# Programming 16-Bit PIC Microcontrollers in C

*Learning to Fly the PIC24*

*By*

***Lucio Di Jasio***

ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Newnes is an imprint of Elsevier

Newnes

# Programming 16-Bit PIC Microcontrollers in C

Newnes is an imprint of Elsevier
30 Corporate Drive, Suite 400, Burlington, MA 01803, USA
Linacre House, Jordan Hill, Oxford OX2 8DP, UK

**Library of Congress Cataloging-in-Publication Data**

(Application submitted.)

**British Library Cataloguing-in-Publication Data**
A catalogue record for this book is available from the British Library.

ISBN-13: 978-0-7506-8292-3
ISBN-10: 0-7506-8292-2

For information on all Newnes publications
visit our Web site at www.books.elsevier.com

07 08 09 10    10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

## Dedication

*To Sara*

# Contents

# Preface

Writing this book turned out to be much more work than I had expected and, believe me, I was already expecting a lot. This project would never have been possible if I did not have 110% support and understanding from my wife, Sara. Special thanks also go to Steve Bowling, a friend, a pilot and an expert on Microchip 16-bit architecture, for reviewing the technical content of this book and providing many helpful suggestions for the demonstration projects and hardware experiments. Many thanks go to Eric Lawson for constantly encouraging me to write and for all the time he spent fixing my eternally long-running sentences and my bad use of punctuation. I owe big thanks also to Thang Nguyen, who was first to launch the idea of the book; Joe Drzewiecky and Vince Sheard for patiently listening to my frequent laments and, always working hard on making MPLAB® IDE a better tool; Calum Wilkie and Guy McCarthy for quickly addressing all my questions and offering so much help and insight into the inner workings of the MPLAB C30 compiler and libraries. I would also like to extend my gratitude to all my friends and colleagues at Microchip Technology and the many embedded-control engineers I have been honored to work with over the years. You have so profoundly influenced my work and shaped my experience in the fantastic world of embedded control.

# Introduction

The story goes that I badly wanted to write a book about one of the greatest passions in my life: flying! I wanted to write a book that would convince other engineers like me to take the challenge and live the dream—learn to fly and become private pilots. However, I knew the few hours of actual flying experience I had did not qualify me as a credible expert on the art of flying. So when I had an opportunity to write a book about Microchip's new 16-bit PIC24 microcontrollers, I just could not resist the temptation to join the two things, programming and flying, in one project. After all, learning to fly means following a well-structured process—a journey that allows you to acquire new capabilities and push beyond your limits. It gradually takes you through a number of both theoretical and practical subjects, and culminates with the delivery of the private pilot license. The pilot license, though, is really just the beginning of a whole new adventure—a license to learn, as they say. This compares very well to the process of learning new programming skills, or learning to take advantage of the capabilities of a new microcontroller architecture.

Throughout the book, I will make brief parallels between the two worlds and in the references for each chapter I will add, here and there, some suggestions for reading about flying. I hope I will stimulate your curiosity and, if you happen to have this dream inside you, I will give you that last final push to help make it happen.

## Who should read this book?

This is the part where I am supposed to tell you that you will have a wonderful experience reading this book, that you will have a lot of fun experimenting with the software and hardware projects, and, that you will learn about programming a shiny new 16-bit RISC processor in C, practically from scratch. But, in all honesty, I cannot! This is only partially true. I do hope you will have a lot of fun reading this book and the experiments are…"playful," and you should enjoy them. However, you will need some preparation and hard work in order to be able to digest the material I am presenting at a pace that will accelerate rapidly through the first few chapters.

This book is meant for programmers having a basic to intermediate level of experience, but not for "absolute" beginners. Don't expect me to start with the basics of binary numbers, hexadecimal notation or the fundamentals of programming. However, we will briefly review the basics of C programming as it relates to applications for the latest generation of general-purpose 16-bit microcontrollers, before moving on to more challenging projects. My assumption is that you, the reader, belong to one of four categories:

- Embedded-control programmer: experienced in assembly-language microcontroller programming, but with only a basic understanding of the C language.
- PIC® microcontroller expert: having a basic understanding of the C language.

- Student or professional: with some knowledge of C (or C++) programming for PCs.
- Other SLF (superior life forms): I know programmers don't like to be classified that easily, so I created this special category just for you!

Depending on your level and type of experience, you should be able to find something of interest in every chapter. I worked hard to make sure that every one of them contained both C programming techniques and new hardware-peripheral details. Should you already be familiar with both, feel free to skip to the experts section at the end of the chapter, or consider the additional exercises, book references and links for further research/reading.

These are some of the things you will learn:

- The structure of an embedded-control C program: loops, loops and more loops
- Basic timing and I/O operations
- Basic embedded-control multitasking in C, using the PIC24 interrupts
- New PIC24 peripherals, in no specific order:
  - Input Capture
  - Output Compare
  - Change Notification
  - Parallel Master Port
  - Asynchronous Serial Communication
  - Synchronous Serial Communication
  - Analog-to-Digital Conversion
- How to control LCD displays
- How to generate video signals
- How to generate audio signals
- How to access mass-storage media
- How to share files on a mass-storage device with a PC

## Structure of the book

Similar to a flying course, the book is composed of three parts. The first part contains five small chapters of increasing levels of complexity. In each chapter, we will review one basic hardware peripheral of the PIC24FJ128GA010 microcontroller and one aspect of the C language, using the MPLAB C30 compiler (Student Version included on the CD-ROM). In each chapter, we will develop at least one demonstration project. Initially, such projects will require exclusive use of the MPLAB SIM software simulator (included on the CD-ROM), and no actual hardware will be necessary, although an Explorer 16 demonstration board might be used.

In the second part of the book, containing five more chapters, an Explorer16 demonstration board (or third-party equivalent) will become more critical, as some of the peripherals used will require real hardware to be properly tested.

The third part of the book contains five larger chapters. Each one of them builds on the lessons learned in multiple previous chapters, while adding new peripherals to develop projects of greater complexity. The projects in the third part of the book require the use of the Explorer 16 demonstration board and basic prototyping-skills, too (yes, you might need to use a soldering iron). If you don't want to or you don't have access to basic hardware-prototyping tools, an ad hoc expansion board containing all the circuitry and components necessary to complete all the demonstration projects will be made available on the companion Web site: *http://www.flyingthepic24.com*.

All the source code developed in each chapter is also available for immediate use on the companion CD-ROM.

