MICROCONTROLLERS

HIGH-PERFORMANCE SYSTEMS AND PROGRAMMING

Julio Sanchez
Maria P. Canton

CRC Press
Taylor & Francis Group

MICROCONTROLLERS

HIGH-PERFORMANCE SYSTEMS AND PROGRAMMING

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

MICROCONTROLLERS

HIGH-PERFORMANCE SYSTEMS AND PROGRAMMING

Julio Sanchez
Eastern Florida State College

Maria P. Canton
Brevard Public Schools

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20130923

International Standard Book Number-13: 978-1-4665-6665-1 (Hardback)

Library of Congress Cataloging-in-Publication Data

Sanchez, Julio, 1938-
Microcontrollers : high-performance systems and programming / Julio Sanchez, Maria P. Canton.
pages cm
Includes bibliographical references and index.
ISBN 978-1-4665-6665-1 (hardback)
1. Microcontrollers. 2. Microcontrollers--Programming. 3. Programmable controllers I. Canton, Maria P. II. Title.

TJ223.P76S362 2013
629.8'95--dc23 2013036871

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Table of Contents

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Preface

Microcontrollers: High-Performance Systems and Programming can be considered a continuation of and a complement to our previous two titles on the subject of microcontroller programming. In the present book we focus on the line of high-perforance microcontrollers offered by Microchip. In addition to their enhanced features, extended peripherals, and improved performance, there are several practical factors that make the high-performance PIC series a better choice than their mid-range predecessors for most systems:

- The possibility of programming high-performance microcontrollers in a high-level language (C language).
- Source code compatibility with PIC16 microcontrollers, which facilitates code migration from mid-range to PIC18 devices.
- Pin compatibility of some PIC18 devices with their PIC16 predecessors. This makes possible the reuse of PIC16 controllers in circuits originally designed for mid-range hardware. For example, the PIC18F442 and PIC18F452 in 40-pin DIP configuration are pin-compatible with the popular PIC16F877. Similarly, the PIC18F242 and PIC18F252, in 28-pin DIP format, are pin compatible with the PIC16F684.
- Microchip pricing policy makes available the high-performance chips at a lower cost than their mid-range equivalents. Recently we have priced the 18F452 at $6.32 while the 16F877 sells from the same source at $6.72.

Expanded functionality, high-level programmability, architectural improvements that simplify hardware implementation, code and pin-layout compatibility, and lower cost make it easy to select a high-performance PIC over its mid-range counterpart. One consideration that is sometimes mentioned in favor of the mid-range devices is the abundance of published application circuits and code samples. Our book attempts to correct this. Although it should also be mentioned that some PIC16 processors with small footprints have no PIC18 equivalent, which explains why some mid-range devices continue to hold a share of the microcontroller marketplace.

Like our preceding titles in this field, the book is intended as a reference and resource for engineers, scientists, and electronics enthusiasts. The book focuses on the needs of the working professional in the fields of electrical, electronic, com-

puter, and software engineering. In developing the material for this book, we have adopted the following rules:

1. The use of standard or off-the-shelf components such as input/output devices, integrated circuits, motors, and programmable microcontrollers, which readers can easily duplicate in their own circuits.
2. The use of inexpensive or freely available development tools for the design and prototyping of embedded systems, such as electronic design programs, programming languages and environments, and software utilities for creating printed circuit boards.
3. Our sample circuits and programs are not copyrighted or patented so that readers can freely use them in their own applications.

Our book is designed to be functional and hands-on. The resources furnished to the reader include sample circuits with their corresponding programs. The circuits are depicted and labeled clearly, in a way that is easy to follow and reuse. Each circuit includes a parts list of the resources and components required for its fabrication. For the most important circuits, we also provide tested PCB files. The sample programs are matched to the individual circuits but general programming techniques are also discussed in the text. There are appendices with useful information and the book's online software contains a listing of all the sample programs developed in the text.

Julio Sanchez

Maria P. Canton

Chapter 1

Microcontrollers for Embedded Systems

1.1 Embedded Systems

An embedded system is a computer with specific control functions. It can be part of a larger computer system or a stand-alone device. Most embedded systems must operate within real-time constraints. Embedded systems contain programmable processors that are either microcontrollers or digital signal processors (DSPs). The embedded system is sometimes a general-purpose device, but more often it is used in specialized applications such as washing machines, telephones, microwave ovens, automobiles, and many different types of weapons and military hardware.

A microcontroller or DSP usually includes a central processor, input and output ports, memory for program and data storage, an internal clock, and one or more peripheral devices such as timers, counters, analog-to-digital converters, serial communication facilities, and watchdog circuits. More than two dozen companies in the United States and abroad manufacture and market microcontrollers. Mostly they range from 8- to 32-bit devices. Those at the low end are intended for very simple circuits and provide limited functions and program space, while the ones at the high end have many of the features associated with microprocessors. The most popular microcontrollers include several from Intel (such as the 8051), from Zilog (derivatives of their famous Z-80 microprocessor) from Motorola (such as the 68HC05), from Atmel (the AVR), the Parallax (the BASIC Stamp), and many from Microchip. Some of the high-end Microchip microcontrollers and DSPs are the topic of this book.

1.2 Microchip PIC

The names PIC and PICmicro are trademarks of Microchip Technology. Microchip prefers the latter designation because PIC is a registered trademark in some European countries. It is usually assumed that PIC stands for Peripheral Interface Controller, although the original acronym was Programmable Interface Controller. More recently, Microchip has stated that PIC stands for Programmable Intelligent Computer, a much nicer, albeit not historically true version of the acronym.

The original PIC was built to complement a General Instruments 16-bit CPU designated the CP-1600. The first 8-bit PIC was developed in 1975 to improve the performance of the CP-1600 by offloading I/O tasks from the CPU. In 1985, General Instrument spun off its microelectronics division. At that time, the PIC was re-designed with internal EPROM to produce a programmable controller. Today, hundreds of versions and variations of PIC microcontrollers are available from Microchip. Typical on-board peripherals include input and output ports, serial communication modules, UARTs, and motor control devices. Program memory ranges from 256 words to 64k words and more. The word size varies from 12 to 14 or 16 bits, depending on the specific PIC family.

1.2.1 PIC Architecture

PIC microcontrollers contain an instructions set that varies in length from 35 instructions for the low-end devices to more than 70 for the high end. The accumulator, which is known as the work register in PIC documentation, is part of many instructions because the low- and mid-range PICs contain no other internal registers accessible to the programmer. The PICs are programmable in their native Assembly Language. C language and BASIC compilers have also been developed. Open-source Pascal, JAL, and Forth compilers are also available, although not very popular.

It is often mentioned that one of the reasons for the success of the PIC is the support provided by Microchip. This support includes development software, such as a professional-quality development environment called MPLAB, which can be downloaded free from the company's website (www.microchip.com). The MPLAB package includes an assembler, a linker, a debugger, and a simulator. Microchip also sells an in-circuit debugger called MPLAB ICD 2. Other development products intended for the professional market are also available from Microchip.

In addition to the development software, the Microchip website contains a multitude of free support documents, including data sheets, application notes, and sample code. Furthermore, the PIC microcontrollers have gained the support of many hobbyists, enthusiasts, and entrepreneurs who develop code and support products and publish their results on the Internet. This community of PIC users is a treasure trove of information and know-how easily accessible to the beginner and useful even to the professional. One such Internet resource is an open-source collection of PIC tools named GPUTILS, which is distributed under the GNU General Public License. GPUTILS includes an assembler and a linker. The software works on Linux, Mac OS, OS/2, and Windows. Another product, called GPSIM™, is an open source simulator featuring PIC hardware modules.

1.2.2 Programming the PIC

Stand-alone programming a PIC microcontroller requires the following tools and components:

- An Assembler or high-level language compiler. The software package usually includes a debugger, simulator, and other support programs.
- A computer (usually a PC) on which to run the development software.

- A hardware device called a programmer that connects to the computer through the serial, parallel, or USB line. The PIC is inserted in the programmer and "blown" by downloading the executable code generated by the development system. The hardware programmer usually includes the support software.
- A cable or connector for connecting the programmer to the computer.
- A PIC microcontroller.

Alternatively, some PIC microcontrollers can be programmed while installed in their applications boards. Although this option can be very useful as a production and distribution tool, for reasons of space it is not discussed in this book.

PIC Programmers

The development system (assembler or compiler) and the programmer driver are the software components. The computer, programmer, and connectors are the hardware elements. Figure 6.1 shows a commercial programmer that connects to the USB port of a PC. The one in the illustration is made by MicroPro.

Figure 1.1 *USB PIC programmer made by MicroPro.*

Many other programmers are available on the market. Microchip offers several high-end models with in-circuit serial programming (ICSP) and low-voltage programming (LVP) capabilities. These devices allow the PIC to be programmed in the target circuit. Some PICs can write to their own program memory. This makes possible the use of so-called bootloaders, which are small resident programs that allow loading user software over the RS-232 or USB lines. Programmer/debugger combinations are also offered by Microchip and other vendors.

Development Boards

A development board is a demonstration circuit that usually contains an array of connected and connectable components. Their main purpose is as a learning and experiment tool. Like programmers, PIC development boards come in a wide range of prices and levels of complexity. Most boards target a specific PIC microcontroller or a PIC family of related devices. Lacking a development board, the other option is to build the circuits oneself, a time-consuming but valuable experience. Figure 1.2 shows the LAB-X1 development board for the 16F87x PIC family.

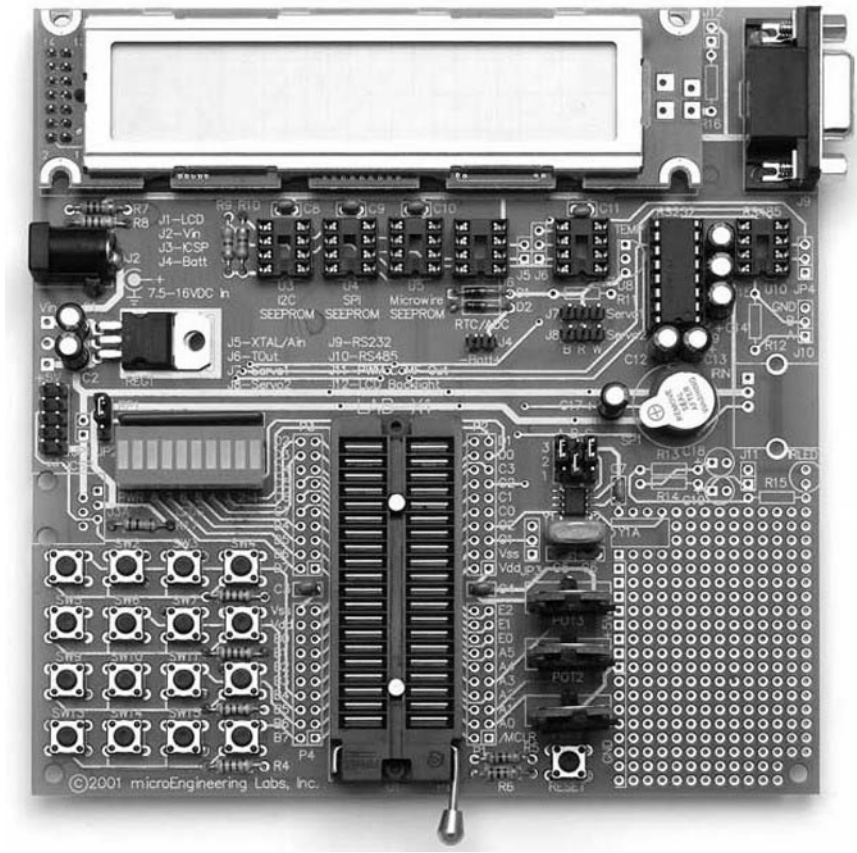


Figure 1.2 *LAB-X1 development board.*

The LAX-X1 board, as well as several other models, are products of microEngineering Labs, Inc. Development boards from Microchip and other vendors are also available.

1.3 PIC Architecture

PIC microcontrollers are roughly classified by Microchip into three groups: baseline, mid-range, and high-performance. Figure 1.3 shows the components of each PIC family at the time of this writing.

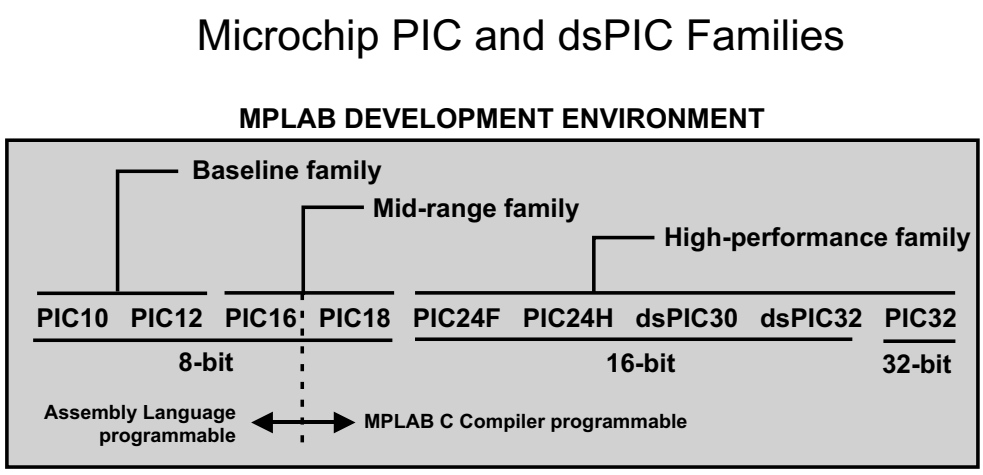


Figure 1.3 Microchip PIC and dsPIC families.

Within each of the groups the PIC are classified based on the first two digits of the PIC's family type. However, the sub-classification is not very strict, as there is some overlap. In fact, we find PICs with 16X designations that belong to the baseline family and others that belong to the mid-range group. In the following sub-sections we describe the basic characteristics of the various sub-groups of the three major PIC families with 8-bit architectures.Table 1.1 shows the principal hardware characteristics of each of the four 8-bit PIC families

Table 1.1

8-bit PIC Architectures Comparison Chart

	BASELINE	MID-RANGE	ENHANCED	PIC18
Pin Count	6-40	8-64	8-64	18-100
Interrupts	No	Single interrupt	Single interrupt Context saved	Multiple Interrupts Context saved
Performance	5 MIPS	5 MIPS	8 MIPS	Up to 16 MIPS
Instructions	33, 12-bit	35, 14-bit	49, 14-bit	83, 16-bit
Program Memory	Up to 3 KB	Up to 14 KB	Up to 28 KB	Up to 128 KB
Data Memory	138 Bytes	368 Bytes	1,5 KB	4 KB
Hardware Stack	2 level	8 level	16 level	32 level
Total Number of Devices	16	58	29	193
Families	PIC10 PIC12 PIC14 PIC16	PIC12 PIC16	PIC12FXXX PIC16F1XX	PIC18

1.3.1 Baseline PIC Family

This group includes members of the PIC10, PIC12, PIC14, and PIC16 families. The devices in the baseline group have 12-bit program words and are supplied in 6- to 28-pin packages. The microcontrollers in the baseline group are described as being suited for

battery-operated applications because they have low power requirements. The typical member of the baseline group has a low pin count, flash program memory, and low power requirements. The following types are in the Baseline group:

- PIC10 devices
- PIC12 devices
- PIC14 devices
- Some PIC16 devices

We present a short summary of the functionality and hardware types of the baseline PICs in the sections that follow, although these devices are not covered in this book.

