

Getting started  
with PIC  
microcontrollers

Fred Stevens

# Getting started with PIC microcontrollers

by  
Fred Stevens

Email: [fred.stevens@ieee.org](mailto:fred.stevens@ieee.org)

URL: <http://www.the-electronics-project.com>

Copyright © (1997) A. L. Stevens

All rights reserved. No part of this book may be reproduced, stored or transmitted in any form or by any means whatsoever without prior written permission from the copyright holder.

The Microchip name, logo, PIC, PICSTART, PICSTART Plus and MPLAB are registered trademarks of Microchip Technology Incorporated, USA. Windows and Windows 95 are registered trademarks of Microsoft Corporation, USA.

While every effort has been made to ensure that the information contained in this book is correct, neither the author nor the publisher shall be held liable for any damage, injury or loss as a result of using this information.

# Acknowledgements

Thanks to Willem Hijbeek and Bernd Pfitzinger of Tempe Technologies (Pty) Ltd. It was Willem who first suggested writing this book and Bernd who provided amazing technical support by always bringing the latest data books, software and equipment to my home.

Glyn Craig of Techlyn (Pty) Ltd. supplied sufficiently large amounts of encouragement for me to actually get started.

To my wonderful wife Ronel, I am indebted for everything in my universe.

# Contents

- 1 Fundamental Concepts** **1**
  - 1.1 Introduction . . . . . 1
    - 1.1.1 What you will need . . . . . 2
    - 1.1.2 What makes a microcontroller useful? . . . . . 2
  - 1.2 Simplified operation of a microcontroller . . . . . 3
    - 1.2.1 Program memory . . . . . 4
    - 1.2.2 Types of ROM . . . . . 4
    - 1.2.3 Data memory . . . . . 5
  - 1.3 Programming a microcontroller . . . . . 5
    - 1.3.1 Execution of an instruction . . . . . 5
    - 1.3.2 The hexadecimal numbering system . . . . . 6
  - 1.4 Microcontroller simulators . . . . . 6
  
- 2 PIC microcontroller details** **8**
  - 2.1 The PIC16F8X family . . . . . 8
  - 2.2 Features of the PIC16F84 . . . . . 8
    - 2.2.1 The PIC16F84 architecture . . . . . 9
    - 2.2.2 Program memory . . . . . 10
    - 2.2.3 Data memory . . . . . 11
  - 2.3 Ports . . . . . 11
    - 2.3.1 Electrical characteristics of the ports . . . . . 12
    - 2.3.2 Clock source options . . . . . 13
    - 2.3.3 The timer/counter module . . . . . 13
    - 2.3.4 Low power operation and the `sleep` instruction . . . . . 14
    - 2.3.5 The watchdog timer . . . . . 14

2.4	Special function registers . . . . .	14
2.4.1	The port and port control registers . . . . .	14
2.4.2	The STATUS register . . . . .	15
2.4.3	The OPTION register . . . . .	15
2.4.4	The INTCN register . . . . .	15
<b>3</b>	<b>PICs in Practice</b>	<b>16</b>
3.1	The Power supply . . . . .	16
3.1.1	Circuit layout and construction . . . . .	17
3.1.2	Powering up CMOS . . . . .	17
3.1.3	Power on reset . . . . .	18
3.2	The structure of a PIC program . . . . .	19
3.2.1	The header file . . . . .	20
3.2.2	The configuration word . . . . .	22
3.3	Project 1 . . . . .	22
3.3.1	Operation of program <code>Port.asm</code> . . . . .	23
3.3.2	Assembling <code>Port.asm</code> . . . . .	24
3.3.3	Simulating <code>Port.asm</code> . . . . .	24
3.3.4	Loading the hex file . . . . .	25
3.3.5	Testing the circuit . . . . .	25
3.4	Project 2 . . . . .	25
3.4.1	The delay . . . . .	26
3.4.2	Calculating the delay . . . . .	27
3.4.3	Program improvements . . . . .	29
3.4.4	Using an input . . . . .	32
3.4.5	Simulating <code>Flash4.asm</code> . . . . .	35
3.5	More programming examples . . . . .	36
3.5.1	Comparing values . . . . .	36
3.5.2	Choosing between alternatives . . . . .	37
3.5.3	Indirect addressing . . . . .	38
3.5.4	Changing bit values in registers . . . . .	39
3.5.5	Bit rotations . . . . .	39
3.6	Project 3 . . . . .	41

3.6.1	Interrupts . . . . .	41
3.6.2	The PIC16F84 interrupt system . . . . .	41
3.6.3	Simulating program <code>Intp.asm</code> . . . . .	44
3.6.4	Context saving during interrupts . . . . .	45
3.7	Using the timer . . . . .	46
3.7.1	Setting up the timer/counter module . . . . .	47
3.7.2	The timer/counter interrupt . . . . .	48
3.7.3	Counting external pulses . . . . .	49
3.7.4	Common errors and pitfalls . . . . .	50
3.8	Some PIC project ideas . . . . .	52

# Chapter 1

## Fundamental Concepts

### 1.1 Introduction

Never before has it been so quick and easy to create microprocessor-based circuits. With the advent of the new PIC range of 8-bit microcontrollers and the high performance, low cost software available, a project can take literally a morning to progress from initial conception to final prototype.

Developing a PIC-based project takes only six easy steps:

1. Type in the program
2. Assemble the program into a binary file
3. Simulate the program and debug it
4. Load the binary program into the PIC's memory
5. Wire up the circuit
6. Switch on and test.

It's as easy as that!

In the early 1980s, the term PIC stood for Peripheral Interface Controller. These devices were originally designed for use in applications with 16-bit microprocessors and computer peripherals, remote control transmitters, domestic products and automotive systems.

While the PIC data sheets are both comprehensive and informative, it is quite difficult and time consuming for the beginner to wade through the documentation to find out where and how to start. The objective of this book is to get the reader up and running in hours rather than days. After reading this book and building the easy projects described, progressing to more advanced systems with other PIC microcontrollers is quite straightforward.

### 1.1.1 What you will need

In the late 1970s and early 1980s, the cost of equipment for developing microcontroller-based systems was beyond the reach of most small companies. Now, however, there are many high school students already developing exciting PIC projects.

To get started you will need the following equipment and documentation:

- One or more PIC16C84s or PIC16F84s (the '83 devices will also suffice)
- A Personal Computer running Windows 3.1 or Windows '95.
- A copy of the latest MPLAB software available free of charge from Microchip Technology Inc.
- PIC microcontroller data sheets and application notes available free of charge from Microchip Technology Inc.
- A PICStart-Plus programmer. Programmers available from third party manufacturers which support the PIC devices will also be suitable.
- A circuit breadboard or similar means of circuit construction.
- A 5V DC power supply.
- Some light emitting diodes (LEDs), resistors and some 32kHz crystals.
- Test equipment such as a digital multimeter and an oscilloscope (not essential).

### 1.1.2 What makes a microcontroller useful?

A microcontroller (or microprocessor) can be viewed as a set of digital logic circuits integrated on a single silicon 'chip' whose connections and behaviour can be specified and later altered when required, by the program in its memory. The great advantage of this, is that in order to change the circuit's *structure* and *operation*, all that is needed is a change in the program - very little, if any, circuit hardware modifications are necessary. An alternative view is that a microcontroller is a *state machine* whose logic states are defined by its program.

A *microprocessor* is the Central Processing Unit (CPU) of a computer and a microcontroller can be regarded as a microprocessor designed specifically for use in applications where machines such as automobile engines or washing machines are to be controlled. Often the distinction between microprocessors and microcontrollers is quite blurred, as there is considerable overlap these days in the classification of different types of computing devices.

A typical microprocessor is a device used in workstation computers, whereas a microcontroller is usually less powerful and has special features such as PWM (pulse width modulation) and timer devices integrated on the IC specifically for use in the applications mentioned above.

## 1.2 Simplified operation of a microcontroller

Humans perform arithmetic using a decimal or base ten numbering system. Computers use a base two system with the digits 0 and 1 and, because there are only two possibilities, they are termed *binary digits* or *bits* for short.

There are many ways of representing a binary set of states, for example

- a mechanical or electronic switch with two states — on or off
- opposite directions of current or fluid flow
- two different pressures
- a positive and a zero voltage.

In digital electronic circuits, the last choice is the most natural. Circuits implemented using TTL (Transistor-Transistor Logic) technology use 5V to represent the binary value 1 and a zero voltage to represent 0. The circuits discussed in this book, although not TTL circuits, will use the approximately the same representation.

To provide some robustness to the representation, voltages above 3.5V will be taken to represent a logical 1 and those below 1.5V to represent a logical 0.

Readers new to the subject of digital electronics are urged to consult one of the many excellent books available on the subject, such as reference [1].

A simple microcontroller consists of the following modules:

- An Arithmetic Logic Unit (ALU)
- One or more working registers (called *accumulators* in the past) for temporary storage during computations. A *register* is a small block of memory, often the size of a byte, where data is stored.
- Program memory (ROM) and data memory (RAM).
- A program counter.
- An instruction register with instruction decoder.
- The control unit.
- A stack.

The ALU is responsible for performing all arithmetic operations such as addition, subtraction and Boolean logical operations, including exclusive-or and bit shifting. Multiplication and division is usually accomplished by repeated use of addition or subtraction, but some devices (such as the PIC17CXX series) have hardware multipliers. The working registers are used by the ALU as temporary ‘scratchpad’ memory, for example, for holding intermediate results of arithmetic operations.

A program is a set of sequential operations on *data*. The program memory is an area of memory where the actual sequence of instructions which make up the program is stored. Data memory is an area of memory where data such as the value of constants are kept for use by the program during its execution.

The program counter is a register used to store the address of the *next* instruction to be executed. Because the program consists of instructions stored sequentially in program memory, the address of the next instruction is obtained by simply incrementing the number (that is, the address), contained in the program counter.

The instruction register contains the actual binary instruction that needs to be executed. The instruction decoder takes the binary instruction and decodes it to determine what operation the instruction must perform and which data it must use.

The control unit controls the timing and sequencing of all operations necessary to correctly schedule and execute instructions. While an instruction is executing, the next instruction is fetched from the program memory and placed in the instruction register with help from the program counter. The instruction decoder then decodes the instruction and it is executed when the next execute cycle occurs.

The stack is an area of memory used to keep track of the contents of the program counter when subroutines are called. When data is written to the stack, it is stored at the ‘top’ of the stack. This operation is referred to as *pushing* data onto the stack. When data is removed from the top of the stack, the stack is said to be *popped*.

A *subroutine* is a block of program code that performs a calculation or operation that the main program needs to do a number of times. Instead of repeatedly inserting the block of code at each position in the main program where it is needed, the subroutine is called when required.

When a subroutine is called, the *return address* (that is, the address of the next instruction that must be executed when the subroutine terminates), is pushed onto the top of the stack. In other words, the program counter is first incremented to specify the address of the *next* instruction to be executed after subroutine completion and then its contents are pushed onto the stack. The address of the beginning of the subroutine is then loaded into the program counter so that it can be executed. When subroutine execution is complete, the top of the stack is *popped* and the address of the next instruction is loaded into the program counter again, so that the program can continue where it left off before the subroutine was called.

### 1.2.1 Program memory

A microcontroller needs a memory to store its program in such a way that it will not be lost when the circuit’s power supply is switched off. This type of memory is called *nonvolatile* and is implemented as Read Only Memory (ROM) because the microcontroller can only read data from it. The initial loading of program data into the ROM is done using a ROM programmer circuit designed specially for the purpose.

### 1.2.2 Types of ROM

There are different types of ROM. An EPROM (Erasable Programmable ROM) can be erased in about 15 minutes using ultraviolet light and EPROM-based microcontrollers such as the

PIC16C71 have a transparent window for erasing the internal ROM.

An EEPROM is an electrically erasable PROM and is a particularly attractive memory technology option as it can be quickly and easily erased and reprogrammed. An EEPROM-based microcontroller such as the PIC16C84 enables simple and rapid program development to take place, without the need for the lengthy ultraviolet erasure procedure needed with EPROM-based devices. Newer devices such as the PIC16F83 and PIC16F84 are advanced Flash memory versions of the older EEPROM-based PIC16C84 and PIC16CR83 microcontrollers.

For these reasons, the PIC16F84 has been chosen for discussion in this book. Once the reader has mastered the use of this device, migration to other devices in the PIC range is quite straightforward.

### 1.2.3 Data memory

During program execution, the processor needs memory space where it can *temporarily* store and retrieve data. This data is stored in a volatile memory known as Random Access Memory (RAM). Information in RAM is lost if power to the memory circuits is removed. A more descriptive term for this type of memory is *Read/Write memory*, as data can be both written to and read from it.

## 1.3 Programming a microcontroller

A microcontroller program consists of a set of instructions stored in program memory which are executed sequentially. Instructions consists of a binary word, or multiple words in some processors and because they can be easily understood by the machine, they make up what is called *machine language*. Machine instructions for the PIC16F84 consist of 14 bit binary words, each of which is manipulated by the processor as a single entity.

It is difficult for humans to remember a set of 14 bit binary instructions, so a mnemonic system of words and symbols is used to refer to them. The mnemonic set that programmers use to remember and manipulate microcomputer program instructions is called the *assembly language* instruction set. So, instead of writing 10110101110101 to tell the processor to move a byte of data to a specified location, the assembly language instruction `movf` is used instead.

The simplest method of writing a program for a microcontroller, therefore, is to write it in assembly language. Another program running on a PC or workstation computer, called an *assembler* is used to convert the assembly language program into a set of binary machine code instructions. In modern English usage, the term *software* is used to describe programs and the electronic circuits which run the software is known as *hardware*.

### 1.3.1 Execution of an instruction

In order to synchronise all the necessary events (logical state changes) taking place during the running of a microcontroller program, a timing reference or *timebase* is needed. A convenient timebase is a fixed frequency oscillator circuit which supplies a square wave signal.

A crystal is commonly used to accurately control the frequency of the oscillator circuit, but in some applications where a precise timebase is unnecessary, such as in electronic toys, a simple RC oscillator will suffice. The timebase signal is referred to as the *clock* signal and the data and instructions are *clocked* through the system.