## What this book is not

This book is not a replacement for the PIC24 datasheet, reference manual and programmer's manual published by Microchip Technology. It is also not a replacement for the MPLAB C30 compiler user's guide, and all the libraries and related software tools offered by Microchip. Copies are available on the companion CD-ROM, but I expect you to download the most recent versions of all those documents and tools from Microchip's Web site (*http://www.microchip.com*). Familiarize yourself with them and keep them handy. I will often refer to them throughout the book, and I might present small block diagrams and other excerpts here and there as necessary. However, my narration cannot replace the information presented in the official manuals. Should you notice a conflict between my narration and the official documentation, ALWAYS refer to the latter. Please do send me an email if a conflict arises. I will appreciate your help and I will publish any corrections and useful hints I receive on the companion Web site: *http://www.flyingthepic24.com*.

This book is also not a primer on the C language. Although a review of the language is given throughout the first few chapters, the reader will find in the references several suggestions on more complete introductory courses and books on the subject.

## Checklists

Pilots, both professional and not, use checklists to perform every single procedure before and during a flight. This is not because the procedures are too long to be memorized or because pilots suffer from more memory problems than others. They use checklists because it is proven that the human memory can fail and that it tends to do so more often when stress is involved. Pilots can perhaps afford fewer mistakes than other proffessionals, and they value safety above their pride.

There is nothing really dangerous that you as a programmer can do or forget to do while developing code for the PIC24. Nonetheless, I have prepared a number of simple checklists to help you perform the most common programming and debugging tasks. Hopefully, they will help you in the early stages, when learning to use the new PIC24 toolset—even later if you are, like most of us, alternating between several projects and development environments from different vendors.

# PART

# I

# CHAPTER 1

# *The first flight*

***In This Chapter***

- *Compiling and linking*
- *Building the first project*
- *PORT initialization*
- *Retesting PORTA*
- *Testing PORTB*

The first flight for every student pilot is typically a blur—a sequence of brief but very intense sensations, including:

- The rush of the first take-off, which is performed by the instructor.
- The white-knuckled, sweaty grip on the yoke while trying to keep the plane flying straight for a couple of minutes, after the instructor gives the standard “anybody that can drive a car can do this” speech.
- Acute motion sickness, as the instructor returns for the landing and performs a sickness-inducing maneuver, called the “side slip.” where it looks like the runway is coming through the side window.

For those who are new to the world of embedded programming, this first chapter will be no different.

## Flight plan

Every flight should have a purpose, and preparing a flight plan is the best way to start.

This is going to be our first project with the PIC24 16-bit microcontroller and, for some of you, the first project with the MPLAB® IDE Integrated Development Environment and the MPLAB C30 language suite. Even if you have never heard of the C language before, you might have heard of the famous “Hello World!” programming example. If not, let me tell you about it.

Since the very first book on the C language, written by Kernighan and Ritchie several decades ago, every decent C-language book has featured an example program containing a single statement to display the words “Hello World” on the computer screen. Hundreds, if not thousands, of books have respected this tradition, and I don’t want this book to be the exception. However, it will have to be just a little different. Let’s be realistic—we are talking about programming microcontrollers because we want to design embedded-control applications. While the availability of a monitor screen is a perfectly safe assumption for any personal computer or workstation, this is definitely not the case in the embedded-control world. For our first embedded application, we better stick to a more basic type of output—a digital I/O pin. In a later and more advanced chapter, we will be able to interface to an LCD display and/or a terminal connected to a serial port. But by then we will have better things to do than writing "Hello World!"

## Preflight checklist

Each flight is preceded by a preflight inspection—simply a walk around the airplane in which we check that, among many other things, gas is in the tank and the wings are still attached to the fuselage. So, let's verify we have all the necessary pieces of equipment ready and installed (from the attached CD-ROM and/or the latest version available for download from Microchip's web site at *http://www.microchip.com/mplab*):

- MPLAB IDE, free Integrated Development Environment
- MPLAB SIM, software simulator
- MPLAB C30, C compiler (free Student Version).

Then, let's follow the "New Project Set-up" checklist to create a new project with the MPLAB IDE:

1. Select "Project→Project Wizard" to activate the new project wizard, which will guide us automatically through the following steps…
2. Select the PIC24FJ128GA010 device, and click Next.
3. Select the MPLAB C30 Compiler Suite and click Next.
4. Create a new folder and name it "Hello"; name the project "Hello Embedded World" and click Next.
5. Simply click Next to the following dialog box—there is no need to copy any source files from any previous projects or directories.
6. Click on Finish to complete the Wizard set-up.

For this first time, let's continue with the following additional steps:

7. Open a new editor window.
8. Type the following three comment lines:

   ```
   //
   //   Hello Embedded World!
   //
   ```

9. Select "File→Save As", to save the file as: `"Hello.c"`.
10. Select "Project→Save" to save the project.

## The flight

It is time to start writing some code. I can see your trepidation, especially if you have never written any C code for an embedded-control application before. Our first line of code is going to be:

```
#include <p24fj128ga010.h>
```

This is not yet a proper C statement, but more of a pseudo-instruction for the preprocessor telling the compiler to read the content of a device-specific file before proceeding any further. The content of the device-specific `".h"` file chosen is nothing more than a long list of the names (and sizes) of all the internal special-function registers (SFRs) of the chosen PIC24 model. If the include file is accurate, those names reflect exactly those being used in the device datasheet. If any doubt, just open the file and take a look—it is a simple text file you can open with the MPLAB editor. Here is a segment of the `p24fj128ga010.h` file where the program counter and a few other special-function registers (SFRs) are defined:

```
...
extern volatile unsigned int  PCL __attribute__((__sfr__));
extern volatile unsigned char PCH __attribute__((__sfr__));
extern volatile unsigned char TBLPAG __attribute__((__sfr__));
extern volatile unsigned char PSVPAG __attribute__((__sfr__));
extern volatile unsigned int  RCOUNT __attribute__((__sfr__));
extern volatile unsigned int  SR __attribute__((__sfr__));
...
```

Going back to our `"Hello.c"` source file, let's add a couple more lines that will introduce you to the `main()` function:

```
main()
{

}
```

What we have now is already a complete, although still empty and pretty useless, C-language program. In between those two curly brackets is where we will soon put the first few instructions of our embedded-control application.

Independently of this function position in the file, whether in the first lines on top or the last few lines in a million-line file, the `main()` function is the place where the microcontroller (program counter) will go first at power-up or after each subsequent reset.

One caveat—before entering the `main()` function, the microcontroller will execute a short initialization code segment automatically inserted by the linker. This is known as the `c0` code. The `c0` code will perform basic housekeeping chores, including the initialization of the microcontroller stack, among other things.

We said our mission was to turn on one or more I/O pins: say PORTA, pins RA0–7. In assembly, we would have used a pair of `mov` instructions to transfer a literal value to the output port. In C it is much easier—we can write an "assignment statement" as in the following example:

```
#include <p24fj128ga010.h>

main()
{
    PORTA = 0xff;
}
```

First, notice how each individual statement in C is terminated with a semicolon. Then notice how it resembles a mathematical equation…it is not!

