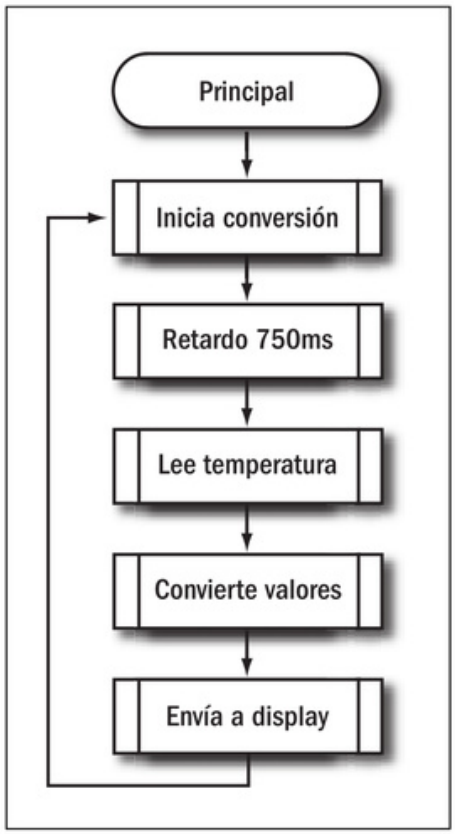


Figura 24. *Rutina principal del programa del termómetro.*

Veamos las subrutinas más importantes llamadas desde la rutina principal. Primero debemos iniciar una conversión, ya que estamos en modo one shot.

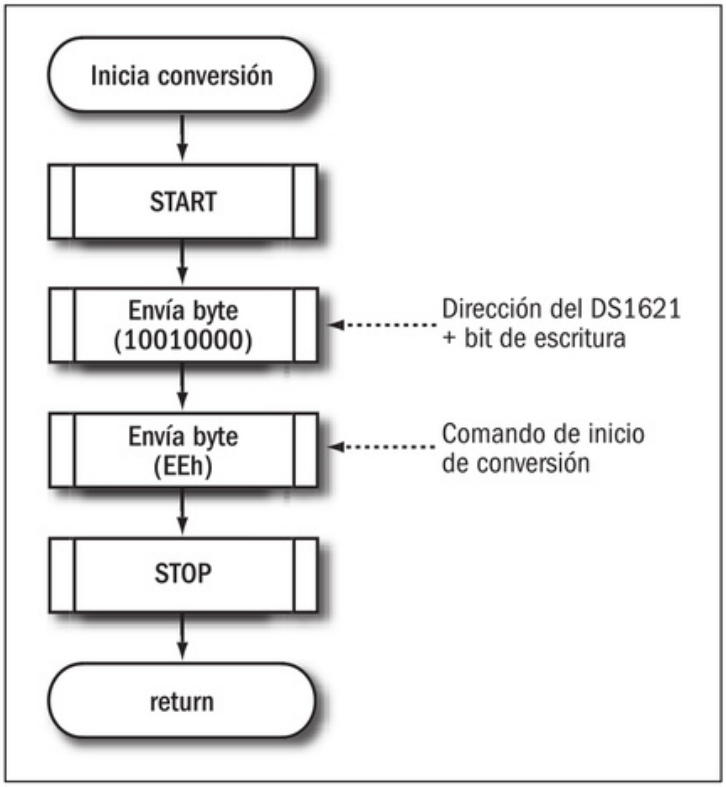


Figura 25. *Subrutina para inicio de conversión de temperatura.*

Luego de iniciar una nueva conversión de temperatura debemos esperar 750 ms a que se complete. Transcurrido ese tiempo, estamos listos para leer la temperatura medida.