Before an instruction can be executed, it must first be fetched from the instruction register. To maximise the speed of instruction execution, the *next* instruction is fetched at the same time that the current one is executing. This time overlapping of the fetch and execute cycles is called *pipelining*.

### 1.3.2 The hexadecimal numbering system

A microcontroller can only ‘think’ in terms of binary numbers. Manipulation of binary numbers by humans is cumbersome and error prone. Clearly a more convenient and compact notation such as the hexadecimal system is preferred. The hexadecimal or ‘hex’ system is a base 16 numbering system consisting of the numbers 0 to 9 and the letters A to F.

The letter A represents the decimal number 10, B the number 11 and so on, with F representing the number 15. Using this notation, the decimal number 17 is written as 11h and AEh is the decimal number 174 ( $10 \times 16 + 14 \times 1$ ). Alternatively, hex numbers can be written as 0xAE, 0x4B etc. The numbering system chosen for use in the assembly language program is referred to as the *radix*.

## 1.4 Microcontroller simulators

It is possible to develop microcontroller circuits by writing a program, loading it into program memory, testing the circuit on the workbench, making a note of the changes required, changing the program, reloading it and testing the new program. It is easy to see that such a repetitive procedure is very inefficient, time consuming and represents a rather opaque design method.

A microcontroller *simulator* is a program running on a PC or workstation, which accurately mimics the behaviour of a given microcontroller device. The exact performance of the simulation in terms of speed and signal voltages will not be fully represented by the simulator program, since the PC and the microcontroller are different.

The Microchip MPLAB simulator program enables the designer to execute and debug an assembly language program for a PIC microcontroller. The advantages of this simulator are:

- Program execution occurs at a lower speed than the actual circuit, so that the behaviour can be carefully examined at the user’s leisure.
- Powerful debugging facilities exist such as adding program breakpoints, inserting new instructions and single stepping, where each instruction is executed one step at a time with pauses in between.
- The facility to view the contents of internal registers and the status of port pins by making use of *watch windows*.

- The possibility of examining the effects of external stimuli such as interrupts and pin voltage state changes on the device.

# Chapter 2

## PIC microcontroller details

### 2.1 The PIC16F8X family

The PIC16F8X family of devices are CMOS (Complementary Metal Oxide Semiconductor) microcontrollers consisting of the PIC16F83, PIC16C83, PIC16F84, PIC16C84, PIC16LCR8X and PIC16LF8X types.

CMOS technology offers a number of advantages over other technologies. For example, CMOS circuits consume very little power, operate over quite a wide voltage range and are quite forgiving of bad layout and electrical noise. The PIC16X8X is available in an 18 pin IC package as shown in Figure 2.1. The IC consists of two pins for the power supply, two pins for the oscillator, OSC1 and OSC2, a pin for the master reset clear line  $\overline{\text{MCLR}}$  and 13 pins for input/output (I/O) ports, RA0 to RA4 and RB0 to RB7.

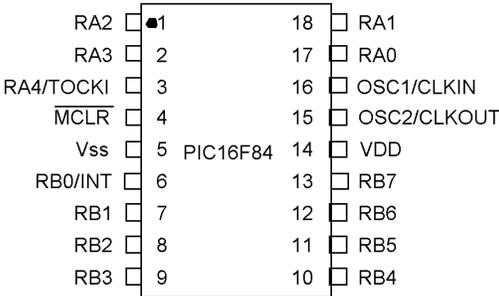


Figure 2.1: The PIC16F84 pin configuration for PDIP and SOIC packages (*Courtesy of Microchip Technology Inc.*).

### 2.2 Features of the PIC16F84

Figure 2.2 is a block diagram of the PIC16F84 microcontroller.

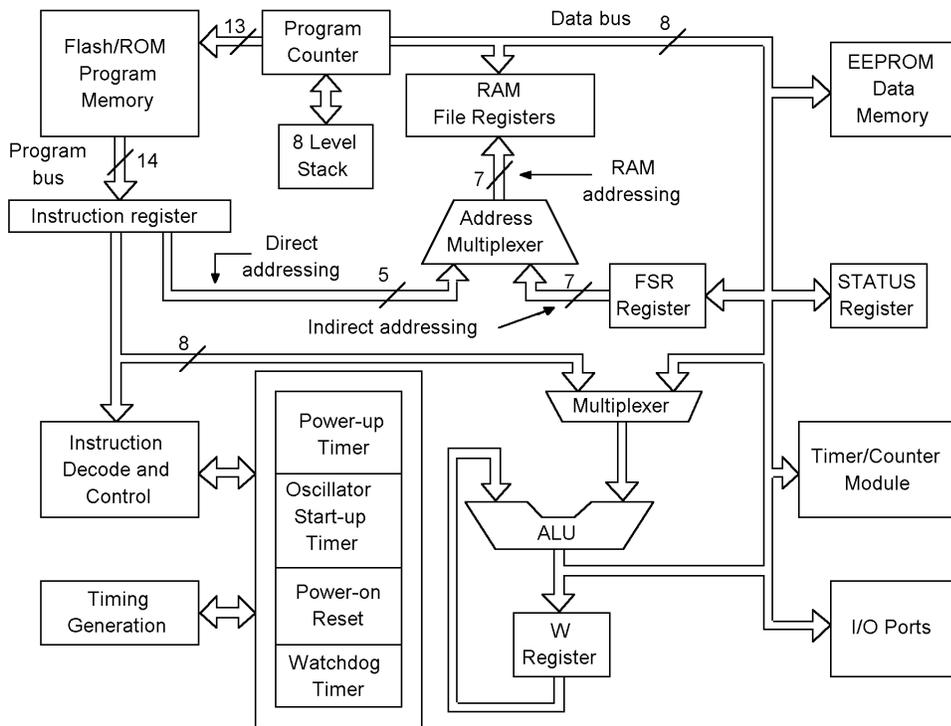


Figure 2.2: Simplified PIC16F84 microcontroller architecture (*Courtesy of Microchip Technology Inc.*).

The system consists of an ALU, a working register W, program and data memories (RAM), a program counter, an instruction register, an instruction decode and control register, a stack and peripheral logic circuitry for timers, I/O control and resets. The PIC operates along the lines discussed in Section 1.2.

### 2.2.1 The PIC16F84 architecture

The PIC architecture is based on a configuration known as a *Harvard machine structure* where separate memories are used for the program and data which are accessed via separate buses. In the PIC16F84, the program bus is 14 bits wide, whereas the data bus is 8 bits wide. In addition, the PIC family is based on a Reduced Instruction Set Computer (RISC) configuration which use fewer instructions than a Complex Instruction Set Computer (CISC). All the PIC devices use less than 60 instructions.

In general, the PIC16XXX devices have only 35 instructions, whereas the PIC17XXX devices have only 58 instructions. There is a substantial amount of program code compatibility amongst

different devices in the PIC family. A program written for one PIC device can easily be assembled and used in another device type with a minimum number of modifications.

The PIC family are fully static devices, meaning that they preserve the contents of their registers when the clock frequency is reduced to zero. In PIC microcontrollers, each instruction takes four clock periods to execute. If a 1MHz clock frequency is used, the corresponding clock period is  $1\mu\text{sec}$ , so each instruction will take  $4\mu\text{sec}$ —this time is called the *instruction cycle time*  $t_i$ . The fastest devices in the PIC family can operate at clock frequencies up to 33MHz, with corresponding instruction cycle times of 121nsec.

Most instructions execute in one instruction cycle, but some require two cycles because they need to branch to some destination other than the next address in the PC. Instructions that need two cycles to execute are `btfs`, `btfs`, `call`, `decfsz`, `goto`, `incfsz`, `retfie`, `retlw` and `return`.

Microchip characterises PIC microcontrollers according to their instruction word lengths. The low-end PICs, such as the eight pin 12C5XX series, have 12 bit word length instructions. The midrange PICs, such as the PIC16XXX, have 14 bit instructions and the high-end 17XXX PICs have 16 bit instructions. All PIC microcontrollers are, however, classified as eight bit microcontrollers as they all manipulate data in byte units on an eight bit wide databus.

A detailed knowledge or understanding of these concepts is not essential to actually use PIC microcontrollers; all that is needed is practice in writing programs and some experimentation with microcontroller circuits to gain experience.

### 2.2.2 Program memory

The map of the program memory and the stack of the PIC16F84 is given in Figure 2.3.

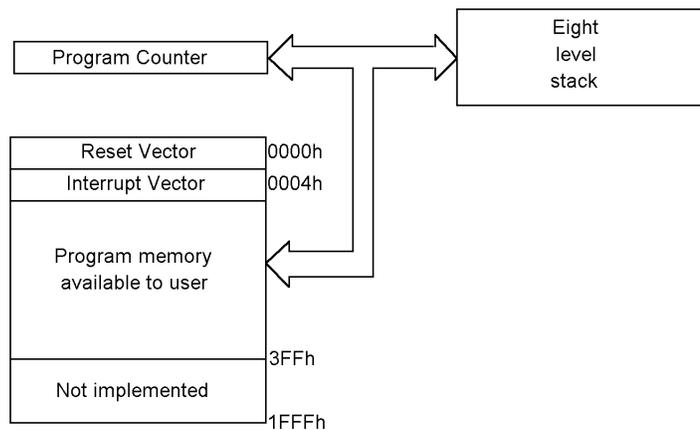


Figure 2.3: The PIC16F84 program memory and stack (*Courtesy of Microchip Technology Inc.*).

The program counter (PC) is a 13 bit wide register which will enable 8K ( $8 \times 1024$ ) program address locations to be addressed. Each program instruction is 14 bits wide, so the PC can

address  $8K \times 14$  bits of memory space. However, only 1K of this program memory is physically implemented.

### 2.2.3 Data memory

The data memory is separated into two areas, one for the special function registers discussed in Section 2.4, and one for the general purpose registers. The PIC16F84 data memory layout is given in Figure 2.4.

	BANK 0	BANK 1	
00h	INDF	INDF	80h
01h	TMR0	OPTION	81h
02h	PCL	PCL	82h
03h	STATUS	STATUS	83h
04h	FSR	FSR	84h
05h	PORTA	TRISA	85h
06h	PORTB	TRISB	86h
07h	Unimplemented	Unimplemented	87h
08h	EEDATA	EECON1	88h
09h	EEADR	EECON2	89h
0Ah	PCLATH	PCLATH	8Ah
0Bh	INTCON	INTCON	8Bh
0Ch	User defined registers	Memory area mapped in Bank 0	8Ch
2Fh			AFh
30h			B0h
7Fh	Unimplemented	Unimplemented	FFh

Figure 2.4: The PIC16F84 register file map (*Courtesy of Microchip Technology Inc.*). Note that INDF and EECON2 are not physical registers.

The memory area is further partitioned into two *banks* which require *bank switching* in the program to correctly access registers. For example, access to register PORTA requires Bank0 to be selected, whereas Bank1 must be selected to access register TRISB. Certain special function registers such as STATUS can be accessed from either of the two banks.

## 2.3 Ports

The microcontroller uses *ports* to interact with external circuitry. A port consists of a number of pins on the microcontroller IC. For example, PORTB consists of the eight IC pins RB0 to RB7 as

shown in Figure 2.1. The PIC16F84 has two ports called PORTA and PORTB with five and eight pins respectively.

For each port there is a corresponding port register, so that the binary values (high or low) represented by voltages on the pins can be read and stored as bits. If the voltages on pins RB0 to RB7 are 5V, 0V, 5V, 5V, 0V, 5V, 0V and 5V respectively, then the binary value of the port register PORTB when the port is read, will be 10101101 with the value of pin RB0 representing the least significant bit (LSB) and RB7 representing the most significant bit (MSB).

A *bidirectional* port is one that can act as either an *input* port, to receive information from external circuitry, or an *output* port, to give information to external circuitry. An input port enables voltage values imposed on the port pins by external circuit elements to be *read* into the port register. An output port enables the binary contents of the port register to be placed on, or *written* to, the port pins with high and low voltages representing binary 1 or binary 0, respectively.

Port pins on the PIC can be configured in the program to behave either as inputs or outputs by changing the relevant bits in the *tristate* register for that particular port. A port can be configured so that some of the pins are inputs and others are outputs. For example, say PORTB is required to have pins RB0, RB1, RB3 and RB7 to be inputs and the rest to be outputs. The tristate register for PORTB is TRISB and to configure a pin as an input requires the corresponding bit of TRISB to be *set* (set to 1). Similarly, an output pin requires the relevant TRISB bit to be *cleared* (set to 0). Therefore, the value that must be loaded into the TRISB register is 10001011.

### 2.3.1 Electrical characteristics of the ports

A circuit diagram of the circuitry associated with the pins RA0 to RA3 is given in Figure 2.5.

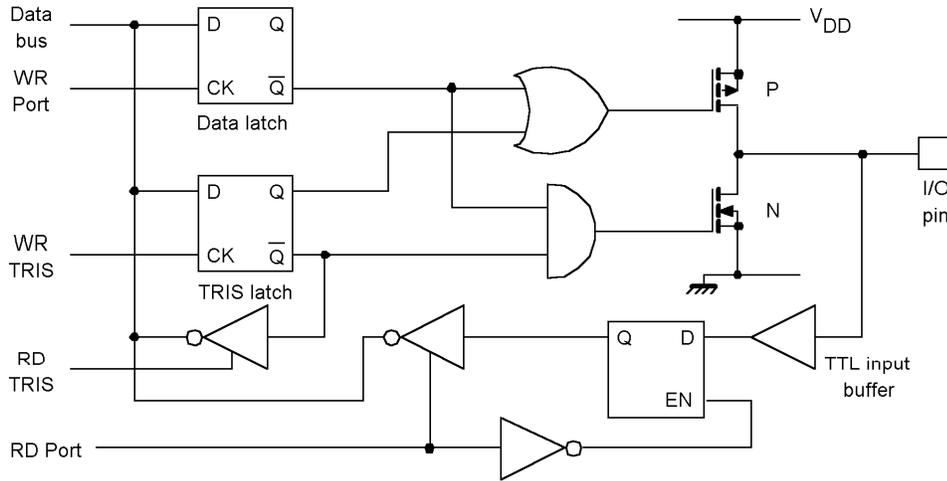


Figure 2.5: Logic circuit diagram for pins RA0 to RA3 (*Courtesy of Microchip Technology Inc.*).