An assignment statement has a right side, which is computed first. A resulting value is obtained (in this case it was simply a literal constant) and it is then transferred to the left side, which acts as a receiving container. In this case it was a special-function 16-bit register of the microcontroller (the name of which was predefined in the `.h` file).

Note: In C language, by prefixing the literal value with `0x`, we indicate the use of the hexadecimal radix. Otherwise the compiler assumes the default decimal radix. Alternatively, the `0b` prefix can be used for binary literal values, while for historical reasons a single `0` (zero) prefix is used for the octal notation. (Does anybody use octal anymore?)

## Compiling and linking

Now that we have completed the `main()` and only function of our first C program, how do we transform the source into a binary executable?

Using the MPLAB Integrated Development Environment (IDE), it is very easy! It's a matter of a single click of your mouse. This operation is called a Project Build. The sequence of events is fairly long and complex, but it is composed mainly of two steps:

- Compiling: The C compiler is invoked and an object code file (`.o`) is generated. This file is not yet a complete executable. While most of the code generation is complete, all the addresses of functions and variables are still undefined. In fact, this is also called a relocatable code object. If there are multiple source files, this step is repeated for each one of them.
- Linking: The linker is invoked and a proper position in the memory space is found for each function and each variable. Also any number of precompiler object code files and standard library functions may be added at this time as required. Among the several output files produced by the linker is the actual binary executable file (`.hex`).

All this is performed in a very rapid sequence as soon as you select the option "Build All" from the Project menu.

Should you prefer a command-line interface, you will be pleased to learn that there are alternative methods to invoke the compiler and linker and achieve the same results without using the MPAB IDE, although you will have to refer to the MPLAB C compiler User Guide for instructions. In the remainder of this book, we will stick to the MPLAB IDE interface and we will make use of the appropriate checklists to make it even easier.

In order for MPLAB to know which file(s) need to be compiled, we will need to add their names (`Hello.c` in this case) to the project Source Files List.

In order for the linker to assign the correct addresses to each variable and function, we will need to provide MPLAB with the name of a device-specific "linker script" file (`.gld`). Just like the include (`.h`) file tells the compiler about the names (and sizes) of device-specific, special-function registers (SFRs), the linker scripts (`.gld`) file informs the linker about their predefined positions in memory (according to the device datasheet) as well as provides essential memory space information such as: total amount of Flash memory available, total amount of RAM memory available, and their address ranges.

The linker script file is a simple text file and it can be opened and inspected using the MPLAB editor.

Here is a segment of the `p24fj128ga010.gld` file where the addresses of the program counter and a few other special-function registers are defined:

```
 ...
 PCL          = 0x2E;
_PCL          = 0x2E;
 PCH          = 0x30;
_PCH          = 0x30;
 TBLPAG       = 0x32;
_TBLPAG       = 0x32;
 PSVPAG       = 0x34;
_PSVPAG       = 0x34;
 RCOUNT       = 0x36;
_RCOUNT       = 0x36;
 SR           = 0x42;
_SR           = 0x42;
...
```

## Building the first project

Let's review the last few steps required to complete our first demo project:

1. Add the current source file to the "Project Source Files" list.
   There are three possible checklists to choose from, corresponding to three different methods to achieve the same result. This first time we will:
   a) Open the Project window, if not already open, selecting "View→Project".
   b) With the cursor on the editor window, right click to activate the editor pop-up menu.
   c) Select "Add to project".

2. Add the PIC24 "linker script" file to the Project.
   Following the appropriate checklist "Add linker script to project":
   a) Right click on the linker scripts list in the project window.
   b) Select "Add file," browse and select the "`p24fj128ga010.gld`" file found in the `support/gld` subdirectory of MPLAB.

Your Project window should now look similar to Figure 1-1.

3. Select the "Project→Build" function and watch the C30 compiler, followed by the linker, work and generate the executable code as well as a few, hopefully reassuring, messages in the MPLAB IDE Build window.

Note: The "Project Build" checklist contains several additional steps that will help you in future and more complex examples. (See Figure 1-2.)

4. Select "Debugger→Select Tool→MPLAB SIM" to select and activate the simulator as our main debugging tool for this lesson. Note: the "MPLAB SIM debugger set-up" checklist will help you properly configure the simulator.

If all is well, before trying to run the code, let's also open a Watch window and select and add the `PORTA` special-function register to it (type or select `PORTA` in the SFR combo box, and then click on the "Add SFR" button). (See Figure 1-3.)

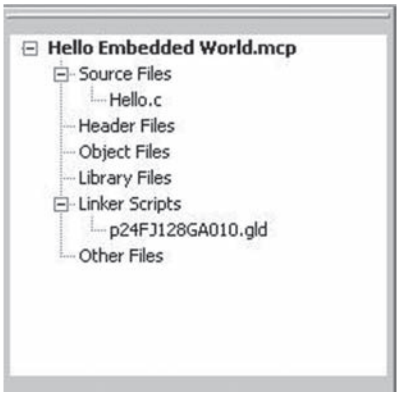
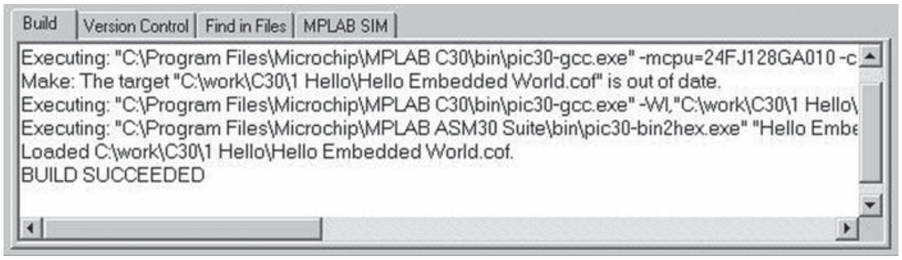


*Figure 1-1. MPLAB IDE Project window set up for the "Hello Embedded World" project.*



```
Executing: "C:\Program Files\Microchip\MPLAB C30\bin\pic30-gcc.exe" -mcpu=24FJ128GA010 -c
Make: The target "C:\work\C30\1 Hello\Hello Embedded World.cof" is out of date.
Executing: "C:\Program Files\Microchip\MPLAB C30\bin\pic30-gcc.exe" -Wl,"C:\work\C30\1 Hello\
Executing: "C:\Program Files\Microchip\MPLAB ASM30 Suite\bin\pic30-bin2hex.exe" "Hello Embe
Loaded C:\work\C30\1 Hello\Hello Embedded World.cof.
BUILD SUCCEEDED
```