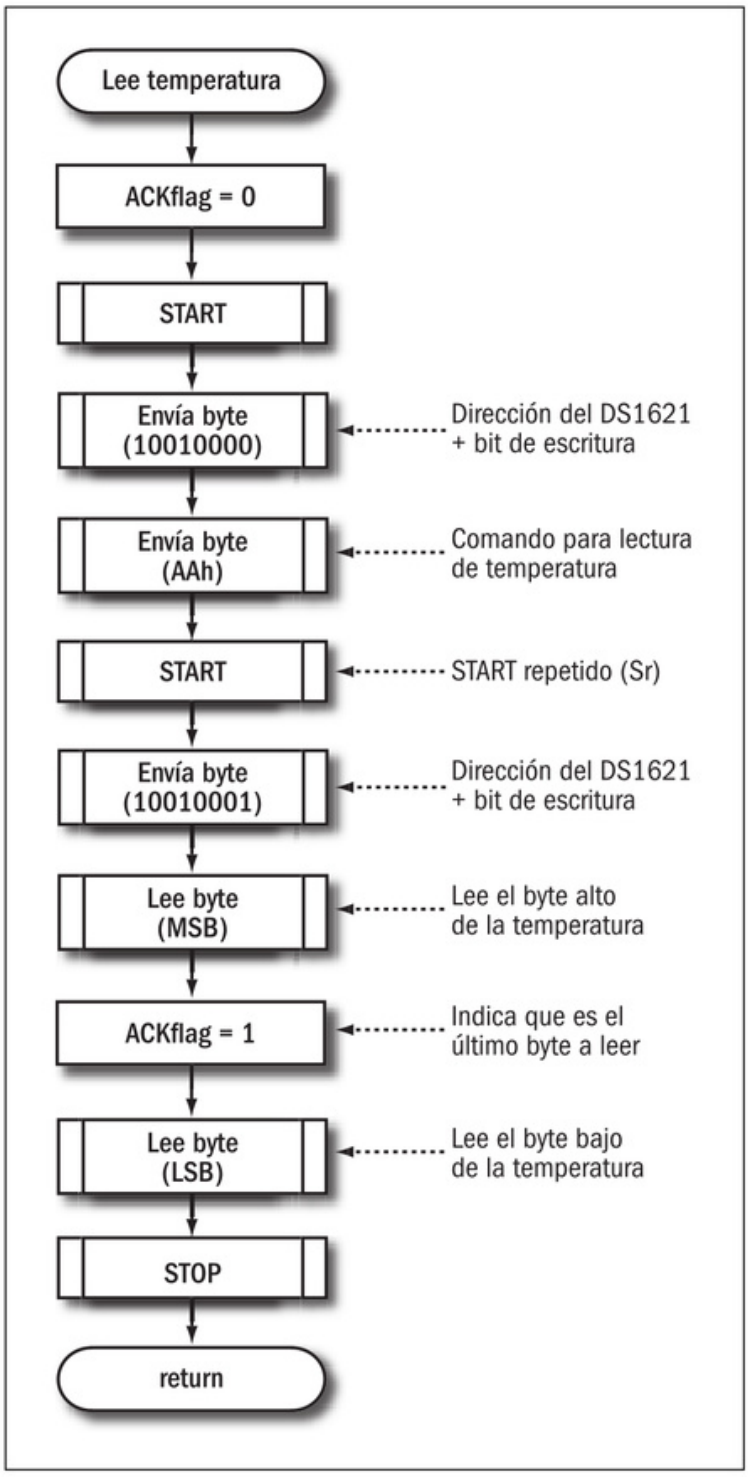


***Figura 26.** Subrutina para lectura de los dos bytes con el valor de la temperatura.*

Observemos cómo en la subrutina de lectura la temperatura se inicia enviando el byte 10010000, para después enviar el comando **AAh**, que es el de lectura de temperatura. Una vez enviado el comando **AAh**, se envía una condición de START

nuevamente, es decir, un START repetido (Sr) para cambiar la dirección de envío al enviar el byte 10010001, con el bit R/W a 1. Luego de esto, ya podemos leer los bytes que enviará el DS1621 con la lectura de la temperatura. Primero se recibe el MSB e inmediatamente después el LSB. Es importante observar cómo, antes de leer el LSB, se coloca el ACKflag en 1 para indicar que éste es el último byte a leer, y de esa forma hacer que la subrutina de lectura de la librería **I2C.INC** envíe un bit NACK al final. Una vez leídos los dos bytes correspondientes a la temperatura, se envía la condición de STOP. Ahora sólo resta convertir los valores leídos a un formato adecuado y enviar la lectura al display para poder observarla.