The pin is connected to a CMOS transistor pair labeled ‘P’ and ‘N’. When the pin is configured as an output, the contents of the port register can be clocked through the data latch via the WR port lines to pull the output either high (upper transistor on, lower one off), or low (lower transistor on, upper one off).

When the port is configured as an input, both the output transistors will be put into a high impedance mode (ie both turned off) so that external circuitry is able to pull the pin up or down. The pin voltage can then be passed to the data bus for use by the processor. It is important to remember that attempting to impose a voltage on a port pin that is configured as an output can permanently damage the device.

The circuits of other port pins, such as RA4 and RB7 are similar to those discussed above, except that they differ by having either open circuited drains or weak programmable pullups. Open circuited drains need to be connected to the positive supply voltage via pullup resistors.

The port pins are capable of sinking 25mA and sourcing 20mA, so that loads connected to the positive supply rail have an extra 5mA of available current, compared to grounded loads.

### 2.3.2 Clock source options

There are five possible source options for the instruction cycle clock oscillator, the simplest being an RC network. Choosing an option is done when the PIC program is loaded into ROM. There are three crystal oscillator options - low power, LP, medium speed, XT and high speed, HS.

The low power option is intended for use with crystals with frequencies up to 200kHz; the XT option for crystals up to 4MHz and the HS option for crystals operating up to 10MHz. The values of the oscillation capacitors in each case is given in the data sheet [3] for the PIC device.

Circuit diagrams showing the use of the RC and crystal oscillator options are given in Figures 3.3 and 3.4.

It is also possible to run the PIC using an external clock source from a square wave signal generator on all but the RC option. The external signal is fed directly into pin 16 (OSC1) and must have a peak to peak voltage of 0 to 5V. Ensure that there is no voltage offset on the signal *before* applying it to the oscillator pins.

### 2.3.3 The timer/counter module

The PIC16F8X has a timer/counter module which can be used for either timing or counting operations. The module has an eight bit register TMR0, which can be configured to increment its value via clock pulses originating from the internal clock, or from an external source applied to pin T0CKI. Incrementing occurs on either the rising or falling edge of the input signal.

The TMR0 register, which can be written to or read from, increments from a starting value of zero. When it overflows, an interrupt signal is generated causing the interrupt flag TOIF to be set. The program will then jump to the interrupt vector at address 0x4 as discussed in Section 3.6.2.

In addition, there is a programmable *prescaler* which is simply a programmable divider, implemented using an asynchronous ripple counter, which can be used to divide the timer/counter

input frequency by a value of up to 256. Prescaler division ratio and edge triggering direction selection can be achieved by setting or clearing the relevant bits in the `OPTION` register as discussed in Section 2.4.3. Details of how to use this module are given in Section 3.7.

### 2.3.4 Low power operation and the sleep instruction

In portable battery powered equipment, it is desirable to minimise power consumption, especially during periods when the microcontroller is idle, waiting for an input or an interrupt to spur it into action. To conserve power while idle, the processor can be placed in a state of ‘suspended animation’ using the `sleep` instruction. The power consumption in sleep mode is very low.

When the processor is in sleep mode, all activity is suspended, except for the watchdog timer. No execution of instructions is possible as the internal clock is stopped and the timer/counter module is disabled.

Waking up the processor can only be achieved via a reset, a pin interrupt or a watchdog timer time-out. To ensure low power consumption, all unused inputs must be connected to one of the supply rails.

### 2.3.5 The watchdog timer

A *watchdog timer* is an internal timer running independently of the system clock. It resets the device in the event of a program or circuit malfunction or if an unknown logical state is encountered. For example, if the program hangs, the watchdog timer will time out and reset the processor. The PIC16F84 has a watchdog timer with a timeout period of approximately 18msec, with no prescaler, determined by a separate internal *RC* oscillator.

## 2.4 Special function registers

An area of data memory is dedicated to registers that are required for configuration and data flow control. This dedicated memory area is divided into a number of *special function registers* which cannot be used as general purpose registers by the programmer. The special function registers are `TMRO`, `OPTION`, `PCL`, `STATUS`, `FSR`, `PORTA`, `PORTB`, `TRISA`, `TRISB`, `EEDATA`, `EECON1`, `EEADR`, `EECON2`, `PCLATH` and `INTCON` and are situated in the data memory locations shown in Figure 2.4.

### 2.4.1 The port and port control registers

`PORTA` and `PORTB` are located in Bank 0 and are the actual *registers* used for holding the contents of the port pin binary values. However, the collection of pins as well as the port registers are usually referred to rather loosely as ‘`PORTA`’ or ‘`PORTB`’. The `TRISA` and `TRISB` registers, situated in Bank 1, are the tristate registers and are used to configure the port pins as inputs or outputs as desired. `PORTA` is associated with five input/output pins, `RA0` to `RA4` and their I/O status is controlled by setting or clearing bits in the `TRISA` register. Similarly, `PORTB` is an eight bit wide port and its I/O status is controlled by the `TRISB` register, as discussed in Section 2.3.

### 2.4.2 The STATUS register

The STATUS register stores the status of the ALU, the power down, the time out and the bank select bits. The first three bits (STATUS<0> to STATUS<2>) are the carry (C), digit carry (DC) and zero (Z) flags of the ALU respectively. The values of these bits change depending on the results of arithmetic or logical operations performed during program execution.

Bits 3 and 4 are the power down  $\overline{PD}$  and watchdog timer timeout  $\overline{TO}$  bits respectively and bits 5 and 6 (RP0 and RP1) are the bank selection bits.

### 2.4.3 The OPTION register

As its name suggests, the OPTION register allows the programmer to select timer settings and other parameters. Bits 0, 1 and 2 are the three prescaler division ratio bits for either the timer/counter module or the watchdog timer WDT, depending on which of these have been allocated for use.

Bit 3 selects the assignment of the prescaler/postscaler to either the timer/counter, or the watchdog timer. Bit 4 selects whether the timer/counter increments on the falling edge or the rising edge of the external square wave signal on pin TOCKI when external timer clocking is selected; otherwise, the waveform is ignored.

Bit 5 is used to choose between external timer clocking via the TOCKI pin, or via the internal instruction cycle clock. Bit 6 selects between interrupt triggering on either a rising or falling edge. The timer/counter parameters are discussed fully in Section 3.7.

Finally, bit 7 enables or disables the weak internal pullups on the pins of PORTB when the port is configured as an input. The pullups are disabled when the port is in output mode.

### 2.4.4 The INTCON register

This register is used to configure the interrupt control logic circuitry. Bits 0 to 6 are used to configure the interrupt enable/disable statuses and the interrupt flags for the four interrupt sources.

No interrupt to the CPU will result unless the GIE bit is set. The GIE bit is the bit INTCON<7> and when set, enables all un-masked interrupts. The use of the parameters associated with this register is explained in detail in Section 3.6.2.

# Chapter 3

# PICs in Practice

## 3.1 The Power supply

The PIC range of microprocessors offers a wide operating voltage range varying from 2VDC to 6VDC, depending on the device. For example, the PIC16LF84 operates over this extended voltage range. Consult the data sheets [3] for device specifications.

For the purposes of this discussion, a simple 5VDC power supply will be used. A 5VDC supply is easy to construct due to the availability of monolithic voltage regulators such as the 7805 positive 5V regulator which provides good regulation as well as automatic thermal shutdown and short circuit overload protection. A suitable circuit for use in the PIC projects discussed is shown in Figure 3.1.

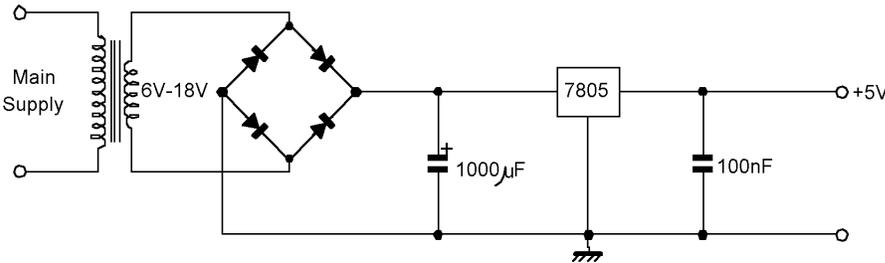


Figure 3.1: A simple 5VDC power supply suitable for PIC experiments.

The circuit consists of a transformer, a full wave bridge rectifier, the 7805 IC voltage regulator and some ripple smoothing capacitors. The four diodes comprising the full wave bridge rectifier can be replaced with a four terminal potted version. The voltage regulator should be mounted on a metal heatsink.

The actual value of the transformer secondary voltage is not critical, provided it falls within the range of 6V (RMS) to 18V (RMS). The voltage regulator needs an input voltage at least 2V higher than its output voltage and it can operate with a maximum recommended voltage of 25V

across its input terminals. Operation at high secondary transformer voltage requires the size of the regulator heatsink be increased accordingly.

### 3.1.1 Circuit layout and construction

It is advisable to pay some attention to construction and layout to prevent frustrating circuit problems from occurring. With low frequency circuits in the kHz range, layout is not critical, as the effect of circuit board and wiring parasitic capacitances and inductances is negligible. At higher frequencies, in the region of tens of MHz, these effects become more pronounced, so that bad wiring can actually prevent a circuit from working.

There are a few general rules to obey when constructing the PIC circuits discussed here.

- The oscillator crystal and capacitors should be located in close proximity to the IC with short copper tracks or connecting leads.
- A small decoupling capacitor ( $\approx 0.1\mu\text{F}$ ) should be placed across the power supply as close as possible to the PIC.
- All circuits should be grounded to the circuit chassis at one point only. Multiple ground points can cause problems due to the nonzero resistance of the ground plane.
- Keep wiring neat - this will improve reliability as well as making the circuit easier to debug and maintain.
- CMOS is sensitive to static electricity which can cause a build up of quite a few thousand Volts - more than enough to destroy a high impedance device. Avoid making contact with the pins. Transfer the IC from the protective foam container to the circuit only after discharging built up static electricity by touching the earth plane to which the circuit is connected. Work on an earthed metal table if possible.
- All pins that act as inputs must be connected either to the positive supply rail or to ground. Inputs left floating can cause unnecessarily high current consumption and spurious triggering of CMOS circuits.

### 3.1.2 Powering up CMOS

CMOS logic transistors are intended to operate in one of two states; either fully switched on, or fully off. CMOS logic circuits, such as inverters, are sometimes used as amplifiers by biasing them in their linear region, that is, in between fully on and fully off. This type of operation results in an increase in the power consumption and, in the context of digital logic circuits, produces unpredictable results.

When power is applied to a CMOS circuit, the supply voltage must rise from zero, pass through an intermediate region where linear operation takes place, to reach the final value. As some circuitry within the microcontroller begins operation at voltages lower than others, the device may power up in an unknown logical state. To ensure reliable and consistent initial conditions on power-up, it is desirable to use special reset circuitry to delay operation until the power supply voltage has settled to its final state.

### 3.1.3 Power on reset

When the power supply to a PIC is switched on, the rise of the voltage on the  $V_{DD}$  pin is detected by internal circuitry which then generates a reset pulse to initialise the device. For the internal power on reset (POR) scheme to work, the rate of rise of the power supply voltage must be sufficiently fast - less than about 50msec. If the power supply rise time is too long, an external POR circuit such as that shown in Figure 3.2 is recommended, although such a circuit is seldom necessary with good power supplies.

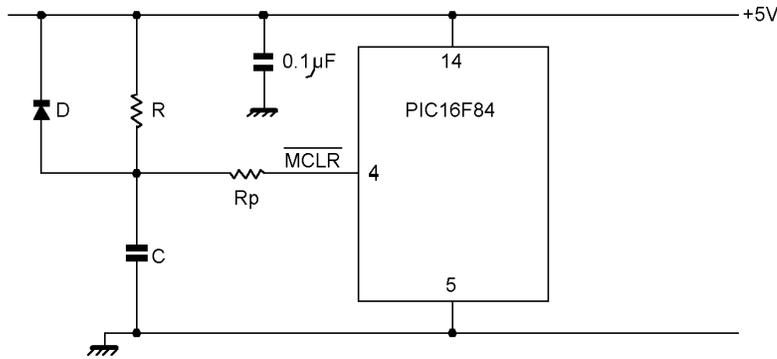


Figure 3.2: An external power on reset circuit (*Courtesy of Microchip Technology Inc.*).

The operation of this circuit is as follows. When the power is first applied, the capacitor is discharged and it starts to charge up through resistor  $R$ . The  $RC$  network delays the rise of voltage on the master clear reset pin  $\overline{\text{MCLR}}$ , so that an effective reset signal is experienced by the device. In other words, the  $\overline{\text{MCLR}}$  pin is held down for a short while until the power supply voltage has stabilised.

The diode  $D$  provides a rapid discharge path for the capacitor when power is removed and resistor  $R_p$  prevents the  $\overline{\text{MCLR}}$  input from being damaged by excessive capacitor discharge currents.

The rise of the capacitor voltage obeys the equation

$$v_C = V_{DD} \left(1 - e^{-t/RC}\right) \quad (3.1)$$

which can be rearranged into the form

$$t = RC \ln \left( \frac{V_{DD}}{V_{DD} - v_C} \right) \quad (3.2)$$

From equation (3.2), the capacitor voltage will take  $0.693RC$  seconds to reach a voltage of  $V_{DD}/2$ , so the approximate values of  $R$  and  $C$  can be determined from the equation

$$RC = 1.44t_{0.5} \quad (3.3)$$

where  $t_{0.5}$  is the time taken for the capacitor to charge to a voltage of  $0.5V_{DD}$ .

The value of  $0.5V_{DD}$  is chosen, as this is the voltage in the middle of the linear region and roughly represents the onset of a logical high. (See Section 3.1.2).

## 3.2 The structure of a PIC program

The exact structure of a PIC program depends to some extent on the programmer's personal programming style. It is advisable to put comments into programs to indicate the purpose and operation of the code, to facilitate maintenance at a later stage.