PIC10 devices

The PIC10 devices are low-cost, 8-bit, flash-based CMOS microcontrollers. They use 33 single-word, single-cycle instructions (except for program branches, which take two cycles. The instructions are 12-bits wide. The PIC10 devices feature power-on reset, an internal oscillator mode which saves having to use ports for an external oscillator. They have a power-saving SLEEP mode, A Watchdog Timer, and optional code protection.

The recommended applications of the PIC10 family range from personal care appliances and security systems to low-power remote transmitters and receivers. The PICs of this family have a small footprint and are manufactured in formats suitable for both through hole or surface mount technologies. Table 1.2 lists the characteristics of the PIC10F devices.

Table 1.2

PIC10F Devices

	10F200	10F202	10F204	10F206
Clock:				
Maximum Frequency of Operation (MHz)	4	4	4	4
Memory:				
Flash Program Memory	256	512	256	512
Data Memory (bytes)	16	24	16	24
Peripherals:				
Timer Module(s)	TMR0	TMR0	TMR0	TMR0
Wake-up from Sleep	Yes	Yes	Yes	Yes
Comparators	0	0	1	1
Features:				
I/O Pins	3	3	3	3
Input Only Pins	1	1	1	1
Internal Pull-ups	es	Yes	Yes	Yes
In-Circuit Serial Programming	Yes	Yes	Yes	Yes
Instructions	33	33	33	33
Packages:	---------- 6-pin SOT-23 ----------			
	---------- 8-pin PDIP ----------			

Two other PICs of this series are the 10F220 and the 10F222. These versions include four I/O pins and two analog-to-digital converter channels. Program memory is 256 words on the 10F220 and 512 in the 10F222. Data memory is 16 bytes on the F220 and 23 in the F222.

PIC12 Devices

The PIC12C5XX family are 8-bit, fully static, EEPROM/EPROM/ROM-based CMOS microcontrollers. The devices use RISC architecture and have 33 single-word, single-cycle instructions (except for program branches that take two cycles). Like the PIC10 family, the PIC12C5XX chips have power-on reset , device reset, and an internal timer. Four oscillator options can be selected, including a port-saving internal oscillator and a low-power oscillator. These devices can also operate in SLEEP mode and have watchdog timer and code protection features.

The PIC12C5XX devices are recommended for applications ranging from personal care appliances, security systems, and low-power remote transmitters and receivers. The internal EEPROM memory makes possible the storage of user-defined codes and passwords as well as appliance setting and receiver frequencies. The various packages allow through-hole or surface mounting technologies. Table 1.3 lists the characteristics of some selected members of this PIC family.

Table 1.3

PIC 12CXXX and 12CEXXX Devices

	12C508(A) 12C509A 12CR509A	12C518	12CE519	12C671 12C672 12C673	12CE674
Clock:					
Maximum Frequency of Operation (MHz)	4	4	4	10	10
Memory:					
EPROM Program Memory (bytes)	25/41/41	25	41	128	128
Peripherals:					
EEPROM Data Memory (bytes)	—	16	16	0/0/16	16
Timer Module(s)	TMR0	TMR0	TMR0	TMR0	TMR0
A/D Converter (8-bit) Channels	—	—	—	4	4
Features:					
Interrupt Sources	—	—	—	4	4
I/O Pins	5	5	5	5	5
Input Pins	1	1	1	1	1

(continues)

Table 1.3

*PIC 12CXXX and 12CEXXX Devices **(continued)***

	12C508(A) 12C509A 12CR509A	12C518	12CE519	12C671 12C672 12C673	12CE674
Internal Pull-ups	Yes/Yes/No	Yes	Yes	Yes	Yes
In-Circuit Serial Programming	Yes/No	Yes	Yes	Yes	Yes
Number of Instructions	33	33	33	35	35
Packages	8-pin DIP SOIC	8-pin DIP JW,SOIC	8-pin DIP JW. SOIC	8-pin DIP SOIC	8-pin DIP JW

Two other members of the PIC12 family are the 12F510 and the 16F506. In most respects these devices are similar to the ones previously described, except that the 12F510 and 16F506 both have flash program memory. Table 1.4 lists the most important features of these two PICs.

Table 1.4

PIC12F510 and 12F675

	12F629	12F675
Clock:		
Maximum Frequency of Operation (MHz)	20	20
Memory:		
Flash Program Memory	1024	1024
Data Memory (SRAM bytes)	64	64
Peripherals:		
Timers 8/16 bits	1/1	1/1
Wake-up from Sleep on Pin Change	Yes	Yes
Features:		
I/O Pins	6	6
Analog comparator module	Yes	Yes
Analog-to-digital converter	No	10-bit
In-Circuit Serial Programming	Yes	Yes
Enhanced Timer1 module	Yes	Yes
Interrupt capability	Yes	Yes
Number of Instructions	35	35
Relative addressing	Yes	Yes
Packages	8-pin PDIP, SOIC, DFN-S	8-pin PDIP SOIC, DFN-S

Two other members of the PIC12F are the 12F629 and 12F675. The only difference between these two devices is that the 12F675 has a 10-bit analog-to-digital converter while the 629 has not A/D converter. Table 1.5 lists some important features of both PICs.

Table 1.5

PIC12F629 and 12F675

	12F629	12F675
Clock:		
Maximum Frequency of Operation (MHz)	20	20
Memory:		
Flash Program Memory	1024	1024
Data Memory (SRAM bytes)	64	64
Peripherals:		
Timers 8/16 bits	1/1	1/1
Wake-up from Sleep on Pin Change	Yes	Yes
Features:		
I/O pins	6	6
Analog comparator module	Yes	Yes
Analog-to-digital converter	No	10-bit
In-circuit serial programming	Yes	Yes
Enhanced Timer1 module	Yes	Yes
Interrupt capability	Yes	Yes
Number of instructions	35	35
Relative addressing	Yes	Yes
Packages	8-pin PDIP SOIC DFN-S	8-pin PDIP SOIC DFN-S

Several members of the PIC12 family, 12F635, 12F636, 12F639, and 12F683, are equipped with special power-management features (called nanowatt technology by Microchip). These devices were especially designed for systems that require extended battery life.

PIC14 Devices

The single member of this family is the PIC14000, which is built with CMOS technology. This makes the PIUC14000 fully static and gives it industrial temperature range. The 14000 is recommended for battery chargers, power supply controllers, power management system controllers, HVAC controllers, and for sensing and data acquisition applications.1.3.2

1.3.3 Mid-range PIC Family

The mid-range PICs includes members of the PIC12 and PIC16 groups as well as the PIC 18 group. According to Microchip the mid-range PICs all have 14-bit program words with either flash or OTP program memory. Those with flash program memory also have EEPROM data memory and support interrupts. Some members of the mid-range group have USB, I2C, LCD, USART, and A/D converters. Implementations range form 8 to 64 pins.

PIC16 Devices

This is by far the largest mid–range PIC group. Currently over 80 versions of the PIC16 are listed in production by Microchip. Although we do not cover the mid-range devices

in this book, we have selected a few of its most prominent members of the PIC16 family to list their most important features. These are found in Table 1.6.

Table 1.6

PIC16 Devices

	16C432	16C58	16C770	16F54	16F84A	16F946
Clock:						
Maximum Frequency MHz	20	40	20	20	20	20
Memory:						
Program memory type	OTP	OTP	OTP	Flash	Flash	Flash
K-bytes	3.5	3	3.5	0.75	1.75	14
K-words	2	2	2	0.5	1	8
Data EEPROM	0	0	0	0	64	256
Peripherals:						
I/O channels	12	12	16	12	13	53
ADC channels	0	0	6	0	0	8
Comparators	0	0	0	0	0	2
Timers	1/8-bit	1/8-bit 1/16-bit	2/8-bit	1/8-bit	1/8-bit	2/8-bit 1/16-bit
Watchdog timer	Yes	Yes	Yes	Yes	Yes	Yes
Features:						
ICSP	Yes	No	Yes	No	Yes	Yes
ICD	No	No	No	No	0	1
Pin count	20	18	20	18	18	64
Communications	-	-	MPC/SPI	-	-	AUSART
Packages	20/CERDIP, 20/SSOP 208mil	18/CERDIP 18/PDIP 18/SOIC 300mil 300mil	20/CERDIP 20/PDIP 20/SOIC	18/PDIP 18/SOIC 300mil	18/PDIp 18/SOIC 300mil	64/TQFP

Microchip documentation refers to an enhanced mid-range family composed of PIC12FXXX and PIC16F1XX devices. These devices maintain compatibility with the previous members of the mid-range family while providing additional performance. Their most important new features include multiple interrupts, fourteen additional instructions, up to 28 KB program memory, and additional peripheral modules.

1.3.3 High-Performance PICs and DSPs

The high-performance PICs belong to the PIC18 and PIC32 groups. The motivation for expanding the PIC arquitecture and modifying the core of the mid-range PICs relate to the following limitations:

- Small-size stack
- Single interrupt vector
- Limited instruction set
- Small memory size
- Limited number of peripherals
- No high-level language programmability

The devices in the PIC16 group have 16-bit program words, flash program memory, a linear memory space of up to 2 Mbytes, as well as protocol-based communications facilities. They all support internal and external interrupts and a much larger instruction set than members of the baseline and mid-range families. The PIC18 family is also a large one, with over seventy different variations currently in produc-

tion. These devices are furnished in 18 to 80 pin packages. Microchip describes the PICs in this family as high-performance with integrated A/D converters.

Digital Signal Processor

The notion of digital signal processing starts with the conversion of analog signal information such as voice, image, temperature, or pressure primitive data to digital values that can be stored and manipulated by a computing device. Converting the data from its primitive analog form to a digital format makes it much easier to analyze, display, store, process, or convert the data to another format. Digital signal processing is based on the fact that computing and data processing operations are easier to perform on digital data than on raw analog signals.

The concept of digital signal processing can be illustrated by means of a satellite-based Earth imagining system (such as the Landsat EROS) shown in Figure 1.4.

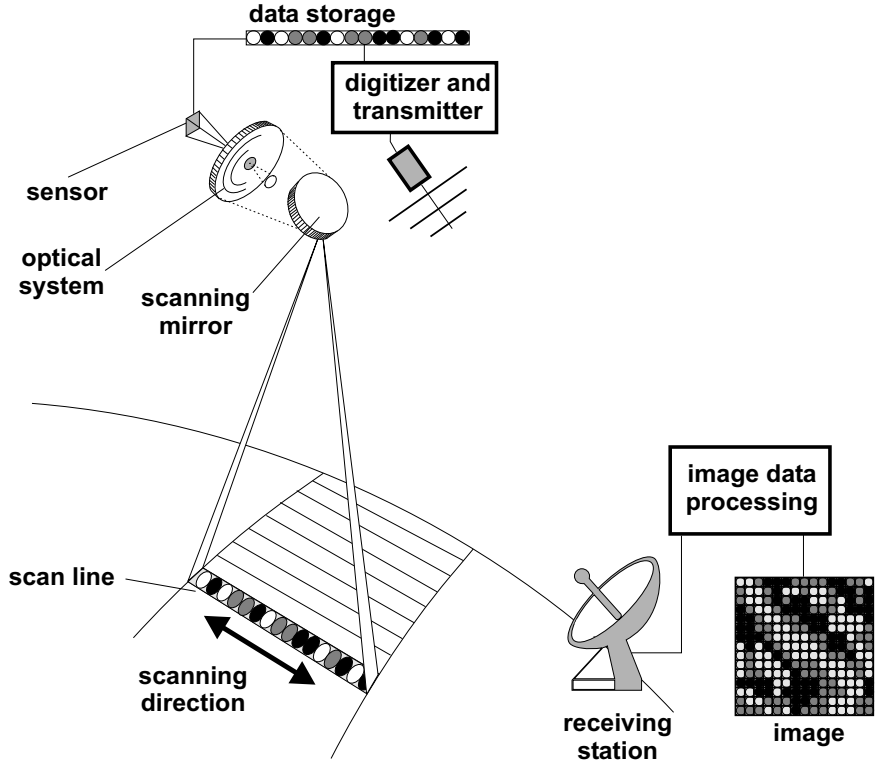


Figure 1.4 *Schematic of a space-borne imaging system.*

The optical-mechanical instrument onboard a spacecraft, shown in Figure 1.4, consists of several subsystems. The scanning mirror collects the radiation, which is imaged by an optical system onto a sensor device. The sensor performs an analog-to-digital conversion and places the digital values in a temporary storage structure. During its orbit, the satellite reaches a location in space from which it can communicate with an Earth receiving station. At this time, the transmitter and support circuitry send the digital data to the receiving station. The receiving station

processes this data and formats it into an image. In this scheme, digital signal processing can take place as the image data is sensed by the instrument and temporarily stored on board the satellite, or when the raw data received by the Earth station is converted into an image that can be manipulated, viewed, stored, or re-processed.

Analog-to-Digital

Conversion from analog-to-digital form and vice versa are not formally operations of a DSP. However, these conversions are so often required during signal processing that most DSP devices include the analog-to-digital and digital-to-analog conversion hardware.

Analog-to-digital conversion is usually performed by sampling the signal at uniform time intervals and using the sampled value as representative of the region between the intervals. Figure 1.5 shows an example of analog-to-digital conversion by sampling.