*Figure 1-2. MPLAB IDE Output window, Build tab after successfully building a project.*

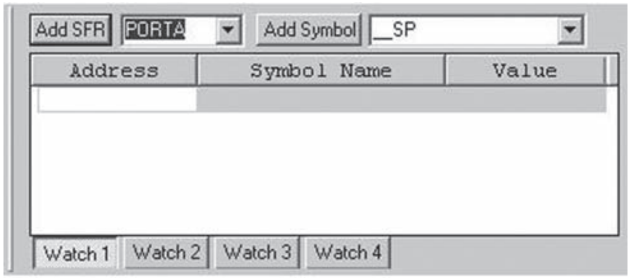


*Figure 1-3. MPLAB IDE Watch window.*

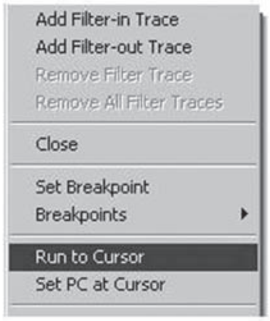


*Figure 1-4. MPLAB IDE Editor context menu (right click).*

5. Hit the simulator Reset button (or select "Debugger→Reset") and observe the contents of `PORTA`. It should be cleared at reset. Then, place the cursor on the line containing the port assignment, inside the main function, and select the "Run to Cursor" option on the right-click menu.

This will let you skip all the C-compiler initialization code (`c0`) and get right to the beginning of our code.

6. Now single-step, (use the Step-Over or Step-In functions) to execute the one and only statement in our first program and observe how the content of `PORTA` changes in the Watch window. Or, notice how nothing happens: surprise!

## PORT initialization

It is time to hit the books, specifically the PIC24FJ128GA datasheet (Chapter 9, for the I/O ports detail). PORTA is a pretty busy, 16-pin wide, port.

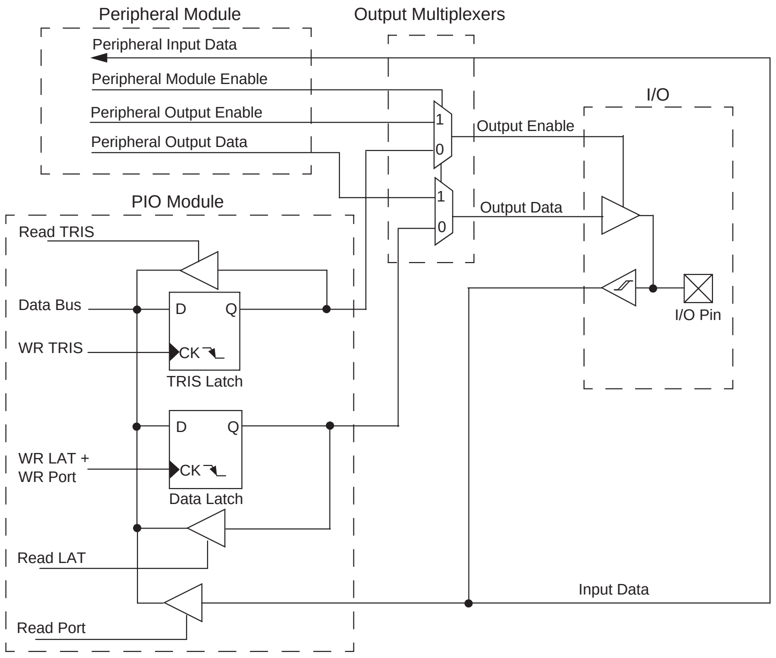


*Figure 1-5. Diagram of a typical PIC24 I/O port.*

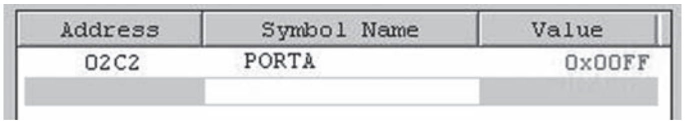
Looking at the pin-out diagrams on the datasheet, we can tell there are many peripheral modules being multiplexed on top of each pin. We can also determine what the default direction is for all I/O pins at reset: they are configured as inputs, which is a standard for all PIC® microcontrollers. The `TRISA` special-function register controls the direction of each pin on PORTA. Hence, we need to add one more assignment to our program, to change the direction of all the pins of PORTA to output, if we want to see their status change:

```
#include <p24fj128ga010.h>
```

```
main()
{
    TRISA = 0;          // all PORTA pins output
    PORTA = 0xff;
}
```

## Retesting PORTA

1. Rebuild the project now.
2. Set the cursor on the `TRISA` assignment.
3. Execute a “Run to Cursor” command to skip all the compiler initialization just as we did before.
4. Execute a couple of single steps and...we have it!



| Address | Symbol Name | Value |
|---|---|---|
| 02C2 | PORTA | 0x00FF |

*Figure 1-6. MPLAB IDE Watch window detail; PORTA content has changed!*

If all went well, you should see the content of `PORTA` change to 0x00FF, highlighted in the Watch window in red. Hello, World!

Our first choice of PORTA was dictated partially by the alphabetical order and partially by the fact that, on the popular Explorer16 demonstration boards, PORTA pins RA0 through RA7 are conveniently connected to 8 LEDs. So if you try to execute this example code on the actual demo board, you will have the satisfaction of seeing all the LEDs turn on, nice and bright.

## Testing PORTB

To complete our lesson, we will now explore the use of one more I/O port, PORTB.

It is simple to edit the program and replace the two PORTA control register assignments with `TRISB` and `PORTB.` Rebuild the project and follow the same steps we did in the previous exercise and…you’ll get a new surprise. The same code that worked for PORTA does not work for PORTB!

Don’t panic! I did it on purpose. I wanted you to experience a little PIC24 migration pain. It will help you learn and grow stronger.

It is time to go back to the datasheet, and study in more detail the PIC24 pin-out diagrams. There are two fundamental differences between the 8-bit PIC microcontroller architecture and the new PIC24 architecture:

- Most of PORTB pins are multiplexed with the analog inputs of the analog-to-digital converter (ADC) peripheral. The 8-bit architecture reserved `PORTA` pins primarily for this purpose—the roles of the two ports have been swapped!

- With the PIC24, if a peripheral module input/output signal is multiplexed on an I/O pin, as soon as the module is enabled, it takes complete control of the I/O pin—independently of the direction (`TRISx`) control register content. In the 8-bit architectures, it was up to the user to assign the correct direction to each pin, even when a peripheral module required its use.

