



INCLUDES

FREE
NEWNES ONLINE
MEMBERSHIP

PROGRAMMING 16-BIT PIC MICROCONTROLLERS IN C

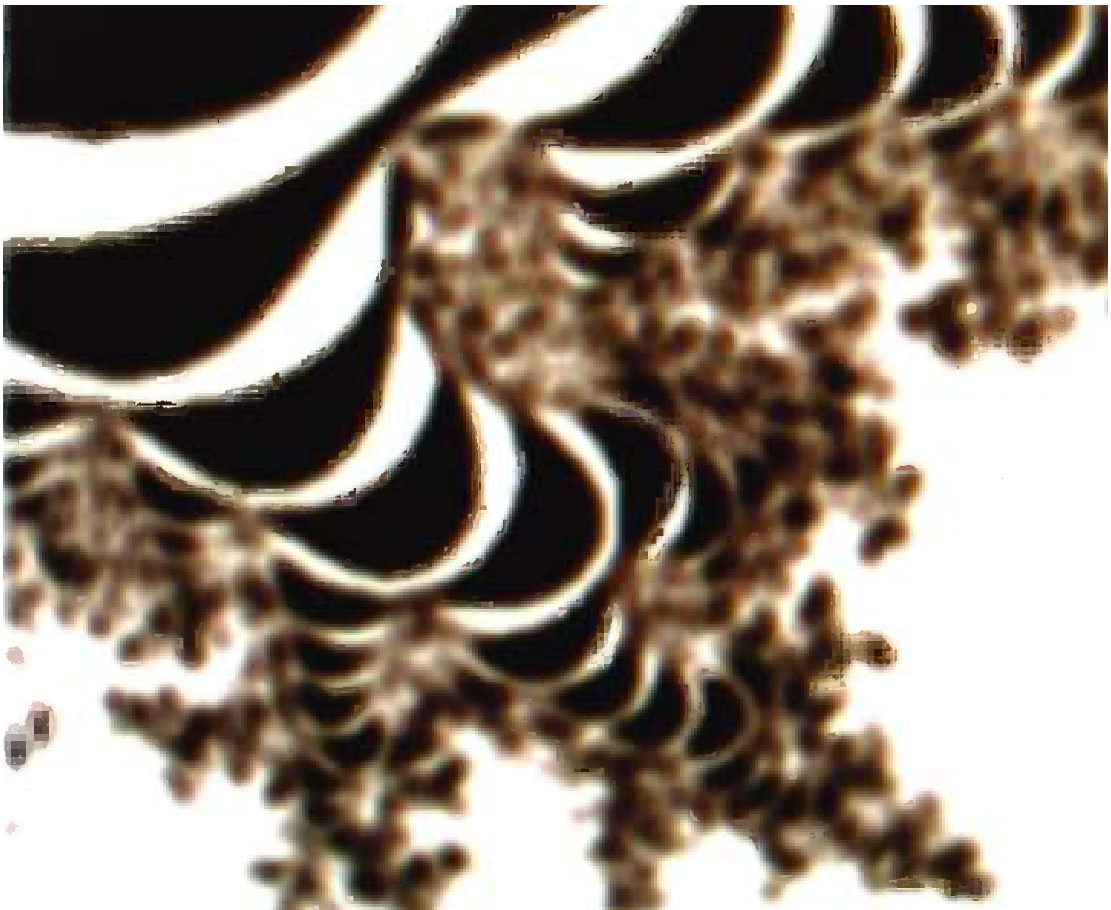
Learning to Fly the PIC 24

Second Edition

- **Expert Assembly Programmers:** Learn how to write embedded control applications in C
- **Expert 8-bit Programmers:** Learn how to boost your applications with a powerful 16-bit architecture
- **Explores the world of embedded control** experimenting with analog and digital peripherals, graphic displays, video and sound

Lucio Di Jasio

Programming 16-Bit PIC Microcontrollers in C



Programming 16-Bit PIC Microcontrollers in C

Learning to Fly the PIC 24

Second edition

Lucio Di Jasio



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

Newnes is an imprint of Elsevier



Newnes is an imprint of Elsevier
The Boulevard, Langford Lane, Kidlington, Oxford, OX5 1GB, UK
225 Wyman Street, Waltham, MA 02451, USA

Copyright © 2012 Elsevier Inc. All rights reserved

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise without the prior written permission of the publisher

Permissions may be sought directly from Elsevier's Science & Technology Rights Department in Oxford, UK: phone (+44) (0) 1865 843830; fax (+44) (0) 1865 853333; email: permissions@elsevier.com. Alternatively, visit the Science and Technology Books website at www.elsevierdirect.com/rights for further information

Notice

No responsibility is assumed by the publisher for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions or ideas contained in the material herein. Because of rapid advances in the medical sciences, in particular, independent verification of diagnoses and drug dosages should be made

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

Library of Congress Cataloging-in-Publication Data

A catalog record for this book is available from the Library of Congress

ISBN: 978-1-85617-870-9

For information on all Newnes publications
visit our website at www.newnespress.com

Typeset by MPS Limited, a Macmillan Company, Chennai, India
www.macmillansolutions.com

Printed and bound in the United States of America

11 12 13 14 15 10 9 8 7 6 5 4 3 2 1

Working together to grow
libraries in developing countries

www.elsevier.com | www.bookaid.org | www.sabre.org

ELSEVIER

BOOK AID
International

Sabre Foundation

This book is dedicated—
To Sara and Luca

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 have never been possible if I did not have 110% support and understanding from my wife. Special thanks also go to Steve Bowling, a friend, a pilot and an expert of 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 to fix 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, always working hard on making MPLAB a better tool; Calum Wilkie and Guy McCarthy for addressing quickly all my questions and offering so much help and insight into the inner workings of the MPLAB C compiler and libraries. Among the technical reviewers of the second edition, I would like to add special thanks to Stewart Cording, Tom Mignone, and Wayne Duquaine who helped me adapt the code to the many new families of PIC24 devices. But I would like to extend my gratitude to all my friends, ex-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 to the Second Edition

In the six years since I completed the first manuscript of this book, there has been a tremendous evolution of the PIC24 architecture and all the supporting set of tools and documentation. What seemed like a promising, if only embryonic, new product line has become one of the main driving forces in Microchip Technology's portfolio. The number of different models available has long passed the 100 mark. It's been growing up toward larger and larger memory sizes and more complex devices as well as down to smaller, simpler devices in tiny packages and with minuscule power consumption figures where, once upon a time, it used to be the undisputed sole domain of the smallest of the 8-bit PIC® architectures.

Back then, when looking for evaluation boards, there was only one choice: the Explorer16 and that is what I had to use for the first edition of this book. Today the choice has been expanded to hundreds of new options offered by numerous and creative third parties worldwide. Interestingly, after much consideration, my final choice for this second edition has been to remain focused mainly on the Explorer16 demonstration board although the reader will be able to reproduce most/all examples and exercises on many other equivalent tools. This choice can be considered a testament to the success of the board in itself and the huge portfolio of expansion boards – known today as the PICTail Plus boards – that complete it. The Explorer16 today can be used to evaluate all 16-bit and 32-bit devices (dsPIC® digital signal controllers included) available in packages of 64 pin or higher, with clock frequencies that have reached 80MHz at the time of this writing and will certainly pass that mark in the coming years.

The same could be said for the programming and debugging tools. When once upon a time the ICD2 used to be the one and only choice, today there is a range of tools, from the simple and inexpensive PICKit3 to the fast and sophisticated REAL ICE, that can cover with agility the entire portfolio of 600+ models of 8-bit, 16-bit and 32-bit microcontrollers offered by Microchip.

So for this second edition my tool of choice has shifted (only slightly) to the generic PICKit3, for its cost, or the ICD3, for its speed, preferring them to more PIC24-specific tools of even lower cost (PIC24F Starter kit, PIC24H Stick...) considering that the work done in this book will represent only a starting point and the PICKit3 and ICD3 in circuit debuggers will have

the ability to continue to support you well beyond these 15 chapters and into the real world of embedded control.

As per the MPLAB Integrated Development Environment, the evolution in these years has been relentless but incremental for the most part. In the last few months, I had the opportunity to test drive the new MPLAB X platform, an almost complete redesign of the tool around an open source platform known as NetBeans. MPLAB X offers tons of new features but also runs on a wider range of operating systems (including MAC OS X, and Linux at last). Most importantly it brings the promise of a greater stability and robustness.

Every chapter in this book has been revised to accommodate for the new MPLAB X interface. In most cases, this has resulted in a simplification of what used to be long sequences of operations, previously detailed in extensive checklists, and now reduced to single mouse clicks.

Finally let me add a word about the MPLAB C compiler, the PIC24F peripheral libraries and the Microchip Application Library. Here the evolution has been steady but nothing short of spectacular. The work on the compiler (now based on the open source GNU GCC v4 project) has been relentless and the real proof of quality of the results today can be quickly summarized in two bullets:

- None of the examples in this book required a single line of assembly code! This is in stark contrast to the first edition, when in several chapters I had to augment the examples with numerous and, at times, cryptic assembly code workarounds.
- Both performance and code size figures had to be corrected in most/all chapters as the compiler produced much tighter code (up to 30% smaller) and faster code (removing the need for extensive hand optimizations) while utilizing only optimization levels 0 and 1 available on all the evaluation (as in free) copies of the compiler.

The PIC24 peripheral libraries have remained an interesting but mixed bag from my point of view. So I have been picking and choosing, very selectively, chapter by chapter when, why and what to take from them. The Microchip Application Libraries (MAL), a large collection of modules that used to be offered as separate application notes (USB, Ethernet, Graphic displays...) deserve a separate mention. Unfortunately the size of the library and the complexity of the subjects covered made it impossible for me to add in this book more than a few hints here and there, and links at the end of related chapters. Each library in the MAL would have perhaps required an entire book dedicated to each subject.

So in the end, far from a definitive work, as too many books claim to be nowadays, this second edition of the “Flying PIC24” is more than ever similar to a private pilot license, a true license to learn, merely the beginning of an adventure that never ends.

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. The thing is that 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 takes you gradually 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 so 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 had 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 you will learn about C programming a shiny new 16-bit RISC processor, practically from scratch. But, in all honesty I cannot! This is only partially true – I really hope you will have a lot of fun reading it and the experiments are ... “playful” and you should enjoy them. However you will need quite 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 of a basic to intermediate level of experience, but not for “absolute” beginners; so don’t expect me to start with the basics of the binary numbers, the hexadecimal notation or the fundamentals of programming. Although, we will briefly review

the basics of C programming as it relates to the 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 microcontrollers programming, but with only a basic understanding of the C language.
- PIC[®] microcontroller expert: with 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 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 peripherals 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 in 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 in 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.

In the third part of the book, there are 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.flyingpic24.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. But, 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. However please send me an email if a conflict arises, I will appreciate your help and I will publish any correction and useful hint I will receive on the companion Web site: <http://www.flyingpic24.com>

This book is also not a primer on the C language. Although a review of the language is performed 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 tends to do so more often when stress is involved. Pilots can perhaps afford less mistakes than other categories, 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 or later if you are, like most of us, alternating between several projects and development environments from different vendors.

New Project Setup	
File > NewProject...	Start
Step 1: Choose Project	Microchip Embedded, C/ASM Standalone
Step 2: Device	PIC24FJ128GA010
Step 3: Select Header	Next
Step 4: Select Tool	PICKit3, ICD3...
Step 5: Select Compiler	C24, C30 or XC16
Step 6: Select Project Name & Folder	Type new name here
Complete Wizard	Click on Finish
Continue adding Include Directory	
Add Include Directory	
Run > Select Project Configuration	Customize
Category	pic30-gcc
Option categories	General
Include directories	click on browse button “...”
Create	Enter “.././include”
Close	Click OK twice
Create New Empty file and Add to Project	
File > New File	Start
Category	Other, Empty File
File Name	name
Folder	Browse or enter
Complete Wizard	Finish
Adding Existing Files to a Project	
Window Menu	
Project Window	
Context Menu	Projects (if not already open) or CTRL + 1 Right Click on Source Files or Header Files Select Add Existing Item
Removing Files from a Project	
Window Menu	Projects (if not already open) or CTRL + 1
Project Window	Right Click on Source File or Header File
Context Menu	Remove from Project
Warning:	Do NOT use the DELETE Button!
Renaming Files	
Window Menu	Projects (if not already open) or CTRL + 1
Project Window	Right Click on Source File or Header File
Context Menu	Rename...
Renaming Projects	
Window Menu	Projects (if not already open) or CTRL + 1
Project Window	Right Click on Project top folder
Context Menu	Rename Project

Project Build for Debugging and Launch	
In Circuit Debugger	Connect
Debug > Debug Project	Select or CTRL + F5
Project Build for Release and Launch	
In Circuit Debugger	Connect
Run > Run Project	Select or F6
Build Project	
Run > Batch Project Build	Select
Clean & Build Project	
Window Menu	Option 1
Project Window	Projects (if not already open) or CTRL + 1
Context Menu	Right Click on Project top folder Clean and Build
Clean & Build Project	
View Toolbar	Option 2
Clean and Build Icon	Select Run (if Build and Run buttons not visible) Select or SHIFT + F11
Access Project Properties	
Run > Set Project Configuration	Option 1
	Customize
Access Project Properties	
	Option 2
View Toolbar	Select Run (if Build and Run buttons not visible)
Configuration Combo box	Select Configure
Access Project Properties	
	Option 3
Window Menu	Projects (if not already open) or CTRL + 1
Project Window	Right Click on Project top folder
Context Menu	Set Configuration > Customize
Enabling Memory Summary in Output window	
Access Project Properties	Option 1, 2 or 3
Category	pic30-ld
Options Category	Diagnostics
Display Memory Usage	Checked
Selecting Compiler Optimization Level	
Access Project Properties	Option 1, 2 or 3
Category	pic30-gcc
Options Category	Optimization
Optimization level	0- debug, 1- free, (2-3 licensed only)

PIC24FJ family characteristics

Vdd range	2.0V to 3.6V
Digital input output pins	5V tolerant
Analog input pins	Vdd + 0.3V max

MPLAB X Debugger Setup

Target Board	Power Up
ICD to PC	Connect (wait for USB connect)
ICD to Target	Connect

Emergency: MPLAB X Cannot Locate Debugging Tool

- 1- Check Connections
- 2- Access Project Properties
- 3- Category Default
- 4- Tool and Compiler Selection Verify Debugging Tool Listed
- 5- Select SN: Click on actual device serial number
- 6- Close Project Properties Click OK
- 7- Retry Project Launch

Emergency: Compiler Fails to Build or Launch Project

- 1- Perform Clean + Build
- 2- Retry Build + Launch

Explorer16 demonstration board

Power Supply	9V to 15V (reversed polarity protected)
Main oscillator	8 MHz crystal (use 4x PLL to obtain 32 MHz)
Secondary oscillator	32,768 Hz (connected to TMR1 oscillator)

Allocating Heap and Extra Stack Space

Access Project Properties	Option 1, 2 or 3
Category	pic30-Id
Options Category	General
Heap size (bytes)	Specify (default: 0)
Stack size (bytes)	Specify (default: All available)

Device Configuration for Explorer16 demonstration board

Configure > Configuration Bits	Select
Primary Oscillator select	HS Oscillator enabled
Primary Oscillator output	OSCO pin has oscillator output
Clock switching and monitor	Disabled both
Oscillator select	Primary Oscillator and PLL
Watchdog Timer Postscaler	1:32,768
Watchdog Prescaler	1:128
Watchdog Timer Enable	Disabled
Comm. Channel Select	EMIC2 EMUD2 shared with PGC2 and PGD2
Set Clip on Emulation Mode	Reset into Operational Mode
General Code Segment Write Protect	Disabled
General Code Segment Code Protect	Disabled
JTAG port Enable	Disabled

Emergency: After pressing Halt, or Breakpoint, MPLAB X Unresponsive

Wait!

- 1- MPLAB could be uploading the content of large variable/array in the Watches window
 - 2- MPLAB could be refreshing the Special Function Registers window (if open)
 - 3- MPLAB could be updating Embedded Memory window (if open)
 - 4- MPLAB could be updating the Variables window (if open and contains a large object)
- After regaining control, close any data window or remove any large object before continuing

Emergency: Breakpoint does not work

- 1- Verify the C source code line is not commented
- 2- Verify you have not used more than 4 breakpoints (see list ALT + SHIFT + S)
- 3- Verify the C source line does not contain only a variable declaration
- 4- Verify the C source file is part of the Project Files list
- 5- Verify the project has been Built before placing a breakpoint

The First Flight

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® X IDE Integrated Development Environment and the MPLAB C language suite. Even if you have never heard before of the C language, 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 where 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/connected:

- MPLAB X IDE, free Integrated Development Environment (obtain the latest version available for download from Microchip's website at <http://www.microchip.com/mplab>)
- MPLAB C30 Lite Compiler v3.30 (or later) or MPLAB XC16 Lite Compiler
- A PIC24FJ128GA010 on a PIM (also known as a mezzanine board)
- An MPLAB X compatible programmer/debugger such as the PICKit3, ICD3 or Real ICE
- The Explorer16 board, or any demo board with a row of eight LEDs connected to PortA.

Let's follow the **New Project Setup** checklist to create a new project.

From the Start Page of MPLAB X, select **Create New Project**, or simply select **File>New Project...** from the main menu to activate the new project wizard, which will guide us automatically through the following six steps:

1. Choose Project: in the *Categories* panel, select the **Microchip Embedded** option. In the *Projects* panel, select **Stand Alone Project** and click **Next**.
2. Select Device: in the *Family* drop box, select **PIC24**. In the *Device* drop box, select **PIC24FJ128GA010**, or other PIC24 model of your choice and click **Next**.
3. Select Header: simply click **Next**.
4. Select Tool: select the **PICKit3**, or other supported programmer/debugger of your choice, and click **Next**.
5. Select Compiler: select **C30** (or XC16 if available), and click **Next**.
6. Select Project Name and Folder: type **1-HelloWorld** as the project name, type **C:\FlyingPIC24** as the folder name or use the *Browse* button to navigate to an existing directory of your choice and click **Finish** to complete the wizard setup.

After a brief moment, you will be presented with a new *projects* window. This is empty at the moment except for a number of *logical folders*. We will learn more about those gradually through the following few lessons.

The Flight

Since this is our very first time, let's take advantage of another little wizard offered by MPLAB X to automate the steps required to create a new source file and to make it part of

the project. From MPLAB X main menu, select: **File>New File...** This is a very short wizard that can be completed in just two steps:

1. Choose File Type: in the *Categories* panel, expand the **Microchip Embedded** folder and click on the **C30 compiler**. In the *File Types* panel select the **mainp24f.c** type.
2. Name and Location: in the *File Name* field type **Hello1.c** and click **Finish** to accept all other default settings.

The MPLAB X built-in Editor window will pop up showing the contents of the newly created *Hello1.c* file. It will contain three initial elements that happen to be common to all C programs:

1. A banner, composed of a few comment lines:

```
/*
 * File:   Hello1.c
 * Author: your name here
 *
 * Created current date here
 */
```

Note

Anything included between the pair of characters `/*` and the pair `*/` is considered a comment by a C compiler and as such it is simply ignored. Notice that such a comment can span multiple lines as was the case of this banner. Should you forget to close the comment, the rest of the file would be completely ignored by the compiler.