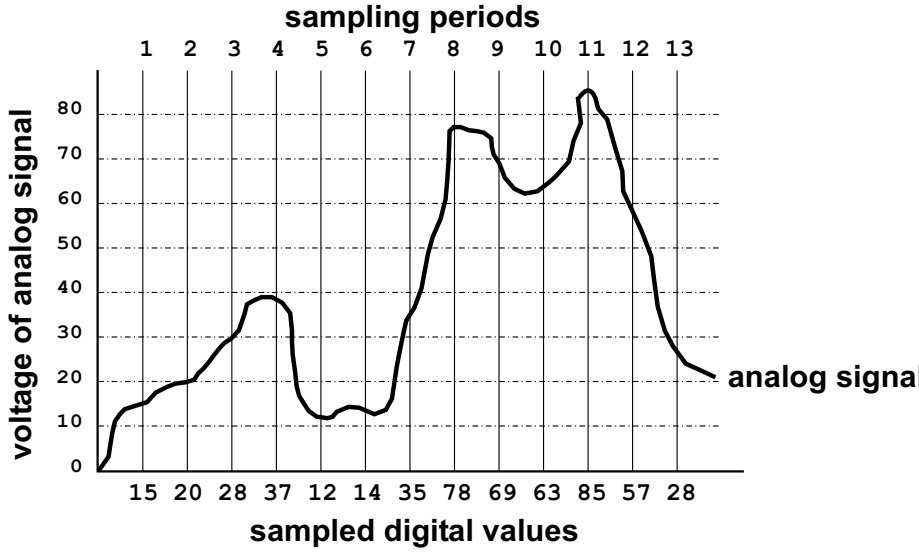


Figure 1.5 *Analog-to-digital conversion by sampling.*

In Figure 1.5 we see that the sampled values are actually an approximation of the analog curve, as the variations between each interval are lost in the conversion process. Therefore, the more sampling periods, the more accurate the approximation. On the other hand, too small a sampling rate tends to reduce the significance of the data by producing repeated values in the digital record.

Chapter 2

PIC18 Architecture

2.1 PIC18 Family Overview

The PIC18 family was designed to provide ease of use (programmable in C), high performance, and effortless integration with previous 8-bit families. In addition to the standard modules found in the PIC16 and previous families, the PIC18 includes several advanced peripherals, such as CAN, USB, Ethernet, LCD and CTMU. Its principal features are

- Nanowatt technology ensures low power consumption
- 83 instructions (16-bit wide)
- C language optimized
- Up to 2 MB addressable program memory
- 4KB maximum RAM
- 32-level hardware stack
- 8-bit file select register
- Integrated 8x8 hardware multiplier

The performance of the PIC18 series is the highest in the Microchip 8-bit architecture. Figure 2.1 is a block diagram of the PIC18 architecture.

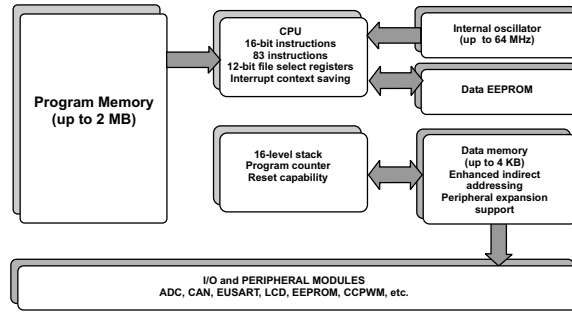


Figure 2.1 *Block diagram of PIC18 architecture.*

Although the PIC16 series has been very successful in the microcontroller marketplace, it also suffers from limitations and constraints. Perhaps the most significant limitation is that the devices of the PIC16 family can only be programmed in Assembly language. Other limitations result from the device's RISC design. For example, the absence of certain types of opcodes, such as the Branch instruction, make it necessary to combine a skip opcode followed by a goto operation in order to provide a conditional, targeted jump. Other limitations relate to the hardware itself: small stack and a single interrupt vector. As the complexity, memory size, and the number of peripheral modules increased, the limitations of the PIC16 series became more evident.

In the PIC18 series, Microchip reconsidered its PIC16 design rules and produced a completely new style microcontroller, with a much more complex core, while limiting the changes to the peripheral modules. The degree of change can be deduced from the expansion of the instruction set from 35 14-bit to 83 16-bit operation codes. Memory has gone from 14 to 128 KB; the stack from 8 levels to 32 levels. These changes made it possible to optimize the PIC18 series for C language programming.

2.1.1 PIC18FXX2 Group

At the present time, Microchip lists 193 different devices in the PIC18 family. These devices are available with pin counts from 28 to 100 and in the SOIC, DIP, PLCC, SPDIP, QFN, SSOP, TQFP, QFN, and LQFP packages. For consistency with the tutorial nature of this book, we have selected the PIC18F4X2 group with identical DIP and SOIC pinouts. Figure 2.2 shows the pin diagram for the PIC18F4X2 devices.

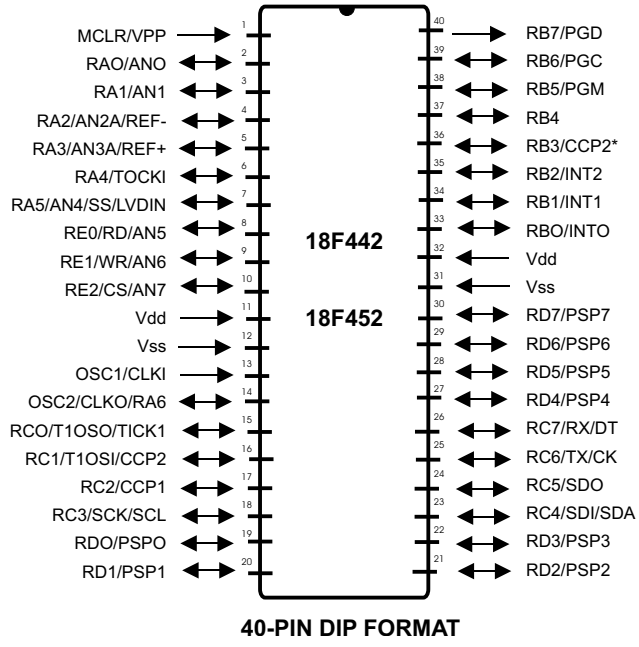


Figure 2.2 *Pin diagram for PIC18F4X2 devices.*

For learning and experimentation the devices in DIP packages are more convenient because they can be easily inserted in the ZIF (zero insertion force) sockets found in most programming devices, development boards, and breadboards. The devices in Figure 1.1 and Figure 1.2 are so equipped. A PLCC (plastic leaded chip carrier) package with 44 pins is also available for 18F442 and 18F452 devices. We do not cover this option.

2.1.2 PIC18FXX2 Device Group Overview

These devices come in 28-pin and 40-pin packages, as well as in a 44-pin PLCC package previously mentioned. The 28-pin devices do not have a Parallel Slave Port (PSP). Also, the number of analog-to-digital (A/D) converter input channels is reduced to 5. An overview of features is shown in Table 2.1

Table 2.1

Principal Features of Devices in the PIC18FXX2 Family

FEATURES	PIC18F242	PIC18F252	PIC18F442	PIC18F452
Operating Frequency	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz	DC - 40 MHz
Program Memory (Bytes)	16K	32K	16K	32K
Program Memory (Instructions)	8192	16384	8192	16384
Data Memory (Bytes)	768	1536	768	1536
Data EEPROM Memory (Bytes)	256	256	256	256
Interrupt Sources	17	17	18	18
I/O Ports	A, B, C	A, B, C	A, B, C, D, E	A, B, C, D, E
Timers	4	4	4	4
Capture/Cornpare /PWM Modules	2	2	2	2
Serial Communications	---------- MSSP, Addressable USART ----------			
Parallel Communications	-	-	PSP	PSP
10-bit Analog-to-Digital Module	5 channels	5 channels	8 channels	8 channels
RESETS (and Delays)	---------- POR, BOR, Reset, Instruction, Stack Full, Stack Underflow, (PWRT, OST) ----------			
Programmable Low Voltage Detect	Yes	Yes	Yes	Yes
Programmable Brown-out Reset	Yes	Yes	Yes	Yes
Instruction Set	75 Instructions	75 Instructions	75 Instructions	75 Instructions
Packages	28-pin DIP 28-pin SOIC SOIC	28-pin DIP 28-pin SOIC SOIC	40-pin DIP PLCC 44-pin SOIC	40-pin DIP QFP PLCC 44-pin SOIC

From Table 2.1 the following general features of the PIC18FXX2 devices can be deduced:

1. Operating frequency is 40 MHz for all devices. They all have a 75 opcode instruction set.
2. Program memory ranges from 16K (8,192 instructions) in the PIC18F2X2 devices to 32K (16,384 instructions) in the PIC18F4X2 devices.
3. Data memory ranges for 768 to 1,536 bytes.
4. Data EEPROM is 256 bytes in all devices.
5. The PIC18F2X2 devices have three I/O poerts (A, B, and C) and the PIC18F4X2 devices have five ports (A, B, C, D, and E).
6. All devices have four timers, two Capture/Compare/PWM modules, MSSP and adressable USART for serial communications and 10-bit analog-to-digital modules.
7. Only PIC18F4X2 devices have a parallel port.

2.1.3 PIC18F4X2 Block Diagram

The block diagram of the 18F4X2 microcontrollers, which correspond to the 40-pin devices of Figure 2.2, is shown in Figure 2.3.

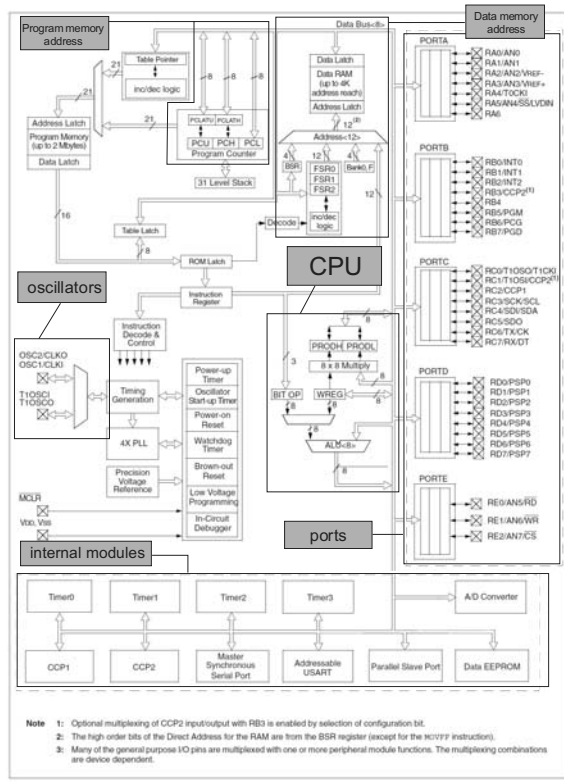


Figure 2.3 *PIC18F4X2 block diagram.*

2.1.4 Central Processing Unit

In Figure 2.3 the dashed rectangle labeled CPU (central processing unit) contains the 8-bit Arithmetic Logic Unit, the Working register labeled WREG, and the 8-bit-by-8-bit hardware multiplier, described later in this chapter. The CPU receives the instruction from program memory according to the value in the Instruction register and the action in the Instruction Decode and Control block. An interrupt mechanism with several sources (not shown in Figure 2.3) is also part of the PIC18FXX2 hardware.

The Status Register

The Status register, not shown in Figure 2.3, is part of the CPU and holds the individual status bits that reflect the operating condition of the individual elements of the device. Figure 2.4 shows the bit structure of the Status register.

bits:	7	6	5	4	3	2	1	0
	-	-	-	N	OV	Z	DC	C

```
bit 4 N:    Negative bit
            1 = Arithmetic result is negative
            0 = Arithmetic result is positive
bit 3 OV:   Overflow bit
            1 = Overflow in signed arithmetic
            0 = No overflow occurred
bit2 Z:     Zero bit
            1 = The result of an operation is zero
            0 = The result of an operation is not zero
bit 1 DC:   Digit carry/borrow bit for ADDWF, ADDLW, SUBLW,
            and SUBWF instructions. For borrow the polarity
            is reversed.
            1 = A carry-out from the 4th bit of the result
            0 = No carry-out from the 4th bit of the result
            For rotate instructions (RRF and RLF) this bit
            is loaded with either bit 4 or bit 3 of the
            source register.
bit 0 C:    Carry/borrow bit for ADDWF, ADDLW, SUBLW, and
            SUBWF instructions. For borrow the polarity
            is reversed.
            1 = A carry-out from the most significant bit
            0 = No carry-out from the most significant bit
            For rotate instructions (RRF and RLF) this bit
            is loaded with either bit 4 or bit 3 of the
            source register.
```

Figure 2.4 *Status register bitmap.*

Program Counter Register

The 21-bit wide Program Counter register specifies the address of the next instruction to be executed. The register mapping of the Program Counter register is shown in Figure 2.5.

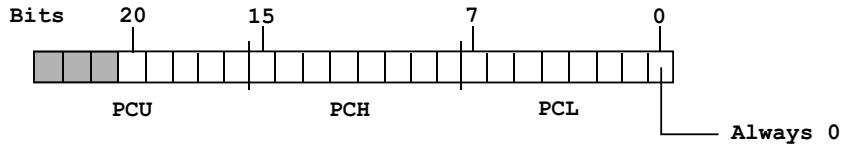


Figure 2.5 *Register map of the Program Counter.*

As shown in Figure 2.5, the low byte of the address is stored in the PCL register, which is readable and writeable. The high byte is stored in the PCH register. The upper byte is in the PCU register, which contains bits <20:16>. The PCH and PCU registers are not directly readable or writeable. Updates to the PCH register are performed through the PCLATH register. Updates to the PCU register are performed through the PCLATU register.

The Program Counter addresses byte units in program memory. In order to prevent the Program Counter from becoming misaligned with word instructions, the LSB of PCL is fixed to a value of '0' (see Figure 2.5). The Program Counter increments by 2 to the address of the next sequential instructions in the program memory.

The CALL, RCALL, GOTO, and program branch instructions write to the Program Counter directly. In these instructions, the contents of PCLATH and PCLATU are not transferred to the program counter. The contents of PCLATH and PCLATU are transferred to the Program Counter by an operation that writes PCL. Similarly, the upper 2 bytes of the Program Counter will be transferred to PCLATH and PCLATU by an operation that reads PCL.

Hardware Multiplier

All PIC18FXX2 devices contain an 8 x 8 hardware multiplier in the CPU. Because multiplication is a hardware operation it completes in a single instruction cycle. Hardware multiplication is unsigned and produces a 16-bit result that is stored in a 16-bit product register pair labeled PRODH (high byte) and PRODL (low byte).

Hardware multiplication has the following advantages:

- Higher computational performance
- Smaller code size of multiplication algorithms

The performance increase allows the device to be used in applications previously reserved for Digital Signal Processors.

Interrupts

PIC18FXX2 devices support multiple interrupt sources and an interrupt priority mechanism that allows each interrupt source to be assigned a high or low priority level. The high-priority interrupt vector is at OOOOO8H and the low-priority interrupt vector is at 000018H. High-priority interrupts override any low-priority interrupts that may be in progress. Ten registers are related to interrupt operation:

- RCON
- INTCON
- INTCON2
- INTCON3
- PIR1, PIR2
- PIE1, PIE2
- IPR1, IPR2

Each interrupt source (except INTO) has three control bits:

- A Flag bit indicates that an interrupt event has occurred.
- An Enable bit allows program execution to branch to the interrupt vector address when the flag bit is set.
- A Priority bit to select high-priority or low priority for an interrupt source.