By default, pins multiplexed with "analog" inputs are disconnected from their "digital"input ports. This is exactly what was happening in the last example. All PORTB pins in the PIC24FJ128GA010 are, by default at power-up, assigned an analog input function; therefore, reading PORTB returns all 0s. Notice, though, that the output latch of PORTB has been correctly set although we cannot see it through the `PORTB` register. To verify it, check the contents of the `LATB` register instead.

To reconnect PORTB inputs to the digital inputs, we have to act on the analog-to-digital conversion (ADC) module inputs. From the datasheet, we learn that the special-function register `AD1PCFG` controls the analog/digital assignment of each pin.

**Upper Byte:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
| PCFG15 | PCFG14 | PCFG13 | PCFG12 | PCFG11 | PCFG10 | PCFG9 | PCFG8 |
| bit 15 | | | | | | | bit 8 |

**Lower Byte:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 | R/W-0 |
| PCFG7 | PCFG6 | PCFG5 | PCFG4 | PCFG3 | PCFG2 | PCFG1 | PCFG0 |
| bit 7 | | | | | | | bit 0 |

| | |
|---|---|
| bit 15-0 | **PCFG15:PCFG0:** Analog Input Pin Configuration Control bits |
| | 1 = Pin for corresponding analog channel is configured in Digital mode; I/O port read enabled |
| | 0 = Pin configured in Analog mode; I/O port read disabled, A/D samples pin voltage |

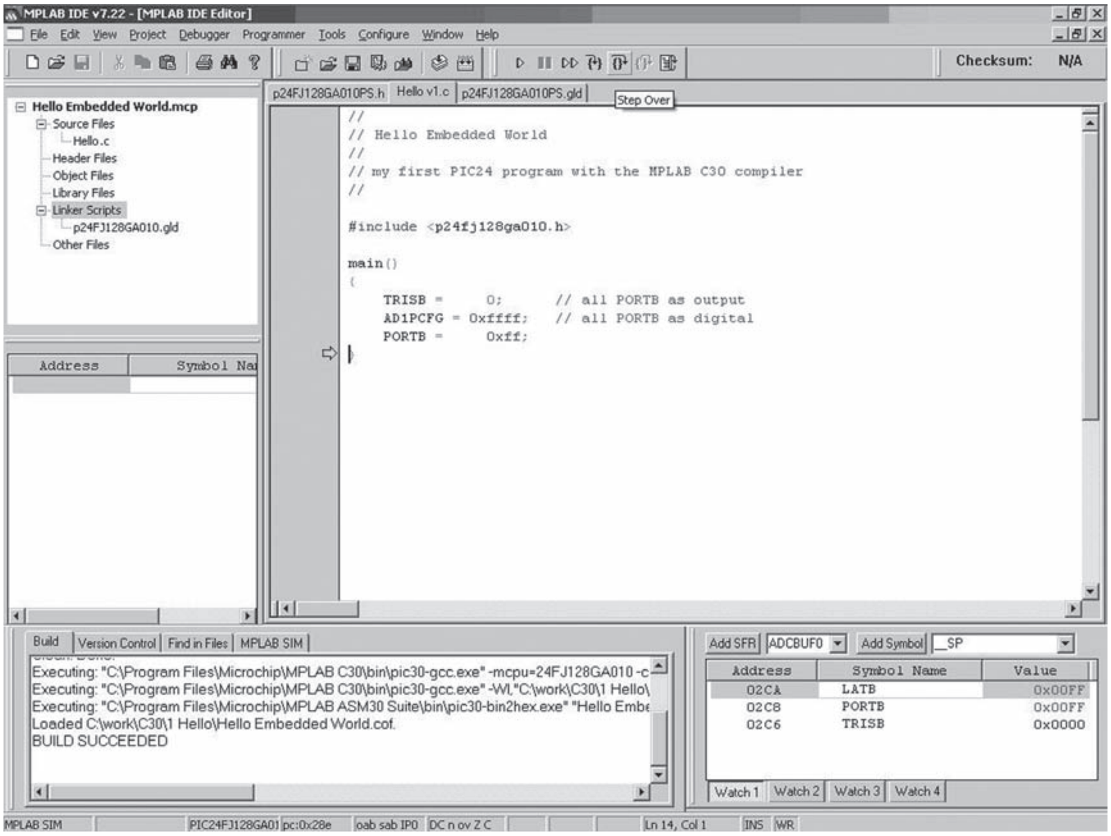
*Figure 1-7.* `AD1PCFG`*: ADC port configuration register.*

Assigning a 1 to each bit in the `AD1PCGF` special-function register will accomplish the task. Our new and complete program example is now:

```
#include <p24fj128ga010.h>

main()
{
   TRISB =       0;            // all PORTB pins output
   AD1PCFG =     0xffff;       // all PORTB pins digital
   PORTB =       0xff;
}
```

This time, compiling and single-stepping through it will give us the desired results.



```
//
// Hello Embedded World
//
// my first PIC24 program with the MPLAB C30 compiler
//

#include <p24fj128ga010.h>

main()
{
    TRISB =     0;      // all PORTB as output
    AD1PCFG = 0xffff;   // all PORTB as digital
    PORTB =     0xff;
}
```

```
Executing: "C:\Program Files\Microchip\MPLAB C30\bin\pic30-gcc.exe" -mcpu=24FJ128GA010 -c
Executing: "C:\Program Files\Microchip\MPLAB C30\bin\pic30-gcc.exe" -Wl,"C:\work\C30\1 Hello\
Executing: "C:\Program Files\Microchip\MPLAB ASM30 Suite\bin\pic30-bin2hex.exe" "Hello Embe
Loaded C:\work\C30\1 Hello\Hello Embedded World.cof.
BUILD SUCCEEDED
```

| Address | Symbol Name | Value |
|---|---|---|
| 02CA | LATB | 0x00FF |
| 02C8 | PORTB | 0x00FF |
| 02C6 | TRISB | 0x0000 |

Figure 1-8. Hello Embedded World Project.

## Post-flight briefing

After each flight, there should be a brief review. Sitting on a comfortable chair in front of a cool glass of water, it's time to reflect with the instructor on what we have learned from this first experience.