2. An *include* directive:

```
#include <p24Fxxxx.h>
```

This is not yet a C statement, but more of an instruction for the MPLAB C compiler *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 containing the names (and structure) of all the internal *special-function registers (SFRs)* of the chosen PIC24 model. If accurate, the names reflect exactly those being used in the device datasheet. If you are curious, just open the file and take a look – it is a simple text file that you can open with the MPLAB editor. Here is a segment of the *p24fj128ga010.h* file (MPLAB automatically replaces those four *xxxx* in the include directive with the actual PIC24 model spelling). This is the part 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__));
...
```

3. The *main()* function:

```
int main( void)
{
    return 0;
}
```

In between those two curly brackets, before the return statement, 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-lines 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 *crt0* code (or simply *c0*). The *c0* code will perform basic housekeeping chores, including the initialization of the microcontroller stack, among other things.

MPLAB X New File wizard has created the *Hello1.c* file, saved it inside the project main directory and is now listing this file among the project *sources* (the first and only so far). You can verify it by going back to the *Projects* window, selecting **Window>Projects** (alternatively using the keyboard shortcut **CTRL+1**). By expanding the *Source Files* logical folder (click on the little box containing a plus sign next to the little blue folder icon), you will see that *Hello1.c* now is in the list (Figure 1.1).

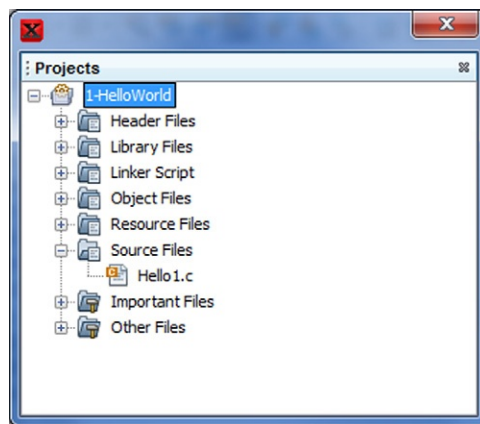


Figure 1.1: The Projects window

Now is the time to start adding our own code to the project. Our mission is to activate for the first time one or more of the output pins of the PIC24.

For historical reasons, and to maintain the greatest compatibility possible with the previous generations of PIC[®] microcontrollers, the input output (I/O) pins of the PIC24 are grouped in modules or *ports*, each comprising up to 16 pins, named in alphabetical order from A to H. We will start logically from the first group, known as *PortA*. Each port has several special-function registers assigned to control its operations. The main one, and the easiest to use, carries traditionally the same name as that of the module (*PORTA*).

To distinguish the control register name from the module name we will use a different notation for the two: *PORTA* (all upper case) will be used to indicate one of the control registers; *PortA* will refer to the entire peripheral module.

According to the PIC24 datasheet, assigning a value of 1 to a bit in the PORTA register turns the corresponding output pin to a logic high level (3.3V). Vice versa, assigning a value of 0 to the same bit will produce a logic level low on the output pin (0V).

Assignments are easy in C language – we can insert a first *assignment statement* in our project as in the following example:

```
PORTA = 0xff;
```

First, notice how each individual statement in C must be terminated with a semicolon. Then notice how they resemble a mathematical equation... they are 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 the special-function PORTA register of the microcontroller.

Note

In C language, by prefixing the literal value with *0x* we indicate the use of the hexadecimal radix. 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?). Otherwise the compiler assumes the default decimal radix.

Compiling and Linking

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

Using the MPLAB X 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 actually pretty long and complex but it can be summarized in two steps:

- *Compiling*, the MPLAB 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 re-locatable code object. If there are multiple source files, this step is repeated for each one of them.
- *Linking*, the MPLAB C linker is invoked and a proper position in the memory space is found for each function and each variable. Also, any number of pre-compiler object code files and standard library functions may be added at this time as required. Among the several output files produced by the linker there is the actual binary executable file (.hex).

All this is performed in a very rapid sequence as soon as you click on the **Build Project** button



in the main toolbar, or you right click in the **Projects** window and select the **Build** option.

Note

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 MPLAB 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, they must be present (as is the case for our *Hello1.c* example) in the projects window *Source Files* logical folder.

Similarly, for the linker to assign the correct addresses to each variable and function, a device-specific “*Linker Script*” file (.gld) must be located. This is taken care of automatically by MPLAB X, so we will not have to worry about it. Just as the include (.h) file tells the compiler about the names (and structure) of device-specific special-function registers (SFRs), the linker script informs the linker about their pre-defined positions in memory (according to the device datasheet) as well as providing essential memory space information such as total amount of flash memory available, total amount of RAM memory available and their address ranges.

Once more, if you are curious, 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 address 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;
_PSVAPAG     = 0x34;
_PSVAPAG     = 0x34;
_RCOUNT     = 0x36;
_RCOUNT     = 0x36;
_SR          = 0x42;
_SR          = 0x42;
...

```

If everything went well, your Output window last few lines should look as in [Figure 1.2](#).

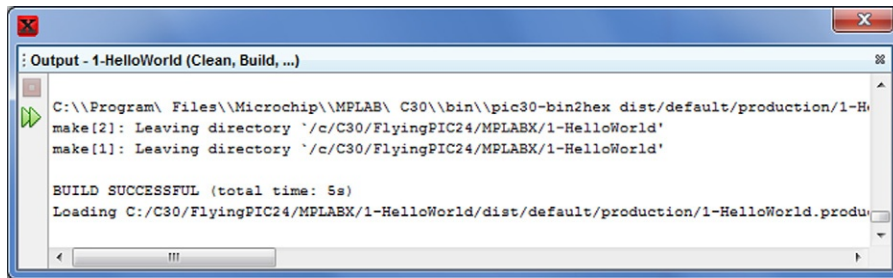



Figure 1.2: MPLAB IDE Output window, after successfully building a project

Debugging the First Project

Let's review the last few steps required to complete our first demo project and start testing it.

1. Hit the **Project Debug**  button or select **Debug>Debug Project** from the MPLAB X main menu and observe the program being written into the part by the connected debugger. If successful, a large green band indicating the device program counter will show up inside the Editor window highlighting the first line of our program, ready to be executed ([Figure 1.3](#)).

Note

In case the green band is not present or should it appear to be positioned at the very end of the program, select **Debug>Reset** from the MPLAB main menu to restart your debugging session from the top. To ensure that the debugger will stop correctly at the top of the main function the next time open the **Tools>Options** dialog box, select the **Embedded** Pane at the top and inside it the **Generic Settings** tab. Finally make sure that both the **Reset@** and the **Debug startup** options are set to **Main**.

In fact, quite a few lines of code have already been executed (remember the C-compiler initialization code *crt0* mentioned earlier) but MPLAB has kept it for itself, sparing us the tedious details.

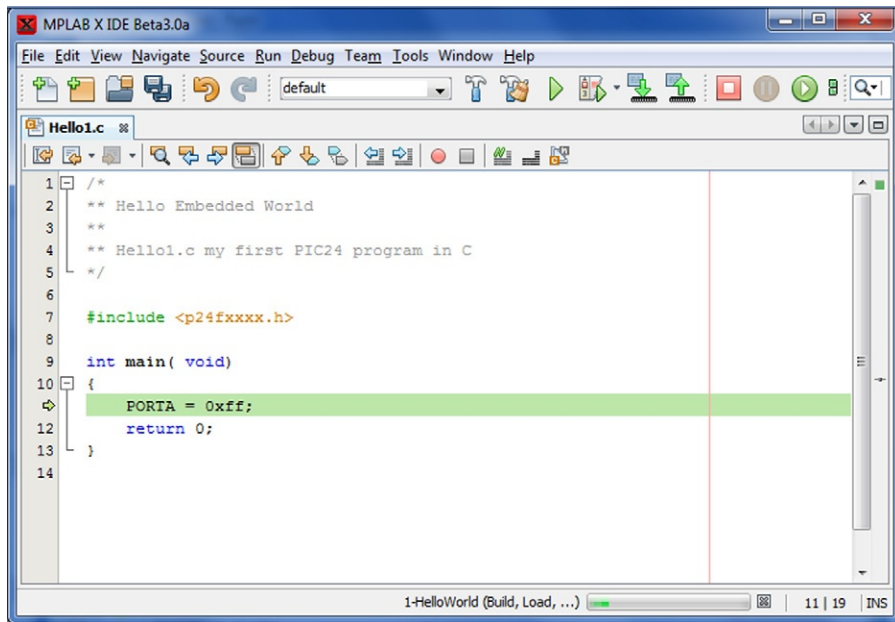


Figure 1.3: MPLAB X in debugging mode

Now, there is one more useful window of the MPLAB X IDE that you should familiarize yourself with: the *Watches* window.

2. From the main menu, select **Window>Debugging>Watches** (or use the keyboard shortcut **ALT+SHIFT+2**).
3. On the first line, click on the **<Enter new watch>** field and type the name of a variable or, in this case, **PORTA** a special function register that we want to observe as it changes during the program execution.

If all is well, your Watches window should now look like Figure 1.4. Now we are ready to start debugging!

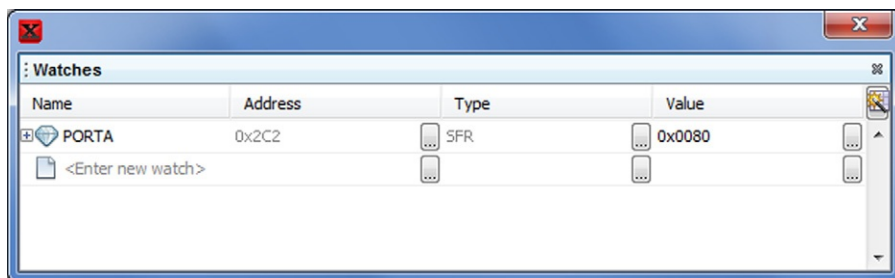




Figure 1.4: MPLAB X Watches window

PORTA register contents are shown as a hexadecimal value. It appears the default content is 0x80. If you click on the little plus sign next to the little diamond icon, the view will be expanded to expose the individual bits and corresponding I/O pins. Only RA7 is initially set.

Now let's advance by a single step using the **Step Over**  or **Step In**  buttons to execute the first statement in our program and observe how the content of *PORTA* changes in the watches window. Or, notice how nothing happens: surprise!

Port Initialization

It is time to hit the books, specifically the PIC24FJ128GA010 datasheet (Chapter 9, for the I/O Ports detail). PortA is a pretty busy, 16-pin-wide port (see [Figure 1.5](#) for a block diagram of a typical I/O pin).

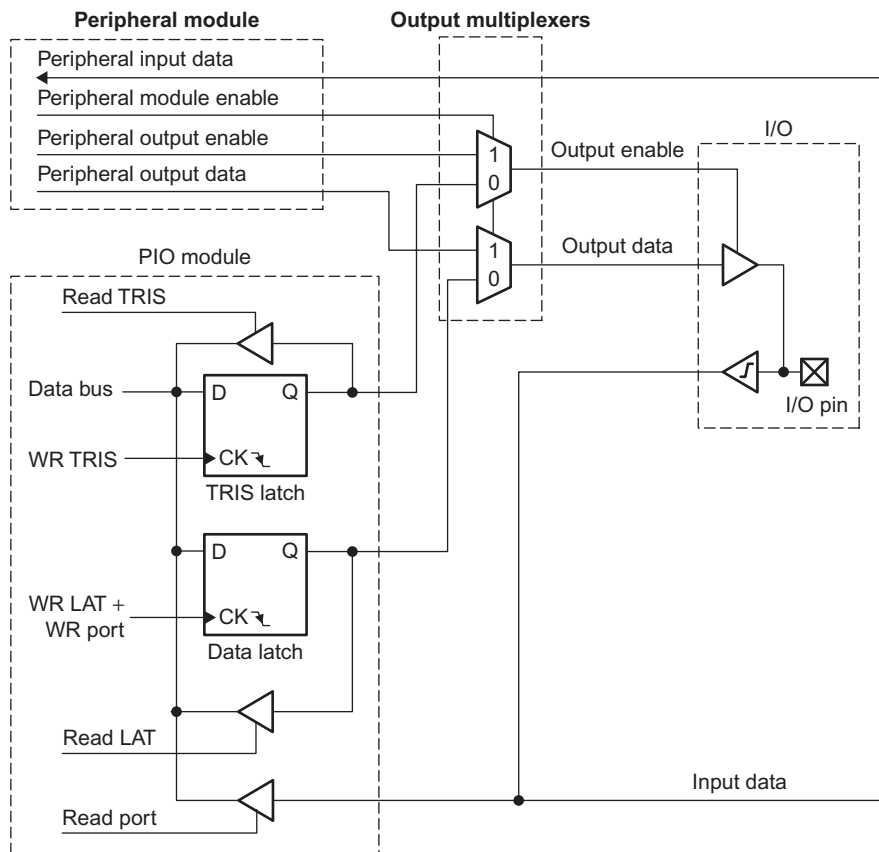




Figure 1.5: Diagram of a typical PIC24 I/O pin

Looking at the pin-out diagrams and pinout descriptions table 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:

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

Re-Testing PortA

Let's exit the debug mode by pressing the  button or selecting **Debug>Finish Debugger Session** from the main menu.

Select the **Debug Project**  button once more, having noticed that the source code has changed, MPLAB X will rebuild the project, reload the flash memory of the device with the resulting executable file and will start executing it only to stop on the very first line of the *main()* function.

Advance by two steps, clicking on the **Step Over**  button and... there you have it, *PORTA* content's finally changed and a few LEDs are lighting up on the Explorer16 board (Figure 1.6):

Hello World!

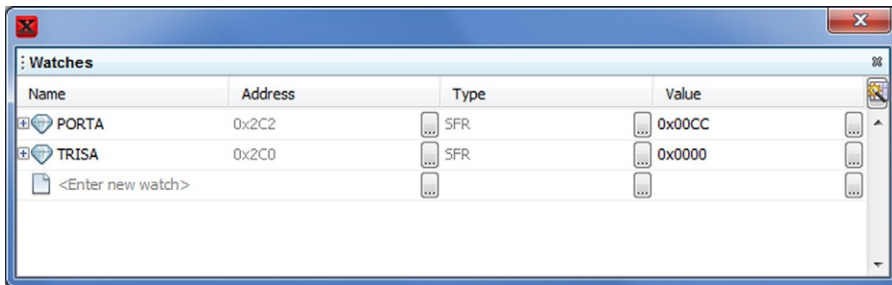


Figure 1.6: PORTA Content has changed!

There is only one problem. The output is not quite what we expected!

After configuring all the pins as outputs, you can verify that the *TRISA* register contains the value 0x00 by adding it to the Watches window list. But the value of *PORTA*, while matching

the LEDs configuration on the Explorer16 board, is not 0xFF. As many as four pins are not responding to our commands: what went wrong?

Configuring the PIC24

If you have ever programmed a microcontroller before, you will be familiar with the concept of *configuration bits*. They are non-volatile memory bits that define some of the most basic behaviors of the device, among which are:

- Selecting the type of oscillator to be used first at power up
- Enabling or disabling the secondary oscillator and the ability to switch the main clock to/from it
- Enabling or disabling the code protection to prevent the contents of the Flash memory to be copied
- Selecting which pair of pins will be used to debug/program the device
- Enabling or disabling the JTAG interface
- ...

While a complete list can be found on the specific device datasheet, it is in particular this last one (the JTAG Enable bit) that we need to understand better in order to explain the strange behavior of our last experiment. In fact, the PIC24 datasheet will reveal that the JTAG interface is by default enabled when the PIC24 memory contents are erased. While such an interface can be useful to perform *boundary scan* tests or to program the device using some third-party tools, when using Microchip development tools it represents just a waste of four otherwise perfectly good pins of PortA. These pins are RA0, RA1, RA4 and RA5, the four pins that had refused to cooperate during our first debugging attempts.

In order to disable the JTAG interface and, since we are at it, to select a convenient configuration that will be common to all our future projects, we will use a couple of handy macros (`_CONFIG1` and `_CONFIG2`). They have been introduced in the latest releases of the MPLAB C compiler. We could add a few lines directly at the top of our main program or, rather, we should save them in a separate file that we will call **config.h**. Here is the basic configuration for a PIC24F GA0 model:

```
/*
** config.h
**
** "Flying PIC24" projects device configuration
*/
#include <p24fxxx.h>

_CONFIG1( JTAGEN_OFF      // disable JTAG interface
          & GCP_OFF       // disable general code protection
          & GWRP_OFF      // disable flash write protection
          & ICS_PGx2      // ICSP interface (2=default)
```

```

        & FWDTEN_OFF)           // disable watchdog timer
_CONFIG2( IESO_OFF            // two speed start up disabled
        & FCKSM_CSDCMD        // disable clk-switching/monitor
        & FNOSC_PRIPLL        // primary oscillator: enable PLL
        & POSCMOD_XT)          // primary oscillator: XT mode

```

Placing this file in a shared **/include** directory will make it available to all the projects in this book. We can now replace the *#include* at the top of our main file and get ready for a final round of debugging. Here is what our main source file should now look like:

```

/*
** Hello Embedded World
**
** Hello2.c controlling PORTA pin direction
*/
#include <config.h>           // set the configuration bits

int main( void)
{
    PORTA =    0xff;
    TRISA =    0;           // configure all PORTA pins as outputs
    return 0;
}

```

Note

There are two ways to specify an include file path: *#include "filename.h"* and *#include <filename.h>*. The first one (using quotes) tells the MPLAB C compiler (preprocessor) to look for the *.h* file in the *local* project directory first. Should this fail, it will extend the search to the *include directories list* (defined in the project options). The latter method (angled brackets) tells the compiler to skip the local search and to go directly to the *include paths list*.

For convenience, we will add one more step to the *New Project* checklist to make sure that the *include directories list* for our projects always contains a reference to the */include* subdirectory so that, in the following, all our projects will be able to access shared header files from there avoiding a lot of duplication and the possibility of errors that comes with it.

The include directories list can be accessed following the four steps below:

1. Select **Run>Set Project Configuration>Customize** from MPLAB main menu. This will open the *Project Properties* dialog box (Figure 1.7).
2. Select the compiler **pic30-gcc** category.
3. In the **General** (default) options category click on the **Include Directories** option.
4. Add a relative path to the **/include** subdirectory ("*../include*" will do if you followed the instructions in the book Introduction).