The assembler will ignore all lines beginning with a semicolon (;) and this is used for inserting comments into the program. For example, the following is a block of header text to document the program:

```
!--Program "Therm.asm" to process a thermocouple signal.
!--Author: Horace Nurk.
!--Version: 1.00
!--Date: 4 July 1997.
```

Comments are useful in the body of the code to annotate the individual lines of code or code fragments, for example

```
!--Read the port pins, load into 'testreg' and test bit 2.
label1 movf PORTA,0      ; Move PORTA register into W register.
        movwf testreg    ; Move contents of W into testreg.
        btfss testreg,2  ; Test bit 2 of testreg, skip next if set.
```

The assembler requires labels of statements, such as `label1`, to begin in the leftmost column. Declarations such as those needed to specify locations of registers in data memory must also begin in the first column. For example,

```
reg1    equ    0xC
```

assigns the register with symbolic name `reg1` to memory location `0xC`.

Program statements such as `movwf testreg`, must be indented from the first column.

The following two lines appear at the top of the program:

```
LIST    p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
```

The `LIST` command tells the assembler to turn the listing option on. The assembler produces a hex file (such as *filename.hex*) for loading into the program memory of the PIC, as well as a listing file (such as *filename.lst*) which is a listing of the program with information such as symbol tables and the total amount of memory used. The latter file is useful as it provides processor resource usage information for use during the development of a PIC project. The `p`, `r` and `f` options specify the processor, the radix (hex, decimal etc.) and the desired hex file output format respectively.

The `include` directive tells the assembler to include the contents of the specified file as if it were actually part of the file. This is useful in avoiding unnecessary typing and, more importantly, in preventing errors from occurring by including the same correct include file each time a new program is developed.

### 3.2.1 The header file

The include file P16C84.inc listed below, contains declarations of variables and memory address allocations.

```
LIST
; P16C84.INC Standard Header File, Version 1.00 Microchip Technology, Inc.
NOLIST
;=====
; Verify Processor
;=====
IFNDEF __16C84
    MESSG "Processor-header file mismatch. Verify selected processor."
ENDIF
;=====
; Register Definitions
;=====
W equ H'0000'
F equ H'0001'
;----- Register Files-----
INDF equ H'0000'
TMRO 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' ;Note the definition of this register.
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'
RPO 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 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'AF'
    __BADRAM H'07', H'30'-H'7F', H'87'

```

```

;=====

```

```

;      Configuration Bits

```

```

;=====

```

```

_CP_ON       equ    H'3FEF'
_CP_OFF      equ    H'3FFF'
_PWRTE_ON    equ    H'3FFF'
_PWRTE_OFF   equ    H'3FF7'

```

```

_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

The `equ` directive tells the assembler to equate the left hand side variable with that on the right hand side. The main reason for this is to make programs more meaningful to the programmer by assigning symbolic names to variables and addresses. For example, instead of writing `bsf H'0003',H'0005'`, which the processor understands, it is more meaningful to the human programmer to write `bsf STATUS,RP0`, using the symbolic name instead.

Referring to Figure 2.4, it can be seen that the register file names are assigned to their addresses using the `equ` directive. Similarly, the numerical value of individual bits can be assigned symbols so that, for example, `RP0` refers to bit 5.

### 3.2.2 The configuration word

The configuration bits are used to configure the hardware operating parameters of the PIC when loading the hex file into the Flash memory or EEPROM. Setting or clearing these bits allows selection of the oscillator type (`LP`, `XT`, `HS` or `RC`) and enabling or disabling the watchdog timer, power-up timer and code protection facilities.

The configuration bits are usually set or cleared from the PICStart-Plus menu, but this can also be done directly in the program source code using the `__CONFIG` instruction. (Note that there are *two* underscore symbols). For example, to select the `RC` oscillator option with power-up timer on, watchdog timer off and code protection off; the following line can be placed after the `include` instruction:

```
__CONFIG _RC_OSC&_PWRTE_ON&_WDT_OFF&_CP_OFF
```

Setting the code protection configuration bit prevents the program in the PIC being read out by an unscrupulous scoundrel intent on stealing your intellectual property by reverse engineering! This feature is useful when a design is at the production stage where thousands of devices are to be programmed, but it should be disabled during system development.

## 3.3 Project 1

A very simple program, `Port.asm`, to set the voltages on pins `RB0` to `RB7` of `PORTB` to the arbitrarily chosen bit pattern `10101101` is listed below. The order is `LSB` first, `MSB` last, because `RB0` is the `LSB`.

Note that there are no line numbers. The program listings with line numbers given in the application notes [4] are *absolute listings* generated by the assembler and are used for analysis and debugging purposes.

```

;--Program "Port.asm". A simple program to write a byte to PORTB.
;--Author: Montague Paravrov-Nikots.
;--Date: 7 July 1997.
;--Version: 1.01

LIST    p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
org 0x0          ; Reset vector.
;--Set up the pins of PORTB to act as outputs.
;--Select Bank 1 to access TRISB register.
    bsf STATUS,RP0      ; Select Bank 1.
    clrf TRISB          ; Make all PORTB pins outputs.
;--Switch back to Bank 0 to access PORTB register.
    bcf STATUS,RP0      ; Back to Bank 0.
;--Load a byte into PORTB.
    movlw B'10110101'   ; Move a byte into the working register.
    movwf PORTB         ; Move contents of W into PORTB register.
wait   goto wait        ; Loop forever until power is removed.
end     ; End of program "Port.asm".

```

### 3.3.1 Operation of program Port.asm

The pins of PORTB must first be put into output mode by correctly setting up the tristate register for PORTB, namely TRISB. To select output mode for all pins, all the TRISB bits must be *cleared*, so the instruction `clrf` is used on TRISB.

Once the ports are set up as outputs, a value must be written to the PORTB register which will then be written to the pins RB0 to RB7 from the register. Since the PIC does not have a single instruction to move a byte into a given register, two instructions are needed. The first is to load the working register W and the second is to move the contents of W into the desired register, PORTB in this case.

To move a byte into the working register, the instruction `movlw` is used to move a literal value (an explicitly specified value) into W. Next, the instruction `movwf` is used to move the contents of W into the specified register.

Since the registers TRISB and PORTB are in different banks of data memory, the correct bank must be selected before they can be accessed. Bank switching is done by setting (for Bank 1) or clearing (for Bank 0), bit 5 (RP0) of the STATUS register. The instruction `bsf` is used to *set* a bit in a register and `bcf` is used to *clear* a bit.

Bit STATUS<6> (RP1) does not need be changed, as it was already cleared on power up by default. The PIC16X8X has only two banks, so that RP1 is always zero.

After the pins have been set to their desired values, the program goes into an infinite loop at label `wait`, where it stays until power is removed.

The `org` instruction near the beginning of the program acts as a marker for the start of the program. When the circuit is either powered up or reset, the program counter is loaded with

a starting address to properly define the program origin. The starting address of the program is called the *reset vector*. Similarly, when an interrupt is encountered, the program will automatically branch to the *interrupt vector* which is a specially reserved location in the program memory (see Figure 2.3). In the PIC16F84, the interrupt vector is at address 0x4.

### 3.3.2 Assembling Port.asm

The assembly language file `Port.asm` should be saved in a suitable directory such as `c:\mplab`. The first step is to create a new project. Go to the **Project** menu and click on **New Project**. In the **Project Path and Name** slot, type `c:\mplab\test.pjt`. In the **Development Mode** slot, use the vertical arrow to select **MPLAB-SIM Simulator** then click **OK**. After clicking **Yes** when asked if the project is to be saved, the **Edit Project** window will be displayed. Go to the **Non Project Files** window, highlight the file `port.asm` and click on the **Add** button. Verify that the file `port.asm` is added to the **Project Files** window, then click **OK**.

Now the source file must be loaded so that it can be assembled. Go to the main **MPLAB** menu and click on **File**. To open the source file, click on **Open source**. In the file name window, insert the file name `port.asm` and click **OK**.

The leftmost **swap toolbar** button (just under the word **File** in the main menu) toggles between the different sets of tool buttons. The function of a tool button will be displayed at the bottom left hand corner of the screen when the mouse arrow is positioned above it. Click on the **swap toolbar** button until a set of nine predominantly green tool buttons appear. Click on the tool button third from the right to build the full project. If there are no errors in the program, green progress bars and a message will appear in the **Compile Status** window, stating that the assembly was successful. Click **OK**. To view the assembled hex file, select **File** followed by **Open**, type in the filename `port.hex` and click **OK**. A set of hexadecimal numbers will be displayed.

### 3.3.3 Simulating Port.asm

Click on **Options** in the main menu and then on **Development Mode**. Ensure that the **MPLAB-SIM Simulator** diamond is filled in and that the PIC16C84 processor is selected in the **Processor select** slot. Now reset the system by clicking on the **Reset** button and save the project.

If the **Editor** diamond is filled, the simulator will not be activated. Similarly, if the wrong processor is selected by mistake, strange behaviour will occur which may cause the user to waste much time trying to debug a program that is actually correct!

After the project has been saved, it must be opened by choosing the **Project** option on the main menu and then clicking on the project name. Toggle the toolbar until the green and red traffic light symbols appear next to the **swap toolbar** button. Open up a watch window by clicking the **Create New Watch Window** and add the variables `PORTB`, `TRISB` and `STATUS`.

Click **Reset Processor**. The black bar in the program source code window will move to a location above the instruction `bsf STATUS,RP0`. Now repeatedly click on the **Step** tool button and watch the black bar move successively down over each instruction of the program until the `wait` instruction is reached. Observe the values of the variables in the watch window and notice how the instructions are executed and the appropriate bits changed. When the contents

of a register change, the corresponding symbol changes colour from blue to red. To repeat the simulation, reset the processor and step through the program again.

### 3.3.4 Loading the hex file

Once the operation of the program has been successfully verified via simulation, the assembled hex file can be loaded into the PIC's memory. The first step is to set up the communications port on the PC. Connect the PICStart-Plus programmer to the appropriate communications port, such as com1 and power up the programmer.

Click on **Options** in the main menu, then go to **Programmer Options** and then to **Communications Port Setup**. Click on **Communications Port Setup** and fill in the appropriate diamond corresponding to the communications port to be used.

To start the actual programming process, select PICStart-Plus on the main menu, followed by **Enable Programmer**. If there are no hardware or software problems, MPLAB will establish communication with the PICStart-Plus programmer. At this point, the PIC IC should be placed in the programmer socket and the gate closed. *It is important that the PIC be absent for those periods when power is being applied or removed, otherwise the PIC may be damaged.*

When communication has been successfully established, go back to the PICStart-Plus menu and select the **Program/Verify** option. A new window containing options for setting up the programming configuration will open. For this example, the watchdog timer should be disabled, the power up timer enabled and the RC oscillator option chosen. When ready, select **Program** and wait for the PICStart-Plus programmer to complete programming. During programming, the **active** LED will be lit on the PICStart-Plus. After programming, select **Verify** to ensure that the programming is correct.

If a programmer other than the PICStart-Plus is to be used for programming the PIC, ensure that it supports the PIC device. The hex file produced by MPLAB can be transferred to the programmer and used to program the PIC.

### 3.3.5 Testing the circuit

Figure 3.3 shows a very simple circuit for testing the program `Port.asm`. The program does not require accurate timing information, so a simple RC oscillator is all that is needed. The actual values of  $R$  and  $C$  are not critical, provided that  $R$  lies between 3k and 100k and  $C$  is greater than 20pF. Consult the data sheet [3] for details.

After the circuit wiring has been checked, apply power and check that the port pins RB0 to RB7 are at the correct logical values using either a voltmeter or by connecting a series LED and resistor to each pin in turn. If a frequency meter is available, the signal on pin 15 (CLKOUT) will be the oscillator frequency divided by four, with a period equal to the instruction cycle time.

## 3.4 Project 2

Section 3.3 demonstrated writing a byte value to the port pins. The next program will flash an LED at a fixed rate by repeatedly raising the voltage on an output pin, waiting for a prescribed

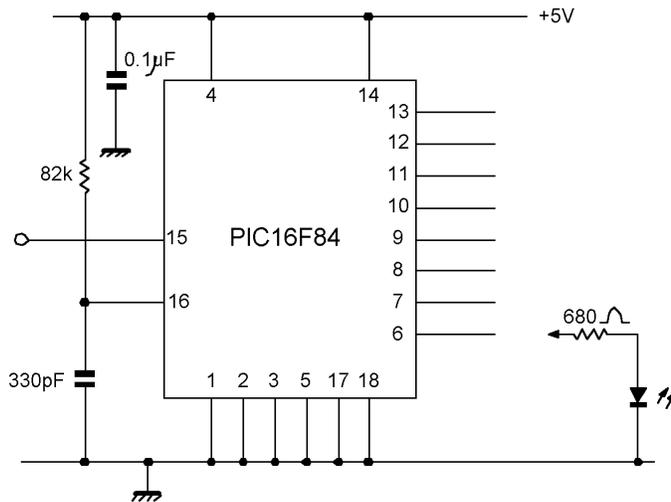


Figure 3.3: Circuit to test program `Port.asm`.

time, then grounding the pin for the same amount of time. To implement this scheme, a simple time delay needs to be devised.

### 3.4.1 The delay

A time delay can be achieved by occupying the processor in counting down to zero from a given value. To accomplish this, a register is loaded with a chosen value which is then decremented until zero is reached, after which the program continues. In assembly code terms, a decrement and a test for zero instruction are needed. A suitable instruction is *decrement f, skip if zero*, `decfsz f,d`. This instruction decrements the register `f`, replaces its contents in the destination register `d`, tests if the value is zero and, if so, skips the following instruction.

The complete delay code is as follows:

```

    movlw 0x20
    movwf delreg      ; Move the value into the delay register.
loop  decfsz delreg,1 ; Move decremented value back into delreg.
      goto loop      ; if delreg value is not yet zero.
      .....         ; Next instruction here.

```

The first two instructions move the specified value into the delay register `delreg`. Next, the value in `delreg` is decremented and if the result is nonzero, the next instruction is executed, causing the program to branch to label `loop`. If the result is zero, then the `goto loop` instruction is skipped and the program continues by executing the instruction immediately after the `goto` statement. Remember that the register `delreg` must be assigned a space in memory in the data memory area at the beginning of the program, using the `equ` directive.

### 3.4.2 Calculating the delay

The delay time depends on the instruction cycle time (four clock periods per instruction) and the number of instructions to be executed before exiting the loop. The number of instructions required is determined as follows.

Instructions `movlw` and `movwf` are each executed once. The number of times `decfsz` (with no instruction skipping) and the `goto` are executed, is given by the value loaded into register `delreg`. A `goto` is an unconditional branch instruction which takes two instruction cycle. A skip instruction, when the delay register value reaches zero, must also be added. The total delay time  $t_d$  is therefore given by

$$t_d = 3(1 + r_v)t_i \quad (3.4)$$

where  $r_v$  is the value loaded into the register `delreg`,  $t_i$  is the the instruction cycle time

$$t_i = \frac{4}{f_0} \quad (3.5)$$

and  $f_0$  is the clock oscillator frequency.

As an example, using a crystal oscillator with a fequency of 32.768kHz, the value of  $t_i$  is

$$t_i = \frac{4}{f_0} = \frac{4}{32768} = 122.07\mu\text{sec}$$

With an  $r_v$  of 0x20 (decimal 32), the delay time will be

$$t_d = 3(1 + 32)122.07 = 12.08\text{msec}$$

The code for the LED flashing program is as follows.