Podemos descargar los archivos **Termómetro DS1621.asm** y **Termómetro DS1621.hex** de **www.redusers.com** para analizar el código fuente en detalle y comprobar el funcionamiento en el circuito. De esta forma, con nuestra librería **I2C.INC** hemos logrado establecer comunicación con un dispositivo I2C para lograr este proyecto del termómetro digital. Con esta base podremos hacer todas las modificaciones y mejoras en el programa y el circuito para lograr poderosas aplicaciones que incluyan medición de temperatura y termostatos para control de temperatura.

OTROS DISPOSITIVOS I2C

Los dispositivos o circuitos integrados que utilizan el protocolo I2C que hemos estudiado son sólo una pequeña muestra de los muchos que existen en el mercado. Existe diversidad de funciones que podemos agregar con los dispositivos que se comunican por medio del bus I2C. Como estudiarlos en detalle requeriría casi un capítulo completo, haremos una breve descripción de algunos para saber qué utilidad podemos obtener de ellos y poder diseñar más proyectos con dispositivos I2C:

DS1624: sensor de temperatura y memoria EEPROM. Este circuito integrado es un sensor de temperatura parecido al DS1621 que estudiamos, pero no tiene termostato, en cambio dispone de una memoria EEPROM de 256 bytes. La operación del termómetro es muy parecida al DS1621.
LM75: sensor de temperatura y termostato. Este circuito integrado es un sensor de temperatura y termostato programable. Su funcionamiento es similar al DS1621 que vimos, aunque el manejo del los registros internos es algo diferente.
PCF8575: expansor de puertos de 16 líneas. Similar al PCF8574 que estudiamos, pero este circuito integrado provee un puerto de 16 líneas de entrada/salida.
24Cxx, **24LCxxx**: memorias EEPROM. Estas series de memorias pueden servirnos para almacenar información en ellas. Al ser del tipo EEPROM son no volátiles, por lo que pueden ser de gran utilidad en muchos proyectos que vayamos

a diseñar. Las x indican la capacidad de la memoria que puede ir de 128 bits (16 bytes x 8) hasta 1024 kbits (1 Mbyte x 8).
DS1307: reloj en tiempo real. Este tipo de circuitos integrados está especialmente diseñado para proyectos donde se necesite medición de tiempo, como relojes, calendarios, o cualquier circuito que requiera de ellos. Puede medir segundos, horas, días, meses y años. Contiene un calendario interno programado hasta el año 2100.
PCF8583: otro reloj en tiempo real.
SAA1064: controlador de displays de 7 segmentos. Este circuito integrado sirve para manejar 2 ó 4 displays de 7 segmentos de ánodo común. Es muy útil en los proyectos donde los utilicemos, ya que se encargará de mantener los displays encendidos mientras el microcontrolador atiende otras tareas. Los datos a mostrar en los displays se envían a través del bus I2C.
PCF8591: convertidor analógico a digital y digital a analógico. En este circuito integrado encontramos un convertidor analógico a digital y un convertidor digital a analógico de 8 bits. Es muy útil si tenemos en cuenta que el PIC16F84A no tiene convertidores, por lo que si necesitamos de ellos podemos recurrir a algún convertidor en bus I2C como éste.

Por supuesto, ésta es sólo una muestra de los dispositivos I2C que podemos encontrar disponibles para nuestros proyectos. Existen muchos más con las mismas o diferentes funciones que los descritos, sólo es cuestión de animarnos a investigar sobre el tema. De cualquier forma, debemos tenerlos en cuenta ya que los dispositivos I2C pueden resultarnos de gran utilidad en multitud de proyectos.

Hasta aquí hemos llegado en la descripción, programación y aplicaciones que podemos darle al PIC16F84A. Como hemos visto, podemos lograr aplicaciones muy variadas y sumamente útiles. Hemos estudiado todas las funciones de este microcontrolador por lo que, con lo aprendido, estamos en condiciones de diseñar nuestros propios proyectos para las funciones que necesitemos. El límite será nuestra imaginación.