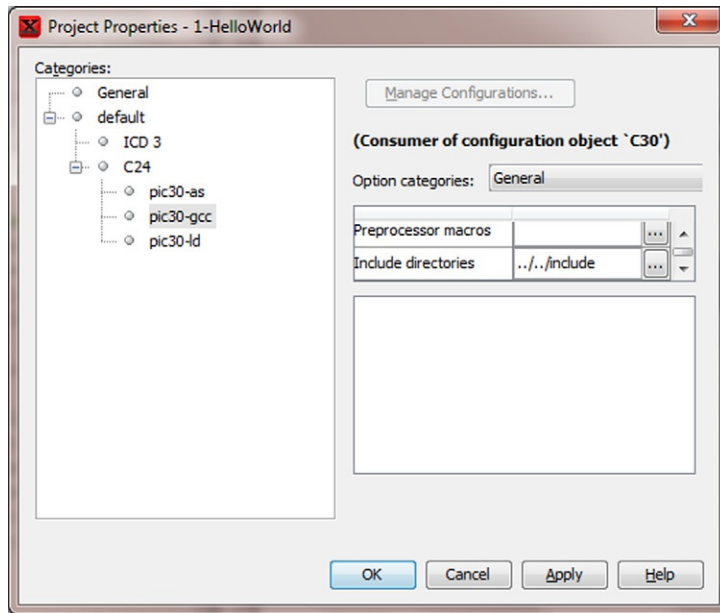


Figure 1.7: The Project Properties dialog box

Let's make sure we have terminated the previous debugging session (select **Debug>Finish Debugger**), and start a new session (select **Debug>Debug Project**). Then select **Debug>Step Over** twice and enjoy the view of a full row of eight LEDs shining bright green at the bottom of the Explorer16 demo board.

The Watches window (Figure 1.8) will confirm: success at last!

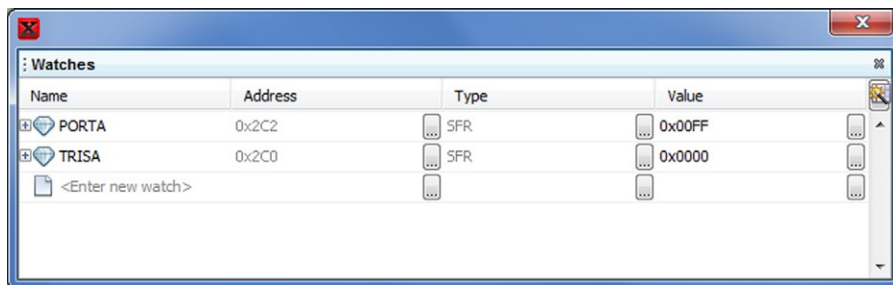


Figure 1.8: Success at last!

Testing PortB

Looking for more trouble, let's explore the possibility of controlling a second alternative I/O port: PortB.

It is simple to edit the program and replace the two control registers 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 (Figure 1.9)!

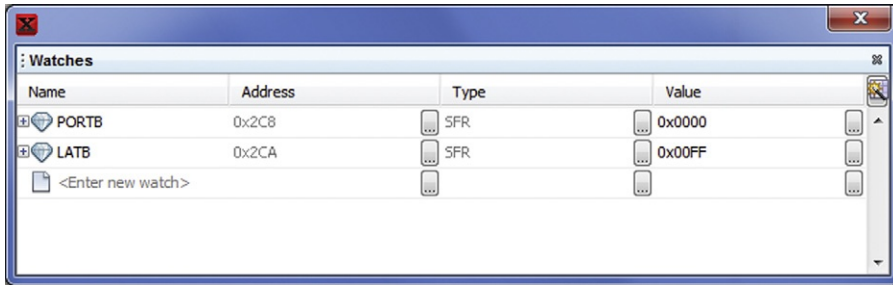


Figure 1.9: The contents of the LATB register

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 newer PIC24 architecture:

- Most of PortB pins are multiplexed with the analog inputs of the Analog-to-Digital Converter (ADC) peripheral. The 8-bit architecture used to reserve 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 explains what was happening in our last (failed) experiment. At power-up, all PortB pins of the PIC24FJ128GA010 are assigned an analog input function. Unless we disable the ADC analog inputs multiplexing, reading *PORTB* will always return all zeros.

Notice 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 (Figure 1.9).

Note

When writing to an I/O port, *PORTx* and *LATx* registers both set a value to the output latches, but it is when reading that the behavior differs dramatically. Reading a *LATx* register will always return the last value that had been written to the output registers. Reading a *PORTx* register instead will return the actual value present on the output pins (Figure 1.5). This could be different from the latched value if there is a short or a heavy load on one of the pins. Also, as in the previous example, if not all pins are configured as digital outputs, only *PORTx* will return a correct picture of the port I/O pins current status. Needless to say, any *read/modify/write* sequence performed on I/O port output pins should be performed using exclusively the *LATx* registers.

Analog vs Digital Pin Control

To reconnect *PORTB* inputs to the digital inputs, we have to modify the analog-to-digital conversion module (ADC) inputs. From the datasheet, we learn that the special-function register *AD1PCFG* controls the analog/digital assignment of each pin (Figure 1.10).

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 1.10: AD1PCFG, ADC Port configuration register

Assigning a 1 to each bit in the *AD1PCFG* special-function register will quickly accomplish the task. Our new and complete program example is now:

```
/*
** Hello Embedded World
**
** Hello4.c learning to control the Analog Pins
*/
#include <config.h>
```

```

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

```

This time, compiling and single stepping through it will give us the desired results (Figure 1.11).

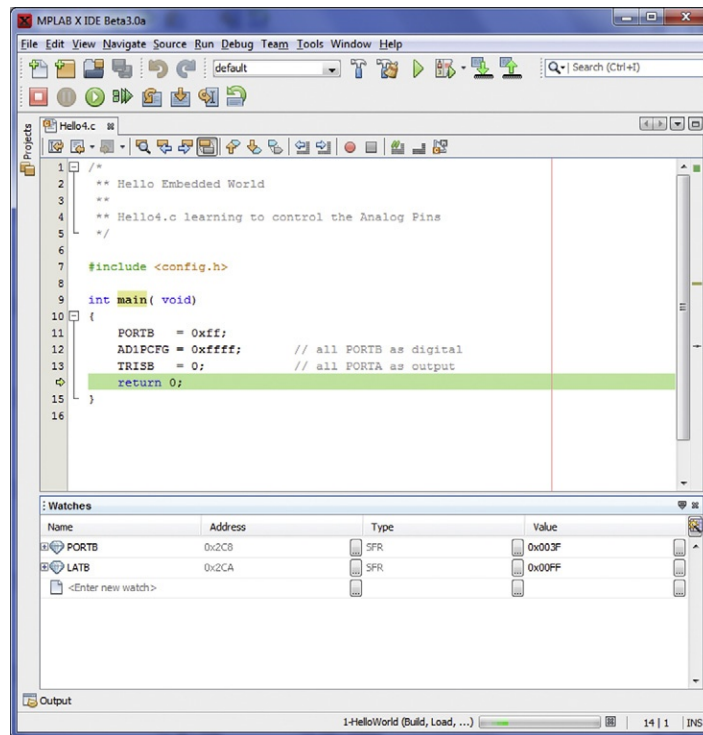


Figure 1.11: PortB Under control

Interestingly, you will notice that the value of *PORTB* reported in the Watches window is still NOT quite what we were expecting!

I will have mercy this time, revealing immediately the solution to this last riddle.

Two pins of PortB are not yet under our full control: they are RB6 and RB7. Turns out those are the two pins used by the two-wire programming/debugging interface (also known as the In Circuit Serial Programming interface or ICSP), used by the PIC24F and most/all other PIC microcontrollers.

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 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 MPLAB C compiler can do to read our mind. Just like in assembly, we are responsible for configuring the device properly and setting the correct direction of the I/O pins. We are also required to study the datasheet and learn about the small differences between the 8-bit PIC microcontrollers we might be familiar with, and the newer 16-bit breed.

As high-level as the C programming language is, writing code for embedded-control devices still requires us to be intimately familiar with the finest details of the hardware we use. In practice, in order to handle correctly the I/O ports we need to understand the function of the *PORTx*, *TRISx* and *LATx* registers. We also need to pay attention to the device pin-out and how multiple functions are often multiplexed on the same I/O pins. These often include analog input functions and possibly programming/debugging functions that require additional control of the *ADIPCFG* register or the device configuration bits.

Notes for the Assembly Experts

If you have difficulties blindly accepting the validity of the code generated by the MPLAB C compiler, you might find comfort in knowing that, at any given point in time, you can decide to switch to the *Embedded Memory* view (select **Windows>Embedded Memory** from the main menu, then choose **Program** as the memory type and **Symbol** as the format). You can quickly inspect the code generated by the compiler, and step through it in sync with the C source file (Figure 1.12).

However, 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 note, I encourage you to scroll back through the Output window, looking for the *memory usage* summary as published by the compiler during the last build.

If it is not present, you can enable this feature by selecting the command **Run>Set Project Configuration>Customize** to get access to the *Project properties* dialog box, selecting the linker **pic30-ld** category, selecting the **Diagnostic** options category, and finally checking the **Display memory usage** option as illustrated in Figure 1.13.

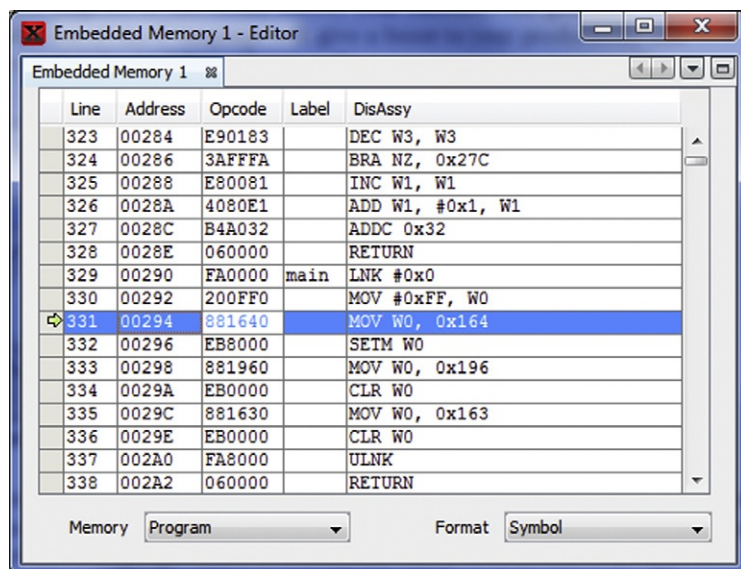
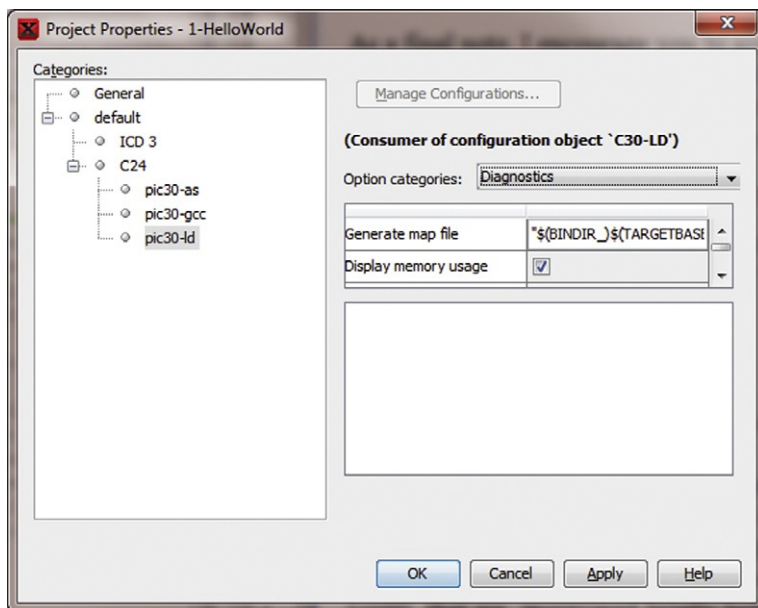


Figure 1.12: Embedded Memory – Disassembly view

Figure 1.13: Enabling the *Display memory usage* linker option

After rebuilding the project with the **Clean and Build Project** command, you should obtain a summary similar to the following:

```

Program Memory [Origin = 0x200, Length = 0x155fc]

section      address      length (PC units)      length (bytes) (dec)
-----
.text        0x200                0x90                0xd8 (216)
.text        0x290                0x14                0x1e (30)
.dinit       0x2a4                 0x2                 0x3 (3)
.isr         0x2a6                 0x2                 0x3 (3)

Total program memory used (bytes):                0xfc (252) <1%

```

Don't be alarmed! Although we wrote only three lines of code in our first example, the amount of program memory used appears to already be up to 200+ 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 MPLAB C compiler. This is the initialization code (*crt0*) that we mentioned briefly before. We will get to it, in more detail, in the following chapters as we discuss variable initialization, memory allocation and interrupts.

Notes for the PIC Microcontroller 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-bit 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 the C Experts

Certainly, we could have used the *printf()* function from the standard C libraries. In fact they are readily available with the MPLAB C compiler. But we are targeting embedded-control applications and we are not writing code for multi-gigabyte 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 & Tricks

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

- To drive a 5 V 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 5 V power supply.
- Digital input pins instead are already capable of tolerating up to 5 V. They can be connected directly to 5 V input signals.

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

Exercises

- 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., 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 C 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://publications.gbdirect.co.uk/c_book/
This is the online version of *The C Book*, second edition by Mike Banahan, Declan Brady and Mark Doran, originally published by Addison Wesley in 1991. This version is made freely available.
- 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.

A Loop in the Pattern

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 the way 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 keep track visually 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 each other in 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® X features will be used for the first time: the *Animate* mode and the *Run* mode.

Preflight Checklist

For this second lesson, we will need the same basic software and hardware components installed and used before, including:

- MPLAB X IDE, free Integrated Development Environment (obtain the latest version available for download from Microchip's Web site at <http://www.microchip.com/mplab>)
- MPLAB C30 Lite Compiler v3.30 (or later) or MPLAB XC16 Lite Compiler
- A PIC24FJ128GA010 on a PIM (also known as a mezzanine board)
- An MPLAB X compatible programmer/debugger such as the PICkit3, ICD3 or Real ICE
- The Explorer16 board, or any demo board with a row of eight LEDs connected to PortA.

We will also re-use the **New Project Setup** checklist to create a new project.

From the Start Page of MPLAB X, select **Create New Project**, or simply select **File>New Project...** from the main menu to activate the new project wizard which will guide us automatically through the following six steps:

1. Choose Project: in the *Categories* panel, select the **Microchip Embedded** option. In the *Projects* panel, select **Stand Alone Project** and click **Next**.
2. Select Device: in the *Family* drop box, select **PIC24**. In the *Device* drop box, select **PIC24FJ128GA010**, or other PIC24 model of your choice and click **Next**.
3. Select Header: simply click **Next**.
4. Select Tool: select the **PICKit3**, or other supported programmer/debugger of your choice, and click **Next**.
5. Select Compiler: select **C30 or XC16** if available, and click **Next**.
6. Select Project Name and Folder: type **2-ALoopInThePattern** as the project name, type **C:\FlyingPIC24** as the folder name or use the *Browse* button to navigate to an existing directory of your choice and click **Finish** to complete the wizard setup.

After a few moments as MPLAB X completes the project creation, let's remember to add the */include* subdirectory to the list of *include directories* with the following four additional steps:

1. Select **Run>Set Project Configuration>Customize** from MPLAB main menu. This will open the *Project properties* dialog box.
2. Select the compiler **pic30-gcc** category.
3. In the **General** (default) options category click on the **Include Directories** option.
4. Add a relative path to the */include* subdirectory ("*../include*" will do if you followed the instructions in the book Introduction).

The Flight

Even in this second flight we will make use of the *New File* wizard (select **File>New File...** in the main menu). It will help us, once more, to create a new source file using a basic PIC24 template and to make it part of the project, adding it to the list of the project *Source Files*.

5. Choose File Type: in the *Categories* panel, expand the **Microchip Embedded** folder and click on the **C30 compiler**. In the *File Types* panel select the **mainp24f.c** type.
6. Name and Location: in the *File Name* field type **Loop.c** and click **Finish** to accept all other default settings.

MPLAB X built-in Editor window will pop up showing the contents of the newly created *Loop.c* file. It will be prepopulated with the three basic elements:

1. The banner, composed of a few comment lines:

```
/*  
 * File:   Loop.c  
 * Author: your name here
```

```
*
* Created current date here
*/
```

2. The *include* directive:

```
#include <p24Fxxx.h>
```

But we have seen already how it would be more advantageous to include the *config.h* file (defining the PIC24 configuration bits) directly and letting that file in its turn include the *p24fxxx.h*. So let's replace it with:

```
#include <config.h>
```

This way, in a single line we have accomplished both the task of configuring the device and including all the special function register definitions.

3. The *main()* function:

```
int main( void)
{
    return 0;
}
```

When the wizard has finished, you can verify that all worked as expected by going back to the *Projects* window, selecting **Window>Projects** (or alternatively using the keyboard shortcut **CTRL+1**). By expanding the *Source Files* logical folder (click on the little box containing a plus sign next to the little blue folder icon), you will see that *Loop.c* is now listed.

Note

You might also be surprised to find that the *Projects* window contains two projects now! In fact *1-HelloWorld* might still be showing up there, alphabetically preceding the newly created project in the list. To clean things up, select the old project **root** and use the **right mouse button** to open the *Projects* window *context menu*. It's quite a long menu and down there somewhere toward the bottom, you will find the **Close** command.

One of the key questions that might have come to mind after working through the previous lesson is probably: “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”. Upon reaching the end of the program and executing a reset, it takes the microcontroller back to the very beginning to execute all the initialization code, including the (now famous) *crt0* code segment. So, as short as this 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, instead, is to design an application *main-loop*. Let’s review the most basic loop coding options in C first.

Note

If the main function is never supposed to return you will ask: why would we add a return instruction at the end of it? The truth is, it is not necessary at all. This is done only to respect the old tradition of C programming on desktops, workstations and mainframe computers where the *main()* function is, in fact, called by an operating system and it is expected to return back a code indicating the correct (or abnormal) termination of the program. For the same reason we could define *main()* as a function returning no value at all as in:

```
void main( void)
{ ... }
```

Or, if you are as lazy as I am, simply:

```
main()
{ ... }
```

This takes advantage of the permissive syntax rules of the C language. If we omit the *void*, we only receive a benign *warning* and even that only if the compiler is set to the highest *error sensitivity* levels (check the Project Options dialog box, pic30-gcc category, *Strict ANSI* warning and *Additional* warnings options).

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 in 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 *zero*
- *true* is represented by any integer except *zero*.

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:

Assuming: $a = 17$ 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, 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-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 parenthesis produces a *true* logic value (that is any integer value but zero), 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 evaluation of the expression is done first, before the curly bracket content is executed (if ever), and is then re-evaluated 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, 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.

```
main()
{
    // init control registers
    TRISA = 0xff00; // all PORTA as output

    // main application loop
```



```

while( 1)
{
    PORTA = 0xff;
    PORTA = 0;
} // main loop
} // main

```

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 once at the beginning.
- The *main loop*, containing all the control functions that define the application behavior, which executes continuously.

Animation

Time to see things moving... let's connect the programmer/debugger tool to our target



(Explorer16 demo) board, power it up and click on the **Debug**  button or select

Debug>Debug Project from the main menu. MPLAB will perform a quick check and, since the project had not been built yet, the compiler and linker will be invoked in a quick sequence to generate an executable code. The programmer/debugger selected for this project will be invoked next to download the new program into the PIC24 and finally, if all went well, the Editor screen will be updated to show a large green band on top of the first line of our *main()* function. We are ready to go!

This time, instead of single stepping as in our first lesson, we will use the **Animate**  button, alternatively select **Debug>Animate** from the main menu.

The green bar, representing our *program counter*, will now start moving quickly through our source file, rolling in the loop and repeating forever the same two assignments.