```

;--Program "Flash1.asm". A simple program to flash an LED.
;--Author: Peter Puntelpiteltot.
;--Date: 9 July 1997.
;--Version: 1.02

LIST p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"

delreg equ 0xC ; Set aside space for the register.

org 0x0

;--Set up the pin RB7 as an output.
    bsf STATUS,RP0 ; Select Bank 1.
    bcf TRISB,7 ; Clear bit 7 so that RB7 is an output.
    bcf STATUS, RP0 ; Go back to Bank 0.
;--Now begin the main program loop.
```

```

begin  bsf PORTB,7      ; Pull pin RB7 high.
;--Start the time delay.
      movlw 0xFF       ; Fill the W register with 11111111.
      movwf delreg     ; Move the value into delreg.
loop1  decfsz delreg,1 ; Decrement delreg, test value and skip next if zero.
      goto loop1
;--End of time delay.
      bcf PORTB,7      ; Pull RB7 down after the delay
;--Now hold the RB7 pin down for a further delay period.
      movlw 0xFF       ; This is a redundant instruction - W unchanged!
      movwf delreg
loop2  decfsz delreg,1
      goto loop2
      goto begin      ; if delreg is zero.
      end             ; End of program "Flash1.asm".

```

Note that different label names must be used for each delay code block, otherwise an error message will be issued by the assembler. The program flashes the LED then cycles back to label `begin` to repeat the process.

The last `movlw` instruction is not required in this instance as the contents of the W register have already been set to the required value by the previous `movlw` instruction. Since no other operation has occurred to change its contents since then, it does not need to be reloaded.

To replace program `Port.asm` with `Flash1.asm`, click on **Project** on the main menu, then select **Edit Project**. Select `Port.asm` and click **Remove** to remove the program from the project. Failure to remove the old file from the project will result in the wrong source code program being assembled. Now select `Flash1.asm` and click on **Add** to add it to the project, then click **OK**.

Assemble the program `Flash1.asm` and set up the program for simulation as described in Section 3.3.3. To test the time delay, open up the stopwatch window by clicking on **Window**, followed by **Stopwatch**. Set the target frequency to 32.768kHz. In the source code window, place the cursor on the line `movlw`, after the line `bcf PORTB,7`. Click the right hand side mouse button and in the resulting window click on **break point(s)**. The selected line will be highlighted in red indicating that a *breakpoint* has been set.

Reset the processor and single step until the cursor is over the first `movlw` instruction at the start of the time delay. Now go to the stopwatch window and click on **Zero** to initialise the timer. Run the program by clicking on the green traffic light symbol. The program will run autonomously until the breakpoint is reached and the stopwatch will display the time taken and the number of cycles. While the program is executing, the bottom status bar will be highlighted in yellow. Verify that 93.75msec elapsed in 768 cycles.

Load the hex file into the PIC. Before programming, the configuration must be chosen to have the watchdog timer disabled, the LP oscillator selected, power up timer enabled and code protection off. If the watchdog timer is not disabled, the device will reset approximately every 18msec, so that the flasher timing will be disrupted. *Forgetting to disable the watchdog timer can lead to hours of unnecessary headscratching and program debugging!*

The circuit diagram of the LED flasher is given in Figure 3.4. A crystal with a resonant frequency of 32.768kHz is used in this circuit. These crystals are common and are used in digital wristwatches as they are used to produce a single pulse per second timebase by dividing the frequency down using a 15 stage binary counter ( $2^{15} = 32768$ ). Note that pins RB0 to RB6

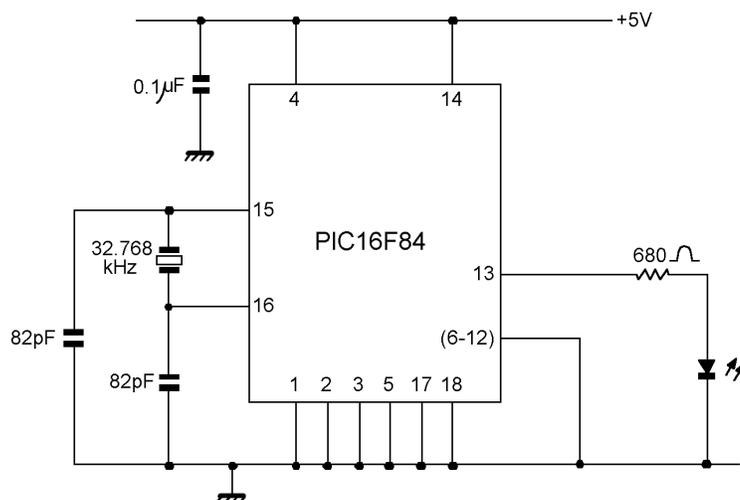


Figure 3.4: Circuit diagram for the LED flasher.

inclusive, must be grounded as they are configured as inputs by default on power up. Wire up the circuit shown and observe the results. Note that the voltage on pin RB7 results in a rectangular waveform. Waveform generation is a typical application for a microcontroller.

### 3.4.3 Program improvements

Program `Flash1.asm` uses two delays with the same code repeated. When a sequence of instructions is used often in a program, it is usual to write a subroutine, as is done in high level languages, and call it whenever it is needed. PIC assembly language allows subroutines to be called using the `call` instruction.

A more pressing problem, however, is the fact that the delay time with the 32.768kHz crystal is too short. With  $r_v = 0xFF$  (decimal 255), which is the largest value that can be loaded into `delreg`, the delay time given by equation (3.4) is only 94 msec. This short delay of around a tenth of a second will cause the LED to flicker rather than flash. One way around this problem would be to use a slow ( $\approx 3\text{kHz}$ ) RC oscillator.

A more elegant solution, however, is to retain the crystal and use an additional register in a way that allows a count down from more than the maximum value of 255 possible using only the one eight bit register `delreg`. This can be accomplished using another register in an outer loop so that the original delay code is called a number of times as specified by the value loaded into register `count` as follows:

```

;--Start a longish delay.
    movlw 0xFF
    movwf delreg ; Fill the register.
    movlw 0xC
    movwf count ; Load 12 into count.
loop2 decfsz count,1
    goto loop1
    goto exit
loop1 decfsz delreg,1
    goto loop1
    goto loop2
exit  .... ; Place the next program instruction here.

```

Registers `delreg` and `count` are filled with values 255 and 12 respectively. At label `loop2`, the value in `count` is decremented and, if nonzero, the program branches to `loop1`. At `loop1`, `delreg` is decremented until it reaches zero, whereupon the second `goto loop1` instruction is skipped and the program branches back to label `loop2`. When `count` reaches zero, the first `goto loop1` instruction is skipped and the instruction at label `exit` is executed. The complete listing of the program for the modified program follows:

```

;--Program "Flash2.asm". Modified LED flasher program.
;--Author: Hercules Uzanoglu.
;--Date: 19 July 1997.
;--Version: 1.03

LIST    p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
delreg  equ 0xC ; Set aside space for the register.
count   equ 0xD
        org 0x0

;--Set up the pin RB7 as an output.
    bsf STATUS,RP0 ; Select Bank 1.
    bcf TRISB,7 ; Clear bit 7 so that RB7 is an output.
    bcf STATUS,RP0 ; Go back to Bank 0.
;--Now begin the main program loop.
begin  bsf PORTB,7 ; Pull pin RB7 high.
;--Start a longish delay.
    movlw 0xFF
    movwf delreg ; Fill the register.
    movlw 0xC
    movwf count ; Load 12 into count.
loop2  decfsz count,1
    goto loop1
    goto exit

```

```

loop1  decfsz delreg,1
        goto loop1
        goto loop2
exit   bcf PORTB,7      ; Pull RB7 down after the delay
;--Now hold the RB7 pin down for a further delay period.
;--Start another longish delay.
        movlw 0xFF
        movwf delreg ; Fill the register again.
        movlw 0xC
        movwf count  ; Load 12 into count.
loop4  decfsz count,1
        goto loop3
        goto exit2
loop3  decfsz delreg,1
        goto loop3
        goto loop4
exit2  goto begin      ; if delreg is zero.
        end            ; End of program "Flash2.asm".

```

Replace Flash1.asm with Flash2.asm in the project and assemble the program. Insert a breakpoint at the movlw instruction at the beginning of the second delay. Step through the program until the cursor is over the movlw instruction at the beginning of the first delay. Set the stop-watch to zero and run the program to the breakpoint to verify that the delay is about one second. Test the circuit to see that the hardware operates correctly.

The last modification is to replace the repeated time delay code with calls to a subroutine as shown in program Flash3.asm.

```

;--Program "Flash3.asm". Modified LED flasher program.
;--Author: Arthur Fishlegge.
;--Date: 19 July 1997.
;--Version: 1.04

        LIST    p=PIC16C84, r=hex, f=INHX8M

        include "P16C84.inc"
delreg  equ  0xC    ; Set aside space for the register.
count   equ  0xD
        org  0x0

;--Set up the pin RB7 as an output.
        bsf  STATUS,RP0 ; Select Bank 1.
        bcf  TRISB,7    ; Clear bit 7 so that RB7 is an output.
        bcf  STATUS, RP0 ; Go back to Bank 0.
;--Now begin the main program loop.
begin   bsf  PORTB,7    ; Pull pin RB7 high.
        call delay

```

```

        bcf PORTB,7      ; Pull pin RB7 down.
        call delay
        goto begin
;--End of main program body.

;--Subroutine to provide a delay of about a second.
delay  movlw 0xFF
        movwf delreg ; Fill the register.
        movlw 0xC
        movwf count  ; Load 12 into count.
loop2  decfsz count,1
        goto loop1
        goto exit
loop1  decfsz delreg,1
        goto loop1
        goto loop2
exit   return
;--End of the subroutine delay.
        end                ; of program "Flash3.asm".

```

Using a subroutine has shortened and simplified the main body of the program. Program operation is almost identical except for the additional overhead of a `call` and a `return` instruction, each of which require two instruction cycles. Note that the subroutine is located before the `end` statement but after the `goto begin` instruction. If the `goto begin` instruction is absent, program execution will begin at the top and run to the `end` statement, executing the delay subroutine a total of *three* times.

When the program reaches the instruction `call delay`, it branches to the label `delay` and executes the code in the subroutine until the `return` instruction is reached. When execution of the subroutine is complete, the program returns to the next instruction after the `call` instruction, so that it continues from the correct place.

Select `Flash3.asm` as the project source file and assemble it. Single step until the cursor is over the `call` instruction. Now zero the stopwatch and click on the tool button next to the step button, which is `Step Over`. The program will then execute the subroutine to its completion. Verify that the delay time is as expected.

If the `Step` button is used, each individual instruction of the subroutine will be stepped through which will take about 8500 instruction cycles!

### 3.4.4 Using an input

The next program `Flash4.asm`, shows how the PIC can read the logical state of an input pin to determine what to do next. The program repeatedly reads the binary value of the pin voltage into the port register, performs a test and, based on the results, initiates a sequence of events. This type of behaviour is a typical requirement in microcontroller based systems. Figure 3.5 shows the circuit diagram for program `Flash4.asm`.

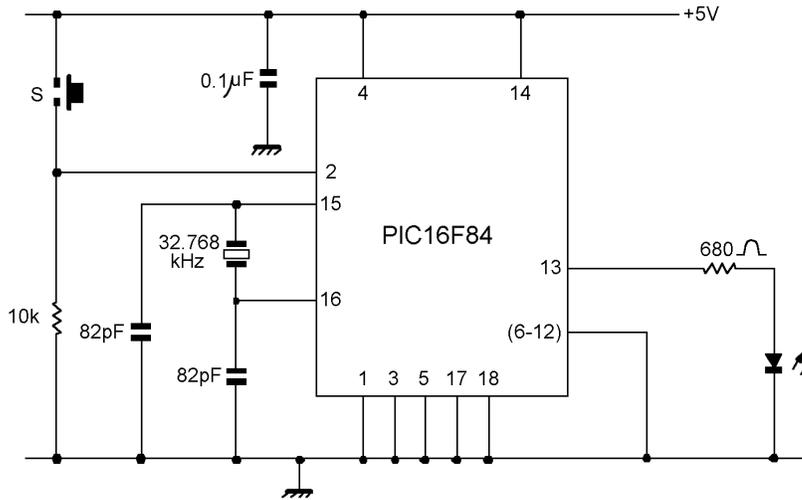


Figure 3.5: Circuit diagram for use with program Flash4.asm.

The input pin RA3 is held at ground potential by the 10k resistor, so that bit PORTA<3> will be read as a 0. When the pushbutton, S, is depressed, the voltage on pin 2 (RA3) becomes +5V and the bit is read as a 1.