... RESUMEN

En este capítulo final hemos estudiado la comunicación entre el PIC16F84A y los dispositivos que usan el protocolo de bus I2C para comunicación serial. La gran variedad de dispositivos I2C en el mercado nos permite lograr aplicaciones de lo más variadas, y complementan al PIC16F84A para lograr diseños muy versátiles y con una gran facilidad. Con esto finalizamos el estudio del PIC16F84A.

ACTIVIDADES

TEST DE AUTOEVALUACIÓN

1 ¿Qué es el bus I2C?

2 ¿Qué función tienen las líneas SDA y SCL en el bus I2C?

3 ¿Para qué sirven los resistores de pull-up (Rp) en el bus?

4 ¿Cuáles son los valores recomendados para los resistores de pull-up?

5 ¿Quién debe generar la señal de reloj para el bus?

6 ¿Qué tipo de condición debe enviar el maestro al bus para iniciar la transmisión de datos?

7 ¿Cuál es la condición para detener la transmisión?

8 ¿Cómo se llama el bit que se debe enviar en el noveno pulso de reloj?

9 ¿Cuál es la función del circuito integrado PCF8574?

10 ¿Cuál es la función del circuito integrado DS1621?

PRÁCTICAS

1 Escriba en MPLAB los dos programas de la sección Expansor de puertos I2C PCF8574, ensámblelos, grábelos en el PIC16F84A y compruebe su funcionamiento en el circuito correspondiente.

2 Diseñe una librería que permita manejar el display LCD conectado a un expansor de puertos PCF8574.

3 Diseñe una librería que permita manejar un teclado de 4x4 conectado a un expansor PCF8574.

4 En el programa del termómetro con DS1621 se leen los valores de TH y TL para comprobar si son los deseados, si no es así, se escriben. Esto no se hace con el registro de configuración. Por lo tanto, agregue las rutinas necesarias al programa para que la comparación se haga también con los dos bits más bajos de este registro y si ya están a 1 no se escribirán.

5 Diseñe un termómetro y termostato para un acuario que le permita mantener la temperatura del agua en los rangos recomendados en forma automática, encendiendo o apagando el calefactor, según se necesite.

Servicios al lector

Para terminar, en este apartado encontraremos un listado de los términos más importantes de esta obra para poder encontrar de forma más rápida lo que queremos aprender.

ÍNDICE TEMÁTICO

Claves para comprar un libro de computación.

1 Sobre el autor y la editorial

Revise que haya un cuadro "sobre el autor", en el que se informe sobre su experiencia en el tema.
En cuanto a la editorial, es conveniente que sea especializada en computación.

2 Preste atención al diseño

Compruebe que el libro tenga guías visuales, explicaciones paso a paso, recuadros con información adicional y gran cantidad de pantallas. Su lectura será más ágil y atractiva que la de un libro de puro texto.

3 Compare precios

Suele haber grandes diferencias de precio entre libros del mismo tema; si no tiene el valor en la tapa, pregunte y compare.

4 ¿Tiene valores agregados?

Desde un sitio exclusivo en la Red hasta un CD-ROM, desde un Servicio de Atención al Lector hasta la posibilidad de leer el sumario en la Web para evaluar con tranquilidad la compra, o la presencia de adecuados índices temáticos, todo suma al valor de un buen libro.

5 Verifique el idioma

No sólo el del texto; también revise que las pantallas incluidas en el libro estén en el mismo idioma del programa que usted utiliza.

6 Revise la fecha de publicación

Está en letra pequeña en las primeras páginas; si es un libro traducido, la que vale es la fecha de la edición original.

usershop.redusers.com

Visite nuestro sitio web

Utilice nuestro sitio **usershop.redusers.com**:

- Vea información más detallada sobre cada libro de este catálogo.
- Obtenga un capítulo gratuito para evaluar la posible compra de un ejemplar.
- Conozca qué opinaron otros lectores.
- Compre los libros sin moverse de su casa y con importantes descuentos.
- Publique su comentario sobre el libro que leyó.
- Manténgase informado acerca de las últimas novedades y los próximos lanzamientos.