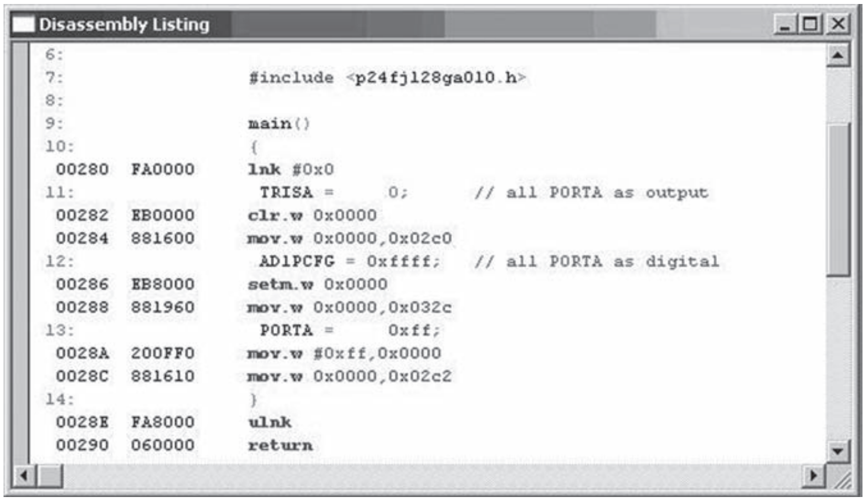
Writing a C program for a PIC24 microcontroller can be very simple, or at least no more complicated than the assembly-language equivalent. Two or three instructions, depending on which port we plan to use, can give us direct control over the most basic tool available to the microcontroller for communication with the rest of the world: the I/O pins.

Also, there is nothing the C30 compiler can do to read our mind. Just like in assembly, we are responsible for setting the correct direction of the I/O pins. And we are still required to study the datasheet and learn about the small differences between the 8-bit PIC microcontrollers we might be familiar with and the new 16-bit breed.

As high-level as the C programming language is touted to be, writing code for an embedded-control device still requires us to be intimately familiar with the finest details of the hardware we use.

## Notes for assembly experts

If you have difficulties blindly accepting the validity of the code generated by the MPLAB C30 compiler, you might find comfort in knowing that, at any given point in time, you can decide to switch to the “Disassembly Listing” view. You can quickly inspect the code generated by the compiler, as each C source line is shown in a comment that precedes the segment of code it generated.



```
                      #include <p24fj128ga010.h>

                      main()
                      {
 00280   FA0000       lnk #0x0
                       TRISA =      0;       // all PORTA as output
 00282   EB0000       clr.w 0x0000
 00284   881600       mov.w 0x0000,0x02c0
                       AD1PCFG = 0xffff;   // all PORTA as digital
 00286   EB8000       setm.w 0x0000
 00288   881960       mov.w 0x0000,0x032c
                       PORTA =      0xff;
 0028A   200FF0       mov.w #0xff,0x0000
 0028C   881610       mov.w 0x0000,0x02c2
                      }
 0028E   FA8000       ulnk
 00290   060000       return
```

*Figure 1-9. Disassembly Listing Window.*

You can even single-step through the code and do all the debugging from this view, although I strongly encourage you not to do so (or limit the exercise to a few exploratory sessions as we progress through the first chapters of this book). Satisfy your curiosity, but gradually learn to trust the compiler. Eventually, use of the C language will give a boost to your productivity and increase the readability and maintainability of your code.

As a final exercise, I encourage you to open the Memory Usage Gauge window—select “View→ Memory Usage Gauge”.

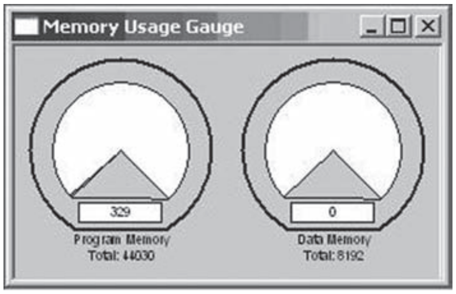


*Figure 1-10. MPLAB IDE Memory Usage Gauge window.*

Don't be alarmed! Even though we wrote only three lines of code in our first example and the amount of program memory used appears to already be up to 300+ bytes, this is not an indication of any inherent inefficiency of the C language. There is a minimum block of code that is always generated (for our convenience) by the C30 compiler. This is the initialization code (`c0`) that we mentioned briefly before. We will get to it, in more detail, in the following chapters as we discuss variables initialization, memory allocation and interrupts.

## Notes for PIC MCU experts

Those of you who are familiar with the PIC16 and PIC18 architecture will find it interesting that most PIC24 control registers, including the I/O ports, are now 16 bits wide. Looking at the PIC24 datasheet, note also how most peripherals have names that look very similar, if not identical, to the 8-bit peripherals you are already familiar with. You will feel at home in no time!

## Notes for C experts

Certainly we could have used the `printf` function from the standard C libraries. In fact the libraries are readily available with the MPLAB C30 compiler. But we are targeting embedded-control applications and we are not writing code for multigigabyte workstations. Get used to manipulating low-level hardware peripherals inside the PIC24 microcontrollers. A single call to a library function, like `printf`, could have added several kilobytes of code to your executable. Don't assume a serial port and a terminal or a text display will always be available to you. Instead, develop a sensibility for the "weight" of each function and library you use in light of the limited resources available in the embedded-design world.

## Tips and tricks

The PIC24FJ family of microcontrollers is based on a 3V CMOS process with a 2.0V to 3.6V operating range. As a consequence, a 3V power supply (Vdd) must be used and this limits the output voltage of each I/O pin when producing a logic high output. However, interfacing to 5V legacy devices and applications is really simple:

- To drive a 5V output, use the `ODCx` control registers (`ODCA` for PORTA, `ODCB` for PORTB and so on…) to set individual output pins in open-drain mode and connect external pull-up resistors to a 5V power supply.
- Digital input pins instead are already capable of tolerating up to 5V. They can be connected directly to 5V input signals.

Be careful with I/O pins that are multiplexed with analog inputs though—they cannot tolerate voltages above Vdd.

## Exercises

If you have the Explorer16 board:

1. Use the ICD2 Debugging Checklist to help you prepare the project for debugging.
2. To test the PORTA example, connect the Explorer16 board and check the visual output on LED0–7.
3. To test the PORTB example, connect a voltmeter (or DMM) to pin RB0 and watch the needle move as you single-step through the code.

## Books

- Kernighan, B. and Ritchie, D., "**The C Programming Language**," Prentice-Hall, Englewood Cliffs, NJ.

  When you read or hear a programmer talk about the "K&R," they mean this book!
  Also known as "the white book," the C language has evolved since the first edition of this book was published in 1978! The second edition (1988) includes the more recent ANSI C standard definitions of the language, which is closer to the standard the MPLAB C30 compiler adheres to (ANSI90).

- "**Private Pilot Manual**," Jeppesen Sanderson, Inc., Englewood, CO.

  This is "the" reference book for every student pilot. Highly recommended, even if you are just curious about aviation.

## Links

- *http://en.wikibooks.org/wiki/C_Programming*

  This is a Wiki-book on C programming. It's convenient if you don't mind doing all your reading online. Hint: look for the chapter called "A taste of C" to find the omnipresent "Hello World!" exercise.