Correspondingly, you will see the LEDs on the board (all eight together) flash on and off.

The speed of animation is determined by a default parameter in MPLAB X. After each line is executed, a quarter-second pause is inserted artificially. This pause is designed to give you the possibility to appreciate the sequence of execution of the program. You can stop the animation pressing the **Pause**  button or by terminating the debugging session using the **Stop**  button.

During debugging sessions you will notice the presence of an animated icon at the bottom right of the MPLAB X main window. This is not a *progress* bar. The little green dot keeps moving from left to right and circles back continuously to indicate that the debugging session is *active*. The **Stop** button makes it go away (Figure 2.1).

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.

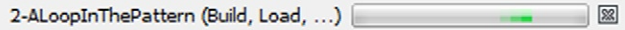



Figure 2.1: The Debugging Active bar

How would our program behave if we were to let the PIC24 processor run to its full speed?

Terminate the current debugging session, and now press the **Run**  button or select **Run>Run Program** from the main menu.

MPLAB X will now recompile the program for immediate execution. It will be downloaded into the PIC24 flash memory and execution will start immediately.

Note

When in run mode, there is no animated icon and MPLAB X seems inactive, but it is your target PIC24 that is alive and executing continuously the application at full speed.

Let me warn you before you are disappointed: you will not be able to see any flashing of the LED bar! This time it is not because of some tricky detail getting in the way of our experiment as in the first lesson examples but, rather, due to a limitation of our human eyes.

The PIC24 is actually flashing those LEDs but, assuming our default configuration of the main oscillator (32MHz), this is happening at the rate of several million times per second!

Not So Fast, Please!

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 into all PIC24 microcontrollers. For this example we will choose the first timer, *Timer1*, of the five timers available inside the PIC24FJ128GA010 models. This is one of the most flexible and simple peripheral modules. All we need is to 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 2.2).

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

- *TMRI*, which contains the 16-bit counter value
- *TICON*, 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 *TMRI* register to start counting from zero.

```
TMRI = 0;
```

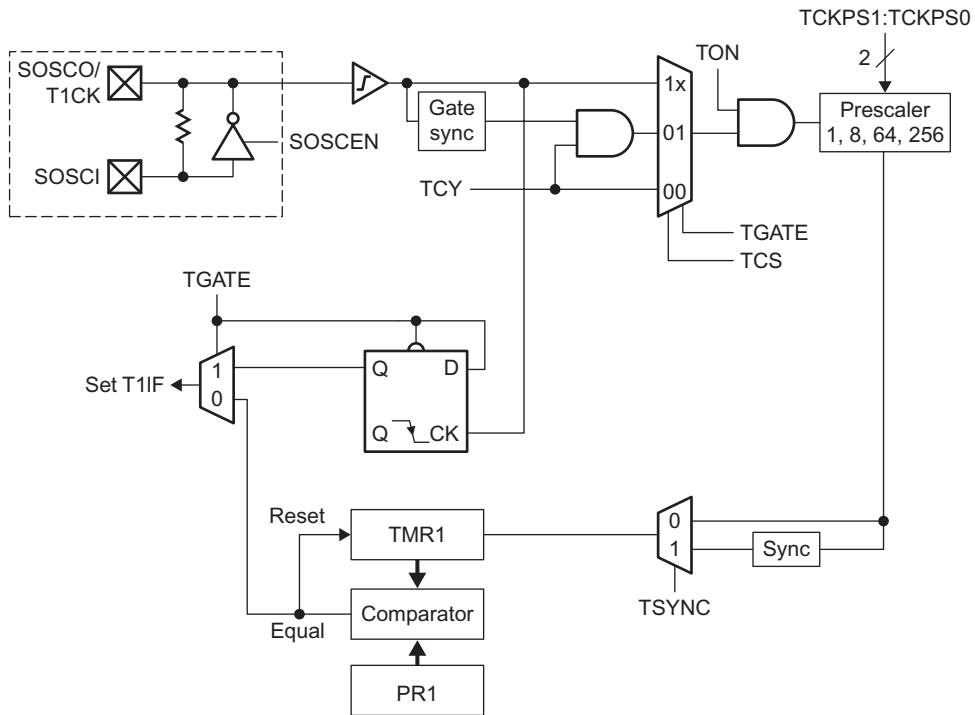


Figure 2.2: Timer1 block diagram

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 2.3: T1CON: Timer1 control register

Then we can initialize *T1CON* so that the timer will operate in a simple configuration (Figure 2.3) where:

- Timer1 is activated: $TON = 1$
- The main MCU clock serves as the source ($F_{osc}/2$): $TCS = 0$
- 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 need to worry about the behavior in IDLE mode: $TSIDL = 0$ (default).

When we assemble all the bits into 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 just 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 as per our default configuration, we need to assign quite a large value to *DELAY* so as to obtain a pause of about a quarter of a second. In fact, the following formula dictates the total delay time produced by the Timer1 loop:

$$T_{\text{delay}} = (2/F_{\text{osc}}) * 256 * \text{DELAY}$$

With $T_{\text{delay}} = 256$ 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 <config.h>
#define DELAY 16000
main()
{
    // init control registers
    TRISA = 0xff00; // all PORTA as output
    T1CON = 0x8030; // TMR1 on, prescale 1:256 Tclk/2

    // main application loop
    while( 1)
    {
        //1. turn pin 0-7 on and wait for 1/4 of a second
        PORTA = 0xff;
        TMR1 = 0;
        while ( TMR1 < DELAY)
        {
        }

        // 2. turn all pin off and wait for a 1/4 of a second
        PORTA = 0;
        TMR1 = 0;
        while ( TMR1 < DELAY)
        {
        }
    } // main loop
} // main
```

Note

Programming in C, the number of opening and closing curly brackets tends to increase rapidly as the code grows. After a very short while, even if you stick religiously to the best indentation rules, it can become difficult to remember which closing curly brackets belong to which opening curly brackets. The MPLAB X editor tries to help by highlighting in yellow each matching bracket when your cursor is resting on one. Still, I try to make my files even more readable by explicitly adding small reminders (comments) after each closing bracket.

It is time now to hit the **Run** button again and verify whether the code is working. The LEDs should flash at a comfortably slow pace, with a frequency of about two flashes per second.

Don't use the **Animate** mode now or things will get really slow!

As we saw before, the animation adds a further delay of about a quarter of a second between the execution of each individual line of code, multiply that by 16,000 ...

Post-Flight Briefing

In this brief lesson, we learned about the way the MPLAB C compiler deals with program termination. For the first time, we gave our little project a bit of structure – separating the *main()* function in an *initialization section* and an *infinite loop*. To do so, we learned about the *while* loop statements and we took the opportunity to touch briefly on the subject of *logical expression* evaluation. We closed the lesson with a final example where we used a *timer* module for the first time and we played with the Animate and Run modes of the in circuit debugging interface.

We will return to all these elements, so don't worry if you have more doubts now than when we started, this is all part of the learning experience.

Notes for the Assembly Experts

Logic expressions in C can be tricky for the assembly programmer who is used to dealing with *binary* operators of identical names (*AND*, *OR*, *NOT*, ...). In C there is a set of binary operators too, but I purposely avoided showing them in this lesson to avoid mixing things up.

Binary logic operators take pairs of bits from each operand and compute the result according to the defined table of truth. *Logic* operators, on the other hand, look at each operand (independently of the number of bits used) as a single *Boolean* value.

See the following examples on byte-sized operands:

	11110101		11110101 (true)
binary OR	00001000	logical OR	00001000 (true)
	-----		-----
gives	11111101	gives	00000001 (true)

Notes for the PIC[®] Microcontroller Experts

I am sure you noticed: *Timer0* has disappeared! The good news is: you are not going to miss it! In fact, the remaining five timers of a PIC24 are so loaded with features that there is no functionality of *Timer0* that you are going to feel nostalgic about. All of the special function registers that control the timers have similar names to the ones used on PIC16 and PIC18 microcontrollers, and are pretty much identical in structure. Still, keep an eye on the datasheet. The designers managed to cram in several new features, including:

- All timers are now 16-bit wide.
- Each timer has a 16-bit period register.
- A new 32-bit mode timer-pairing mechanism is available for *Timer2/3* and *Timer4/5*.
- A new external clock gating feature has been added on *Timer1*.

Notes for the C Experts

If you are used to programming in C on a personal computer, you expect that, upon termination of the *main()* function, control will be returned to the operating system. While several real time operating systems (RTOSs) are available for the PIC24, a large number of applications don't need and won't use one. This is certainly true for all the simple examples in this book. By default, the MPLAB C compiler assumes there is no operating system to return to and does the safest possible thing – it resets.

Tips & Tricks

Some embedded applications are designed to run their main loop for months or years in a row without ever being turned off or receiving a reset command. But the control registers of a microcontroller are simple RAM memory cells. As small as it can be, the probability that a power supply fluctuation (undetected by the brown-out reset circuit), an electromagnetic pulse emitted by some noisy equipment in the proximity, or even a cosmic ray could alter their contents is a small but finite number. Given enough time, depending on the application, you will see it happen. When you design applications that have to operate reliably on large time scales (months, years ...), you should start considering seriously the need to provide a periodic *refresh* of the most important control registers of the essential peripherals used by the application.

Group the sequence of initialization instructions in one or more functions. Call the functions once at power up, before entering the main loop, but also make sure that inside the main loop the initialization functions are called when no other critical task is pending and every control register is re-initialized periodically.

Exercises

- Output a counter on the PortA pins instead of the alternating on and off patterns.
- Use a rotating pattern instead of alternating on and off.

Books

- Adams, N., 2003, *The Flyers, in Search of Wilbur and Orville Wright*, Three Rivers Press, New York, NY.
A trip back in time to the first powered flight in history, just 120 feet on the sands of Kitty Hawk.
- Ullman, L., Liyanage, M., 2005, *C Programming*, Peachpit Press, Berkeley, CA.
This is a fast reading and modern book, with a simple step-by-step introduction to the C programming language.

Links

- http://en.wikipedia.org/wiki/Control_flow#Loops
A wide perspective on programming languages and the problems related to coding and taming loops.
- http://en.wikipedia.org/wiki/Spaghetti_code
Your code gets out of control when you cannot fly the pattern.

More Pattern Work, More Loops

In aviation, a proper “loop” is an “aerobatic” maneuver performed only by pilots who have received advanced training, using airplanes that are specially equipped for the task. You could take this as a disappointment or a reassurance, but you can be certain that, when preparing for the private-pilot license, you will not be asked to perform any such trick. There will be plenty of other challenges, though, as you will be asked to perform and repeat to perfection a variety of “turns”, including turns around a point, S turns, steep turns and standard rate turns. In all these exercises, you will discover how difficult it can be – while navigating in a three-dimensional environment – to control only one of the dimensions at a time. When circling around a reference point on the ground, you will initially struggle to maintain a constant altitude and a constant speed. A little bit of wind will add to the challenge of maintaining a constant distance from the reference point, and performing a nice and smooth circle. Practice will make you perfect.

In C language programming, there are several types of loops, too. Learning which one to choose, when and why, will take a little practice, but will make you a better embedded-control programmer.

Flight Plan

In the previous lesson, we learned there is a loop at the core of every embedded-control application. In this lesson, we will continue exploring a variety of other techniques available to the C programmer to perform loops. Along the way, we will take the opportunity to briefly review integer variable declarations, increment and decrement operators as well as quickly touch the subject of array declaration and use. As in any good flight lesson the theory is immediately followed by the practice, and we will conclude the lesson with an entertaining exercise that will make use of all the concepts and tools acquired during the lesson.

Preflight Checklist

In this lesson we will continue using:

- MPLAB[®] X IDE, free Integrated Development Environment (obtain the latest version available for download from Microchip’s Web site at <http://www.microchip.com/mplab>)
- MPLAB C30 Lite compiler v3.30 (or later) or MPLAB XC16 Lite compiler
- A PIC24FJ128GA010 on a PIM (also known as a mezzanine board)

- An MPLAB X compatible programmer/debugger such as the PICkit3, ICD3 or Real ICE
- The Explorer16 board, or any demo board with a row of eight LED connected to PortA.

In preparation for the new demonstration project, you can make use of the **New Project Setup** checklist to create a new project called **3-MoreLoops**. Also, use the **New File** wizard to create a new source file to be called **MoreLoops.c**.

The Flight

In a *while* loop, a block of code enclosed by two curly brackets is executed if, and for as long as, a logic expression returns a Boolean *true* value (not zero). The logic expression is evaluated before the loop, which means that if the expression returns *false* right from the beginning, the code inside the loop might never be executed.

do Loops

If you need a type of loop that gets executed at least once but only subsequent repetitions are dependent on a logic expression, then you have to look at a different type of loop.

Let me introduce you to the *do* loop syntax:

```
do {  
    // your code here  
} while ( x);
```

Don't be confused by the fact that the *do* loop syntax is using the *while* keyword again to close the loop – the behavior of the two loop types is very different.

In a *do* loop, the code (if any) found between the curly brackets is always executed first, and only then is the logic expression evaluated. Of course, if all we want to achieve is an infinite loop for our *main()* function, then it makes no difference if we choose the *do* or the *while*...

```
main()  
{  
    // initialization code  
    ...  
    // main application loop  
    do {  
        ...  
    } while ( 1);  
} // main
```

Looking for curious cases, we might analyze the behavior of the following loop:

```
do{  
    // your code segment here...  
} while ( 0);
```

You will realize that the code segment inside the loop is going to be executed once and, no matter what, only once. In other words, the loop syntax around the code is, in this case, a total

waste of your typing efforts and another good candidate for the “Most Useless Piece of Code in the World” contest.

Let’s now look at a more useful example, where we use a *while* loop to repeatedly execute a piece of code for a predefined and exact number of times. First of all, we need a variable to perform the count. In other words, we need to allocate one or more RAM memory locations to store a counter value.

Note

In the previous two lessons we were able to skip, almost entirely, the subject of variable *declarations* as we relied exclusively on the use of what are in fact predefined variables: the special function registers of the PIC24.

Variable Declarations

We can declare an integer variable with the following syntax:

```
int c;
```

Since we used the keyword *int* to declare *c* as a 16-bit (signed) integer, the MPLAB C compiler will make arrangements for two bytes of memory to be used. Later, the linker will determine where those two bytes will be allocated in the physical RAM memory of the selected PIC24 model. As defined, the variable *c* will allow us to count from a negative minimum value of $-32,768$ to a maximum positive value of $+32,767$. If we need a larger integer numerical range, we can use the *long* (signed) integer type as in:

```
long c;
```

The MPLAB C compiler will use 32 bits, (four bytes) for this variable.

If we are looking for a smaller counter, and we can accept a range of values from -128 to $+127$, we can use the *char* integer type instead:

```
char c;
```

In this case, the MPLAB C compiler will use only 8 bits (a single byte).

All three types can further be modified by the *unsigned* attribute as in:

```
unsigned char c;    // ranges from 0..255
unsigned int  i;    // ranges from 0..65,535
unsigned long x;    // ranges from 0..4,294,967,295
```

There are then variable types defined for use in floating point arithmetic:

```
float  f;           // defines a 32 bit precision floating point
long double d;      // defines a 64 bit precision floating point
```

for Loops

We can now return to our counter example. All we need is a simple integer variable to be used as index/counter, capable of covering the range from zero to five, therefore a *char* integer type will do:

```
char i;    // declare i as an 8-bit integer with sign
i = 0;     // init the index/counter
while ( i<5)
{
    // insert your code here...
    // it will be executed for i= 0, 1, 2, 3, 4
    i = i+1;    // increment
}
```

Whether counting up or down, this is something you are going to do a lot in your everyday programming life.

In C language, there is a third type of loop that has been designed specifically to make coding this common case easy. It is called the *for* loop, and this is how you would have used it in the previous example:

```
for ( i=0; i<5; i=i+1)
{
    // insert your code here ...
    // it will be executed for i=0, 1, 2, 3, 4
}
```

You will agree that the *for* loop syntax is compact, and it is certainly easier to write. It is also easier to read and debug later. The three expressions separated by semicolons and enclosed in the brackets following the *for* keyword are exactly the same three expressions we used in the prior example to:

- Initialize the index
- Check for termination, using a logic expression
- Advance the index/counter... in this case incrementing it.

You can think of the *for* loop as an abbreviated syntax of the *while* loop. In fact, the logic expression is evaluated first and if *false* from the beginning, the code inside the loop's curly brackets may never be executed.

Perhaps this is also a good time to review another convenient shortcut available in C. There is a special notation reserved for the increment and decrement operations that uses the operators `++` and `--`:

```
++ to increment, as in  i++;   is equivalent to  i=i+1;
-- to decrement, as in  i--;   is equivalent to  i=i-1;
```

There will be much more to say on this subject in later chapters, but this will suffice for now.

More Loop Examples

Let's see some more examples of the use of the *for* loop and the increment/decrement operators. First, a count from zero to four:

```
for ( i=0; i<5; i++)
{
    // insert your code here ...
    // it will be executed for i= 0, 1, 2, 3, 4
}
```

Then, a count down from four to zero:

```
for ( i=4; i>=0; i--)
{
    // insert your code here ...
    // it will be executed for i= 4, 3, 2, 1, 0
}
```

Can we use the *for* loop to code an (infinite) main program loop? Sure we can, here is an example:

```
main()
{
    // 0. initialization code
    ...
    // 1. the main application loop
    for ( ; 1; )
    {
        ...
    }
} // main
```

If you like it, feel free to use this form. As for me, from now on I will stick to the *while* syntax (it is just an old habit).

Arrays

Before starting to code our next project, we need to review one last C language feature: *array* variable types. An array is just a contiguous block of memory containing a given number of identical elements of the same type. Once the array is defined, each element can be accessed via the array name and an index. Declaring an array is as simple as declaring a single variable – just add the desired number of elements in square brackets after the variable name:

```
char c[10]; // declares c as an array of 10 x 8-bit integers
int i[10]; // declares i as an array of 10 x 16-bit integers
long x[10]; // declares x as an array of 10 x 32-bit integers
```

The same squared brackets notation is used to refer to the content or assign a value to each element of an array as in:

```
a = c[0]; // copy the 1st element of c[] into a
c[1] = 123; // assign 123 to the second element of c[]
```

```
i[2] = 12345;    // assign 12,345 to the third element of i[]
x[3] = 123* i[4]; // compute 123 x the 5th element of i[] and
                // assign result to the 4th element of x[]
```

Note

In C language, the elements of an array of size N have indexes 0, 1, 2 ... ($N - 1$). It is when manipulating arrays that the *for* type of loop shows all its merits.

Let's see an example where we declare an array of 10 integers and we initialize each element of the array to a constant value of 1:

```
int a[10];    // declare array of 10 integers: a[0], a[1] ... a[9]
int i;        // the loop index
for ( i=0; i<10; i++)
{
    a[ i] = 1;
}
```

Sending a Message

The best way to conclude this lesson would be to take all the elements of the C language we have reviewed so far and put them to use in our next project. We will take another cut at the old “Hello World!” exercise. Once more, we will use only a few I/O ports to send our message to the world. Here is the code:

```
#include <config.h>

// 1. define timing constant
#define SHORT_DELAY 100
#define LONG_DELAY 800

// 2. declare and initialize an array with the message bitmap
char bitmap[30] = {
    0b11111111, // H
    0b00001000,
    0b00001000,
    0b11111111,
    0b00000000,
    0b00000000,
    0b11111111, // E
    0b10001001,
    0b10001001,
    0b10000001,
    0b00000000,
    0b00000000,
    0b11111111, // L
    0b10000000,
    0b10000000,
    0b10000000,
    0b00000000,
```

```

    0b00000000,
    0b11111111, // L
    0b10000000,
    0b10000000,
    0b10000000,
    0b00000000,
    0b00000000,
    0b01111110, // 0
    0b10000001,
    0b10000001,
    0b01111110,
    0b00000000,
    0b00000000
};

// 3. the main program
main()
{
    // 3.1 variable declarations
    int i;           // i will serve as the index

    // 3.2 initialization
    TRISA = 0;       // all PORTA as output
    T1CON = 0x8030;  // TMR1 on, prescale 1:256 Tclk/2

    // 3.3 the main loop
    while( 1)
    {
        // 3.3.1 display loop, hand moving to the right
        for( i=0; i<30; i++)
        { // 3.3.1.1 update the LEDs
            PORTA = bitmap[i];

            // 3.3.1.2 short pause
            TMR1 = 0;
            while ( TMR1 < SHORT_DELAY)
            {
            }
        } // for i

        // 3.3.2 long pause, hand moving back to the left
        PORTA = 0;    // turn LEDs off
        TMR1 = 0;
        while ( TMR1 < LONG_DELAY)
        {
        }
    } // main loop
} // main

```

For convenience I have numbered out separate sections of the code.

In section 1. we define a couple of timing constants, so that we can easily adjust later the flashing sequence speed if desired.

In section 2. we declare and initialize an 8-bit integer array of 30 elements, each containing an LED configuration in the sequence.


Hint

Using a highlighter you can mark the “1s” on the page to see the message emerge ...

Section 3. contains the main program, with the variable declarations (3.1) at the top, followed by the microcontroller initialization (3.2) and eventually the main loop (3.3).

The main (*while*) loop, in its turn, is divided into two sections:

- Section 3.3.1 contains the actual LED flash sequence, all 30 steps to be played when the board is swept from left to right. A *for* loop is used for accessing each element of the array in order. A *while* loop is used to wait on Timer1 for the proper sequence timing.
- Section 3.3.2 contains a pause for the sweep back, implemented using a *while* loop with a longer delay on Timer1.

It's time to test the program. MPLAB X makes it very easy – simply press the **Run**  button or select **Run>Program Run** from the main menu. In a quick sequence MPLAB will:

1. Launch the C compiler
2. Launch the linker to produce a binary executable
3. Program the flash memory of the PIC24 using the selected programmer/debugger tool
4. Start the execution of the program.

If all went well, what you will see at this point is just a faint “trembling” of the LEDs bar on the demo board. But if you had access to an oscilloscope or one of those small USB logic analyzers you could verify that the LEDs are not just trembling, they are trying to tell us something.

Looking at [Figure 3.1](#), where I captured the PortA signals at the terminals of the LED, can you see the message yet?

No, then I will make it really easy for you. Since we are in *run mode*, you can now disconnect the programmer/debugger from the demo board. The program will continue to run independently, just make sure to keep the power supply connected though! Now, dimming the light a bit in the room, move the board from left to right with a regular wave of the hand. Now you should be able to see it: HELLO!

Consider adjusting the timing constants to your optimal speed. After some experimentation, I found the values 100 and 800, for the short and long delays respectively, were ideal, but your preferences might differ.

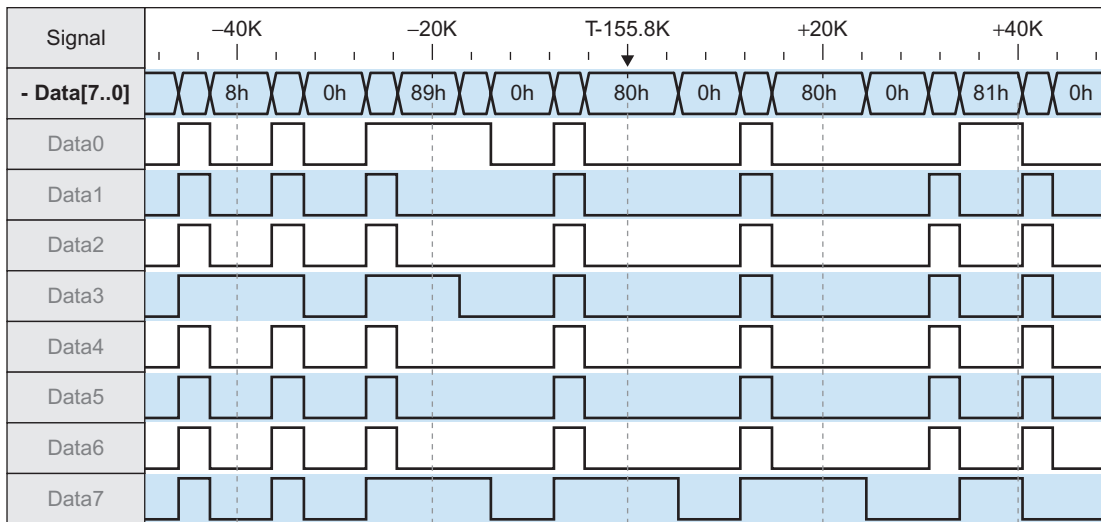


Figure 3.1: Logic analyzer capture of PortA

Post-Flight Briefing

In this lesson we reviewed the declaration of a few basic variable types, including integers and floating points of different sizes. Array declarations and their initialization were also used to create an LED display sequence and a *for* loop was used to play it back.

Notes for the Assembly Experts

If you were wondering whether the increment and decrement operators were going to be translated by the MPLAB C compiler with the *inc* and *dec* assembly instructions, you were mostly right. I am saying mostly and not always because the *++* and *--* operators are actually much smarter than that. If the variable they are applied to is an integer, as in the trivial examples above, this is certainly the case. But if they are applied to a pointer (which is a variable type that contains a memory address) they actually increase the address by the exact number of bytes required to represent the quantity pointed to. For example, a pointer to 16-bit integers will increment its address by two, a pointer to a 32-bit long integer will increment its address by four, and so on. To satisfy your curiosity, switch to the disassembly view and see how the MPLAB C compiler chooses the best assembly code dependent on the situation.

Loops in C can be confusing: should you test the condition at the beginning or the end? Should you use the *for* type or not? The fact is, in some situations the algorithm you are coding will dictate which one to use, but in many situations you will have a degree of freedom and more than one type might do. Choose the one that makes your code more readable, and if it really doesn't matter, as in the main loop, just choose the one you like and be consistent.

Notes for the PIC[®] Microcontroller Experts

Depending on the target microcontroller architecture, and ultimately the arithmetic logic unit (ALU), operating on bytes versus operating on word quantities can make a big difference in terms of code compactness and efficiency. While in the PIC16 and PIC18 8-bit architectures there is a strong incentive to use byte-sized integers wherever possible, in the PIC24 16-bit architecture word-sized integers can be manipulated just with the same efficiency. The only limiting factor preventing us from always using 16-bit integers with the MPLAB C compiler is the consideration of the relative preciousness of the internal resources of the microcontroller, in this case the RAM memory.

Notes for the C Experts

Even if PIC24 microcontrollers have a relatively large RAM memory array, embedded control applications will always have to contend with the reality of cost and size limitations. If you learned to program in C on a PC or a workstation, you probably never considered using anything smaller than an *int* as a loop index. Well, this is the time to think again. Shaving one byte at a time off the requirements of your application might, in some cases, mean the ability to select a smaller model of PIC24 microcontroller, saving fractions of a dollar that, when multiplied by the thousands or millions of units (depending on your production run rates), can mean real money saved from the bottom line. In other words, if you learn to keep the size of your variables to the strict minimum necessary, you will become a better embedded-control designer and ultimately this is what engineering is all about.

Tips & Tricks

In this last exercise we declared an array called *bitmap[]* and we asked for it to be prefilled with a specific series of values. The array, being a data structure, resides in RAM during execution. But since RAM is volatile, the compiler has to make sure that the right values (assigned with the curly brackets { } notation) are copied from a non-volatile memory – namely a table in FLASH memory. As mentioned already in both previous chapters, this is the kind of task that the *crt0* code is meant to do for us. The compiler and linker will take complete care of it though. You won't have to move a finger. Expect more on this in the coming chapters as we will learn how and when to optimize RAM usage, a precious resource in all embedded control applications.

Notes for PIC24 GA1 and GB1 Users

For those among you who have planned on using the newer GA1, GB1 and later families of PIC24F devices, there is an important enhancement to the I/O system that will require a little additional study: the *Peripheral Pin Select* or PPS. This feature was designed to allow a much increased flexibility in the allocation of pins for devices that are particularly rich in

peripherals as is the case of the GA1 and GB1 families. By increasing the pinout options available on a particular device, users can better tailor the microcontroller to their entire application, rather than trimming the application to fit the device.

The Peripheral Pin Select feature operates over a fixed subset of digital I/O pins. Users may independently map the input and/or output of any one of many digital peripherals to any one of these I/O pins. The actual I/O mapping is performed dynamically, in software, but special hardware safeguards have been included that prevent accidental changes to the peripheral mapping once established.

The total number of pins supported is very large (44 in current models, but could be extended up to 64 in future models) although not all pins are fully interchangeable. There are actually only 32 pins that can be used freely as inputs or outputs. An additional 12 pins support only input functions. Also, the PPS feature is limited to digital input/output functions and only from a reduced selection of peripherals. Interestingly, some of the pins are multiplexed on top of the analog-to-digital converter input multiplexer, which means that, for the PPS to be able to take control of those pins, as we have seen in the previous chapters, we will have to disable the individual analog input function first. If the PPS is not initialized though, each pin remains simply a generic digital I/O or analog input.

The list of peripherals that can share pins via the PPS is quite extensive and includes:

- 4 × UART (full handshake signals included)
- 3 × SPI™ (four control signals)
- 4 × External interrupt inputs
- 9 × Input Captures
- 4 × Timers (clock input)
- 9 × Output Compares
- 2 × OC Fault inputs
- 3 × Analog comparators (digital) outputs.

Most notably absent are the I²C ports and the USB port.

But it is also particularly interesting to note that multiple peripheral inputs can be connected to a single PPS pin. This means that interconnections between peripherals can be achieved directly on chip, requiring virtually no external wiring. Similarly, peripheral outputs can be connected to a number of output pins (to increase the fan-out for example) and/or to other peripheral inputs.

Even more interesting is the fact that the architects of the PIC24 have designed the PPS mapping to make it impossible for the user to create “conflicts” among peripherals!

This has been achieved by defining the mapping of inputs and outputs from two opposite perspectives:

- Output pin selections are performed at the pin. So that for each output pin you can select only one peripheral output signal out of the 64 (max. theoretical) options available.
- Input selections are performed at the peripheral instead. So for each peripheral input you can select only one input signal out of the 64 (max. theoretical) options available.

The actual number of options does vary from PIC24F model to model and depending on the specific package chosen. Brilliant!

All this means that there is going to be a very large number of control registers devoted to the PPS selection – in fact, there are as many as thirty-seven 16-bit special function registers in the GB1 family. You will be glad to know that you are not required to initialize them all; on the contrary, in most applications you will have to carefully initialize only a handful, leaving the rest to their default value.

Here you will find some useful examples that will configure several PPS pins on the PIC24 GA1 and GB1 series to take the default role they have on the PIC24 GA0 series.

```
// SPI1
PPSOutput( PPS_RP15, PPS_SD01);    // SD01 =RP15 F8/pin 53

// SPI2
PPSInput( PPS_SDI2,  PPS_RP26);    // SDI2 =RP26 G7/pin 11
PPSOutput( PPS_RP22, PPS_SCK2OUT); // SCK2 =RP21 G6/pin 10
PPSOutput( PPS_RP21, PPS_SD02);    // SD02 =RP19 G8/pin 12

// UART
PPSInput( PPS_U2RX,  PPS_RP10);    // U2RX =RP10 F4/pin 49
PPSInput( PPS_U2CTS, PPS_RP32);    // U2CTS=RP32 F12/pin 40
PPSOutput( PPS_RP17, PPS_U2TX);    // U2TX =RP17 F5/pin 50
PPSOutput( PPS_RP31, PPS_U2RTS);   // U2RTS=RP31 F13pin 39

// IC
PPSInput( PPS_IC1,   PPS_RP2);     // IC1  =RP2  D8/pin 68

// OC
PPSOutput( PPS_RP19, PPS_OC1);     // OC1  =RP11 D0/pin 72
PPSOutput( PPS_RP11, PPS_OC2);     // OC2  =RP24 D1/pin 76
PPSOutput( PPS_RP24, PPS_OC4);     // OC4  =RP22 D3/pin 78
```

In fact, these assignments are exactly what you will need to follow the examples in the next few chapters of this book. As you can see, this is also one of those cases where the use of the peripheral libraries (*pps.h*) pays off. Performing these assignments directly on the control registers would have required a significant amount of time just to be able to match each PPS pin input/output with one of the 37 special function registers, not a fun activity.

Unfortunately, if you were planning on using a PIC24F GB1 series device PIM on the Explorer16 board, there is an additional complication that you need to be aware of. Since the two USB pins (D+ and D−) would have been in conflict with the Explorer16 assigned connections, the designers had to re-route (swap) some of the I/Os on the PIM to new locations. So now you would have to consider carefully how the “PIM remapping” adds on top of the “PPS remapping” for you to be able to connect the peripherals back the way a GA0 device would have!

All things considered, you might want to add the following assignment in case a GB1 PIM is used (GA1 users won’t have to worry about this!):

```
// GB110 PIM pin-remapping to accomodate additional USB pins
// GB110 shares usage of D2/pin 77 between SDI1 and OC3
//      pin 54 SDI1 is remapped to Explorer pin 77/D2
// NOTE: we will use it only as OC3
//      pin 55 SCK1 is remapped to Explorer pin 25/B0
// NOTE: pin 55 is input only, connecting it to SCK1
//      restricts usage to "slave mode" only
//      pin 56 RG3 is remapped to Explorer pin 89/G1
//      pin 57 RG2 is remapped to Explorer pin 90/G0

#ifdef __PIC24FJ256GB110__
    PPSOutput( PPS_RP23, PPS_OC3OUT);    // OC3=RP23 D2/pin 77
#endif
```

For obvious safety reasons, the PIC24 architects added a hardware locking mechanism to ensure that no unintentional modifications are performed by rogue code. To close the lock we can use another library provided function:

```
// Done, lock the PPS configuration
PPSLock;
```

Now we can wrap all of the above inside a function that we might call *InitPPS()*.

```
int InitPPS(void)
{
    ...
    return 0;
} // InitPPS
```

The ideal place to perform such initializations is obviously at the top of the *main()* function and before the main loop, but rather than retyping all of the above in each project that makes use of the PPS remappable peripherals (Chapter 7 and later will start making use of them) I recommend that we put it in a small file that we will call *EX16.c* and that we will save in the */lib* directory for later use.

In the same file we can add a more generic *InitEX16()* function, which might perhaps be useful to all future projects where the *InitPPS()* function is inserted automatically as soon as the presence of a PIC24 GA1 or GB1 processor is detected by the compiler.

```
/*
** EX16.c
*/
#include <EX16.h>

void InitEX16( void)
{
    // if using a GA1 or GB1 PIM, initialize the PPS module
    #if defined(__PIC24FJ256GB110__) || defined(__PIC24FJ256GA110__)
    #include <pps.h>
    InitPPS();
    #endif

    // prepare Port A for use with LED bar
    LATA = 0;           // all LED off
    TRISA = 0xFF00;     // all output
} // InitEX16
```

Note

The two symbols `__PIC24FJ256GA110__` and `__PIC24FJ256GB110__` are defined automatically by MPLAB X when you select the target device during the project creation initial steps. They are passed to the MPLAB C compiler and linker every time you invoke a build command.

A corresponding header file **EX16.h** (to be saved in the */include* subdirectory) will come in handy to publish the new *InitEX16()* function and to offer additional useful information about the Explorer16 board, such as the instruction clock FCY that is really dependent on the PIC24 oscillator configuration but also on the crystal (8MHz) mounted on the board.

```
/*
** EX16.h
**
** Standard definitions for use with the Explorer16 board
*/
#ifndef _EX16
#define _EX16

#include <p24fxxxx.h>

#if defined(__PIC24FJ256GB110__) || defined(__PIC24FJ256GA110__)
#include <pps.h>
#endif

#define FCY 16000000UL // instruction clock 16MHz

// prototypes
void InitEX16(void); // initialize the Explorer16 board
#endif
```

Since the external crystal is an 8 MHz quartz, and since in the device configuration we chose to enable the 4x PLL circuit, the resulting PIC24F clock will be at 32 MHz, and the internal instruction execution (often referred to as Fcy on the PIC24 datasheet) will be 16 MHz.

Speaking of the device configuration, the PIC24F GA1 and GB1 series will require a separate treatment there as well since additional configuration bits (`_CONFIG2`) are available. You might want to add the following lines to the `config.h` file we created in the first chapter:

```
#if defined ( __PIC24FJ128GA010__ ) || defined ( __PIC24FJ256GA110__ )
_CONFIG2( IESO_OFF           // two speed start up disabled
          & FCKSM_CSDCMD     // disable clock-switching/monitor
          & FNOSC_PRIPLL     // primary oscillator: enable PLL
          & POSCMOD_XT)      // primary oscillator: XT mode

#else // GB1 configuration requires additional detail
_CONFIG2( PLL_96MHZ_ON      // enable USB PLL module
          & PLLDIV_DIV2     // 8MHz/2 = 4MHz input to USB PLL
          & IESO_OFF         // two speed start up disabled
          & FCKSM_CSDCMD     // disable clock-switching/monitor
          & FNOSC_PRIPLL     // primary oscillator: enable PLL
          & POSCMOD_XT)      // primary oscillator: XT mode