También puede conseguir nuestros libros en kioscos o puestos de periódicos, librerías, cadenas comerciales, supermercados y casas de computación.

SuperBlogger

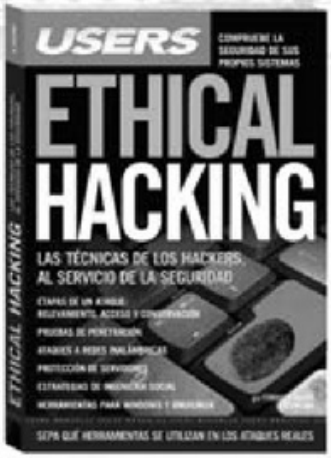
Esta obra es una guía para sumarse a la revolución de los contenidos digitales. En sus páginas, aprenderemos a crear un blog, y profundizamos en su diseño, administración, promoción y en las diversas maneras de obtener dinero gracias a Internet.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1347-96-4

UML

Este libro es la guía adecuada para iniciarse en el mundo del modelado. Conoceremos todos los constructores y elementos necesarios para comprender la construcción de modelos y razonarlos de manera que reflejen los comportamientos de los sistemas.

→ COLECCIÓN: DESARROLLADORES
→ 320 páginas / ISBN 978-987-1347-95-7



Ethical Hacking

Esta obra expone una visión global de las técnicas que los hackers maliciosos utilizan en la actualidad para conseguir sus objetivos. Es una guía fundamental para conseguir sistemas seguros y dominar las herramientas que permiten lograrlo.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-93-3

Unix

Esta obra contiene un material imperdible, que nos permitirá dominar el sistema operativo más sólido, estable, confiable y seguro de la actualidad. En sus páginas encontraremos las claves para convertirnos en expertos administradores de FreeBSD.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-94-0

200 Respuestas Excel

Una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos conocer para dominar la versión 2007 de Microsoft Excel. Definiciones, consejos, claves y secretos, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-1347-91-9

Hardware Extremo

En esta obra aprenderemos a llevar nuestra PC al límite, aplicar técnicas de modding, solucionar fallas y problemas avanzados, fabricar dispositivos inalámbricos caseros de alto alcance, y también a sacarle el máximo provecho a nuestra notebook.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-90-2

usershop.redusers.com

Marketing en Internet

Este libro brinda las herramientas de análisis y los conocimientos necesarios para lograr un sitio con presencia sólida y alta tasa de efectividad. Una obra imprescindible para entender la manera en que los negocios se llevan a cabo en la actualidad.

→ COLECCIÓN: PROFESSIONAL TOOLS
→ 288 páginas / ISBN 978-987-1347-82-7

200 Respuestas: Hardware

Esta obra es una guía básica que responde, en forma visual y práctica, a todas las preguntas que necesitamos hacernos para dominar el hardware de la PC. Definiciones, consejos, claves y secretos de los profesionales, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-1347-83-4

Curso de programación PHP

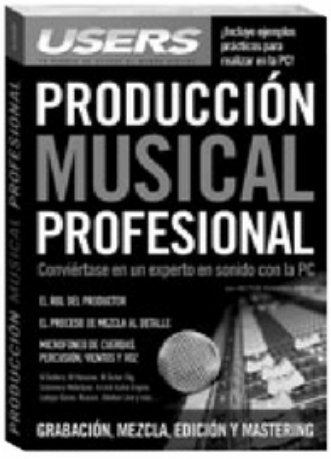
Este libro es un completo curso de programación con PHP desde cero. Ideal tanto para quienes desean migrar a este potente lenguaje, como para los que quieran aprender a programar, incluso, sin tener conocimientos previos.

→ COLECCIÓN: DESARROLLADORES
→ 368 páginas / ISBN 978-987-1347-81-0

Curso de programación C#

Este libro es un completo curso de programación con C# desde cero. Ideal tanto para quienes desean migrar a este potente lenguaje, como para quienes quieran aprender a programar, incluso, sin tener conocimientos previos.