```

;--Program "Flash4.asm". Modified LED flasher program.
;--Author: Marmaduke St. John Woolhooter.
;--Date: 21 July 1997.
;--Version: 1.05

LIST    p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
delreg  equ 0xC    ; Set aside space for the register.
count   equ 0xD
org 0x0

;--Set up the pin RB7 as an output.
bsf STATUS,RP0    ; Select Bank 1.
bcf TRISB,7      ; Clear bit 7 so that RB7 is an output.
bsf TRISA,3      ; Make RA3 an input.
bcf STATUS, RP0  ; Go back to Bank 0.
clrf PORTA      ; Clear the PORTA and PORTB registers to enforce
clrf PORTB      ; desired power up conditions.

;--Now begin the main program loop.
read  movf PORTA,1    ; Read the contents of PORTA into itself.

```

```

        btfss PORTA,3    ; Test bit 3, skip next instruction if set.
        goto read       ; if bit 3 is not set.
;--After we know the button has been pressed, continue.
        bsf PORTB,7     ; Pull pin RB7 high.
        call delay
        bcf PORTB,7     ; Pull pin RB7 down.
        call delay
        goto read
;--End of main program body.

```

```

;--Subroutine to provide a delay of about a second.

```

```

delay  movlw 0xFF
        movwf delreg    ; Fill the register.
        movlw 0xC
        movwf count     ; Load 12 into count.
loop2  decfsz count,1
        goto loop1
        goto exit
loop1  decfsz delreg,1
        goto loop1
        goto loop2
exit   return
;--End of the subroutine delay.
        end             ; End of program "Flash4.asm".

```

To produce `Flash4.asm`, program `Flash3.asm` has been modified by the addition of the following lines of code:

```

        bsf TRISA,3     ; Make RA3 an input.
        clrf PORTA     ; Clear the PORTA and PORTB registers to enforce
        clrf PORTB     ; desired power up conditions.
;--Now begin the main program loop.
read   movf PORTA,1    ; Read the contents of PORTA into itself.
        btfss PORTA,3  ; Test bit 3, skip next instruction if set.
        goto read     ; if bit 3 is not set.
        .....        ; Next instruction here.

```

The first three modifications are to set up pin RA3 as an input and to clear the port registers. Pin RA3 has been arbitrarily chosen as an input for illustrative purposes - any unused pin from ports A or B could have been chosen instead. Also, setting up register TRISA to make RA3 an input at the beginning of the program is not strictly necessary in this particular case, as the ports are set up as inputs on power-up by default.

At the label `read`, the instruction `movf` is used to read the contents of the register PORTA and move it into itself. Moving the contents of a register into itself is a convenient trick that enables

the status of the port pins to be read into the register. It is also used when testing the contents of a register for a zero value, because the Z flag in the STATUS register is affected.

An alternative code sequence is as follows:

```
read  movf PORTA,0    ; Read the contents of PORTA into working register W.
      movwf temp      ; Move the port contents into register temp.
      btfss temp,3    ; Test bit 3, skip next instruction if set.
      goto read       ; if bit 3 is not set.
      .....         ; Next instruction here.
```

Here, an auxiliary register `temp` is used to move the port contents into. Testing of bit 3 is carried out on the `temp` register in this case, which results in an extra instruction. Note that although 0 is used to refer to the W register in commands such as `movf reg,0`, the W register is *not* the 0th register. The 0th register is the indirect addressing register INDF in the special function register area of data memory. In fact, W is a special working register located elsewhere as is evident from Figure 2.2.

Returning to the original code fragment, bit 3 of the PORTA register is tested by instruction `btfss` (*bit test, skip next if set*) to see whether it is set or cleared. If it is set, indicating that the button has been pressed, the next instruction (`goto read`) will be skipped and the program will execute the instruction after that. If `PORTA<3>` is clear, indicating that the button has not been pressed, then the instruction `goto read` will be executed. In this way, the voltage of pin RA3 will be checked repeatedly to determine when the button is pressed. This technique is known as *polling*.

### 3.4.5 Simulating Flash4.asm

To simulate the effect of an external voltage on an input pin using MPLAB, a *stimulus* file is used in conjunction with the stopwatch. The stimulus file has a column with headings `Step`, followed by the names of the pins being stimulated. The step number corresponds to the instruction cycle number displayed on the stopwatch. The stimulus file `Flash.sti` is shown below:

```
!--File "flash.sti". Stimulus file for program "Flash4.asm".
Step RA3
1      0
21     0
22     1
23     1
24     1
```

At step 1, pin RA3 will be grounded. During steps 22 to 24, RA3 will be high, simulating the pushbutton being pressed for a number of instruction cycles. In this simple example, no contact debouncing strategy has been included.

Set up the simulator with `Flash4.asm` assembled and ready to run. Reset the processor and zero the stopwatch. Open watch windows to observe PORTA and PORTB. To load the stimulus file,

click on **Debug** in the main menu, move down to **Simulator Stimulus**, across to **Pin Stimulus** and finally to **Enable**. Click on **Enable** to open the **Select Pin Stimulus File** window. Select file `flash.sti` and click on **OK**.

Single step through the program and watch the program loop around the `read` statement until step 22 is reached. At step 22, `RA3` goes high, the program exits the loop, flashes the LED and returns to the label `read`.

At this point, only a minor step is needed to implement a simple programmable time delay by reading a byte from a port into a register and using the value to determine the length of the time delay. The values on the port pins can be set using thumbwheel switches. Alternatively, a programmable rectangular waveform generator (or pulse width modulator) can be developed. The following code fragment could be used:

```
read    movf PORTA,0    ; Read the contents of PORTA into W.
        movwf count    ; Load the count register.
```

All that is required is a rearrangement of the code in the `delay` subroutine to create a variable delay. By using the `bsf` and `bcf` commands with the programmable delay code, a waveform generator can be designed.

## 3.5 More programming examples

The previous examples have used move and compare instructions to accomplish simple decisions. One of the fundamental constructs of any of the higher programming languages is the `if` statement which is used to test whether a given number is greater than, equal to, or less than another number. Using assembly language, it is possible to develop such programming constructs, although the number of instructions required is larger than would be required in a high level language such as C.

### 3.5.1 Comparing values

To determine whether a given number is less than, greater than or equal to another number, it is necessary to subtract them and test the zero and carry bits generated by the ALU. If the result is negative, then the carry bit is zero. If the carry bit is 1, then the result is either positive or zero. If the result is zero, then the zero bit is 1. The carry and zero bits are bits 0 and 2 of the `STATUS` register respectively. The following code can be used:

```
        movlw 0x3      ; Assume W has some value in it.
        sublw 0x5      ; (k-W) into W.
        btfss STATUS,0 ; Test the carry bit.
        goto LT       ; carry=0, so k<W.
        btfsc STATUS,2 ; carry=1, so k>W or k=W, so test Z.
        goto equal    ; k=W.
        goto GT       ; k>W.
```

The value in the W register is subtracted from the literal value `k`, using the instruction `sublw` and the result is stored in W. Next, the carry bit is tested and, if set, `goto LT` is skipped and the zero bit (`STATUS<2>`) is tested. If the zero bit is clear, indicating that the result was *not* zero, `goto equal` is skipped and the program branches to label `GT` because `k > W`. If the carry bit is clear, then the program will branch to `LT` as `k < W`.

An alternative way of implementing this is as follows:

```

movlw 0x3      ; Assume W has some value in it.
sublw 0x5      ; (k-W) into W.
btfsc STATUS,2 ; Test for a zero result first.
got equal     ; k=W because zero bit is set.
btfss STATUS,0 ; Test the carry bit.
goto LT       ; k<W.
goto GT       ; k>W.

```

The last three lines of the last piece of code can be replaced by

```

btfsc STATUS,0
goto GT
goto LT

```

if the instruction `btfsc` is used instead of `btfss`.

### 3.5.2 Choosing between alternatives

Often a value needs to be generated which depends on the value of a counter or on the result of a calculation. A typical example would be where seven segment decoding is required to display a measured temperature.

Consider the following code fragment which uses the subroutine `selector` and the instructions `addwf f,d`, which adds the contents of W to the contents of `f` and stores the result in destination `d`; and `retlw` which returns from a subroutine with a specified value loaded in W.

```

movlw 0x2      ; Assume the value in W is given.
.....        ; Some instructions
call selector
.....        ; Some instructions.
selector addwf PCL,1
retlw 0x41
retlw B'11010110'
retlw B'10110101'
retlw B'0x53
sel_end retlw 0x0

```

Assume that the number 2 has been loaded into W. The program counter (PC) always contains the address of the next instruction to be executed. When the program is at the `call` instruction, the PC will contain the appropriate address for that instruction. However, at the label `selector`, execution of the instruction `addwf PCL` will add the value in W to the PC, so that the program will branch to the new location. If `W = 0`, the next instruction executed will be `retlw 0x41` which is the first return value in the selector subroutine. Therefore, the value contained in W will be the *offset* value in the table, so if `W = 2`, then the value returned will be 10110101, which is the third value in the table. This is a simple example of *relative addressing*.

### 3.5.3 Indirect addressing

Indirect addressing is needed in situations where an operation must be performed on different registers and where the particular register used depends on events determined at some other point in the program. For example, suppose a program needs to fill 16 registers, such that register `reg1` contains the number 1, `reg2` the number 2 and so on. Assume that the registers `reg1` to `reg16` are at addresses 0xC to 0x1B respectively. The brute force way of doing this would be as follows:

```

movlw 0x1
movwf reg1
.....
movlw 0x10
movwf reg16

```

Using indirect addressing, this can be done as follows:

```

movlw 0x10
movwf count
movlw 0x1B
movwf FSR
loop movf count,0
movwf INDF
decf FSR
decfsz count
goto loop
.....
end

```

An operation performed on register `INDF` performs the operation on the register *pointed to* by register `FSR`. Register `FSR` contains the address of the register on which the operation is to be performed. Performing the operation on `INDF`, therefore performs the operation on the desired register via the pointer `FSR`. In other words, `INDF` addresses the register pointed to by `FSR`.

The code works as follows. The number 16 is loaded into `count`. The address of register 16 is loaded into `FSR`. Next, the value of `count` is moved into the register whose address is contained in `FSR`, by moving it into `INDF`.

### 3.5.4 Changing bit values in registers

It is often necessary to change the value of individual bits in a register. Changing a single bit is easily done using either `bcf` or `bsf`. However, when two or more bits need to be changed, it becomes inefficient to make repeated use of `bcf` or `bsf`. A better method is to use logical operators such as AND, OR and Exclusive-OR.

The value of a bit will not be changed if it is used in an AND operation with a 1 or in an OR operation with a 0. To change a 1 to a 0, perform an AND with a 0 and to change a 0 to a 1, perform an OR with a 1.

The PIC instruction set contains the instructions `andlw`, `andwf`, `iorlw`, `iorwf`, `xorlw`, `xorwf`, `comf`, `rrf` and `rlf` for logical and shifting operations. The use of some of these will be illustrated using the following fragments of code.

#### Example 1

Given the byte 11010110 in register `modereg`, change bits 2, 4 and 6 to zeros, leaving the remaining bits unchanged. This can be done using `andwf` as follows:

```
movlw B'10101011'  
andwf modereg,1 ; (W AND modereg) into modereg.
```

#### Example 2

Change bits 2,3 and 4 of the byte 11100001 in register `reggie` to ones, leaving the remaining bits unchanged. Using the inclusive OR function `iorwf` gives:

```
movlw B'00011100'  
iorwf reggie,1 ; (W OR reggie) into reggie.
```

To determine whether two bytes are the same, they can be subtracted and the zero bit of the `STATUS` register tested to see if it is set as was done in previous code examples. Alternatively, an Exclusive-OR operation can be performed and the `STATUS` bit tested using either `btfsX` or `btfss` as follows:

```
movlw B'10101101'  
xorlw B'10101101'  
btfsX STATUS,2 ; X could be 's' or 'c'.
```

When the numbers are the same, the result of the Exclusive-OR operation will be zero.

### 3.5.5 Bit rotations

The instructions *rotate left through carry* (`rlf`) and *rotate right through carry* (`rrf`) are used when a byte needs to be shifted left or right by one bit position through the carry bit. A typical application would be when a byte must be transmitted serially, one bit at a time out of a port pin.

Because the ALU carry bit (`STATUS<0>`) is affected by the shifting operation, it should be initialised to either 0 or 1.

### Example 3

Consider the following program that repeatedly right shifts the byte 11010101 in register `reg` and stores the result back in the register. The carry bit has been initialised to 1 in this example.