Interrupt priority is enabled by setting the IPEN bit {mapped to the RCON<7> bit}. When interrupt priority is enabled, there are 2 bits that enable interrupts globally. Setting the GIEH bit (1NTCON<7>) enables all interrupts that have the priority bit set. Setting the GIEL bit (INTCON<6>) enables all interrupts that have the priority bit cleared. When the interrupt flag, the enable bit, and the appropriate global interrupt enable bit are set, the interrupt will vector to address OOOOO8h or 000018H, depending on the priority level. Individual interrupts can be disabled through their corresponding enable bits.

When the IPEN bit is cleared (default state), the interrupt priority feature is disabled and the interrupt mechanism is compatible with PIC mid-range devices. In this compatibility mode, the interrupt priority bits for each source have no effect and all interrupts branch to address OOOOO8H.

When an interrupt is handled, the Global Interrupt Enable bit is cleared to disable further interrupts. The return address is pushed onto the stack and the Program Counter is loaded with the interrupt vector address, which can be OOOOO8H or 000018H. In the Interrupt Service Routine, the source or sources of the interrupt can be determined by testing the interrupt flag bits. To avoid recursive interrupts, these bits must be cleared in software before re-enabling interrupts. The “return from interrupt“ instruction, RETFIE, exits the interrupt routine and sets the GIE bit {GIEH or GIEL if priority levels are used), which re-enables interrupts.

Several external interrupts are also supported, such as the INT pins or the PORTB input change interrupt. In these cases, the interrupt latency will be three to four instruction cycles. Interrupts and interrupt programming are the subject of Chapter 8.

2.1.5 Special CPU Features

Several CPU features are intended for the following purposes:

- Mmaximize system reliability
- Minimize cost through the elimination of external components
- Provide power-saving operating modes
- Offer code protection

These special features are related to the following functions and components:

- SLEEP mode
- Code protection
- ID locations
- In-circuit serial programming
- SLEEP mode

SLEEP mode is designed to offer a very low current mode during which the device is in a power-down state. The application can wakeup from SLEEP through the following mechanisms:

1. External RESET
2. Watchdog Timer Wake-up
3. An interrupt

The Watchdog Timer is a free running on-chip RC oscillator, that does not require any external components. This RC oscillator is separate from the RC oscillator of the OSC1/CLKI pin. That means that the WDT will run, even if the clock on the OSC1/CLKI and OSC2/CLKO/ RA6 pins of the device has been stopped, for example, by execution of a SLEEP instruction.

Watchdog Timer

A Watchdog Timer time-out (WDT) generates a device RESET. If the device is in SLEEP mode, a WDT causes the device to wakeup and continue in normal operation (Watchdog Timer Wake-up). If the WDT is enabled, software execution may not disable this function. When the WDTEN configuration bit is cleared, the SWDTEN bit enables/disables the operation of the WDT. Values for the WDT postscaler may be assigned using the configuration bits.

The CLRWDT and SLEEP instructions clear the WDT and the postscaler (if assigned to the WDT) and prevent it from timing out and generating a device RESET condition. When a CLRWDT instruction is executed and the postscaler is assigned to the WDT, the postscaler count will be cleared, but the postscaler assignment is not changed.

The WDT has a postscaler field that can extend the WDT Reset period. The postscaler is selected by the value written to 3 bits in the CONFIG2H register during device programming.

Wake-Up by Interrupt

When global interrupts are disabled (the GIE bit cleared) and any interrupt source has both its interrupt enable bit and interrupt flag bit set, then one of the following will occur:

When an interrupt occurs before the execution of a SLEEP instruction, then the SLEEP instruction becomes a NOP. In this case, the WDT and WDT postscaler will not be cleared, the TO bit will not be set, and PD bits will not be cleared.

If the interrupt condition occurs during or after the execution of a SLEEP instruction, then the device will immediately wakeup from SLEEP. In this case, the SLEEP instruction will be completely executed before the wake-up. Therefore, the WDT and WDT postscaler will be cleared, the TO bit will be set, and the PD bit will be cleared.

Even if the flag bits were checked before executing a SLEEP instruction, it may be possible for these bits to set before the SLEEP instruction completes. Code can test the PD bit in order to determine whether a SLEEP instruction executed. If the PD bit is set, the SLEEP instruction was executed as a NOP. To ensure that the WDT is cleared, a CLRWDT instruction should be executed before a SLEEP instruction.

Low Voltage Detection

For many applications it is desirable to be able to detect a drop in device voltage below a certain limit. In this case, the application can define a low voltage window in which it can perform housekeeping tasks before the voltage drops below its defined operating range. The Low Voltage Detect feature of the PIC18FXX2 devices can be used for this purpose. For example, a voltage trip point for the device can be specified so that when this point is reached, an interrupt flag is set. The program will then branch to the interrupt's vector address and the interrupt handler software can take the corresponding action. Because the Low Voltage Detect circuitry is completely under software control, it can be “turned off” at any time, thus saving power.

Implementing Low Voltage Detect requires setting up a comparator that reads the reference voltage and compares it against the preset trip-point. This trip-point voltage is software programmable to any one of sixteen values by means of the 4 bits labeled LVDL3:LVDLO. When the device voltage becomes lower than the preselected trip-point, the LVDIF bit is set and an interrupt is generated.

Device Configuration

Several device configurations can be selected by programming the configuration bits. These bits are mapped, starting at program memory address 300000H. Note that this address is located in the configuration memory space (300000H to 3F0000H), which is only accessed using table read and table write operations. When the configuration bits are programmed, they will read as '0; when left unprogrammed they will read as '1'.

MPLAB development tools provide an __CONFIG directive, together with a set of device-specific operands, that simplify selecting and setting the desired configuration bits. This topic is explored in the book's chapters related to programming.

2.2 Memory Organization

Devices of the PIC18FXX2 family contain three independent memory blocks:

- Program Memory
- Data Memory
- Data EEPROM

Because the device uses a separate buss, the CPU can concurrently access the data and program memory blocks.

2.2.1 Program Memory

The Program Counter register is 21 bit wide and therefore capable of addressing a maximum of 2-Mbyte program memory space. Accessing a location between the physically implemented memory and the 2-Mbyte maximum address will read all zeroes. The PIC18F242 and PIC18F442 devices can store up to 8K of single-word instructions. The PIC18F252 and PIC18F452 devices can store up to 16K of single-word instructions. The RESET vector address is at OOOOH and the interrupt vector addresses are at 0008H and 0018H. Figure 2.6 shows the memory map for the PIC18FXX2 family.

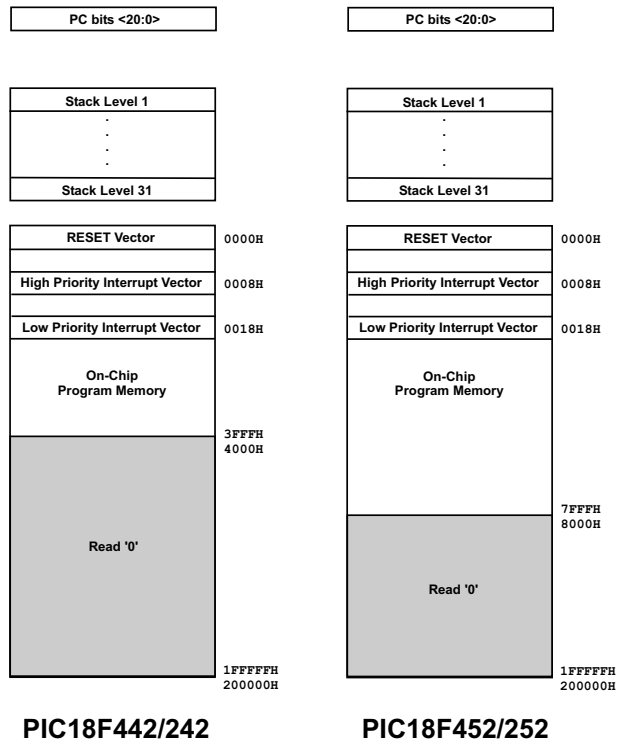


Figure 2.6 *Program memory map for the PIC18FXX2 family.*

2.2.2 18FXX2 Stack

The PIC18FXX2 stack is 31 address deep and allows as many combinations of back-to-back calls and interrupts to occur. When a CALL or RCALL instruction is executed, the Program Counter is pushed onto the stack. When a CALL or RCALL instruction is executed, or an interrupt is acknowledged, the Program Counter is pulled off the stack. This also takes place on a RETURN, RETLW, or RETFIE instruction. PCLATU and PCLATH registers are not affected by any of the RETURN or CALL instructions.

The stack consists of a 31-word deep and 21-bit wide RAM structure. The current stack position is stored in a 5-bit Stack Pointer register labeled STKPTR. This register is initialized to OOOOOB after all RESETS. There is no RAM memory cell associated with Stack Pointer value of OOOOOB. When a CALL type instruction executes (PUSH operation), the stack pointer is first incremented and the RAM location pointed to by STKPTR is written with the contents of the PC. During a RETURN type instruction (POP operation), the contents of the RAM location pointed to by STKPTR are transferred to the PC and then the stack pointer is decremented.

The stack space is a unique memory structure and is not part of either the program or the data space in the PIC18FXX2 devices. The STKPTR register is readable and writeable, and the address on the top of the stack is also readable and writeable through SFR registers. Data can also be pushed to or popped from the stack using the top-of-stack SFRs. Status bits indicate if the stack pointer is at, or beyond the 31 levels provided.

Stack Operations

Figure 2.7 shows the bit structure of the STKPTR register. The STKPTR register contains the stack pointer value, as well as a stack full and stack underflow) status bits. The STKPTR register can be read and written by the user. This feature allows operating system software to perform stack maintenance operations. The 5-bit value in the stack pointer register ranges from 0 through 31, which correspond to the available stack locations. The stack pointer is incremented by push operations and decremented when values are popped off the stack. At RESET, the stack pointer value is set to 0.

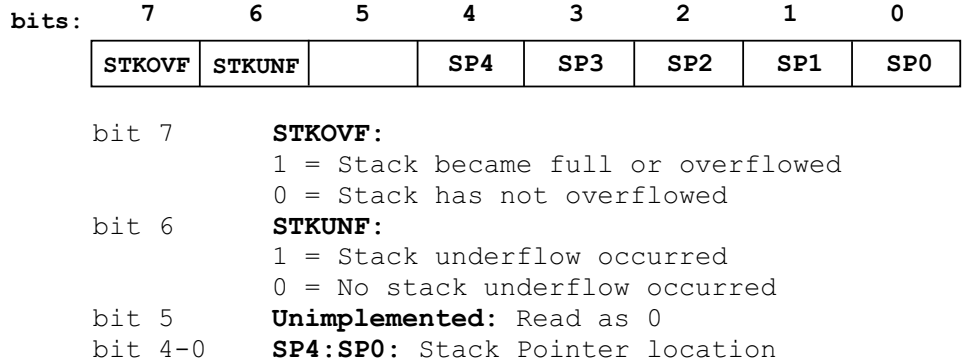


Figure 2.7 *STKPTR register bit map.*

The STKOVF bit is set after the program counter is pushed onto the stack 31 times without popping any value off the stack. Notice that some Microchip documentation refers to a STKFUL bit, which appears to be a synonym for the STKOVF bit. To avoid confusion, we only use the STKOVF designation in this book.

The STKOVF bit can only be cleared in software or by a Power-On Reset (POR) operation. The action that takes place when the stack becomes full depends on the state of the STVREN (Stack Overflow Reset Enable) configuration bit. The STVREN bit is bit 0 of the CONFIG4L register. If the STVREN bit is set, a stack full or stack overflow condition will cause a device RESET. Otherwise, the RESET action will not take place. When the stack pointer has a value of 0 and the stack is popped, a value of zero is entered to the Program Counter and the STKUNF bit is set. In this case, the stack pointer remains at 0. The STKUNF bit will remain set until cleared in software or a POR occurs. Returning a value of zero to the Program Counter on an underflow condition has the effect of vectoring the program to the RESET vector. User code can provide logic at the RESET vector to verify the stack condition and take the appropriate actions.

Three registers, labeled TOSU, TOSH and TOSL, hold the contents of the stack location pointed to by the STKPTR register. The address mapping of these registers is shown in Figure 2.8.

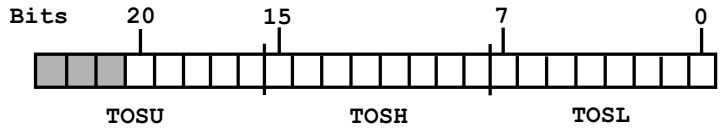


Figure 2.8 *Address mapping of the stack contents registers.*

Users can implement a software stack by manipulating the contents of the TOSU, TOSH, and TOSL registers. After a CALL, RCALL, or interrupt, user software can read the value in the stack by reading the TOSU, TOSH, and TOSL. These values can then be placed on a user-defined software stack. At return time, user software can replace the TOSU, TOSH, and TOSL with the stored values. At this time, global interrupts should have been disabled in order to prevent inadvertent stack changes.

Fast Register Stack

A fast return from interrupts is available in the PIC18FXX2 devices. This action is based on a Fast Register Stack that saves the STATUS, WREG, and BSR registers. The fast version of the stack is not readable or writable and is loaded with the current value of the three registers when an interrupt takes place. The FAST RETURN instruction is then used to restore the working registers and terminate the interrupt.

The fast register stack option can also be used to store and restore the STATUS, WREG, and BSR registers during a subroutine call. In this case, a fast call and fast return instruction are executed. This is only possible if no interrupts are used.

Instructions in Memory

Program memory is structured in byte-size units but instructions are stored as two bytes or four bytes. The Least Significant Byte of an instruction word is always stored in a program memory location with an even address, as shown in Figure 2.5. Figure 2.9 shows three low-level instructions as they are encoded and stored in program memory

		LOCATIONS IN PROGRAM MEMORY:		
		LSB = 1	LSB = 0	Word Address
				00000H
				00002H
				00004H
INSTRUCTIONS:				00006H
MOVLW	055H	0FH	55H	00008H
GOTO	000006H	EFH	03H	0000AH
		00H	00H	0000CH
MOVFF	12H, 456H	C1H	23H	0000EH
		04H	56H	00010H
				00012H
				00014H

Figure 2.9 *Instruction encoding.*

The CALL and GOTO instructions have an absolute program memory address embedded in the instruction. Because instructions are always stored on word boundaries, the data contained in the instruction is a word address. This word address is written to Program Counter bits <20:1>, which accesses the desired byte address. Notice in Figure 2.9 that the instruction

```
GOTO      000006H
```

is encoded by storing the number of single-word instructions that must be added to the Program Counter (03H). All program branch instructions are encoded in this manner.

2.2.3 Data Memory

Data memory is implemented as static RAM. Each register in the data memory has a 12-bit address, allowing up to 4096 bytes of data memory in the PIC18FXX2 devices. Data memory consists of Special Function Registers (SFRs) and General Purpose Registers (GPRs). The SFRs are used for control and status operations and for implementing the peripheral functions. The GPRs are for user data storage. Figure 2.10 is a map of data memory in the PIC18FXX2 devices.