CHAPTER 2

# A loop in the pattern

***In This Chapter***

- `while` *loops*
- *An animated simulation*
- *Using the Logic Analyzer*

The “pattern” is a standardized rectangular circuit, where each pilot flies in a loop. Every airport has a pattern of given (published) altitude and position for each runway. Its purpose is to organize traffic around the airport and its working is not too dissimilar to how a roundabout works. All airplanes are supposed to circle in a given direction consistent with the prevailing wind at the moment. They all fly at the same altitude, so it is easier to visually keep track of each other’s position. They all talk on the radio on the same frequencies, communicating with a tower if there is one, or among one another with the smaller airports. As a student pilot, you will spend quite some time, especially in the first few lessons, flying in the pattern with your instructor to practice continuous sequences of landings immediately followed by take-offs (touch-and-goes), refining your newly acquired skills. As a student of embedded programming, you will have a loop of your own to learn—the main loop.

## Flight plan

Embedded-control programs need a framework, similar to the pilots’ pattern, so that the flow of code can be managed. In this lesson, we will review the basics of the loops syntax in C and we’ll also take the opportunity to introduce a new peripheral module: the 16-bit Timer1. Two new MPLAB® SIM features will be used for the first time: the “Animate” mode and the “Logic Analyzer.”

## Preflight checklist

For this second lesson, we will need the same basic software components installed (from the attached CD-ROM and/or the latest versions, available for download from Microchip’s website) and used before, including:

- MPLAB IDE, Integrated Development Environment
- MPLAB SIM, software simulator
- MPLAB C30 compiler (Student Version)

We will also reuse the “New Project Set-up” checklist to create a new project with the MPLAB IDE:

1. Select “Project→Project Wizard”, to start creating a new project.
2. Select the PIC24FJ128GA010 device, and click Next.
3. Select the MPLAB C30 compiler suite and click Next.
4. Create a new folder and name it “Loop.” name the project “A Loop in the Pattern,” and click Next.
5. There is no need to copy any source files from the previous lessons; click Next once more.
6. Click Finish to complete the Wizard set-up.

This will be followed by the “Adding Linker Script file” checklist, to add the linker script “`p24fj128ga010.gld`” to the project. It can typically be found in the MPLAB IDE installation directory “`C:/Program Files/Microchip/`”, within the subdirectory “`MPLAB C30/support/gld/`”.

After completing the “Create New File and Add to Project” checklist:

7. Open a new editor window.
8. Type the main program header:

```
//
//   A loop in the pattern
//
```

9. Select “Project→AddNewFiletoProject”, to save the file as: `“loop.c”` and have it automatically added to the project source files list.
10. Save the project.

## The flight

One of the key questions that might have come to mind after working through the previous lesson is “What happens when all the code in the `main()` function has been executed?” Well, nothing really happens, or at least nothing that you would not expect. The device will reset, and the entire program will execute again…and again.

In fact, the compiler puts a special software reset instruction right after the end of the `main()` function code, just to make sure. In embedded control we want the application to run continuously, from the moment the power switch has been flipped on until the moment it is turned off. So, letting the program run through entirely, reset and execute again might seem like a convenient way to arrange the application so that it keeps repeating as long as there is “juice.” This option might work in a few limited cases, but what you will soon discover is that, running in this “loop.” you develop a “limp.” Reaching the end of the program and executing a reset takes the microcontroller back to the very beginning to execute all the initialization code, including the `c0` code segment briefly mentioned in the previous lesson. So, as short as the initialization part might be, it will make the loop very unbalanced. Going through all the special-function register and global-variables initializations each time is probably not necessary and it will certainly slow down the application. A better option is to design an application “main loop.” Let’s review the most basic loop-coding options in C first.

## `While` loops

In C there are at least three ways to code a loop; here is the first—the `while` loop:

```
while ( x)
{
    // your code here...
}
```

Anything you put between those two curly brackets ({}) will be repeated for as long as the logic expression in parenthesis `( x)` returns a true value. But what is a logic expression in C?

First of all, in C there is no distinction between logic expressions and arithmetic expressions. In C, the Boolean logic TRUE and FALSE values are represented just as integer numbers with a simple rule:

- FALSE is represented by the integer 0.
- TRUE is represented by *any* integer except 0.

So `1` is true, but so are `13` and `-278`. In order to evaluate logic expressions, a number of logic operators are defined, such as:

| | |
|---|---|
| `\|\|` | the logic OR operator, |
| `&&` | the logic AND operator, |
| `!` | the logic NOT operator. |

These operators consider their operands as logical (Boolean) values using the rule mentioned above, and they return a logical value. Here are some trivial examples:

(when `a = 17` and `b = 1`, or in other words they are both true)

| | |
|---|---|
| `( a \|\| b)` | is true, |
| `( a && b)` | is true |
| `( !a)` | is false |

There are, then, a number of operators that compare numbers (integers of any kind and floating-point values, too) and return logic values. They are:

| | |
|---|---|
| `==` | the "equal-to" operator; notice it is composed of two equal signs to distinguish it from the "assignment" operator we used in the previous lesson, |
| `!=` | the "NOT-equal to" operator, |
| `>` | the "greater-than" operator, |
| `>=` | the "greater-or-equal to" operator, |
| `<` | the "less-than" operator, |
| `<=` | the "less-than-or-equal to" operator. |

Here are some examples:

assuming `a = 10`

| | |
|---|---|
| `( a > 1)` | is true |
| `(-a >= 0)` | is false |

```
( a == 17) is false
( a != 3)  is true
```

Back to the `while` loop, we said that as long as the expression in parentheses produces a true logic value (that is any integer value but 0), the program execution will continue around the loop. When the expression produces a false logic value, the loop will terminate and the execution will continue from the first instruction after the closing curly bracket.

Note that the expression is evaluated first, before the curly bracket content is executed (if ever), and is then reevaluated each time.

Here are a few curious loop examples to consider:

```
 While ( 0)
{
   // your code here...
}
```

A constant false condition means that the loop will never be executed. This is not very useful. In fact I believe we have a good candidate for the "world's most useless code" contest!

Here is another example:

```
 while ( 1)
{
   // your code here...
}
```

A constant true condition means that the loop will execute forever. This is useful, and is in fact what we will use for our main program loops from now on. For the sake of readability, a few purists among you will consider using a more elegant approach, defining a couple of constants:

```
#define TRUE      1
#define FALSE     0
```

and using them consistently in their code, as in:

```
While ( TRUE)
{
   // your code here…
}
```

It is time to add a few new lines of code to the `"loop.c"` source file now, and put the `while` loop to good use.

```
#include <p24fj128ga010.h>
```

```
main()
{
    // init the control registers
    TRISA =    0xff00;// PORTA pin 0..7 as output

    // application main loop
    while( 1)
    {
        PORTA = 0xff; // turn pin 0-7 on
        PORTA = 0;            // turn all pin off
    }
}
```

The structure of this example program is essentially the structure of every embedded-control program written in C. There will always be two main parts:

- The initialization, which includes both the device peripherals initialization and variables initialization, executed only once at the beginning.
- The main loop, which contains all the control functions that define the application behavior, and is executed continuously.