→ COLECCIÓN: DESARROLLADORES
→ 400 páginas / ISBN 978-987-1347-76-6



Producción musical profesional

Esta obra es un manual preciso y detallado que permite alcanzar la perfección a quienes quieren lograr el sonido ideal para sus composiciones. Está enfocado en el rol del productor, lugar desde donde construye los cimientos para producciones profesionales.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-75-9

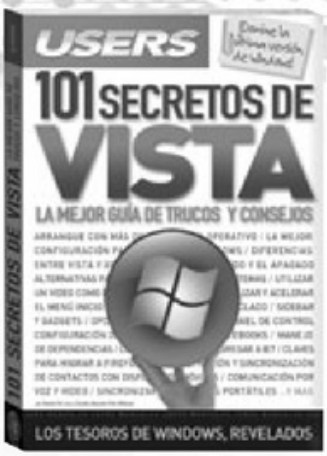
Desarrollador .NET

Ésta es una obra teórica y práctica para aprender a programar. Basado en el curso Desarrollador cinco estrellas de Microsoft, este material brinda las habilidades necesarias para iniciar el camino que nos lleve a convertirnos en desarrolladores de la plataforma .NET.

→ COLECCIÓN: DESARROLLADORES
→ 400 páginas / ISBN 978-987-1347-74-2

¡Léalo antes Gratis!

En nuestro sitio, obtenga GRATIS un capítulo del libro de su elección antes de comprarlo.



101 Secretos de Vista

Una obra absolutamente increíble, con los mejores 101 secretos para dominar la última versión de Windows. En sus páginas encontraremos un material sin desperdicios, ideal para quienes quieren llevar el rendimiento de su PC al límite.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1347-80-3

Excel 2007: Guía de funciones

Este libro es una guía de referencia y consulta permanente que brinda acceso instantáneo a las funciones de Excel 2007, y sin duda se convertirá en un material indispensable para quienes quieran potenciar sus planillas de cálculo.

→ COLECCIÓN: MANUALES USERS
→ 368 páginas / ISBN 978-987-1347-79-7

El gran libro de la vida digital

Ésta es una obra visual y práctica, en la que aprenderemos a usar los principales dispositivos tecnológicos de la actualidad. Un libro fundamental para estar preparados y ser partícipes del cambio tecnológico que estamos viviendo.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-78-0

Electrónica digital

Una obra ideal para quienes desean conocer el mundo de la electrónica digital, y la simulación de circuitos y componentes. Un repaso de los principios de electricidad y electrónica, explicando en detalle las herramientas e instrumentos más importantes.

→ COLECCIÓN: MANUALES USERS
→ 352 páginas / ISBN 978-987-1347-73-5

200 Respuestas Fotografía digital

Esta obra es una guía básica que responde en forma visual y práctica a todas las preguntas que se nos plantean sobre la fotografía digital. Definiciones, consejos, claves y secretos de los profesionales, explicados de manera clara, sencilla y didáctica.

→ COLECCIÓN: 200 RESPUESTAS
→ 320 páginas / ISBN 978-987-1347-71-1

Desarrollo Web profesional

Este libro resulta una herramienta imprescindible, de consulta permanente, que brinda las técnicas, claves y consejos que permitirán desarrollar sitios profesionales y perfeccionar nuestra labor en las principales tecnologías: XHTML, CSS, JavaScript y AJAX.

→ COLECCIÓN: DESARROLLADORES
→ 400 páginas / ISBN 978-987-1347-70-4

usershop.redusers.com

Armado Profesional de PC

En las páginas de este libro, encontraremos cada procedimiento para aprender a armar computadoras de manera profesional. Todo detallado paso a paso, acompañado de consejos, técnicas y secretos de los máximos expertos en el área.

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-69-8

Office 2007

Este manual ofrece un recorrido visual y práctico por cada aplicación de Office 2007, mediante explicaciones teóricas, guías visuales y procedimientos paso a paso. Con él aprenderemos a obtener el máximo provecho de todos sus programas.