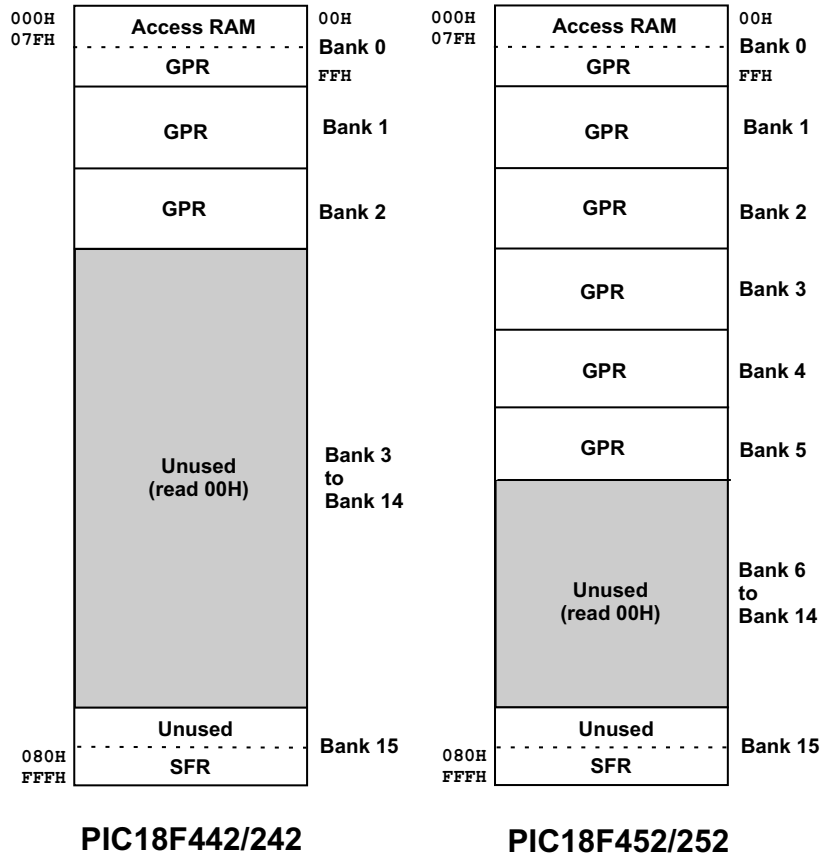


Figure 2.10 *Data memory map in PIC18FXX2.*

In Figure 2.10, GPRs start at the first location of Bank 0 and grow to higher memory addresses. Memory is divided into 255-byte units called banks. Seven banks are implemented in the PIC18F452/252 devices and four banks in the PIC18F442/242 devices. A read operation to a location in an unimplemented memory bank always returns zeros.

The entire data memory may be accessed directly or indirectly. Direct addressing requires the use of the BSR register. Indirect addressing requires the use of a File Select Register (FSRn) and a corresponding Indirect File Operand (INDFn). Addressing operations are discussed in Chapter 11 in the context of LCD programming.

Each FSR holds a 12-bit address value that can be used to access any location in the Data Memory map without banking. The SFRs start at address F80H in Bank 15 and extend to address 0FFFH in either device. This means that 128 bytes are assigned to the SFR area although not all locations are implemented. The individual SFRs are discussed in the context of their specific functionality. Figure 2.11 shows the names and addresses of the Special Function Registers.

Address	Name	Address	Name	Address	Name	Address	Name
FFFh	TOSU	FDFh	INDF2(3)	FBFh	CCPR1H	F9Fh	IPR1
FFEh	TOSH	FDEh	POSTINC2(3)	FBEh	CCPR1L	F9Eh	PIR1
FFDh	TOSL	FDDh	POSTDEC2(3)	FBDh	CCP1CON	F9Dh	PIE1
FFCh	STKPTR	FDCh	PREINC2(3)	FBCh	CCPR2H	F9Ch	—
FFBh	PCLATU	FDBh	PLUSW2(3)	FBBh	CCPR2L	F9Bh	—
FFAh	PCLATH	FDAh	FSR2H	FBAh	CCP2CON	F9Ah	—
FF9h	PCL	FD9h	FSR2L	FB9h	—	F99h	—
FF8h	TBLPTRU	FD8h	STATUS	FB8h	—	F98h	—
FF7h	TBLPTRH	FD7h	TMR0H	FB7h	—	F97h	—
FF6h	TBLPTRL	FD6h	TMR0L	FB6h	—	F96h	TRISE(2)
FF5h	TABLAT	FD5h	T0CON	FB5h	—	F95h	TRISD(2)
FF4h	PRODH	FD4h	—	FB4h	—	F94h	TRISC
FF3h	PRODL	FD3h	OSCCON	FB3h	TMR3H	F93h	TRISB
FF2h	INTCON	FD2h	LVDCON	FB2h	TMR3L	F92h	TRISA
FF1h	INTCON2	FD1h	WDTCON	FB1h	T3CON	F91h	—
FF0h	INTCON3	FD0h	RCON	FB0h	—	F90h	—
FEFh	INDF0(3)	FCFh	TMR1H	FAFh	SPBRG	F8Fh	—
FEEh	POSTINC0(3)	FCEh	TMR1L	FAEh	RCREG	F8Eh	—
FEDh	POSTDEC0(3)	FCDh	T1CON	FADh	TXREG	F8Dh	LATE(2)
FECh	PREINC0(3)	FCCh	TMR2	FACh	TXSTA	F8Ch	LATD(2)
FEBh	PLUSW0(3)	FCBh	PR2	FABh	RCSTA	F8Bh	LATC
FEAh	FSR0H	FCAh	T2CON	FAAh	—	F8Ah	LATB
FE9h	FSR0L	FC9h	SSPBUF	FA9h	EEADR	F89h	LATA
FE8h	WREG	FC8h	SSPADD	FA8h	EEDATA	F88h	—
FE7h	INDF1(3)	FC7h	SSPSTAT	FA7h	EECON2	F87h	—
FE6h	POSTINC1(3)	FC6h	SSPCON1	FA6h	EECON1	F86h	—
FE5h	POSTDEC1(3)	FC5h	SSPCON2	FA5h	—	F85h	—
FE4h	PREINC1(3)	FC4h	ADRESH	FA4h	—	F84h	PORTE(2)
FE3h	PLUSW1(3)	FC3h	ADRESL	FA3h	—	F83h	PORTD(2)
FE2h	FSR1H	FC2h	ADCON0	FA2h	IPR2	F82h	PORTC
FE1h	FSR1L	FC1h	ADCON1	FA1h	PIR2	F81h	PORTB
FE0h	BSR	FC0h	—	FA0h	PIE2	F80h	PORTA

Note 1: Unimplemented registers are read as '0'.
2: This register is not available on PIC18F2X2 devices.
3: This is not a physical register.

Figure 2.11 *PIC18FXX2 Special Function Registers map.*

2.2.4 Data EEPROM Memory

EEPROM stands for Electrically Erasable Programmable Read-Only Memory. This type of memory is used in computers and embedded systems as a nonvolatile storage. You find EEPROM in flash drives, BIOS chips, and in memory facilities such flash memory and EEPROM data storage memory found in PICs and other microcontrollers.

EEPROM memory can be erased and programmed electrically without removing the chip. The predecessor technology, called EPROM, required that the chip be removed from the circuit and placed under ultraviolet light in order to erase it. In embedded systems, the typical use of serial EEPROM on-board memory, and EEPROM ICs, is in the storage of passwords, codes, configuration settings, and other information to be remembered after the system is turned off.

Data EEPROM is readable and writable during normal operation. EEPROM data memory is not directly mapped in the register file space. Instead, it is indirectly addressed through the SFRs. Four SFRs used to read and write the program and data EEPROM memory. These registers are

- EECON1
- EECON2
- EEDATA
- EEADR

In operation, EEDATA holds the 8-bit data for read/write and EEADR holds the address of the EEPROM location being accessed.

All devices of the PIC18FXX2 family 256 bytes of data EEPROM with an address range from Oh to FFh. EEPROM access and programming are discussed in Chapter 10.

2.2.5 Indirect Addressing

The instruction set of most processors, including the PICs, provide a mechanism for accessing memory operands indirectly. Indirect addressing is based on the following capabilities:

1. The address of a memory operand is loaded into a register. This register is called the pointer.
2. The pointer register is then used to indirectly access the memory location at the address it "points to."
3. The value in the pointer register can be modified (usually incremented or decremented) so as to allow access to other memory operands.

Indirect addressing is useful in accessing data tables in manipulating software stacks.

In the PIC18FXX2 architecture, indirect addressing is implemented using one of three Indirect File Registers (labeled INDFx) and the corresponding File Select Register (labeled FSRx). Any instruction using an INDFx register actually accesses the register pointed to by the FSRx. Reading an INDF register indirectly (FSR = 0) will read OOH. Writing to the INDF register indirectly, results in a no operation.

The INDFx registers are not physical registers in the sense that they cannot be directly accessed by code. The FSR register is the pointer register that is initialized to the address of a memory operand. Once a memory address is placed in FSRx, any action on the corresponding INDFx register takes place at the memory location pointed at by FSR. For example, if the FSR0 register is initialized to memory address 0x20, then clearing an INDF0 register has the effect of clearing the memory location at address 0x20. In this case, the action on the INDF0 register actually takes place at the address contained in the FSR0 register. Now if FSR (the pointer register) is incremented and INDF is again cleared, the memory location at address 0x21 will be cleared. Indirect addressing is covered in detail in the programming chapters later in the book.

2.3 PIC18FXX2 Oscillator

In the operation of any microprocessor or microcontroller, it is necessary to provide a "clock" signal in the form of a continuously running, fixed-frequency, square wave. The operation and speed of the device are entirely dependent on this clock frequency. In addition to the fetch/execute cycle of the CPU, other essential timing functions are also derived from this clock signal ranging from timing and counting operations pulses required in communications. In many PIC microcontrollers, the internal or external component that generates this clock signal is called the oscillator.

Every microcontroller or microprocessor must operate with a clock signal of a specified frequency. The principal clock signal is divided internally by a fixed value, thus creating a lower-frequency signal. Each cycle of this slower signal is called an instruction cycle by Microchip. The instruction cycle can be considered the primary unit of time in the action of the CPU because it determines how long an instruction takes to execute. The original clock signal is also used to create phases or time stages within the instruction cycle or in other microcontroller operations. In PIC18FXX2 devices, the main oscillator signal is divided by four. For example, a clock signal frequency of 40 MHz produces an instruction cycle frequency of 10 MHz. Many microcontrollers, including PICs, provide an internal oscillator signal; however, this is not the case with the PIC18FXX2 devices, which require an external device to provide the clock signal. The pins labeled OSC1 and OSC2 (see Figure 2.2) are used with the oscillator function.

2.3.1 Oscillator Options

The PIC18FXX2 can be operated in eight different oscillator modes. The configuration bits labeled FOSC2, FOSC1, and FOSCO allow selecting one of these eight modes during start-up. Table 2.2 shows the designations and description of the eight oscillator modes.

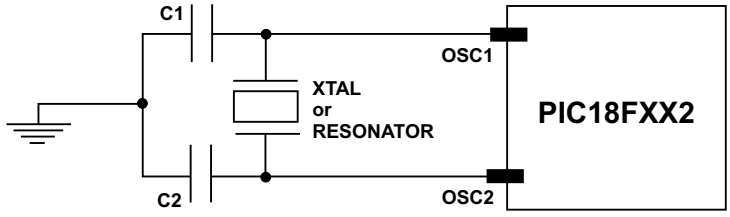
Table 2.2

Oscillator Types

CODE	TYPE
LP	Low-Power Crystal
XT	Crystal/Resonator
HS	High-Speed Crystal/Resonator
HS + PLL	High-Speed Crystal/Resonator with PLL enabled
RC	External Resistor/Capacitor
RCIO	External Resistor/Capacitor withI/O pin enabled
EC	External Clock
ECIO	External Clock with I/O pin enabled

Crystal Oscillator and Ceramic Resonator

The designations XT, LP, HS or HS+PLL in Table 2.2 refer to modes in which a crystal or ceramic resonator os connected to the OSC1 and OSC2 pins to establish a clock signal for the device. The PIC18FXX22 requires that crystals be parallel cut because serial cut crystals can give frequencies outside the manufacturer's specifications. Figure 2.12 shows the wiring and components required for oscillator modes LP, XT, and HS.



CAPACITOR SELECTION (C1 and C2)
FOR CERAMIC RESONATORS

Mode	Freq	C1 and C2
XT	455 kHz 2.0 MHz 4.0 MHz	68 - 100 pF 15 - 68 pF 15 - 68 pF
HS	8.0 MHz 16.0 MHz	10 - 68 pF 10 - 22 pF

CAPACITOR SELECTION (C1 and C2)
FOR CRYSTAL OSCILLATOR

Mode	Freq	C1 and C2
LP	32.0 kHz 200 kHz	33 pF 15 pF
XT	200 kHz 1.0 MHz 4.0 MHz	22 - 68 pF 15 pF 15 pF
HS	4.0 MHz 8.0 Mhz 20.0 MHz 25.0 Mhz	15 pF 15 - 33 pF 15 - 33 pF 15 - 33 pF

Figure 2.12 *Oscillator schematics for LP, XT, and HS modes.*

An external clock may also be used in the HS, XT, and LP oscillator modes. In this case, the clock is connected to the device's OSC1 pin while the OSC2 pin is left open.

RC Oscillator

The simplest and least expensive way of providing a clocked impulse to the PIC is with an external circuit consisting of a single resistor and capacitor. This circuit is usually called an RC oscillator. The major drawback of the RC option is that the frequency of the pulse depends on the supply voltage, the normal variations in the actual values of the resistor and capacitor, and the operating temperature. This makes the RC oscillator option only suitable for applications that are timing insensitive. Figure 2.13 shows the circuit required for the RC and RCIO oscillator modes.

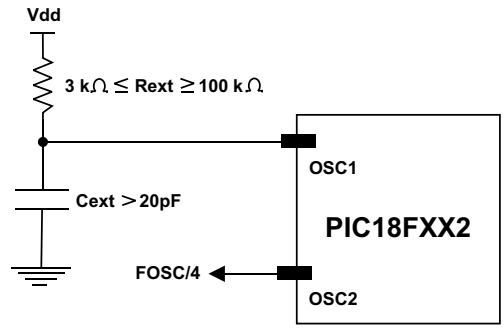


Figure 2.13 *RC and RCIO oscillator modes.*

The two variations of the RC option are designated RC and RCIO. In the RC option, the OSC2 pin is left open. In the RCIO variation, the OSC2 pin provides a signal with the oscillator frequency divided by 4 (FOSC/4 in Figure 2.13). This signal can be used for testing or to synchronize other circuit components.