```

;--Program "rotate.asm". A simple program to rotate a byte.
;--Author: Ethel Skonzblik.
;--Date: 7 July 1997.
;--Version: 1.06

LIST    p=PIC16C84, r=hex, f=INHX8M

        include "P16C84.inc"
reg equ 0xC
count equ 0xD
        org 0x0

        clrf count
        movlw B'11010101'
        movwf reg
        bsf STATUS,0    ; Set the carry flag.
loop    rrf reg,1
        incf count
        goto loop
        end

```

The output from this program is given in Figure 3.6. The values in the last column are the

Step number	Carry Bit	Register Value
0	1	11010101
1	1	11101010
2	0	11110101
3	1	01111010
4	0	10111101
5	1	01011110
6	0	10101111
7	1	01010111
8	1	10101011
9	1	11010101

Figure 3.6: Sequential right rotation of a byte.

register values *after* the rotate right instruction has executed. After nine shift operations, the

original byte results. The carry bit appears as the MSB of the value in the next row. The original byte also appears as the sequence of carry bit values from step 8 (MSB) to step 1 (LSB).

If the carry bit was initialised to zero, the original byte would appear after the ninth shift operation as before, but the intermediate register values would be different. This can be easily verified by simulating the program with watch windows for `reg`, `STATUS` and `count`. Similar results are obtained by using the rotate left operator `rlf` instead of `rrf`.

## 3.6 Project 3

In this project, the use of simple interrupts will be discussed using a modification of previous programs.

### 3.6.1 Interrupts

An *interrupt* is an unscheduled input, which interrupts the normal flow of a program, causing it to temporarily suspend what it is doing and perform a specified sequence of instructions required by the interrupting agent before returning to its original point in the program. This behaviour is similar to a subroutine call, except that the program jump is caused by an event *external* to the processor. An example of an interrupt in a computer system, is the signal from a printer informing the CPU that it has finished printing.

Unlike the polling of an input, an interrupt can occur at any point in the program and, because its occurrence is unpredictable, its arrival cannot simply be catered for by special programming. To handle interrupts, a combination of hardware and software must be specifically designed into the architecture of the processor.

An interrupt is a signal originating in a *peripheral*. In PIC microcontrollers, these peripherals need not necessarily be external devices, although they usually are, but they are *peripheral to the main processing logic*. For example, the PIC's on-board counter/timer can generate an interrupt signal, because, although it is part of the PIC's circuitry, it is considered to be external to the processing part.

When a subroutine is called, the return address must be pushed onto the stack. A subroutine call is simple to handle because the jump occurs in a well defined position in the program. However, because an interrupt is unscheduled and can therefore occur at any point in the program, more than just the return address needs to be stored. For example, the status of the registers (including those for ports) needs to be stored, so that when the interrupt has been serviced, the program can commence with the system in the same state as it was before the interrupt occurred.

### 3.6.2 The PIC16F84 interrupt system

The PIC16X8X can handle four main interrupt sources

1. A rising or falling voltage pulse on pin `RBO/INT`

2. A change in one or more of the voltage levels on the group of pins RB4 to RB7
3. An overflow of the timer register TMR0 from 0xFF to 0x0
4. A ‘write complete’ signal generated when writing to the EEPROM is complete

The interrupt logic circuit diagram is shown in Figure 3.7.

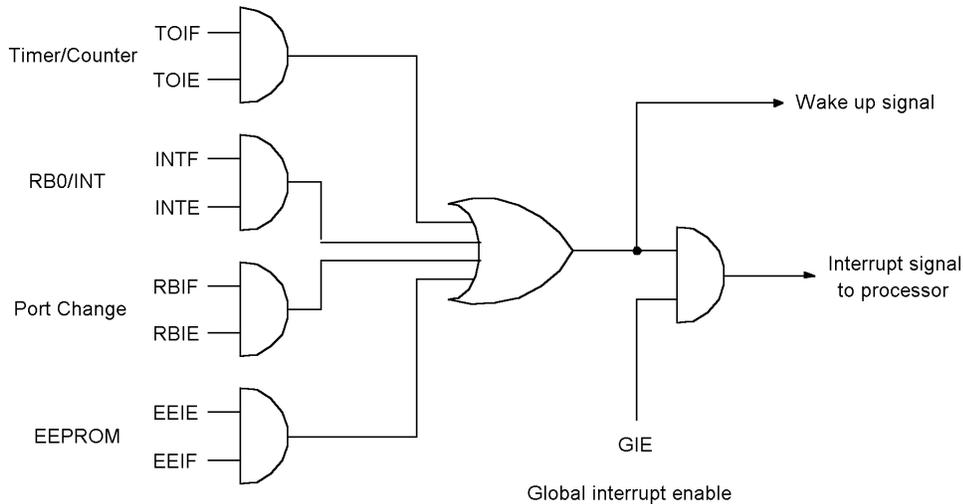


Figure 3.7: Logic circuit diagram of PIC16F84 interrupt system (*Courtesy of Microchip Technology Inc.*).

There are four *interrupt flags*, namely TOIF, INTF, RBIF and EEIF which are bits in the INTCON register and are set whenever that particular interrupt triggers, irrespective of whether it has been enabled by setting its corresponding enable flag. The *interrupt enable* flags are TOIE, INTE, RBIE and EEIE. From the diagram, it is evident that in order for any of the interrupt signals to have any effect, it is necessary to set the global interrupt bit, GIE, to enable any signal from the OR gate to pass through the rightmost AND gate.

When an interrupt occurs, the processor immediately pushes the return address onto the stack, branches to the *interrupt vector* at address 0x4 in program memory and starts executing the code at that address. The sequence of instructions executed is called the *interrupt service routine*.

As there is more than one possible interrupt source, one of the duties of the service routine is to determine which interrupts occurred and what their individual priorities are. Interrupts with the highest priority are serviced first. It is important that the interrupt flag be cleared by the program otherwise the `retfie` instruction will not operate correctly.

The source code listing for a simple interrupt system is given in program `Intp.asm`. The program is a modified version of program `Flash4.asm` which could, for example, form the basis of a simple burglar alarm. The PIC flashes the LED (the ‘alarm armed’ indicator) until the interrupt pin

(RB0) experiences a signal with a rising edge, caused by some alarm condition. The LED stops flashing and pin RB6 goes high, setting off some audible alarm annunciator which remains on until the 'alarm reset' pin (RA3) is momentarily pulled high by a pushbutton. After the reset signal, normal operation commences with the LED flashing as before.

```

;--Program "Intp.asm" to demonstrate simple interrupt operation.
;--Author: Nugent Zakatak
;--Date: 4 August 1997.
;--Version: 1.07

LIST p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
delreg equ 0xC ; Set aside space for the registers.
count equ 0xD

org 0x00 ; Reset vector.
goto start

org 0x04 ; Interrupt vector.
intpt bcf INTCON,INTF
      bsf PORTB,6 ; Pull the alarm line high.
;--Poll input pin RA3 to determine when alarm clear signal occurs.
read movf PORTA,1 ; Read the contents of PORTA into itself.
      btfs PORTA,3 ; Test bit 3, skip next instruction if set.
      goto read ; if bit 3 is not set.
;--Now pull the alarm line low again.
      bcf PORTB,6
      retfie ; Return from interrupt and continue as normal.

;--Set up the interrupt system initially.
start movlw B'10010000' ; Set GIE and INTE bits in INTCON register.
      movwf INTCON
;--Set up the interrupt for rising edge triggering.
      movlw B'01000000'
      movwf OPTION_REG
;--Set up RB6 and RB7 as outputs and RB0 an input for the interrupt.
      bsf STATUS,RP0 ; Select Bank 1.
      movlw B'00111111'
      movwf TRISB
      bsf TRISA,3 ; Make RA3 an input.
      bcf STATUS, RP0 ; Go back to Bank 0.
      clrf PORTA ; Clear the PORTA and PORTB registers to enforce
      clrf PORTB ; desired power up conditions.

;--Now begin the main program loop.

```

```

begin    bsf PORTB,7      ; Pull pin RB7 high.
         call delay
         bcf PORTB,7      ; Pull pin RB7 down.
         call delay
         goto begin
;--End of main program body.

;--Subroutine to provide a delay of about a second.
delay    movlw 0xFF
         movwf delreg     ; Fill the register.
         movlw 0xC
         movwf count      ; Load 12 into count.
loop2    decfsz count,1
         goto loop1
         goto exit
loop1    decfsz delreg,1
         goto loop1
         goto loop2
exit     return
;--End of the subroutine delay.
         end              ; of program "Intp.asm".

```

When the system is switched on, the program begins at the reset vector address 0x0 which specifies a branch to label `start`. The branch is needed as a bypass in order to avoid the program simply running into the interrupt routine and executing it without an interrupt actually occurring.

Firstly the global interrupt enable bit `GIE` and the interrupt bit for the interrupt pin `RB0`, must be set. The processor needs to be told whether to trigger on a rising or a falling voltage edge on pin `RB0`, by setting or clearing bit `INTEDG` in the `OPTION` register.

The pins `RA3`, `RB0`, `RB6` and `RB7` need to be configured as input or outputs as before. The main program loop for flashing the LED is the same as that in program `Flash4.asm`. The starting point for the interrupt routine originates at label `intpt` as defined by the interrupt vector address 0x4. The first action of the interrupt routine is to clear the interrupt flag `INTF` to prevent perpetuating the interrupt request while it is being serviced. Next, the alarm pin `RB6` is pulled high and a loop is initiated which polls the voltage status of pin `RA3`. When `RA3` is pulled high, the alarm pin goes low, indicating that the alarm condition has been cleared. Finally, the `retfie` instruction loads the return address into the program counter, sets the `GIE` bit and the program continues executing the LED flashing routine where it left off.

Note that the `OPTION` register is referred to as `OPTION_REG` as it is defined by this name in the include file to distinguish it from an obsolete PIC instruction also named `OPTION`.

### 3.6.3 Simulating program `Intp.asm`

Assemble and reset the program ready for simulation. Open watch windows for the variables `INTCON`, `PORTA` and `PORTB` and set up the pin stimulus with the following stimulus file:

Step	RB0	RA3
17050	1	0
17076	0	1
17077	0	1
17078	0	1
17079	0	1
17080	0	1

Simulate the program by using the **Step over** option until just before the first delay is due to repeat (the second iteration of the loop at the label **begin**), then single step using the **Step** option. Note how the program jumps to the label **intpt** at cycle 17050. Watch the **GIE** and **INTF** bits get cleared and pin **RB6** go high.

Continue stepping and observe that beginning at cycle 17076, the pin **RA3** goes high and pin **RB6** goes low. Finally watch instruction **retfie** set **GIE** high again and jump back to the position in the **delay** subroutine where it was before the interruption took place.

Program **Intp.asm** is a simple demonstration of the use of the interrupt **RB0/INT**. For details on using the ‘port interrupt on change’ and the remaining interrupt source options, the reader is urged to consult the detailed application notes (reference [4]).

### 3.6.4 Context saving during interrupts

When an interrupt occurs, the return address is automatically pushed onto the stack and the program branches to execute the interrupt service routine.

Sometimes the contents of other registers such as the **W** and **STATUS** registers need to be saved before executing the interrupt service routine. In such cases, the *programmer* must ensure that the appropriate measures are taken during program development.

Saving the contents of the **W** register and other user-defined registers is accomplished by simply using the usual **movf** and **movwf** instructions. In the case of the **STATUS** register, however, the **movf** instruction cannot be used as it has the effect of possibly altering the zero flag of the **STATUS** register.

To save the contents of the **STATUS** register successfully, the **swapf** instruction is used. This instruction can be used to move the contents of a register without affecting any **STATUS** bits, but it has the side-effect that the upper and lower nibbles (four bit blocks) are interchanged when the contents are moved to the destination register. This side-effect is harmless, provided that another **swapf** instruction is invoked to cancel the first nibble swap when the original program context is restored after the completion of the interrupt.

An example code fragment of a typical context save is as follows:

```
save    movwf wsave          ; Save the W register first.
        swapf STATUS, 0      ; Swap STATUS into W.
        movwf statussave     ; Save STATUS.
        .....               ; Place interrupt service routine here.
restore swapf statussave,0
```

```

movwf STATUS
swapf wsave,1      ; Swap and store in wsave.
swapf wsave,0      ; Swap and store in W.

```

### 3.7 Using the timer

One way of using the PIC to measure time intervals or count external pulses, is to write a simple program using the interrupt pin RB0/INT or to poll an ordinary input pin. The disadvantage of this is that the PIC is then unable to do anything else. To free the processor from total dedication to this task, the timer/counter module can be used.

Unlike devices such as the PIC16C92X which have fully asynchronous timer/counter modules which operate during sleep and independently of the system clock, those of the PIC16X8X range operate in conjunction with the system clock and are disabled during sleep. Consequently, the PIC16X8X timer/counter cannot be used as an interrupt to wake the processor from sleep.

The timer/counter module can operate either in *counter mode* or in *timer mode*. In timer mode, the TMR0 register is incremented via the internal system clock, whereas in the counter mode, it is incremented via an external signal on pin TOCKI. The TMR0 register can be written to or read from, so that the timing period or count value can be adjusted in the software.

In order to achieve flexible counting and timing, an eight bit programmable divider is available, allowing its input signal frequency to be divided by one of eight binary values between 2 and 256. This divider can be used either with the timer/counter module or the watchdog timer. When selected for use with the timer/counter module, the divider is called the *prescaler* as it is located *before* the TMR0 register. When selected for use with the watchdog timer, the divider is called the *postscaler* as it is located *after* the watchdog timer module.

All operations that write to the timer register TMR0 such as `clrf`, `movwf`, `bcf` and `bsf` will automatically clear the prescaler.

In the counter mode, the external signal from the TOCKI pin is passed to a sampling circuit either directly, or via the prescaler, to form a trigger waveform. The sampling circuit samples the trigger waveform every second period of the internal clock signal in order to determine whether a rising or falling edge has occurred. If the trigger waveform is high for at least two consecutive samples and then low for at least two consecutive samples, the TMR0 register is incremented. For this reason, the minimum period of the external signal is

$$T_{min} > t_i + 40\text{nsec} \quad (3.6)$$

where  $t_i = 4/f_0$  and the additional 40nsec delay is required for the logic circuitry to settle. This gives a maximum frequency of

$$f_{max} < \frac{f_0 PS}{4} = \frac{PS}{t_i} \quad (3.7)$$

where  $PS$  is the prescaler division ratio (between 2 and 256).

After sampling, the information is processed, resulting in a total delay from the input signal transition to when the TMR0 register is incremented, of up to two instruction cycles.

### 3.7.1 Setting up the timer/counter module

The first decision to be made is whether to use timer or counter mode. Next, if the input frequency to the module needs to be scaled down by the divider, the prescaler must be selected and the division ratio chosen. In such a situation, the watchdog timer will not be able to use the postscaler. Finally, if the counter mode is chosen, a decision to trigger on the rising edge or the falling edge of the input waveform must be taken.

The first six bits of the OPTION register are used to configure the module. To select counter mode, the TOCS (clock source) bit OPTION<5> must be set and to select timer mode, it must be cleared.

To assign the prescaler to the counter/timer module, bit PSA (OPTION<3>) must be cleared. The division ratio is set by the binary value of the bits OPTION<2:0> (PS2, PS1 and PS0). A value of 000 will give a 2:1 ratio, a value of 001 a 4:1 ratio and so on, up to 111 which gives a ratio of 256:1.

If the prescaler is not used, then it must be assigned to the watchdog timer by setting the PSA bit. This step is necessary even if the watchdog timer is permanently disabled when programming the device. The following code is recommended:

```
bcf STATUS,RP0    ; Select Bank0.
clrf TMRO         ; Clear TMRO register and prescaler.
bsf STATUS,RP0    ; Select Bank1.
clrwdt           ; Clear watchdog timer.
movlw B'XXXX1XXX' ; (User to choose the X values.)
movwf OPTION_REG ; Load the OPTION register
bcf STATUS, RP0   ; Select Bank0 again.
```

Program Timer1.asm listed below, illustrates the operation of the counter/timer module using the internal clock signal as a source for incrementing the TMRO register.