→ COLECCIÓN: MANUALES USERS
→ 400 páginas / ISBN 978-987-1347-68-1

Ruby

Este manual presenta Ruby, uno de los lenguajes de programación más flexibles y poderosos de la actualidad para el desarrollo de aplicaciones de escritorio o web. Por su sencillez, es ideal para aprenderlo rápidamente, aun sin tener grandes conocimientos de programación.

→ COLECCIÓN: DESARROLLADORES
→ 432 páginas / ISBN 978-987-1347-67-4

PHP Master

Este libro acerca al trabajo diario del desarrollador los avances más importantes incorporados en las últimas versiones de PHP. En sus páginas se exponen las técnicas actuales que permiten potenciar el desarrollo de sitios Web.

→ COLECCIÓN: DESARROLLADORES
→ 400 páginas / ISBN 978-987-1347-61-2

Reparación de componentes

En las páginas de este libro se detalla en profundidad el desarme de cada pieza de hardware, la metodología para realizar el diagnóstico y efectuar la reparación. Un material imperdible que incluye técnicas para convertirse en un profesional.

→ COLECCIÓN: MANUALES USERS
→ 240 páginas / ISBN 978-987-1347-58-2

Fotografía digital

El objetivo de este libro es brindar las herramientas y los conocimientos necesarios para comprender a fondo la fotografía digital y aprender a realizar tomas con la calidad que nuestros recuerdos se merecen. ¡Con guías paso a paso y a todo color!

→ COLECCIÓN: MANUALES USERS
→ 320 páginas / ISBN 978-987-1347-60-5

LOS MEJORES PROYECTOS REALES PARA LA PC

Una obra ideal para quienes desean conocer el mundo de la electrónica digital, y la simulación de circuitos y componentes. Un repaso por los principios de electricidad y electrónica, explicando en detalle las herramientas e instrumentos más importantes.

MANUALES USERS | 352 páginas
ISBN 978-987-1347-73-5

usershop.redusers.com

Adquiéralo con todos los medios de pago*

• Capítulo GRATIS • Avant Premiére • Promoción • Ofertas

* Sólo válido para la República Argentina

USERS

CONTENIDO

NIVEL DE USUARIO

PRINCIPIANTE | INTERMEDIO | AVANZADO | EXPERTO

ELECTRÓNICA & MICROCONTROLADORES PIC

Una obra ideal para quienes desean aprovechar al máximo las aplicaciones prácticas de los microcontroladores PIC y entender su funcionamiento.
Aprenderemos a diseñar y escribir programas reales, simularlos en la PC por medio de MPLAB IDE y ¡escribir los PICs con un grabador creado por nosotros mismos! Cada tema es tratado desde cero, dando todas las herramientas necesarias para el aprendizaje y la práctica de Assembler, de modo que aun quienes nunca hayan utilizado ningún lenguaje de programación puedan escribir programas eficientes y poderosos. Un material con procedimientos paso a paso y guías visuales, para crear proyectos sin límites, como una cerradura electrónica, un reloj digital o la escritura de mensajes en un display LCD.
En definitiva, un libro que expone las bases necesarias para ingresar en este fascinante mundo, sin sobresaltos.

redusers.com

En este sitio encontrará una gran variedad de recursos y software relacionado, que le servirán como complemento al contenido del libro. Además, tendrá la posibilidad de estar en contacto con los editores, y de participar del foro de lectores, en donde podrá intercambiar opiniones y experiencias.

Para obtener más información sobre el libro comuníquese con nuestro

Servicio de Atención al Lector

usershop@redusers.com

ARGENTINA (11) 4110 8700
CHILE (2) 335 7477
ESPAÑA (93) 635 4120

PIC MICROCONTROLLERS

This book is the ideal introduction for those who seek to enter the world of microcontrollers.
Projects aimed to learn about the theory and the practical aspects of this devices. A material full of visual guides and step by step guides to program, write and erase chips for usage in real projects.

ISBN 978-987-663-002-3

SERS MANUALES USERS MANUALES USERS MA

INCLUYE PROYECTOS REALES