External Clock Input

The EC and ECIO Oscillator modes are used with an external clock source connected to the OSC1 pin. Figure 2.14 shows the circuit for the EC oscillator mode.

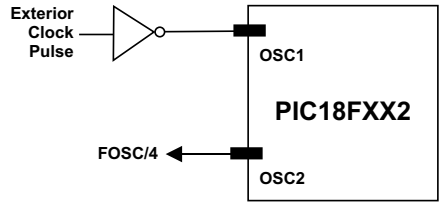


Figure 2.14 *External clock oscillator mode.*

In the EC mode (Figure 2.14), the oscillator frequency divided by 4 is available on the OSC2 pin. This signal may be used for test purposes or to synchronize other circuit components.

The ECIO oscillator is similar to the EC mode except that the OSC2 pin becomes an additional general-purpose I/O source; specifically, the OSC2 pin becomes bit 6 of PORTA (RA6).

Phase Locked Loop Oscillator Mode

With the Phase Locked Loop (PLL) a circuit is provided as a programmable option. This is convenient for users who want to multiply the frequency of the incoming crystal oscillator signal by 4, as in Figure 2.12. For example, if the input clock frequency is 10 MHz and the PLL oscillator mode is selected, the internal clock frequency will be 40 MHz. The PLL mode is selected by means of the FOSC<2:0> bits. This requires that the oscillator configuration bits are programmed for the HS mode. Otherwise, the system clock will come directly from OSC1.

2.4 System Reset

The PIC18FXX2 documentation refers to the following eight possible types of RESET.

1. Power-On Reset (POR)
2. Master Clear Reset during normal operation (MCLR)
3. Reset during SLEEP (MCLR)
4. Watchdog Timer Reset
5. Programmable Brown-Out Reset
6. Action of the RESET Instruction
7. Stack Full Reset
8. Stack Underflow Reset

The status of most registers is unknown after Power-on Reset (POR) and unchanged by all other RESETS. The remaining registers are forced to a "RESET state" on Power-on Reset, MCLR, WDT Reset, Brownout Reset, MCLR Reset during SLEEP, and by the RESET instruction.

2.4.1 Reset Action

Most registers are not affected by a WDT wake-up, as this is viewed as the resumption of normal operation. Status bits from the RCON register, Rl, TO, PD, POR, and BOR, are set or cleared differently in the various RESET actions. Software can read these bits to determine the type of RESET. Table 2.3 shows the RESET condition for some Special Function Registers.

Table 2.3

RESET State for some SFRs

Condition	Program Counter	RCON Register	$\overline{RI}$	$\overline{TO}$	$\overline{PD}$	$\overline{POR}$	$\overline{BOR}$	STKFUL	STKUNF
Power-On Reset	0000h	0--1 1100	1	1	1	0	0	u	u
$\overline{MCLR}$ Reset during normal operation	0000h	0--u uuuu	u	u	u	u	u	u	u
Software Reset during normal operation	0000h	0--0 uuuu	0	u	u	u	u	u	u
Stack Full Reset during normal operation	0000h	0--u uu11	u	u	u	u	u	u	1
Stack Underflow Reset during normal operation	0000h	0--u uu11	u	u	u	u	u	1	u
$\overline{MCLR}$ Reset during SLEEP	0000h	0--u 10uu	u	1	0	u	u	u	u
WDT Reset	0000h	0--u 01uu	1	0	1	u	u	u	u
WDT Wake-up	PC + 2	u--u 00uu	u	0	0	u	u	u	u
Brown-out Reset	0000h	0--1 11u0	1	1	1	1	0	u	u
Interrupt wake-up from SLEEP	PC + 2(1)	u--u 00uu	u	1	0	u	u	u	u

Legend: u = unchanged, x = unknown, - = unimplemented bit, read as '0'

Note 1: When the wake-up is due to an interrupt and the GIEH or GIEL bits are set, the PC is loaded with the interrupt vector (0x000008h or 0x000018h).

Some circuits include a hardware reset mechanism that allows the user to force a RESET action, usually by activating a switch that brings low the MCLR line. The same circuit holds high the MCLR line during device operation by tying it to the Vdd source. Figure 2.15 shows a possible schematic for a pushbutton reset switch on the MCLR line.

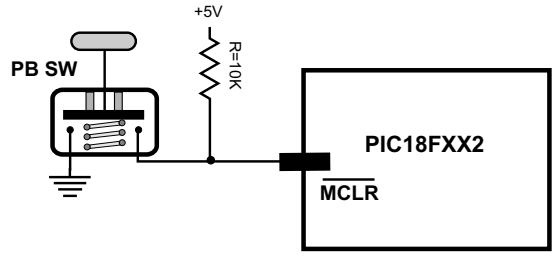


Figure 2.15 *RESET switch on the MCLR line.*

Power-On Reset (POR)

A Power-on Reset pulse is generated on-chip when a Vdd rise is detected. Users can take advantage of the POR circuitry by tying the MCLR pin to Vdd either directly or through a resistor, as shown in Figure 2.15. This circuit eliminates external RC components usually needed to create a Power-on Reset delay.

Power-Up Timer (PWRT)

The Power-up Timer (PWRT) provides a fixed nominal time-out from POR. The PWRT operates on an internal RC oscillator. The chip is kept in RESET as long as the PWRT is active. This action allows the Vdd signal to rise to an acceptable level. A configuration bit is provided to enable/disable the PWRT.

Oscillator Start-Up Timer (OST)

The Oscillator Start-up Timer (OST) provides a delay of 1024 oscillator cycles from the time of OSC1 input until after the PWRT delay is over. This ensures that the processor fetch/execute cycle does not start until the crystal oscillator or resonator has started and stabilized. The OST time-out is invoked only for XT, LP, and HS modes and only on Power-on Reset or wake-up from SLEEP.

PLL Lock Time-Out

When the Phase Locked Loop Oscillator Mode is selected, the time-out sequence following a Power-on Reset is different from the other oscillator modes. In this case, a portion of the Power-up Timer is used to provide a fixed time-out that is sufficient for the PLL to lock to the main oscillator frequency. This PLL lock time-out (TPLL) is typically 2 ms and follows the oscillator start-up time-out (OST),

Brown-Out Reset (BOR)

A temporary reduction in electrical power (brown-out condition) can activate the chip's brown-out reset mechanism. A configuration bit (BOREN) can be cleared to disable or set to enable the Brown-out Reset circuitry. If Vdd falls below a predefined value for a predetermined period, the brown-out situation will reset the chip. The chip will remain in Brown-out Reset until Vdd rises above the predefined value.

Time-Out Sequence

On power-up, the time-out sequence follows this order:

1. After the Power-on Reset (POR) time delay has expired, the Power-up Time (PWRT) time-out is invoked
2. The Oscillator Start-up Time (OST) is activated

The total time-out will vary based on the particular oscillator configuration and the status of the PWRT. In RC mode with the PWRT disabled, there will be no time-out at all. Because the time-outs occur from the POR pulse, if MCLR is kept low long enough, the time-outs will expire. Bringing MCLR high will begin execution immediately. This is useful for testing purposes or to synchronize more than one PIC18FXXX device operating in parallel.

2.5 I/O Ports

PIC18FXX2 devices come equipped with either five or three ports. PIC18F4X2 devices have five ports and PIC18F2X2 devices have three ports. Ports provide access to the outside world and are mapped to physical pins on the device. Some port pins are multiplexed with alternate functions of peripheral modules. When a peripheral module is enabled, that pin ceases to be a general-purpose I/O.

Ports are labeled with letters of the alphabet and are designated as port A (PORTA) to port E (PORTE). Port pins are bi-directional, that is, each pin can be configured to serve either as input or output. Each port has three registers for its operation. These are

- A TRIS register that determines data direction
- A PORT register used to read the value stored in each port pin or to write values to the port's data latch
- A LAT register that serves as a data latch and is useful in read-modify-write operations on the pin values

The status of each line in the port's TRIS register determines if the port's line is designated as input or output. Storing a value of 1 in the port's line TRIS register makes the port line an input line, while storing a value of 0 makes it an output line. Input port lines are used in communicating with input devices, such as switches, keypads, and input data lines from hardware devices. Output port lines are used in communicating with output devices, such as LEDs, seven-segment displays, liquid-crystal displays (LCDs), and data output line to hardware devices.

Port pins are bit mapped, however, they are read and written as a unit. For example, the PORTA register holds the status of the eight pins possibly mapped to port A, while writing to PORTA will write to the port latches. Write operations to ports are actually read-modify-write operations. Therefore, the port pins are first read, then the value is modified, and then written to the port's data latch.

As previously mentioned, some port pins are multiplexed; for example, pin RA4 is multiplexed with the Timer0 module clock input, labeled TOCKI. In Figure 2.2 the port pin is shown as RA4/T0CKI. Other port pins are multiplexed with analog inputs and with other peripheral functions. The device data sheets contain information regarding the functions assigned to each device pin.

2.5.1 Port Registers

In PIC18FXX2 devices, ports are labeled PORTA, PORTB, PORTC, PORTD, and PORTE. PORTD and PORTE are only available in PIC18F4X2 devices. The characteristics of each port are detailed in the device's data sheet. For example, PORTA is a 7-bit wide, bi-directional port. The corresponding Data Direction register is TRISA. If software sets a TRISA bit to 1, the corresponding PORTA pin will serve as an input pin. Clearing a TRISA bit (= 0) will make the corresponding PORTA pin an output. It is easy to remember the function of the TRIS registers because the number 1 is reminiscent of the letter I and the number 0 of the letter O.

Reading the PORTA register reads the status of the pins, whereas writing to it will write to the port latch. The Data Latch register (LATA) is also memory mapped. Read-modify-write operations on the LATA register read and write the latched output value for PORTA. The RA4 pin is multiplexed with the Timer0 module clock input to become the RA4/TOCKI pin. This pin is a Schmitt Trigger input and an open drain output. All other RA port pins have TTL input levels and full CMOS output drivers.

All other PORTA pins are multiplexed with analog inputs and the analog VREF+ and VREF- inputs. The operation of each pin is selected by clearing/setting the control bits in the ADCON1 register (A/D Control Register). The TRISA register controls the direction of the PORTA pins. This is so even when the port pins are being used as analog inputs.

2.5.2 Parallel Slave Port

The Parallel Slave Port is implemented on the 40-pin devices only, that is, those with the PIC18F4X2 designation. In these devices, PORTD serves as an 8-bit wide Parallel Slave Port when the control bit labled PSPMODE (at TRISE<4>) is set. It is asynchronously readable and writable through the RD control input pin (REO/RD) and WR control input pin (RE1/WR).

The Parallel Slave Port can directly interface to an 8-bit microprocessor data bus. In this case, the microprocessor can read or write the PORTD latch as an 8-bit latch. Programming and operation of the Parallel Slave Port is discussed later in this book.

2.6 Internal Modules

In electronics a module can be loosely defined as an assembly of electronic circuits or components that performs as a unit. All PIC microcontrollers contain internal modules to perform specific functions or operations. In this sense we can refer to the Timer module, the Capture/Compare/PWM module, or the Analog-to-Digital Converter module. By definition, a module is an internal component.

A peripheral or peripheral device, on the other hand, is an external component, such as a printer, a modem, or a Liquid Crystal Display (LCD). Microcontrollers often communicate with peripheral devices through their I/O ports or through their internal modules. We make this clarification because sometimes in the literature we can find references to the “peripheral components” or the “peripherals” of a microcontroller when actually referring to modules.

2.6.1 PIC18FXX2 Modules

Most PIC microcontrollers contain at least one internal module, and many devices contain ten or more different modules. The following are the standard modules of the PIC18FXX2 family of devices:

- Timer0 module: 8-bit/16-bit timer/counter with 8-bit programmable prescaler
- Timerl module: 16-bit timer/counter
- Timer2 module: 8-bit timer/counter with 8-bit period register

- Timer3 module: 16-bit timer/counter
- Two Capture/Compare/PWM (CCP) modules
- Master Synchronous Serial Port (MSSP) module with two modes of operation
- Universal Receiver and Transmitter (USART)
- 10-bit Analog-to-Digital Converter (A/D)
- Controller Area Network (CAN)
- Comparator Module
- Parallel Slave Port (PSP) module

The structure and details of the internal modules are discussed in the programming chapters later in this book.

Chapter 3

Programming Tools and Software

3.1 Environment

In order to learn and practice programming microcontrollers in embedded systems, you will require a development and testing environment. This environment will usually include the following elements:

1. A software development environment in which to create the program's source file and generate an executable program that can later be loaded into the hardware device. This environment often includes debuggers, library managers, and other auxiliary tools.
2. A hardware device called a “programmer” that transfers the executable program to the microcontroller itself. In the present context, this process is usually called “burning” or “blowing” the PIC.
3. A circuit or demonstration board in which the program (already loaded onto a PIC microcontroller) can be tested in order to locate defects or confirm its functionality.

In the present chapter, we discuss some of the possible variations in these elements for PIC18F programming and system development.

3.1.1 Embedded Systems

An embedded system is designed for a specific purpose, in contrast to a computer system, which is a general-purpose machine. The embedded system is intended for executing specific and predefined tasks, for example, to control a microwave oven, a TV receiver, or to operate a model railroad. In a general-purpose computer, on the other hand, the user may select among many software applications. For example, the user may run a word processor, a Web browser, or a database management system on the desktop. Because the software in an embedded system is usually fixed and cannot be easily changed, it is called “firmware.”

At the heart of an embedded system is a microcontroller (such as a PIC), sometimes several of them. These devices are programmed to perform one, or, at most, a few tasks. In the most typical case an embedded system also includes one or more "peripheral" circuits that are operated by dedicated ICs or by functionality contained in the microcontroller itself. The term "embedded system" refers to the fact that the programmable device is often found inside another one; for instance, the control circuit is embedded in a microwave oven. Furthermore, embedded systems do not have (in most cases) general-purpose devices such as hard disk drives, video controllers, printers, and network cards.

The control for a microwave oven is a typical embedded system. The controller includes a timer (so that various operations can be clocked), a temperature sensor (to provide information regarding the oven's condition), perhaps a motor (to optionally rotate the oven's tray), a sensor (to detect when the oven door is open), and a set of pushbutton switches to select the various options. A program running on the embedded microcontroller reads the commands and data input through the keyboard, sets the timer and the rotating table, detects the state of the door, and turns the heating element on and off as required by the user's selection. Many other daily devices, including automobiles, digital cameras, cell phones, and home appliances, use embedded systems and many of them are PIC-based.

3.1.2 High- and Low-Level Languages

All microcontrollers can be programmed in their native machine language. The term "machine language" refers to the primitive codes, internal to the CPU, that execute the fundamental operations that can be performed by a particular processor. The processor's fetch/execute cycle retrieves the machine code from program memory and performs the necessary manipulations and calculations. For example, the instruction represented by the binary opcode

```
00000000 00000100
```

clears the watchdog timer register in the PIC18FXX devices.