#endif
```

Exercises

- Improve the display/hand synchronization by waiting for a button to be pressed before the hand sweep is started.
- Add a switch to sense the sweep movement reversal and play the LED sequence backwards on the back sweep.

Books

- Rony, P., Larsen, D., Titus, J., 1976, *The 8080A Bugbook*, Microcomputer Interfacing and Programming, Howard W. Sams & Co., Inc., Indianapolis, IN.
This is the book that introduced me to the world of microprocessors and changed my life forever. No high-level language programming here, just the basics of assembly programming and hardware interfacing. (Too bad this book is already considered museum material; see link below.)
- Shulman, S., 2003, *Unlocking the Sky, Glenn Hammond Curtis and the Race to Invent the Airplane*, Harper Collins, New York, NY.
A beautiful recount of the “struggle to innovate” in the early days of aviation.

Links

- <http://www.bugbookcomputermuseum.com/BugBook-Titles.html>.
A link to the “Bugbooks museum”. It is forty years since the introduction of the INTEL 8080 microprocessor and it is like centuries have already passed.

Numb3rs

The human sense of equilibrium is based on a device (the labyrinth or vestibular apparatus) located inside the ear that gives us feedback on gravity and motion. But, unlike that of birds, ours was just not designed for flight. It can be easily tricked by a little centrifugal acceleration and, in the absence of visual clues (in fog, clouds or during a night flight), it can have us flying happily into a tightening spiral ... into the ground. To overcome our shortcomings, we have to rely on instruments to tell us how fast we are flying, in which direction, and perhaps most importantly, which way is up. Practically, this means that most of the information that directly reaches the brain of a bird from his senses will arrive to our brain only in the form of numbers.

A good portion of the time spent by a student pilot in an airplane after the first few flights will be spent learning the “right” numbers for his airplane – like the best climb speed, the best glide speed, the take-off (rotation) speed, the approach speeds and so on. Most of the time, these numbers are available inside the Pilot Operating Handbook (POH), the airplane datasheet, and, for convenience, on the related checklists. Each pilot tries his best to follow them religiously so that his flying performance gains consistency as he improves his command of the machine. However, even the most experienced aerobatic pilots, and certainly all the airline pilots who fly thousands of hours every year, will tell you how flying can be extremely spontaneous, if you know exactly all your numbers!

Similarly, in embedded control, we need to know the numeric types well, their relative performance, as well as the costs and benefits of each one.

Flight Plan

In this lesson we will review all the numerical data types offered by the MPLAB® C compiler for the PIC24. We will learn how much memory the compiler allocates for the numerical variables and we will investigate the relative efficiency of the routines used to perform arithmetic operations by using a timer as a stopwatch or a profiling tool. This experience will help you choose the *right* numbers for your embedded control application, and understand when and how to balance performance, memory resources, real time constraints and complexity.

Preflight Checklist

In this lesson we will continue using:

- MPLAB X IDE, free Integrated Development Environment (obtain the latest version available for download from Microchip's Web site at <http://www.microchip.com/mplab>)
- MPLAB C30 Lite compiler v 3.30 (or later) or MPLAB XC16 Lite compiler
- Any PIC24 model currently supported by MPLAB X
- An MPLAB X compatible programmer/debugger such as the PICKit3, ICD3 or Real ICE
- The Explorer16 board, or pretty much any PIC24 demo board you might have available

In preparation for the new demonstration project, you can use the **New Project Setup** checklist to create a new project called **4-Numb3rs**. Also use the **New File** wizard to create a new source file to be called **Numbers.c**.

The Flight

To review all the data types available, I recommend you take a look at the MPLAB C Compiler User Guide. You can start in Chapter 5 where you can find a first list of the supported integer types.

As you can see (Table 4.1), there are 10 different integer types as specified in the ANSI C standard including: *char*, *int*, *short*, *long*, and *long long*, both in the *signed* (default) and *unsigned* variant. The table shows the number of bits allocated specifically by the MPLAB C compiler for each type, and, for your convenience, spells out the minimum and maximum value that can be represented.

It is expected that when the type is signed, one bit must be dedicated to the sign itself and the resulting numerical range is therefore halved. It is also interesting to note how the

Table 4.1: Integer data types

Type	Bits	Min.	Max.
char, signed char	8	−128	127
unsigned char	8	0	255
short, signed short	16	−32768	32767
unsigned short	16	0	65535
int, signed int	16	−32768	32767
unsigned int	16	0	65535
long, signed long	32	−2 ³¹	2 ³¹ − 1
unsigned long	32	0	2 ³² − 1
long long**, signed long long**	64	−2 ⁶³	2 ⁶³ − 1
unsigned long long**	64	0	2 ⁶⁴ − 1

**ANSI-89 extension.

C compiler treats *int* and *short* as synonyms by allocating 16 bits for both of them. Both 8- and 16-bit quantities can be processed efficiently by the PIC24 Arithmetic and Logic Unit (ALU), so that most of the arithmetic operations can be coded by the compiler using few and efficient instructions. The *long* integers are treated as 32-bit quantities, using four bytes, while the *long long* type (specified by the ANSI C extensions in 1989) requires eight bytes. Operations on *long* integers are performed by the compiler using short sequences of instructions inserted inline. So, there is a small performance penalty to pay for using *long* integers, and a proportionally larger penalty to pay for *long long* integers, that must be taken into account.

Let's look first at an integer example; we'll start typing the following code:

```
unsigned int i,j,k;

main ()
{
    i = 0x1234;    // assign an initial value to i
    j = 0x5678;    // assign an initial value to j
    k = i * j;      // perform product and store the result in k
}
```

After building the project, click on the **Build Project**  button (alternatively select

Run>Batch Build Project). We can open the Embedded Memory window by choosing **Window>Embedded Memory** and by selecting **Program** from the *Memory* drop box, and **Symbol** from the *Format* drop-box.

From here we can take a look at the code generated by the compiler. Even without knowing the PIC24 instruction set in detail, we can recognize the two assignments. They are performed by loading the literal values to register *W0* first and from there to the memory locations reserved for the variable *i*, and later for variable *j*.

```

                i = 0x1234;
212340    MOV #0x1234, W0          // move literal to W0
884280    MOV W0, 0x400           // move from W0 to i
                j = 0x5678;
256780    MOV #0x5678, W0        // move literal to W0
884290    MOV W0, 0x401           // move from W0 to j
                k = i * j;
804281    MOV 0x400, W1           // move from i to W1
804290    MOV 0x401, W0           // move from j to W0
B98800    MUL.SS W1, W0, W0
8842A0    MOV W0, 0x402           // move result to k
```

The multiplication is performed by transferring the values from the locations reserved for the two integer variables *i* and *j* back to registers *W0* and *W1*, and then performing a single *mul* instruction. The result, available in *W0*, is stored back into the locations reserved for *k*. Pretty straightforward!

You will notice how the overall program, as compiled, is somewhat redundant. The value of *j*, for example, is still available in register *W0* when it is reloaded again – just before the multiplication. Can't the compiler see that this operation is unnecessary?

In fact, the compiler does not see things this clearly – its role is to create *safe* code, avoiding (at least initially) any assumption and using standard sequences of instructions. Later on, if the proper optimization options are enabled, a second pass (or more) is performed to remove the redundant code. During the development and debugging phases of a project though, it is always good practice to disable all optimizations (level 0) as they might modify the structure of the code being analyzed and render single stepping and breakpoint placement problematic. In the rest of this book we will consistently avoid making use of compiler optimizations until the very last minute and, even when we have verified that the required levels of performance cannot be obtained by any other means, we will step up only to optimization level 1. As a consequence you will be able to execute all the examples presented in this and the following chapters using the MPLAB C Lite compiler which is free and available from the Microchip website.

Going Long

At this point, modifying only the first line of code, we can change the entire program to perform operations on *long* (32-bit) integer variables.

```
unsigned long i,j,k;

main ()
{
    i = 0x01234567L;    // assign an initial value to i
    j = 0x89ABCDEFL;    // assign an initial value to j
    k = i * j;          // perform product and store the result in k
}
```

Re-building the project and switching again to the Embedded Memory window, we can see how the newly generated code is considerably longer than the previous version.

Note

If you had the Editor window maximized and you have not closed the Embedded Memory window yet, you could use the **CTRL-Tab** command to quickly alternate between the Editor and the Embedded Memory windows.

While the initializations are still straightforward, the multiplication is now performed using several more instructions.

```

                                k = i * j;
804038      MOV 0x403, W8
804049      MOV 0x404, W9
```

```

804056      MOV 0x405, W6
804067      MOV 0x406, W7
780088      MOV W8, W1
780006      MOV W6, W0
B80A00      MUL.UU W1, W0, W4
B9C007      MUL.SS W8, W7, W0
780105      MOV W5, W2
410100      ADD W2, W0, W2
B9B009      MUL.SS W6, W9, W0
410100      ADD W2, W0, W2
780282      MOV W2, W5
884074      MOV W4, 0x407
884085      MOV W5, 0x408

```

The PIC24 Arithmetic and Logic Unit can only process 16 bits at a time, so the 32-bit multiplication is actually performed as a sequence of 16-bit multiplications and additions. The sequence used by the compiler is generated with pretty much the same technique that we learned to use in elementary school, only performed on a word at a time rather than a digit at a time. In practice, to perform a 32-bit multiplication using 16-bit instructions, there should be four multiplications and two additions, but you will note how the compiler has actually inserted only three multiplication instructions. What is going on here?

The fact is that multiplying two long integers (32-bit each) will produce a 64-bit-wide result. But in the example above, we have requested the result to be stored in yet another long variable, therefore limiting the result to a maximum of 32 bits. Doing so, we have clearly left the door open for the possibility (not so remote) of an overflow, but we have also given the compiler the permission to ignore the most significant bits of the result. Knowing those bits are not going to be missed, the compiler has eliminated completely the fourth multiplication step and, in a way, has already optimized the code.

Long Long Multiplications

Changing the variables declarations to the *long long* integer type (64-bit) is just as simple:

```

unsigned long long i,j,k;

main ()
{
    i = 0x0123456789ABCDEFLL; // assign an initial value to i
    j = 0xFEDCBA9876543210LL; // assign an initial value to j
    k = i * j;      // perform product and store the result in k
}

```

Recompiling and inspecting the new code in the Embedded Memory window reveals how this time the compiler has chosen a different approach. Instead of a longer sequence inserted inline, there are now only a few instructions to transfer the data into predefined registers and there is a call to a subroutine.

```

07FDFC      RCALL 0x290

```

The subroutine will appear in the disassembly listing, after all the main function code. This subroutine is clearly separated and identified by a comment line that indicates it is part of a library, a module called *muldi3.c*. The source for this routine is actually available as part of the complete documentation of the MPLAB C compiler and can be found in the subdirectory *src/libm/src/* under the same directory tree where the compiler has been installed on your hard disk.

By selecting a subroutine in this case, the compiler has clearly made a compromise. Calling the subroutine means adding a few extra instructions and using extra space on the stack. On the other hand, fewer instructions will be added each time a new multiplication (among *long long* integers) is required in the program; therefore code space will be saved.

Floating Point

Beyond integer data types, the C compiler offers support for a few more data types that can capture fractional values – the floating point data types (Table 4.2). There are three types to choose from corresponding to two levels of resolution: *float*, *double* and *long double*.

Notice how the MPLAB C compiler, by default, allocates for both the *float* and the *double* types the same number of bits, using the single-precision floating point format defined in the IEEE754 standard. Only the *long double* data type is treated as a *true* double-precision IEEE754 floating-point type.

Notes for the C Experts

It is my belief that these floating-point settings were intentionally used by the MPLAB C compiler designers to simplify and make more efficient the porting of complex math algorithms to embedded-control target applications. Most of the algorithms and libraries available in literature are designed for the performance and resources of personal computers and workstations, and make use of double-precision floating-point arithmetic whenever possible to maximize accuracy. Most often in embedded control, we are willing to compromise some of that accuracy for the level of performance necessary to achieve real-time response.

Table 4.2: Floating point types

Type	Bits	E Min.	E Max.	N Min.	N Max.
float	32	−126	127	2^{-126}	2^{128}
double*	32	−126	127	2^{-126}	2^{128}
long double	64	−1022	1023	2^{-1022}	2^{1024}

E = Exponent.

N = Normalized (approximate).

*double is equivalent to long double if no short double is used.

If needed, this behavior can be changed either locally, by turning doubles in *long doubles* in selected cases, or globally by using a special compiler option (Figure 4.1):

- Open the Project Properties dialog box by selecting **Run>Set Project Configuration>Customize...**
- Select the **pic30-gcc** options category
- Check or un-check the **Use 64-bit double** check box

Since the PIC24 doesn't have a hardware Floating Point Unit (FPU), all operations on floating point types must be coded by the compiler using floating-point arithmetic libraries whose size and complexity are considerably larger/higher than any of the integer libraries. You should expect a major performance penalty if you choose to use these data types but, again, if the problem calls for fractional quantities to be taken into account, the C compiler certainly makes dealing with them easy.

Let's modify our previous example to use floating-point variables:

```
float i,j,k;

main ()
{
    i = 12.34;    // assign an initial value to i
    j = 56.78;    // assign an initial value to j
    k = i * j;    // perform product and store the result in k
}
```

After recompiling and inspecting the Embedded Memory window, you will notice that the compiler has immediately chosen to use a subroutine instead of inline code.

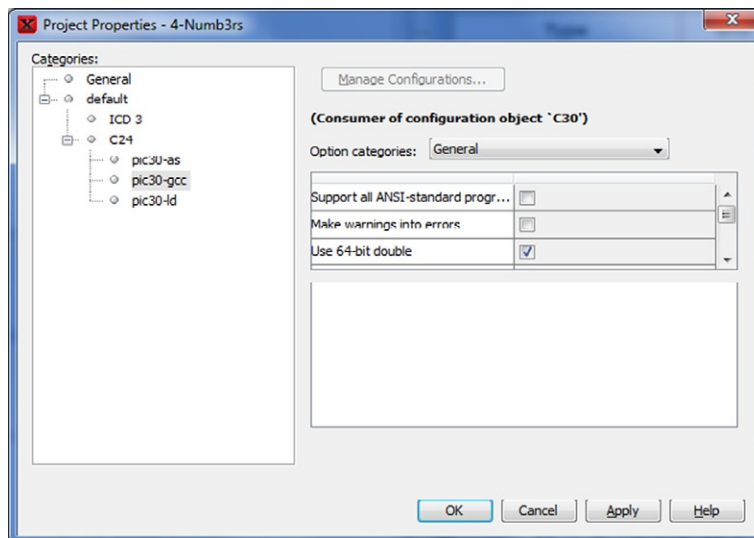


Figure 4.1: The Project Properties dialog box

Changing the program again to use double-precision floating-point type, *long double*, produces very similar results. Only the initial assignments seem to be affected, and all we can see is a subroutine call.

The MPLAB C compiler makes using any data type so easy that we might be tempted to always use the largest integer or floating-point type available, just to stay on the safe side and avoid the risk of overflows and underflows. On the contrary, choosing the right data type for each application can be critical in embedded control to balance performance and optimize the use of resources. In order to make an informed decision, we need to know more about the level of performance we can expect when choosing data types of various precision.

Measuring Performance

In the following exercise we will use Timer1 as a very rudimentary profiling tool to measure the actual relative performance of the arithmetic libraries, both integer and floating point, used by the MPLAB C compiler. We can start using the following code:

```
/*
** Numbers.c
*/
#include <config.h>

int      i1, i2, i3;
long     x1, x2, x3;
long long y1, y2, y3;
float    f1, f2, f3;
long double d1, d2, d3;

main()
{
    T1CON = 0x8000;    // enable Timer1 1:1 with main clock

    i1 = 0x1234;       // testing integers (16-bit)
    i2 = 0x5678;
    TMR1 = 0;          // clear the timer
    i3 = i1 * i2;

    x1 = 0x01234567L;  // testing long integers (32-bit)
    x2 = 0x89ABCDEFL;
    TMR1 = 0;          // clear the timer
    x3 = x1 * x2;


    y1 = 0x0123456789ABCDEFLL; // testing 64-bit integers
    y2 = 0xFEDCBA9876543210LL;
    TMR1 = 0;          // clear the timer
    y3 = y1 * y2;

    f1 = 12.34;        // testing single precision floating point
    f2 = 56.78;
    TMR1 = 0;          // clear the timer
    f3 = f1 * f2;
```

```

d1 = 12.34L;    // testing double precision floating point
d2 = 56.78L;
TMR1 = 0;
d3 = d1 * d2;
} // main

```

Enter the Project Debug mode (select **Debug>Debug Project** from the main menu or press **CTRL+F5**) and press the **Reset**  button to position the green bar (representing the processor program counter) to the first line of the main function.

Then proceed by single stepping by clicking on the **Step Over**  button.

By configuring the Timer1 to count the main clock cycles (1:1 pre-scaler setting) and taking care to clear the counter just before each multiplication, we can easily record the number of cycles required by each integer type multiplication.

To read the count, it will suffice to open the Watches window (select **Window>Watches** from the main menu), adding the register **TMR1** to the list.

Continue single stepping through the code until all five types have been performed and note the corresponding values in a table.

In [Table 4.3](#), obtained using MPLAB C compiler with all optimizations disabled, I have recorded the results of the experiment in the first column (expressed in cycle counts) and then I added three more columns to show the relative performance ratios, obtained by dividing the cycle count of each row by the cycle count recorded for the reference type. Don't be alarmed if you happen to record different values as several factors can affect the measure. Future versions of the compiler could possibly use more efficient libraries, and/or optimization features could be introduced or enabled at the time of testing.

Keep in mind that this type of test lacks any of the rigorousness required by a true benchmark. What we are looking for here is just a basic understanding of the impact on the performance we can expect from choosing to perform our calculations with one data type versus another. We are looking for the big picture – relative orders of magnitude. For that purpose, the table we just obtained can already give us some interesting indications.

Table 4.3: Relative performance test summary

Multiplication Test	Cycle Count	Performance Relative To		
		<i>Int</i>	<i>Long</i>	<i>Float</i>
Integer	4	1	—	—
Long integer	15	3.75	1	—
Long long integer	108	27	7	—
Single precision f.p.	121	30	8	1
Double precision f.p.	321	80	21	2.6

As expected, 16-bit operations appear to be the fastest. Long-integer (32-bit) multiplications are about four times slower, while long-long-integer (64-bit) multiplications are one order of magnitude slower. Again, it was expected that single-precision floating-point operations would require more effort than integer operations. Multiplying a 32-bit integer is only about four times slower than multiplying a 16-bit integer. However, multiplying 32-bit floating-point numbers is more than thirty times slower than multiplying 16-bit integers. That means it is eight times slower than the corresponding 32-bit integer multiplication, or about an order of magnitude. Going to double-precision floating point (64-bit), though, only doubles the number of cycles. This tells us that, apparently, the double-precision floating-point libraries used by the compiler are more efficient than the corresponding 64-bit integer libraries.

So, when should we use floating point and when should we use integer arithmetic?

Beyond the obvious, from the little we have learned so far we can perhaps extract the following rules:

1. Use integers every time you can (i.e. when fractions are not required, or the algorithm can be re-written for integer arithmetic).
2. Use the smallest integer type that will not produce an overflow or underflow.
3. If you have to use a floating-point type (fractions are required), expect an order-of-magnitude reduction in the performance of the compiled program.
4. Double-precision floating point (*long double*) seems to only reduce the performance by a further factor of two.

Keep in mind also that floating-point types offer the largest value ranges, but also are always introducing approximations. As a consequence, floating-point types cannot be used for financial calculations. Use *long* or *long long* integers instead, and perform all operations in cents (instead of dollars and fractions).

Post-Flight Briefing

In this lesson, we have learned not only what data types are available and how much memory is allocated to them, but also how they affect the resulting compiled program – code size and the execution speed. We used Timer1 as a stopwatch of sorts to measure the number of instruction cycles required for the execution of a series of code segments. Some of the information gathered will, hopefully, be useful to guide our actions in the future when balancing our needs for precision and performance in embedded control applications.

Notes for the Assembly Experts

The brave few assembly experts who have attempted to deal with floating-point numbers in their applications tend to be extremely pleased and forever thankful for the great

simplification achieved by the use of the C compiler. Single- or double-precision arithmetic becomes just as easy to code as integer arithmetic has always been.

When using integer numbers, though, there is sometimes a sense of loss of control, as the compiler hides the details of the implementation and some operations might become obscure or much less intuitive/readable. Here are some examples of conversion and byte manipulation operations that can induce some anxiety:

1. Converting an integer type into a smaller/larger one.
2. Extracting or manipulating the most or least significant byte of a 16-bit data type.
3. Extracting or manipulating one bit out of an integer variable.

The C language offers convenient mechanisms for covering all such cases via *implicit* type conversions as in:

```
int    x;      // 16-bit
long   y;      // 32-bit
y = x;
```

The value of *x* is transferred into the least significant byte (LSB) of *y* while the most significant byte (MSB) of *y* is cleared.

Otherwise *explicit* conversions might be required (these are often referred to as *type casting*) in cases where the compiler would otherwise assume an error, as in:

```
int    x;      // 16-bit
long   y;      // 32-bit
x = (int) y;
```

Here *(int)* is a *type cast* forcing the most significant byte (MSB) of *y* to be discarded as *y* is treated as a 16-bit value.

Bit fields are used to cover the conversion to and from integer types that are smaller than one byte. Bit fields are treated by the MPLAB C compiler with great efficiency and will result in the use of bit manipulation instructions whenever possible. The PIC24 library files contain numerous examples of definitions of bit fields for the manipulation of all the control bits in the peripheral and the core special function registers.

Here is an example extracted from the include file used in our project, where the Timer1 control register *TICON* is defined and each individual control bit is exposed in a structure defined as *TICONbits*:

```
extern unsigned int TICON;
extern union {
    struct {
        unsigned :1;
        unsigned TCS:1;
        unsigned TSYNC:1;
```

```
    unsigned :1;
    unsigned TCKPS0:1;
    unsigned TCKPS1:1;
    unsigned TGATE:1;
    unsigned :6;
    unsigned TSIDL:1;
    unsigned :1;
    unsigned TON:1;
};
struct {
    unsigned :4;
    unsigned TCKPS:2;
};
} T1CONbits;
```

Notes for the PIC[®] Microcontroller Experts

The PIC microcontroller user, familiar with the 8-bit PIC microcontrollers and their respective compilers, will notice a considerable improvement in the performance both with integer arithmetic and with floating-point arithmetic. The 16-bit ALU available in the PIC24 architecture is clearly providing a great advantage by manipulating twice the number of bits per cycle, but the performance improvement is further accentuated by the availability of up to eight working registers which make the coding of critical arithmetic routine and makes numerical algorithms more efficient.

Tips & Tricks

Math Libraries

The MPLAB C compiler supports several standard ANSI C libraries including:

- *limits.h* – contains many useful macros defining implementation dependent limits, such as, for example, the number of bits composing a char type (CHAR_BIT) or the largest integer value (INT_MAX).
- *float.h* – contains similar implementation dependent limits for floating point data types, such as, for example, the largest exponent for a single precision floating point variable (FLT_MAX_EXP).
- *math.h* – contains trigonometric functions, rounding functions, logarithms and exponentials.

Complex Data Types

The MPLAB C compiler supports *complex* data types as an extension of both integer and floating-point types. Here is an example declaration for a single-precision floating-point type:

```
__complex__ float z;
```

Notice the use of a double underscore before and after the keyword *complex*.

The variable *z*, defined so, has now a real and an imaginary part that can be individually addressed using the syntax: `__real__ z` and `__imag__ z` respectively.

Similarly, the next declaration produces a complex variable of 16-bit integer type:

```
__complex__ int x;
```

Complex constants are easily created adding the suffix *i* or *j*, as in the following examples:

```
x = 2 + 3j;
z = 2.0f + 3.0fj;
```

All standard arithmetic operations (+, −, *, /) are performed correctly on complex data types. Additionally the “~” operator produces the complex conjugate.

Complex types could be pretty handy in some types of applications, making the code more readable and helping avoid trivial errors. Unfortunately, as of this writing, the MPLAB IDE support of complex variables during debugging is only partial, giving access only to the *real* part through the Watches window and the mouse-over function.

Exercises

1. Write a program that uses Timer2 and Timer3 joined in the new 32-bit timer mode.
2. Test the relative performance of the division for the various data types.
3. Test the performance of the trigonometric functions relative to standard arithmetic operations.
4. Test the relative performance of the multiplication for complex data types.

Books

- Gahlinger, P.M., 2000. *The Cockpit, A Flight of Escape and Discovery*, Sagebrush Press, Salt Lake City, UT.
It's an interesting journey around the world. Follow the author in search of ... his soul. Every instrument in the cockpit triggers a memory and starts a new chapter.

Links

- http://en.wikipedia.org/wiki/Taylor_series
If you are curious about how the C compiler can approximate some of the functions in the math library.

Interrupts

Every pilot is taught to keep his eyes constantly scanning the horizon, looking for visual clues about position and direction of flight, and looking for other airplanes. But he also needs to check the airplane instruments momentarily to verify his speed and his altitude and keep an eye on the map. Now and then there might be the need to focus longer on one of the inputs and it is essential to learn how some instruments need a more frequent check than others, depending on the phase of the flight and a number of other conditions. In other words, pilots need to learn to multi-task, assigning the correct priority to each instrument and optimizing the use of time so as to always stay ahead of the machine.

In the world of embedded control, for reasons of efficiency, size and ultimately cost, the smallest applications, which happen to be implemented in the highest volumes, most often cannot afford the “luxury” of a multi-tasking operating system and use the interrupt mechanisms instead to “divide their attention” between the many tasks at hand.

Flight Plan

In this lesson we will see how the MPLAB[®] C compiler for the PIC24 allows us to easily manage the interrupt mechanisms offered by the PIC24 microcontroller architecture. After a brief review of some of the C language extensions and some practical considerations, we will present a short example on how to use the secondary (low frequency) oscillator to maintain a real-time clock.

Preflight Checklist

In this lesson we will continue using:

- MPLAB X IDE, free Integrated Development Environment (obtain the latest version available for download from Microchip’s Web site at <http://www.microchip.com/mplab>)
- MPLAB C30 Lite compiler v.3.30 (or later) or the MPLAB XC16 Lite compiler
- Any PIC24 model currently supported by MPLAB X
- An MPLAB X compatible programmer/debugger such as the PICkit3, ICD3 or Real ICE
- The Explorer16 board, or pretty much any PIC24 demo board you might have available

Use the **New Project Setup** checklist to create a new project called **5-Interrupts** and a new source file similarly called **Interrupts.c**.

The Flight

An interrupt is an internal or external event that requires quick attention from the CPU. The PIC24 architecture provides a rich interrupt system that can manage as many as 118 distinct sources of interrupts. Each interrupt source can have a unique piece of code, called the Interrupt Service Routine (ISR), directly associated via a pointer (which is also called a *vector*), to provide the required response action. Interrupts can be completely asynchronous with the execution flow of the main program. They can be triggered at any point in time and in an unpredictable order. Responding quickly to interrupts is essential to allow prompt reaction to the trigger event and a fast return to the main program execution flow. Therefore the goal is to minimize the interrupt latency, defined as the time between the triggering event and the execution of the first instruction of the *Interrupt Service Routine* (ISR). In the PIC24 architecture, not only is the latency very short but it is also fixed for each given interrupt source – only three instruction cycles for internal events and up to four instruction cycles for external events. This is a highly desirable quality that makes the PIC24 interrupt management superior to most other architectures.

The MPLAB C compiler helps managing the complexity of the interrupt system by providing a few language extensions. The PIC24 keeps all interrupt vectors in one large *Interrupt Vector Table* (IVT) and the MPLAB C compiler can automatically associate interrupt vectors with “special” user-defined C functions as long as a few limitations are kept in consideration such as:

- They are not allowed to return any value (use type *void*).
- No parameter can be passed to the function (use parameter *void*).
- They cannot be called directly by other functions.
- Ideally, they should not call any other function.

The first three limitations should be pretty obvious given the nature of the interrupt mechanism – since it is triggered by an external event, there cannot be parameters or a return value because there is no proper function call in the first place. The last limitation is more of a recommendation to keep in mind for efficiency considerations.

The following example illustrates the syntax that could be used to associate a function to the Timer1 interrupt vector:

```
void __attribute__(( interrupt )) _T1Interrupt ( void)
{
    // interrupt service routine code here...
} // _InterruptVector
```

The function name `_T1Interrupt` is not an accidental choice. It is actually the predefined identifier for the Timer1 interrupt as found in the Interrupt Vectors Table of the PIC24 (defined in the datasheet), and as coded in the linker script, the `.gld` file loaded for the current project.

The `__attribute__ (())` mechanism is used by the C compiler in this and many other circumstances as a way to specify special features as C language extension.

Personally, I find this syntax too lengthy and hard to read. I recommend the use of a couple of macros that can be found in each PIC24 include (.h) file and that greatly improve the code readability. In the following example the `_ISR` macro is used to the same effect as the previous code snippet:

```
void _ISR _T1Interrupt (void)
{
    // interrupt service routine code here...
} // _InterruptVector
```

From [Table 5.1](#), taken from the PIC24FJ128GA010 family datasheet, you can see which events can be used to trigger an interrupt.

Among the external sources available for the PIC24FJ128GA010, there are:

- 5 × external pins with level trigger detection
- 22 × external pins connected to the Change Notification module
- 5 × Input Capture modules
- 5 × Output Compare modules
- 2 × serial port interfaces (UARTs)
- 4 × synchronous serial interfaces (SPI™ and I²C™)
- Parallel Master Port.

And among the internal sources we count:

- 5 × 16-bit Timers
- 1 × analog-to-digital converter
- 1 × Analog Comparators module
- 1 × real-time clock and calendar
- 1 × CRC generator.

Note

The PIC24F GA1, GB1, DA2 and GB2 families add several more peripheral sources each capable of generating additional interrupt events, including several additional Timers, a USB interface (GBx), and a CTMU or touch sensing interface.

Table 5.1: Interrupt vector table of the PIC24F GA0 family

Interrupt Source	Vector Number	IVT Address	Interrupt Bit Locations		
			Flag	Enable	Priority
adc1 Conversion Done	13	00002Eh	IFS0<13>	IEC0<13>	IPC3<6:4>
Comparator Event	18	000038h	IFS1<2>	IEC1<2>	IPC4<10:8>
crc generator	67	00009Ah	IFS4<3>	IEC4<3>	IPC16<14:12>
External Interrupt 0	0	000014h	IFS0<0>	IEC0<0>	IPC0<2:0>
External Interrupt 1	20	00003Ch	IFS1<4>	IEC1<4>	IPC5<2:0>
External Interrupt 2	29	00004Eh	IFS1<13>	IEC1<13>	IPC7<6:4>
External Interrupt 3	53	00007Eh	IFS3<5>	IEC3<5>	IPC13<6:4>
External Interrupt 4	54	000080h	IFS3<6>	IEC3<6>	IPC13<10:8>
12c1 Master Event	17	000036h	IFS<1>	IEC1<1>	IPC4<6:4>
12c1 Slave Event	16	000034h	IFS1<0>	IEC1<0>	IPC4<2:0>
12c2 Master Event	50	000078h	IFS3<2>	IEC3<2>	IPC12<10:8>
12c2 Slave Event	49	000076h	IFS3<1>	IEC3<1>	IPC12<6:4>
Input capture 1	1	000016h	IFS0<1>	IEC0<1>	IPC0<6:4>
Input capture 2	5	00001Eh	IFS0<5>	IEC0<5>	IPC1<6:4>
Input capture 3	37	00005Eh	IFS2<5>	IEC2<5>	IPC9<6:4>
Input capture 4	38	000060h	IFS2<6>	IEC2<6>	IPC9<10:8>
Input capture 5	39	000062h	IFS2<7>	IEC2<7>	IPC9<14:12>
Input change Notification	19	00003Ah	IFS1<3>	IEC1<3>	IPC4<14:12>
Output compare 1	2	000018h	IFS0<2>	IEC0<2>	IPC0<10:8>
Output compare 2	6	000020h	IFS0<6>	IEC0<6>	IPC1<10:8>
Output compare 3	25	000046h	IFS1<9>	IEC1<9>	IPC6<6:4>
Output compare 4	26	000048h	IFS1<10>	IEC1<10>	IPC6<10:8>
Output compare 5	41	000066h	IFS2<9>	IEC2<9>	IPC10<6:4>
Parallel master port	45	00006Eh	IFS2<13>	IEC2<13>	IPC11<6:4>
Real-Time Clock/Calendar	62	000090h	IFS3<14>	IEC3<13>	IPC15<10:8>
spi1 Error	9	000026h	IFS0<9>	IEC0<9>	IPC2<6:4>
spi1 Event	10	000028h	IFS0<10>	IEC0<10>	IPC2<10:8>
spi2 Error	32	000054h	IFS2<0>	IEC0<0>	IPC8<2:0>
spi2 Event	33	000056h	IFS2<1>	IEC2<1>	IPC8<6:4>
Timer1	3	00001Ah	IFS0<3>	IEC0<3>	IPC0<14:12>
Timer2	7	000022h	IFS0<7>	IEC0<7>	IPC1<14:12>
Timer3	8	000024h	IFS0<8>	IEC0<8>	IPC2<2:0>
Timer4	27	00004Ah	IFS1<11>	IEC1<11>	IPC6<14:12>
Timer5	28	00004Ch	IFS1<12>	IEC1<12>	IPC7<2:0>
uart1 Error	65	000096h	IFS4<1>	IEC4<1>	IPC16<6:4>
uart1 Receiver	11	00002Ah	IFS0<11>	IEC0<11>	IPC2<14:12>
uart1 Transmitter	12	00002Ch	IFS0<12>	IEC0<12>	IPC3<2:0>
uart2 Error	66	000098h	IFS4<2>	IEC4<2>	IPC16<10:8>
uart2 Receiver	30	000050h	IFS1<14>	IEC1<14>	IPC7<10:8>
uart2 Transmitter	31	000052h	IFS1<15>	IEC1<15>	IPC7<14:12>

Many of these sources in their turn can generate several different interrupts. For example, a serial port interface peripheral (UART) can generate three types of interrupts:

- When new data has been received and is available in the receive buffer for processing.
- When data in the transmit buffer has been sent and the buffer is empty and ready and available to transmit more.
- When an error condition has been generated and action might be required to re-establish communication.

Each interrupt source has five associated control bits, allocated in various special-function registers (Table 5.1):

- The *Interrupt Enable* bit (typically represented with a suffix *-IE*). When cleared, the specific trigger event is prevented from generating interrupts. When set, it allows the interrupt to be processed. At power on, all interrupt sources are disabled by default.
- The *Interrupt Flag* (typically represented with a suffix *-IF*), a single bit of data, is set each time the specific trigger event is activated. Notice how, once set, it must be cleared (manually) by the user. In other words, it must be cleared before exiting the interrupt service routine, or the same interrupt service routine will be immediately called again.
- The *Priority Level* (typically represented with a suffix *-IP*). Interrupts can have up to seven levels of priority. Should two interrupt events occur at the same time, the highest-priority event will be served first. Three bits encode the priority level of each interrupt source. The PIC24 execution priority level value is kept in the *SR* register in three bits referred to as *IPL0..IPL2*. Interrupts with a priority level lower than the current value of *IPL* will be ignored. At power on, all interrupt sources are assigned a default level of four and the processor priority is initially set at level zero.

Within an assigned priority level there is also a *natural* (or default) priority amongst the various sources in the fixed order of appearance in the IVT table.

Nesting of Interrupts

Interrupts can be nested so that a lower-priority interrupt service routine can be, in its turn, interrupted by a higher-priority routine. This behavior can be changed by the *NSTDIS* bit in the *INTCON1* register of the PIC24.

When the *NSTDIS* bit is set, as soon as an interrupt is received, the priority level of the processor (*IPL*) is set to the highest level (7) independently of the specific interrupt level assigned to the event. This prevents new interrupts being serviced until the present one is completed. In other words, when *NSTDIS* bit is set, the priority level of each interrupt is used only to resolve conflicts in the event that multiple interrupts should occur simultaneously. In this case all interrupts will be serviced sequentially.

Table 5.2: Trap vector table

Vector Number	IVT Address	Trap Source
0	000004h	Reserved
1	000006h	Oscillator failure
2	000008h	Address error
3	00000Ah	Stack error
4	0000Ch	Math error
5	0000Eh	Reserved
6	000010h	Reserved
7	000012h	Reserved

Traps

Eight additional vectors occupy the first locations at the top of the IVT table (Table 5.2). They are used to capture special error conditions such as a failure of the selected CPU oscillator, an incorrect address (word access to an odd address), stack underflow/overflow or a divide by zero (math error).

Trap Vector Details

Since these types of errors have generally fatal consequences for a running application, they have been assigned fixed priority levels above the seven basic levels available to all other interrupts. This means also that they cannot be inadvertently masked (or delayed by the *NSTDIS* mechanism) and it provides an extra level of security for the application. The MPLAB C compiler associates all trap vectors to a single default routine that will produce a processor reset. You can change such behavior using the same technique illustrated for all generic interrupt service routines.

A Template and an Example for a Timer1 Interrupt

This all might seem extremely complicated, but we will quickly see that by following some simple guidelines, we can put it to use in no time. Let's create a template – that we will re-use in future practical examples – that demonstrates the use of the Timer1 peripheral module as the interrupt source. We will start by writing the interrupt service-routine function:

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
    // insert your code here
    // ...
    // remember to clear the interrupt flag before exit
    _T1IF = 0;
} //T1Interrupt
```

We used the `_ISR` macro just like before and made sure to declare the function type and parameters as `void`. Remembering to clear the interrupt flag (`_T1IF`) before exiting the function is extremely important, as we have seen. In general, the application code should be extremely concise. The goal of any interrupt service routine is to perform a simple task, quickly and efficiently, in rapid response to an event.

Note

In extreme cases, when interrupt events can happen so frequently that you worry you might not even complete the service routine before the next event is detected, you might want to clear the interrupt flag at the very top of the function in order to be able to “catch” the next one. Clearing the flag later, at the very end, would otherwise *swallow* it.

As a general rule, I would say that if you find yourself writing a long interrupt service routine, more than a page of code long (or contemplating calling other functions), you should most probably stop and reconsider the goals and the structure of your application. Lengthy calculations have a place in the main function and specifically in the main loop, not inside an interrupt service routine where time is at premium.

Let’s complete the template with a few lines of code that we will add to the main function:

```
main()
{
    // 2. initializations
    _T1IP = 4;        // this is the default value anyway
    TMR1 = 0;         // clear the timer
    PR1 = PERIOD-1;  // set the period register

    // 3. configure Timer1 module
    T1CON = 0x8020;  // enabled, prescaler 1:64, internal clock

    // 4. init the Timer1 Interrupt control bits
    _T1IF = 0;        // clear the interrupt flag, before
    _T1IE = 1;        // enable the T1 interrupt source

    // 5. main loop
    while( 1)
    {
        // your main code here
    } // main loop
} // main
```

In 2. we assign a priority level to the Timer1 interrupt source, although this might not be strictly necessary as we know that all interrupt sources are assigned a default level 4 priority at power on. We also clear the timer and assign a value to its period register.

In 3. we complete the configuration of the timer module by turning the timer on with the chosen settings.

In 4. we clear the interrupt flag just before enabling the interrupt source.