## An animated simulation

Use the Project Build checklist to compile and link the `"loop.c"` program. Also use the "MPLAB SIM simulator set-up" checklist to prepare the software simulator.

To test the code in this example with the simulator, I recommend you use the "Animate" mode (Debugger→Animate). In this mode, the simulator executes one C program line at a time, pausing for ½ second after each one to give us the time to observe the immediate results. If you add the `PORTA` special-function register to the Watch window, you should be able to see its value alternating rhythmically between 0xff and 0x00.

The speed of execution in Animate mode can be controlled with the "Debug→Settings" dialog box, selecting the "Animation/Real Time Updates" tab, and modifying the "Animation Step Time" parameter, which by default is set to 500 ms. As you can imagine, the "Animate" mode can be a valuable and entertaining debugging tool, but it gives you quite a distorted idea of what the actual program execution timing will be. In practice, if our example code was to be executed on a real hardware target, say an Explorer16 demonstration board (where the PIC24 is running at 32 MHz), the LEDs connected to the PORTA output pins would blink too fast for our eyes to notice. In fact, each LED would be turned on and off several million times each second.

To slow things down to a point where the LEDs would blink nicely just a couple of times per second, I propose we use a timer, so that in the process we learn to use one of the key peripherals integrated in all PIC24 microcontrollers. For this example, we will choose the first timer, Timer1, of the five timers available inside the PIC24FJ128GA010. This is one of the most flexible and simple peripheral modules. All we need to do is take a quick look at the PIC24 datasheet, check the block diagram and the details of the Timer1 control registers, and find the ideal initialization values.

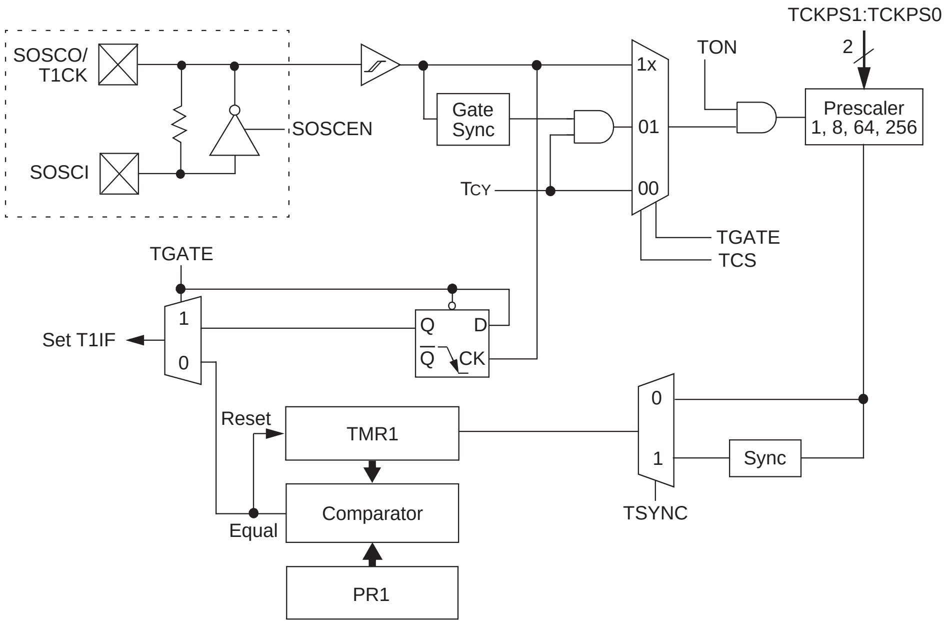


*Figure 2-1. 16-bit Timer1 Module block diagram.*

We quickly learn that there are three special-function registers that control most of the Timer1 functions. They are:

- `TMR1`, which contains the 16-bit counter value.
- `T1CON`, which controls activation and the operating mode of the timer.
- `PR1`, which can be used to produce a periodic reset of the timer (not required here).

We can clear the `TMR1` register to start counting from zero.

```
TMR1 = 0;
```

Then we can initialize `T1CON` so that the timer will operate in a simple configuration where:

- Timer1 is activated: `TON = 1`
- The main MCU clock serves as the source (Fosc/2): `TCS = 0`
- The prescaler is set to the maximum value (1:256): `TCKPS = 11`
- The input gating and synchronization functions are not required, since we use the MCU internal clock directly as the timer clock: `TGATE = 0, TSYNC = 0`
- We do not worry about the behavior in IDLE mode: `TSIDL = 0 (default)`

**Upper Byte:**

| R/W-0 | U-0 | R/W-0 | U-0 | U-0 | U-0 | U-0 | U-0 |
|---|---|---|---|---|---|---|---|
| TON | — | TSIDL | — | — | — | — | — |
| bit 15 | | | | | | | bit 8 |

**Lower Byte:**

| U-0 | R/W-0 | R/W-0 | R/W-0 | U-0 | R/W-0 | R/W-0 | U-0 |
|---|---|---|---|---|---|---|---|
| — | TGATE | TCKPS1 | TCKPS0 | — | TSYNC | TCS | — |
| bit 7 | | | | | | | bit 0 |

*Figure 2-2. T1CON: Timer1 control register.*

Once we assemble all the bits in a single 16-bit value to assign to `T1CON`, we get:

```
T1CON = 0b1000000000110000;
```

or, in a more compact hexadecimal notation:

```
T1CON = 0x8030;
```

Once we are done initializing the timer, we enter a loop where we wait for `TMR1` to reach the desired value set by the constant `DELAY`.

```
while( TMR1 < DELAY)
{
    // wait
}
```

Assuming a 32-MHz clock will be used, we need to assign quite a large value to DELAY so as to obtain a delay of about a quarter of a second. In fact the following formula dictates the total delay time produced by the `TMR1` loop:

`Tdelay` = (2/`Fosc`) * 256 * `DELAY`

With `Tdelay` = 256 ms and resolving for `DELAY`, we obtain the value 16,000:

```
#define DELAY 16000
```

By putting two such delay loops in front of each `PORTA` assignment inside the main loop, we get our latest and best code example:

```
#include <p24fj128ga010.h>

#define DELAY      16000

main()
{
    // init the control registers
    TRISA = 0xff00;              // PORTA pin 0..7 as output
    T1CON = 0x8030;      // TMR1 on, prescaler 1:256 Tclk/2

    // main application loop
    while( 1)
    {
```