Programming, loosely defined, refers to selecting, configuring, and storing in program memory a sequence of primitive opcodes so as to perform a specific function or task. The machine language programmer has to manually determine the operation code for each instruction in the program and places these codes, in a specific order, in the designated area of program memory.

Assembly language is based on a software program that recognizes a symbolic language where each machine code is represented by a mnemonic instruction and a possible operand. A program, called an "assembler," reads these instructions and operands from a text file and generates the corresponding machine codes. For example, in order to encode the instruction that clears the watchdog (00000000 00000100 binary in the previous example), the assembly language programmer inserts in the text file the keyword

```
CLRWDT
```

The assembler program reads the programmer's text file, parses these mnemonic keywords and their possible operands, and stores the binary opcodes in a file for later

execution. Because assembly language references the processor opcodes, it is a machine-specific language. An assembler program operates only on devices that have a common machine language, although some minor processor-specific variations can, in some cases, be selectively enabled. Because of their association with the hardware, machine language and assembly language are usually referred to as "low-level languages."

High-level programming languages, such as C, Pascal, and Fortran, provide a stronger level of abstraction from the hardaware device. It is generally accepted that compared to low-level languages, high-level programming is more natural, easier to learn, and simplifies the process of developing program languages. The result is a simpler and more understandable development environment that comes at some penalties regarding performance, hardware control, and program size.

Rather than dealing with registers, memory addresses, and call stacks, a high-level language deals with variables, arrays, arithmetic or Boolean expressions, subroutines and functions, loops, threads, locks, and other abstract concepts. In a high-level language, the design focuses on usability rather than optimal program efficiency. Unlike low-level assembly languages, high-level languages have few, if any, elements that translate directly into the machine's native opcodes.

The term "abstraction penalty" is sometimes used in the context of high-level languages in reference to limitations that are evident when computational resources are limited, maximum performance is required, or hardware control is mandated. In some cases, the best of both worlds can be achieved by coding the noncritical portions of a program mostly in a high-level language while the critical portions are developed in assembly language. This results in mixed-language programs, which are discussed later in this book.

It should be noted that many argue that modern developments in high-level languages, based on well-designed compilers, produce code comparable in efficiency and control to that of low-level languages. Another advantage of high-level languages is that their design is independent of machine structures, and the hardware features of a specific device result in code that can be easily ported to different systems. Finally we should observe that the terms "low-level" and "high-level" languages are not cast in stone: to some, assembly language with the use of macros and other tools becomes a high-level language, while C is sometimes categorized as low-level due to its compact size, direct memory addressing, and low-level operands.

As previously mentioned, the major argument in favor of high-level languages is their ease of use and their faster learning curve. The advantages of assembly language, on the other hand, are better control and greater efficiency. It is true that arguments that favor high-level languages find some justification in the computer world, but these reasons are not always valid in the world of microcontroller programming. In the first place, the microcontroller programmer is not always able to avoid complications and technical details by resorting to a high-level language because the programs relate closely to hardware devices and to electronic circuits. These devices and circuits must be understood at their most essential level if they

are to be controlled and operated by software. For example, consider a microcontroller program that must provide some sort of control baseded on the action of a thermostat. In this case, the programmer must become familiar with temperature sensors, analog-to-digital conversions, motor controls, and so on. This is true whether the program will be written in a low- or a high-level language. For these reasons we have considered both high-level and low-level programming of the microcontrollers is discussed in this book.

3.1.3 Language-Specific Software

Developing programs in a particular programming language requires a set of matching software tools. These software development tools are either generic, that is, suitable for any programming language, or multi-language. Fortunately, for PIC programming, all the necessary software tools are furnished in a single development environment that includes editors, assemblers, compilers, debuggers, library managers, and other utilities. This development package, called MPLAB, discussed in the following sections.

3.2 Microchip's MPLAB

MPLAB is the name of the PIC assembly language development system provided by Microchip. The package is furnished as an integrated development environment (IDE) and can be downloaded from the company's website at www.microhip.com. The MPLAB package is furnished for Windows, Linux, and Mac OS systems. At the time of this writing, the current MPLAB version is 8.86.

3.2.1 MPLAB X

A new implementation of MPLAB is named MPLAB X. This new package, available free on the Microchip website, is described by Microchip as “an integrated environment to develop code for embedded microcontrollers.” This definition matches the one for the conventional MPLAB; however, the MPLAB X package brings many changes to the conventional MPLAB environment. In the first place, MPLAB X is based on the open source NetBeans IDE from Oracle. This has allowed Microchip to add many features and to be able to quickly update the software in the context of a more extensible architecture. Microchip also states that MPLAB X provides many new features that will be especially beneficial to users of 16- and 32-bit microprocessor families, where programs can quickly become extremely complex.

Because MPLAB X is still considered “work in progress,” we have not used it in developing the programs that are part of this book. Furthermore, the expanded features of this new environment have added complications in learning and using this package. For the processors considered in this book, and the scope of the developed software, we have considered these new features an unnecessary complication.

3.2.2 Development Cycle

The development cycle of an embedded system consists of the following steps

1. Define system specifications. This step includes listing the functions that the system is to perform and determining the tests that will be used to validate their operations.
2. Select system components according to the specifications. This step includes locating the microcontroller that best suits the system.
3. Design the system hardware. This step includes drawing the circuit diagrams.
4. Implement a prototype of the system hardware by means of breadboards, wire boards, or any other flexible implementation technology.
5. Develop, load, and test the software.
6. Implement the final system and test hardware and software.

The commercial development of an embedded system is hardly ever the work of a single technician. More typically, it requires the participation of computer, electrical, electronic, and software engineers. Note that, in the present context, we consider computer programmers as software engineers. In addition, professional project managers are usually in charge of the development team.

3.3 An Integrated Development Environment

The MPLAB development system, or integrated development environment, consists of a system of programs that run on a PC. This software package is designed to help develop, edit, test, and debug code for the Microchip microcontrollers. Installing the MPLAB package is usually straightforward and simple. The package includes the following components:

1. MPLAB editor. This tool allows creating and editing the assembly language source code. It behaves very much like any Windows™ editor and contains the standard editor functions, including cut and paste, search and replace, and undo and redo functions.
2. MPLAB assembler. The assembler reads the source file produced in the editor and generates either absolute or relocatable code. Absolute code executes directly in the PIC. Relocatable code can be linked with other separately assembled modules or with libraries.
3. MPLAB linker. This component combines modules generated by the assembler with libraries or other object files, into a single executable file in .hex format.
4. MPLAB debuggers. Several debuggers are compatible with the MPLAB development system. Debuggers are used to single-step through the code, breakpoint at critical places in the program, and watch variables and registers as the program executes. In addition to being a powerful tool for detecting and fixing program errors, debuggers provide an internal view of the processor, which is a valuable learning tool.
5. MPLAB in-circuit emulators. These are development tools that allow performing basic debugging functions while the processor is installed in the circuit.

Figure 3.1 is a screen image of the MPLAB program. The application on the editor window is one of the programs developed later in this book.

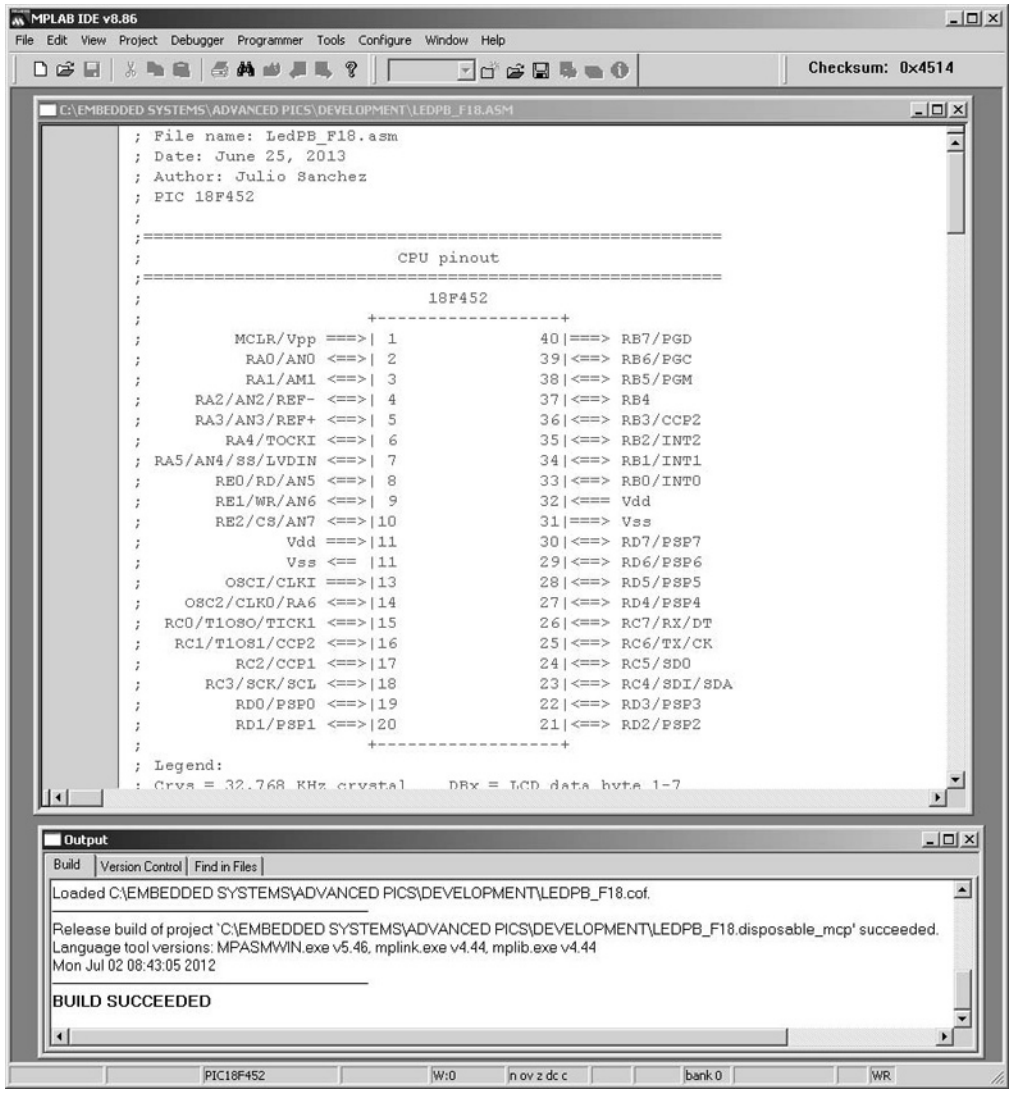


Figure 3.1 *Screen snapshot of MPLAB IDE version 8.64.*

3.3.1 Installing MPLAB

In the normal installation, the MPLAB executable will be placed in the following path:

```
C:\Program Files\Microchip\MPASM Suite
```

Although the installation routine recommends that any previous version of MPLAB be removed from the system, we have found that it is unnecessary, considering that several versions of MPLAB can peacefully coexist.

Once the development environment is installed, the software is executed by clicking the MPLAB IDE icon. It is usually a good idea to drag and drop the icon onto the desktop so that the program can be easily activated.

With the MPLAB software installed, it may be a good idea to check that the applications were placed in the correct paths and folders. Failure to do so produces assembly-time failure errors with cryptic messages. To check the correct path for the software, open the Project menu and select the Set Language Tool Locations command. Figure 3.2 shows the command screen.

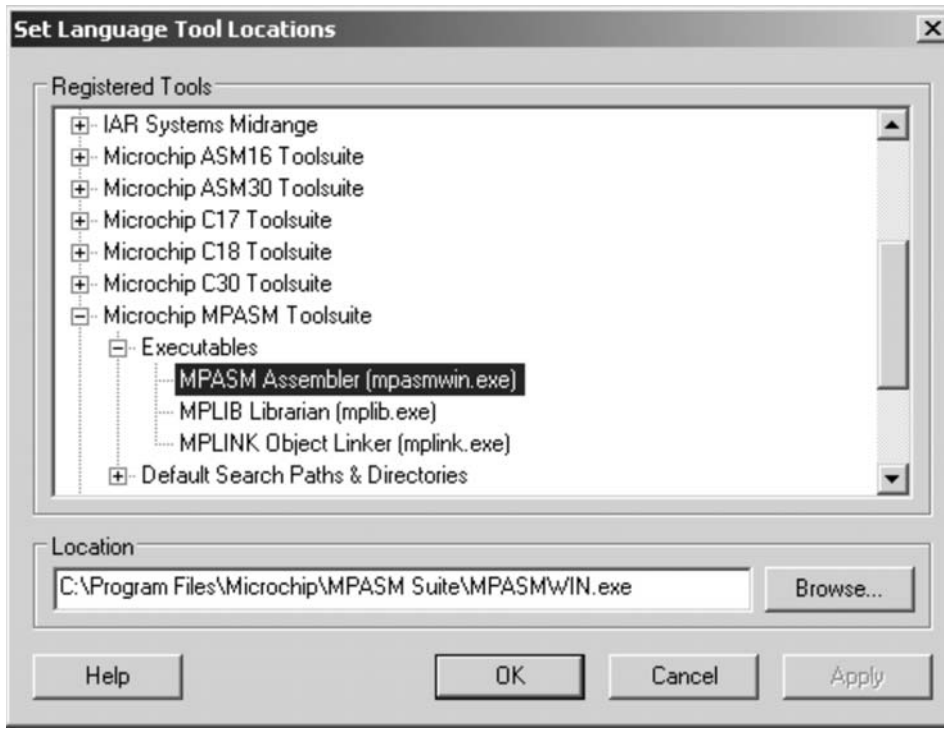


Figure 3.2 *MPLAB 8.64 set language tool locations screen.*

In the Set Language Tool Locations window, make sure that the file location coincides with the actual installation path for the software. If in doubt, use the <Browse> button to navigate through the installation directories until the executable program is located. In this case, mpasmwin.exe. Follow the same process for all the executables in the tool packages that will be used in development. For assembly language programs this is the Microchip MPASM Toolsuite shown in Figure 3.2

3.3.2 Creating the Project

In MPLAB, a project is a group of files generated or recognized by the IDE. Figure 3.3 shows the structure of an assembly language project.