```
!--Program "Timer1.asm". A program using the counter/timer module.
!--Author: Humbert Snetherswaite.
!--Date: 30th August 1997.
!--Version: 1.08
```

```
LIST    p=PIC16C84, r=hex, f=INHX8M

include "P16C84.inc"
org 0x0

clrf TMRO          ; Clear counter/timer register and prescaler.
bsf STATUS,RP0    ; Select Bank 1.
movlw B'01010000' ; Set up the timer/counter parameters.
movwf OPTION_REG  ; Load the OPTION register.
bcf STATUS,RP0    ; Select Bank 0.
nop
```

```

        nop
        nop
        nop
        nop
wait    goto wait
        end                ; End of program Timer1.asm.

```

Bits 0, 1 and 2 of the `OPTION` register are set to zero to give a prescaler division ratio of 2:1. Bit 3 is cleared to assign the prescaler to the counter/timer module. The value of bit 4 is irrelevant in this case, because pin `T0CKI` is not being used by the module. Bit 5 is cleared so that the internal clock is used and bit 6 is set so that the register `TMR0` increments on the rising edge of the clock signal. The value of bit 7 is also not relevant and can be either cleared or set.

The `nop` instruction is used here merely for demonstrative purposes so that the program can be stepped through one instruction at a time. This instruction is often used to introduce a one instruction cycle delay into a program where required.

Assemble the program and set up the MPLAB simulator. Open a watch window for the `TMR0` register and set the stopwatch to zero. Step through the program using the simulator `Step` option and observe how the `TMR0` register increments every second instruction. At the label `wait`, the register *appears* to increment every step, but as can be seen from the stopwatch cycle counter, it also increments every second step. The reason for this behaviour is that the `goto` instruction is a two cycle instruction, so every time it is executed, the register increments.

To change the prescaler setting to a division ratio of 256:1, the `OPTION` register must be loaded with the value 01010111. This can be accomplished by changing the value moved in by the `movlw` instruction and then reassembling the program. Alternatively, the value loaded into the `OPTION` register can be changed *without reassembling the program*, by using the simulator option `Execute an Opcode`.

Single step through the program until *after* the instruction `movlw` has executed, then click on the `Execute an Opcode` toolbar button. In the `Opcode` window, type the instruction `movlw B'01010111'`, then click on `Execute` followed by `Close`. Continue stepping beyond the `movwf` instruction until the `wait` label is reached, then reset the stopwatch to zero. Verify that the `TMR0` register only increments after 256 steps.

### 3.7.2 The timer/counter interrupt

To enable the timer interrupt, bit `INTCON<5>` (`TOIE`) must be set by adding the line `bsf INTCON, 5` before the first `nop` instruction. Open a watch window to view the `INTCON` register and step through the program to verify that bit `TOIE` of the `INTCON` register gets set.

Continue stepping through the program until the label `wait` is reached. Click on the `Execute an Opcode` tool button and execute the two instructions `movlw 0xFF` and `movwf TMR0` to load the value `0xFF` into the `TMR0` register. Set the stopwatch to zero and step for 256 cycles until the `TMR0` register overflows to zero. Verify that the counter/timer interrupt flag `TOIF` (`INTCON<2>`) gets set as `TMR0` overflows.

A modification of the program for operation with interrupts is as follows:

```

;--Program "Timer2.asm". Demonstration of counter/timer interrupt generation.
;--Author: Major Dennis Bloodnok.
;--Date: 30th August 1997.
;--Version: 1.09

```

```

LIST    p=PIC16C84, r=hex, f=INHX8M

        include "P16C84.inc"
        org 0x0
        goto start
        org 0x4
intpt   bcf INTCON, 5      ; Clear the interrupts flag.
        nop               ; Put in some useful code here.
        retfie

start   clrf TMRO
        bsf STATUS,RP0    ; Select Bank 1
        movlw B'01010000'
        movwf OPTION_REG
        bcf STATUS,RP0
        movlw B'10100000' ; Select prescaler settings (bits 0 to 2).
        movwf INTCON
        nop               ; Put in some useful code here.
wait    goto wait
        end

```

To set up the interrupt system, the global interrupt enable bit GIE (INTCON<7>) must be set. To enable the counter/timer interrupt, the TOIE bit (INTCON<5>) must be set. The interrupt flag must be cleared in the interrupt service routine otherwise the `retfie` instruction will not work.

Step through the program and observe how the program jumps to the interrupt routine when TMRO overflows. To avoid stepping through 256 cycles, the program can be verified by stepping to label `wait` then executing the instructions `movlw 0xFE` and `movwf TMRO` using the `Execute an Opcode` option and then stepping until TMRO overflows as previously discussed.

### 3.7.3 Counting external pulses

To investigate the counting of pulses applied to the TOCKI pin, consider the following program.

```

;--Program "Timer3.asm". Counting external pulses via the TOCKI pin.
;--Author: Zebulon Humdelay.
;--Date: 1st September 1997.
;--Version: 1.10

```

```

LIST    p=PIC16C84, r=hex, f=INHX8M

```

```

include "P16C84.inc"
org 0x0

clrf TMR0
bsf STATUS,RP0      ; Select Bank 1
movlw B'01110000'   ; Set up counter/timer parameters.
movwf OPTION_REG
bcf STATUS,RP0      ; Select Bank 0.
wait goto wait
end                  ; End of program.

```

The timer/counter module is set up to increment on the rising edge (bit `OPTION<4>` set) of the `TOCKI` pin voltage, with a prescaler division ratio of 2:1 (bits `OPTION<2:0>` cleared).

To simulate the operation of this code fragment, use a stimulus file with the voltage of pin `RA4/TOCKI` having five cycles at logical 1 followed by five at logical 0 as shown below.

Step	RA4
5	0
6	0
7	0
8	0
9	0
10	1
11	1
12	1
13	1
14	1
15	0
.....	

The voltage on pin `TOCKI` has a rising edge every ten cycles and because the prescaler division ratio is 2:1, the `TMR0` register is incremented every 20 cycles as displayed on the stopwatch. Step through the program and verify the operation of the program.

As with the previous counter/timer examples, this program can be modified to use interrupts with various prescaler division ratios.

### 3.7.4 Common errors and pitfalls

It is almost certain that the beginner will encounter some ‘teething problems’ with the projects described and there are a few common errors that the beginner is bound to make. However, with experience and some attention to detail, the probability of getting the next project to work ‘first time’ will increase. This section will list some of the errors and problems that are likely to be encountered when first starting out.

- The circuit, including the oscillator, does not work at all. Check that the master-reset-clear line  $\overline{\text{MCLR}}$  (pin 4), is neither disconnected nor held low. Verify that the voltage is at the positive supply rail as an unsuspected open circuit may be present. Alternatively, check that the PIC has not been put to sleep with a `sleep` instruction.
- The circuit draws much more current than expected, even when it is placed in sleep mode. With a CMOS device, all unused pins configured as inputs must be connected to one of the supply rails. Output pins should be sourcing or sinking current.
- The oscillator does not work, but when an external clock signal is applied (in LP, XT or HS modes) the circuit works correctly. Make sure that the crystal is physically close to the IC and that the oscillation capacitors are the correct value. Higher value capacitors than recommended in the data sheet will improve the oscillator's stability but will increase the start-up time. Other possibilities are that the wrong oscillator configuration was selected when the program was loaded into the ROM, or the crystal being used is too fast for the device. For example, a device with an identification number PIC16F84-04/P will only operate up to a maximum frequency of 4MHz.
- A register does not seem to respond to an instruction. This is often caused by the program trying to access a register in a memory bank other than the one currently selected. Referring to Figure 2.4, check that the correct bank is selected prior to the use of the instruction. Some registers such as `STATUS`, `INTCON` and `PCL` can be accessed from any bank as their memory addresses are mapped in both banks.
- A pin does not go high or low as desired. Check that the pins are correctly configured as either input or outputs by checking that the tristate direction register `TRISA` or `TRISB` is set up correctly. If a pin is configured as an input, then a command such as `bcf` or `movlw` will change the contents of the corresponding port register but not the pin itself. Alternatively trying to pull a pin high or low using an external voltage may damage the device if the pin is configured as an output.

Remember that operations that write to a port, first read the contents of the port register, then write the new values to it. If a port register is configured so that some of the pins are inputs and others are outputs, then when a write operation occurs, a voltage on the input pin will be read into the port register overwriting its previous value. This behaviour will not cause any problems as long as the input pin stays an input pin. However, if the pin is later changed to an output pin, the value written out will be that of the previous pin's input.

- The program operates correctly when simulated, but behaves in an unpredictable manner in the circuit. If the watchdog timer is not disabled when setting up the configuration and loading the program into the PIC, then it will automatically time-out after 18msec (without the postscaler) and cause a system reset. To prevent this from happening, either the watchdog timer should be permanently disabled when programming the device, or else a `clrwdt` instruction must be executed in the program before time-out occurs. If the timer/counter module is not using the prescaler, and it is difficult to repeatedly issue the `clrwdt` instruction every 18msec, then the timeout can be lengthened by using the postscaler and clearing the watchdog timer less frequently.

- When reading values into the `PORTB` register via high impedance sources, incorrect values are read in. Port B has weak internal pullups that could pull the pins into a logical high state. The pullups can be disabled by setting bit  $\overline{\text{RBPU}}$  (`OPTION<7>`).
- When simulating, the pin stimulus option does not work. Check that the stimulus file is correct, that it has been enabled in the `Debug` menu and that the step numbers are within range of that displayed on the stopwatch. Always reset the stopwatch to zero when resetting the system.
- The wrong file gets assembled. In the `Edit Project` menu, check that the desired file is added to the project using the `Add` option and that all other irrelevant files are removed using `Remove`.
- Strange behaviour occurs during simulation. Check that the correct PIC processor has been selected and ensure that the correct include file is used.

### 3.8 Some PIC project ideas

The PIC range of microcontrollers are high performance, low cost devices that are so easy to use that the applications they can be used for is limited only by the designer's imagination. Here is a short list of some ideas:

- Square or rectangular waveform generators. Use the instructions `bsf` followed by `bcf` in an iterative fashion with a delay in between, to produce waveforms on a port pin with various periods and duty cycles. A square wave of period  $2t_i$  will be produced on pin `RA0` if the sequence `bsf PORTA,RA0 bcf PORTA,RA0 bsf PORTA,RA0 . . .` is used with no other instructions in between.
- A programmable delay can be produced by reading a byte from a port and then using the value to set the delay time.
- A model train controller or digital event sequencer for toys.
- A controller for an elevator of a building.
- A sequencer to switch appliances and lights on or off in a random sequence at different times to give the impression that a home is occupied.
- An automatic security gate controller to open and close gates.
- An irrigation controller to water the garden under computer control. Temperature and humidity sensors can be used to trigger a PIC-based circuit to control the opening and closing of valves in the water line.
- A frequency counter. Count pulses on a pin using either the timer/counter module or via the interrupt pin `RBO/INT`
- A simple musical instrument. Use resistors and keyboard switches on a port to set the value of a byte read into the port to change the frequency of the note produced by a waveform generator.

- A sinusoidal and triangular wave signal generator can be constructed by writing the binary value of the waveform at each sample to a digital to analogue converter. For slow waveforms, these values can be calculated as described in the application notes [4]. Walsh functions can also be used to create waveforms.
- A burglar alarm can be developed that reads the status of sensors and activates an alarm. Time delays and other ‘intelligence’ can be incorporated.
- A servo motor or stepper motor controller using a PIC and power driver ICs can be used to produce quarter, half and full step drive waveform configurations for stepper motors or pulse width modulated signals for DC servo motors.
- A digitally controlled power supply unit (PSU). Thumbwheel switches can be used to set up the binary value of the required power supply voltage which the PIC can then use to produce a PWM signal for a switched mode PSU, or a processed binary value for a digital to analogue converter for a linear PSU. Interrupt and pin polling can be used to perform sensing and supervisory functions.
- A digital filter can be built using a PIC16C71 or similar PIC device that has an analogue to digital converter. See the application notes [4] for design details and program listings on FFT (fast Fourier transform) analysis and discrete filtering applications.
- An access control system using the PORTB ‘interrupt on change’ facility with a keypad code can be developed.
- A digital thermometer with LCD display for use in fish tanks, rooms, brewing, photography etc. can be produced. A DS1820 from Dallas Semiconductor Corp. or similar digital temperature transducer and a Microchip AY0438 CMOS LCD display driver can be used. Alternatively, the PIC16C923 or PIC16C924 microcontroller can be used to perform all the functions required to process the transducer signal and produce the LCD driver signals.
- A pulse width modulator to control power delivered to a load via thyristors. Typical applications are light dimmers and motors.
- A digital phase locked loop (PLL) can be developed along the lines of a 4046 CMOS PLL.
- A programmable battery management system.
- A programmable timer with timing intervals from seconds to hours. The PIC16F84 has many registers enabling long time delays to be implemented.
- A Flash/PROM or EEPROM programmer for PIC and other microcontrollers.
- An electronic guitar tuner.
- A serial port controller for PCs.
- A logic analyser.
- A heart rate monitor.
- A radio controlled vehicle controller.
- An infra red telemetry system which sends and receives bit streams via an infra red beam.

# References

- [1] Gothmann W. H., “Digital Electronics - An introduction to Theory and Practice.”  
Prentice-Hall, Englewood Cliffs, NJ, USA, 1977.
- [2] Carter J. W., “Microprocessor Architecture and Microprogramming - A State  
Machine Approach.”  
Prentice-Hall, Englewood Cliffs, NJ, 1996.
- [3] Microchip PIC16/17 Microcontroller Data Book.  
Microchip Technology Inc., 1996.
- [4] Microchip Embedded Control Handbook 1994/95  
Microchip Technology Inc., 1994.
- [5] Microchip PICStart Design Contest Application Brief Notebook.  
Microchip Technology Inc., 1993.
- [6] Wilkinson B. and Makki R., “Digital System Design.”  
Prentice Hall, International (UK) Ltd., 1992.