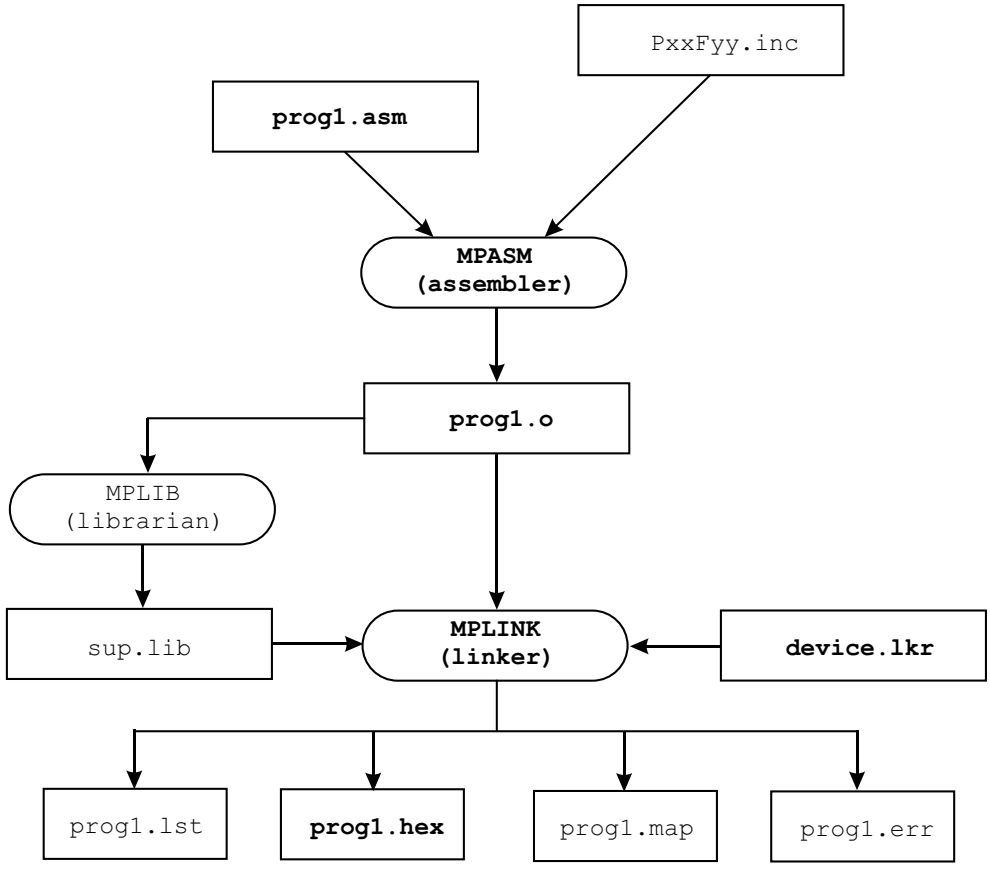


Figure 3-3 *MPLAB project files.*

Figure 3.3 shows an assembly language source file (prog1.asm) and an optional processor-specific include file that are used by the assembler program (MPASM) to produce an object file (prog1.o). Optionally, other sources and other include files may form part of the project. The resulting object file, as well as one or more optional libraries, and a device-specific script file (device.lkr), are then fed to the linker program (MPLINK), which generates a machine code file (prog1.hex) and several support files with listings, error reports, and map files. It is the .hex file that is used to blow the PIC.

Other files, in addition to those in Figure 3.3, may also be produced by the development environment. These vary according to the selected tools and options. For example, the assembler or the linker can be made to generate a file with the extension .cod, which contains symbols and references used in debugging.

Projects can be created using the <New> command in the Project menu. The programmer then proceeds to configure the project manually and add to it the required files. An alternative option, much to be preferred when learning the environment is using the <Project Wizard> command in the project menu. The wizard will prompt you for all the decisions and options that are required, as follows

1. Device selection. Here the programmer selects the PIC hardware for the project, for example, 18F452.
2. Select a language toolsuite. The purpose of this screen is to make sure that the proper development tools are selected and their location is correct.
3. Next, the wizard prompts the user for a name and directory. The Browse button allows for navigating the system. It is also possible to create a new directory at this time.
4. In the next step, the user is given the option of adding existing files to the project and renaming these files if necessary. Because most projects reuse a template, an include file, or other prexisting resources, this can be a useful option.
5. Finally the wizard displays a summary of the project parameters. When the user clicks on the <Finish> button the project is created and programming can begin.

Figure 3.4 is a snapshot of the final wizard screen.

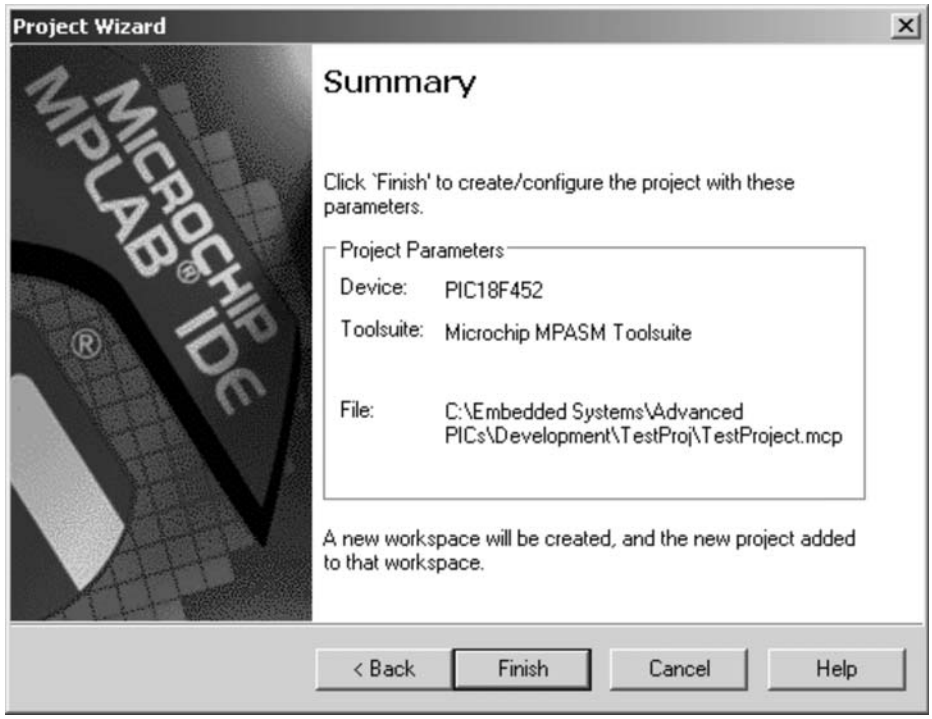


Figure 3-4 *Final screen of the Project Creation Wizard.*

3.3.3 Setting the Project Build Options

The <Build Options: Project> command in the Project menu allows the user to customize the development environment. For projects to be coded in assembly language, the MPASM Assembler Tab on the Build Options for Project screen is one of the the most used. The screen is shown in Figure 3.5.

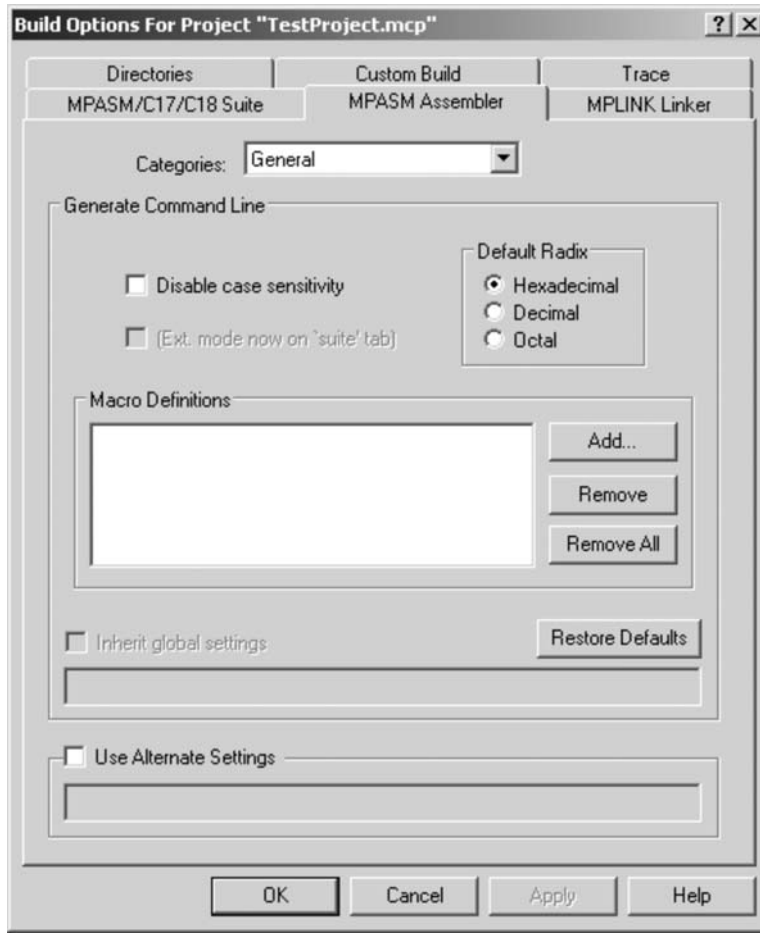


Figure 3-5 *MPASM assembler tab in the build options screen.*

The MPASM Assembler tab allows performing the following customizations:

1. Disable/enable case sensitivity. Normally, the assembler is case-sensitive. Disabling case sensitivity turns all variables and labels to uppercase.
2. Select the default radix. Numbers without formatting codes are assumed to be hex, decimal, or octal according to the selected option. Assembly language programmers usually prefer the hexadecimal radix.
3. The macro definition windows allows adding macro directives. The use of macros is not discussed in this book.
4. The Use Alternate Settings textbox is provided for command line commands in non-GUI environments.
5. The Restore Defaults box turns off all custom configurations.
6. Selecting Output in the Categories window provides command line control options in the output file.

3.3.4 Adding a Source File

The code and structure of an assembly language program is contained in a text file referred to as the source file. The name of the source file is usually descriptive of its purpose or function. At this point in the project, the programmer will usually import an existing file or a template that serves as a skeleton for the project's source file, the name of the project or a variation on this name. Alternatively, the source file can be coded from scratch, although the use of a template saves considerable effort and avoids errors.

Click on the Add New File to Project command in the Project menu to create a new source file from scratch. Make sure that the new file is given the .asm extension and the development environment will automatically save it in the Source Files group of the Project Directory. At this time, a blank editor window will be opened and you can start typing the source code for your program.

Select the Add Files to Project command in the Project menu to import an existing file or a template into the project. In either case, you may have to rename the imported file and remove the old one from the project. If this precaution is not taken, an existing file may be overwritten and its contents lost. For example, to use the template file names PIC18F_Template.asm in creating a source file named PIC18F_Test1.asm, proceed as follows:

1. Make sure that the Project window is displayed by selecting the Project command in the View menu.
2. Right-click the Source Files option in the Project window and select the Add Files...
3. Find the file PIC18F_Template.asm dialog and click on the file name and then the Open button. At this point, the selected file appears in the Source Files box of the Project window.
4. Double click the file name (PIC18F_Template.asm) and MPLAB will open the Editor Window with the file loaded.
5. Rename the template file (PIC18F_Template.asm) by selecting the Editor window, then the Save As command in the MPLAB File menu. Enter the name under which the file is to be saved (PIC18F_Test1.asm in this walkthrough). Click on the Save button.
6. At this point, the file is renamed but not inserted in the Project. Right-click the Source Files option in the Project window and select the Add Files... Click on the PIC18F_Test1.asm file name and on the Open button. The file PIC18F_Test1.asm now appears in the Source Files option of the Project window.
7. Right-click on the PIC18F_Template.asm filename in the Project window and select the Remove command. This removes the template file from the project.

In many cases the MPLAB environment contains duplicate commands that provide alternative ways for achieving the same results, as is the case in the previous walkthrough and in others listed in this book. When there are several ways to obtain the same result, we have tried to select the more intuitive, simpler, and faster one.

3.3.5 Building the Project

Once all the options have been selected, the installation checked, and the assembly language source file written or imported, the development environment can be made to build the project. Building consists of calling the assembler, the linker, and any other support program in order to generate the files shown in Figure 3.3 and any others that may result from a particular project or IDE configuration. The build process is initiated by selecting the <Build All> command in the Project menu. Once the building concludes, a screen labeled Output is displayed showing the results of the build operation. If the build succeeded, the last line of the Output screen will show this result. Figure 3.6 shows the MPLAB program screen after a successful build in the preceding walkthrough.

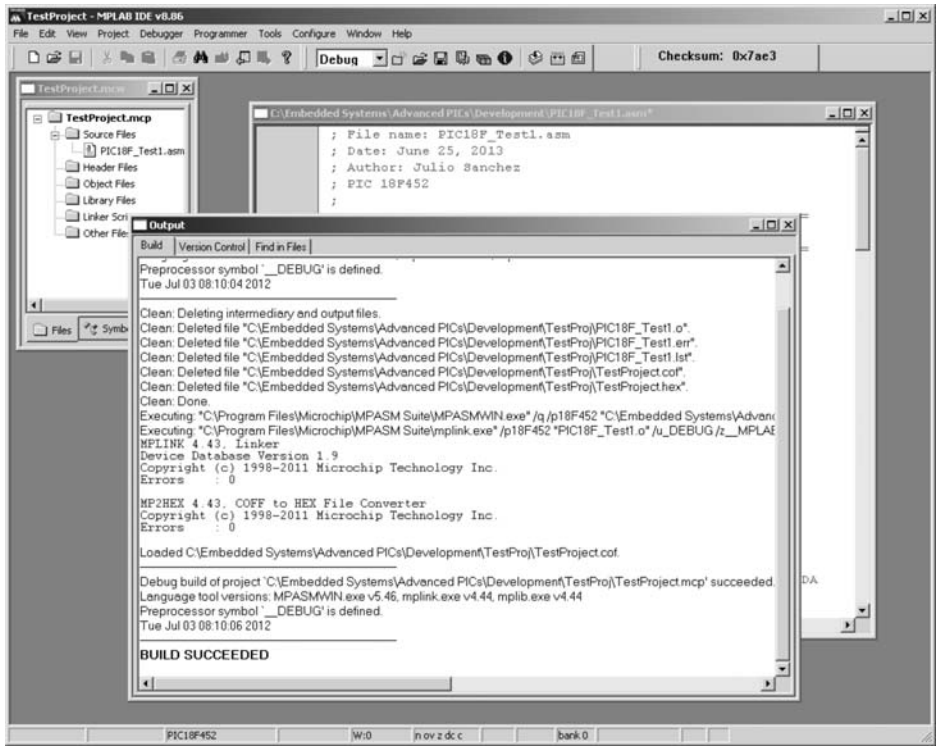


Figure 3-6 *MPLAB program screen showing the Build All command result.*

3.3.6 .hex File

The build process results in several files, depending on the options selected during project definition. One of these files is the executable, which contains the machine codes, addresses, and other parameters that define the program. This is the file that is "blown" in the PIC. The location of the .hex file depends on the option selected during project creation (see Figure 3.5). The Directories tab of the Build Options for Project dialog box contains a group labeled Build Directory Policy, as shown in Figure 3.7.