The interrupt trigger event for the timer module is defined as the instant the timer value reaches the value assigned to the period register. In that instant, the interrupt flag is set and the timer is reset to begin a new cycle. If the interrupt enable bit is set as well, and the priority level is higher than the processor's current priority (*IPL*), the interrupt service function is immediately called.

In 5. we will insert the main loop code. If everything goes as planned, the main loop will execute continuously, interrupted periodically by a brief call to the interrupt service routine.

A Real Example with Timer1

By adding only a couple of lines of code, we can turn this template into a more practical example where Timer1 is used to maintain a real-time clock, with tenths of a second, seconds and minutes. As a simple visual feedback we can use the lower 8 bits of PortA as a binary display showing the seconds running. Here is what we need to add:

Before 1. add the declaration of a few new integer variables that will act as the seconds and minutes counters

```
int dSec = 0;
int Sec = 0;
int Min = 0;
```

In 1.1. have the interrupt service routine increment the counter: `dSec++`; a few additional lines of code will be added to take care of the carryover into seconds and minutes.

In 2. set the period register for Timer1 to a value that (assuming a 32 MHz clock) will give us a tenth of a second period between interrupts.

```
PR1 = 25000-1; // 25,000 * 64 * 1 cycle (62.5ns) = 0.1s
```

Set *PORTA* least significant byte as output:

```
TRISA = 0xff00;
```

In 3. set the Timer1 prescaler to 1:64 to help achieve the desired period.

```
T1CON = 0x8020;
```

In 3. add code inside the main loop to continuously refresh the content of *PORTA* with the current value of the milliseconds counter.

```
PORTA = Sec;
```

The new project is ready to build:

```
#include <config.h>

int dSec = 0;
int Sec = 0;
int Min = 0;

// 1. Timer1 interrupt service routine
void _ISR_T1Interrupt( void)
{
    // 1.1 your code here
    dSec++;          // increment the tenths of a second counter
    if ( dSec > 9)    // 10 tenths in a second
    {
        dSec = 0;
        Sec++;       // increment the seconds counter

        if ( Sec > 59) // 60 seconds make a minute
        {
            Sec = 0;
            Min++;    // increment the minute counter

            if ( Min > 59) // 59 minutes in an hour
                Min = 0;
        } // minutes
    } // seconds

    // 1.2 clear the interrupt flag
    _T1IF = 0;
} //T1Interrupt

main()
{
    // 2. initializations
    _T1IP = 4;          // this is the default value anyway
    TMR1 = 0;           // clear the timer
    PR1 = 25000-1;      // set the period register
    TRISA = 0xff00;     // set PORTA LSB as output

    // 3. configure Timer1 module
    T1CON = 0x8020;     // enabled, prescaler 1:64, internal clock

    // 4. init the Timer1 Interrupt control bits
    _T1IF = 0;          // clear the interrupt flag, before
    _T1IE = 1;          // enable the T1 interrupt source
```

```
// 5. main loop
while( 1)
{
    // your main code here
    PORTA = Sec;
} // main loop
} // main
```

Testing the Timer1 Interrupt

Let's first build the project for debugging using the **Debug>Debug Project** command from the main menu.


Then let's open the **Watches** window selecting **Debug>New Watch...** from the main menu (or pressing **CTRL+SHIFT+F7**) then add the following variables:

- **dSec**: select from the Symbol pull down box, then click on Add
- **TMR1**: select from the SFR pull down box, then click on Add
- **SR**: select from the SFR pull down box, then click on Add.

Now let's set a breakpoint: set the cursor on the first line of the Timer1 interrupt service routine (1.1) and right click with the mouse and select **Toggle Line Breakpoint**.

Alternatively you can press **CTRL+F8** or perhaps, even more simply, you can click with your mouse pointer right **on the line number** in the editor *gutter*, that gray area on the immediate left border of the editor window. In either case, a little red square will be superimposed on the line number and the whole line will be highlighted in red as in [Figure 5.1](#).

By setting a breakpoint here, we will be able to observe whether the interrupt is actually being triggered.

Start executing the code by pressing the **Continue** button  (or select **Debug>Continue**). The debugger should stop quickly, with the program counter cursor (the green bar) pointing right at the breakpoint inside the interrupt service routine ([Figure 5.2](#)).

So we did stop inside the interrupt service routine! This means that the trigger event was activated; that is, the Timer1 reached a count of 24,999 (0x61A7). Remember though that the

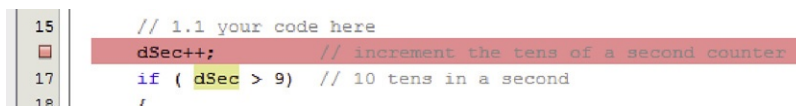


Figure 5.1: Breakpoint line highlighting

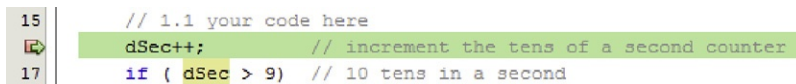


Figure 5.2: Breakpoint reached

Timer1 count starts with 0, therefore 25,000 counts have been performed, which multiplied by the prescaler value means that $25,000 \times 64$ or exactly 1.6 million cycles have elapsed. At the PIC24's execution rate (16 million instructions per second or 62.5 ns per cycle) this all happened in a tenth of a second!

From the Watches window we can now observe the current value of processor priority level (*IPL*). Since we are inside an interrupt service routine, configured to operate at level 4, we should be able to verify that bits 3, 4 and 5 of the status register (*SR*) contain exactly this value. For maximum convenience, click on the little plus sign icon next to the line of the Watches window that shows the value of the *SR* register to expand it and show the individual bit fields contained.

In Figure 5.3, I have circled the *IPL* (bitfield) value in the Watches window showing that an interrupt priority level four is being serviced.

Single stepping from the current position, using either the **StepOver** or the **StepIn** commands, we can monitor the execution of the next few instructions inside the interrupt service routine. Upon its completion, we can observe how the priority level returns back to the initial value – look for the *IPL* bit field to be cleared.

After executing another **Continue** command, we should find ourselves again in the interrupt routine, and another 1.6 million cycles will have been executed.

Add the **Sec** and **Min** variables to the *Watches* window.

Execute the **Continue** command a few more times to verify that, after 10 iterations, the seconds counter is incremented.

To test the minutes, you might want to remove the current breakpoint and place a new one a few lines below – otherwise you will have to execute the **Continue** command exactly 600 times!

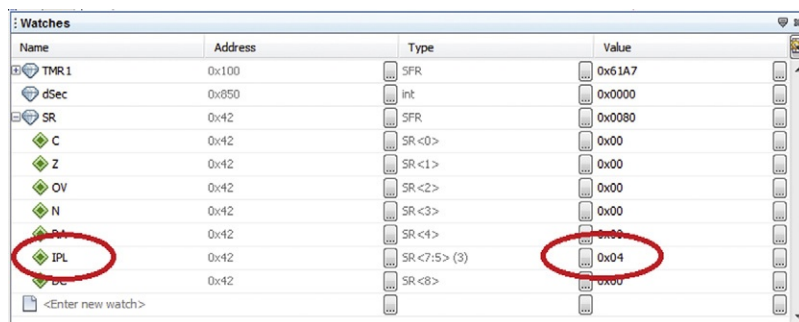


Figure 5.3: The Status Register (SR) expanded in the Watches window

Place the new interrupt on the *Min++* statement.

Execute the **Continue** command once more and while you wait watch the LED bar count in binary all the way to 0x3B (59). Now observe that the seconds counter has already been cleared in the Watches window, but the new value has not been updated yet on the LED bar.

Execute the **Step Over** command once and the minute counter will be incremented.

The interrupt routine has been executed 600 times in total, at precise intervals of one tenth of a second. Meanwhile the code present in the main loop has been executed continuously to use the vast majority of the grand total of 960 million cycles. In all honesty, our demo program did not make much use of all those cycles, wasting them all in a continuous update of the PortA content. In a real application, we would have performed a lot of work, all the while maintaining a precise real-time clock count.

The Secondary Oscillator

There is another feature of the PIC24 Timer1 module (common to all previous generations of 8-bit PIC[®] microcontrollers) that we could have used to obtain a real-time clock. In fact, there is a *low-frequency* oscillator, known as the secondary oscillator, that can be used to feed just the Timer1 module in place of the high-frequency *main clock*. Since it is designed for low-frequency operation, typically it is used in conjunction with an inexpensive 32,768 Hz crystal, it requires very little power to operate. And since it is independent of the main clock circuit, it can be maintained in operation when the main clock is disabled and the processor enters one of the many possible low-power modes. In fact, the secondary oscillator is an essential part of many of those low-power modes. In some cases it is used to replace the main clock, in others it remains active only to feed the Timer1 or a selected group of peripherals.

To convert our previous example for use with the secondary oscillator, we will need to perform only a few minor modifications, such as:

- Change the interrupt routine to count only seconds and minutes (the much slower clock rate does not require the extra step for the tenth of a second).

```
// 1. Timer1 interrupt service routine
void _ISR _T1Interrupt( void)
{
    // 1.1 clear the interrupt flag
    _T1IF = 0;

    // 1.2 your code here
    Sec++;      // increment the seconds counter

    if ( Sec > 59) // 60 seconds make a minute
    {
        Sec = 0;
        Min++;   // increment the minute counter
    }
}
```

```

        if ( Min > 59)// 59 minutes in an hour
            Min = 0;
    } // minutes
} //T1Interrupt

```

- In 2. change the period register to generate one interrupt every 32,768 cycles
- In 3. change the Timer1 configuration word (the prescaler is not required anymore)

```
PR1 = 32768-1;    // set the period register
```

```
T1CON = 0x8002;    // T1 on, 1:1 off secondary oscillator
```

According to the PIC24 datasheet, to activate the secondary low-power oscillator you will also need to set the *SOSCEN* bit in the *OSCCON* register. But before you rush to type in the code and try to execute it on the target board, notice that the *OSCCON* register contains vital controls for the MCU affecting the choice of the main active oscillator and its speed (Figure 5.4).

Because of its importance, it is protected by a locking mechanism. As a safety measure, you will have to perform a special unlock sequence first or your command will be ignored. The sequence would be hard, if not impossible, to code in C. Even if we managed, the optimizer could disrupt it later. So there are only two options: use *inline* assembly language or, better, use a *built-in* function of the MPLAB C compiler for the PIC24.

```
__builtin_write_OSCCONL( 2);
```

This function does perform the required unlock sequence and writes the desired value “2” to set the *SOSCEN* bit in the *OSCCON* register for us.

Now we are finally ready. Remove all the breakpoints, then re-build the project. This time elect to use the Run mode: select **Run>Run Project**. You will be able to quickly verify that the LED bar is being updated once a second as desired. The difference is that here the timing is independent of the main clock frequency, which could change or could be generated by a low-accuracy clock source such as the fast (8 MHz) internal oscillator (FRC) of the PIC24.

U-0	R-0	R-0	R-0	U-0	R/W-x ⁽¹⁾	R/W-x ⁽¹⁾	R/W-x ⁽¹⁾
—	COSC2	COSC1	COSC0	—	NOSC2	NOSC1	NOSC0
bit 15				bit 8			
R/SO-0	U-0	R-0 ⁽²⁾	U-0	R/CO-0	U-0	R/W-0	R/W-0
CLKLOCK	—	LOCK	—	CF	—	SOSCEN	OSWEN
bit 7				bit 0			

Figure 5.4: OSCON register

The Real-Time Clock Calendar (RTCC)

Building on the previous two examples, we could evolve the real-time clock implementations to include the complete functionality of a calendar, adding the count of days, day of the week, months and years. These few new lines of code would be executed only once a day, once a month or once a year, and therefore would produce no decrease in the performance of the overall application whatsoever. Although it would be somewhat entertaining to develop such code, considering leap years and working out all the details, the PIC24FJ128GA010 already has a complete *Real-Time Clock and Calendar* (RTCC) module built in and ready for use.

How convenient! Not only is it fed from the same low-power secondary oscillator, but it comes with all the bells and whistles – including a built in Alarm function that can generate interrupts. In other words, once the module is initialized, it is possible to activate the RTCC alarm and wait for an interrupt to be generated, for example, on the exact month, day, hour, minute and second desired once a year (or, if set on February 29th, even once every four years!).

Setting the RTCC time requires a similar combination lock mechanism (as seen with *OSCCON*) to protect the key RTCC register *RCFGCAL*. A special bit (*RTCWEN*) must be set to allow writing to the register, but this bit requires its own special unlock sequence to be executed first. Here another built-in function of the MPLAB C compiler for the PIC24 comes to our rescue:

```
__builtin_write_RTCWEN();
```

This time, no parameter is required. The unlock sequence for the RTCC register is performed and the *RTCWEN* bit is set.

After this step, initializing the RTCC, setting the date and time is trivial:

```
_RTCEN = 0;           // disable the module
// example set 12/01/2006 WED 12:01:30
_RTCPTR = 3;          // start the loading sequence
RTCVAL = 0x2006;       // YEAR
RTCVAL = 0x1100;       // MONTH-1/DAY-1
RTCVAL = 0x0312;       // WEEKDAY/HOURS
RTCVAL = 0x0130;       // MINUTES/SECONDS

// optional calibration
//_CAL = 0x00;

// enable and lock
_RTCEN = 1;           // enable the module
_RTCWREN = 0;         // lock settings
```

Setting the alarm does not require any special unlock combination. Here is an example that will help you remember my birthday:

```
// disable alarm
_ALRMEN = 0;
```

```

// set the ALARM for a specific day (my birthday)
_ALRMPTR = 2;           // start the loading sequence
ALRMVAL = 0x1124;       // MONTH-1/DAY-1
ALRMVAL = 0x0006;       // WEEKDAY/HOUR
ALRMVAL = 0x0000;       // MINUTES/SECONDS

// set the repeat counter
_ARPT = 0;              // once
_CHIME = 1;             // indefinitely

```

Now all you need to do is to set up the *Alarm Mask* as in:

```
_AMASK = 0b1001;       // set alarm once a year
```

And enable the alarm and RTC interrupt in place of the Timer1 interrupt:

```

_ALRMEN = 1;           // enable alarm
_RTCIF = 0;            // clear interrupt flag
_RTCIE = 1;            // enable interrupt

```

Finally, this is what the interrupt service routine will look like...

```

// 1. RTCC interrupt service routine
void _ISR_RTCCInterrupt( void)
{
    // 1.1 clear the interrupt flag
    _RTCIF = 0;

    // 1.2 your code here, will be executed only once a year
    // or once every 365 x 24 x 60 x 60 x 16,000,000 MCU cycles
    // that is once every 504,576,000,000,000 MCU cycles
} // RTCCInterrupt

```

Managing Multiple Interrupts

It is typical of an embedded control application to require several interrupt sources to be serviced. For example, a serial communication port might only require periodic attention, while a PWM might require periodic updates to control an analog output. Multiple timer modules might be used simultaneously to produce pulsed outputs, while multiple inputs could be sampled by the Analog to Digital converter and their values would need to be buffered. There is no limit to the number of things that can be done with 118 interrupt sources available. At the same time, there is no limit to the complexity of the bugs that can be generated thanks to the same sophisticated mechanisms if a little discipline and some common sense are not applied.

Here are some of the rules to keep in mind:

1. Keep it short and simple. Make sure the interrupt routines are the shortest/fastest possible, and under no circumstances should you attempt to perform any processing of the incoming data. Limit the activity to buffering, transferring and flagging.
2. Use the priority levels to determine which event deserves to be serviced first, in case two events are triggered simultaneously.

But consider very carefully whether you want to face the additional complexity and headaches that result from enabling the use of nested interrupt calls. After all, if the interrupt service routines are short and efficient, the extra latency introduced by waiting for the current interrupt to be completed before a new one is serviced is going to be extremely small. If you determine that you don't really need it, make sure the *NSTDIS* control bit is set to prevent nesting:

```
_NSTDIS = 1; // disable interrupt nesting (default)
```

Post-Flight Briefing

In this lesson we have seen how an interrupt service routine can be simple to code thanks to the language extensions built into the MPLAB C compiler, and the powerful interrupt control mechanisms offered by the PIC24 architecture. Interrupts can be an extremely efficient tool in the hands of the embedded-control programmer to manage multiple tasks while maintaining precious timing and resources constraints. At the same time, they can be a great source of trouble. In the PIC24 reference manual and the MPLAB C Compiler User Guide you will find more useful information than we could possibly cram into one single lesson. Finally, in this lesson we took the opportunity to learn more about the uses of Timer1 and the secondary oscillator, and we got a glimpse of the features of the new Real-Time Clock and Calendar (RTCC) module.

Notes for the C Experts

The interrupt vector table (IVT) is an essential part of the *crt0* code segment for the PIC24. Actually, two copies of it are required to be present in the first 256 locations of the program memory. One is used during normal program execution, and the second (or Alternate IVT) during debugging. These tables account for most of the size of the *crt0* code in all the examples we have been developing in these first five lessons. Subtract 256 words (or 768 bytes) from the file size of each example to obtain the “net” code size.

Notes for the Assembly Experts

The *_ISRFAST* macro can be used to declare a function as an interrupt service routine and to further specify that it will use an additional and convenient feature of the PIC24 architecture: a set of four shadow registers. By allowing the processor to automatically save the content of the first four working registers (*W0–W3*, i.e. the most frequently used ones) and most of the content of the SR register in special reserved locations, without requiring the use of the stack, the shadow registers provide the fastest possible interrupt response time. Naturally, since there is only one set of such registers, their use is limited to applications where only one interrupt will be serviced at any given time. This does not limit us to use only one interrupt in the entire application, but rather to use *_ISRFAST* only in applications that have all interrupts with the same priority level or, if multiple levels are in use, reserve the *_ISRFAST* options only for the interrupt service routines with the highest level of priority.

Notes for the PIC Microcontroller Experts

It is about time that we start using more of the PIC24 peripheral libraries. In the previous chapters I have often specifically avoided them to illustrate the need for the embedded control programmer to understand the minute details of the microcontroller architecture and, to a degree, to allow those readers transitioning from assembly language to appreciate how even C language can be close to the “machine”.

Libraries (collections of ready to use functions) can speed up considerably the development of a project by hiding complex details and exposing simple-to-use interfaces. In fact, in the next few chapters we will start developing our own little library (moving new modules in the */lib* subdirectory and their corresponding headers in the */include* subdirectory) to simplify a multitude of interesting applications.

But libraries can also obscure your code and make debugging very difficult if they are not very well designed and documented. For this reason, I am very selective when choosing to use function libraries in my applications and in the rest of this book.

Although Microchip offers a peripheral library for the PIC24 that contains a module for each peripheral contained in each device family, not all of them are equally useful!

For example, the *ports.h* library offers hundreds of functions (and macros) to perform the simplest of the operations on the I/O pins of the PIC24, such as:

```
mPORTBRead()
```

As you would have guessed, this macro reads the contents of the *PORTB* register.

What do you think the following function does then?

```
mPORTACloseAll()
```

Closing a Port is not a defined operation, not documented at least in the general Microchip literature, so you will be hard pressed to guess what will really be accomplished.

As a consequence, generally I do not recommend using the *ports.h* library but, rather, I encourage my readers to use the Port registers directly. There is absolutely no advantage in using the library form over the register name.

A different case can be made for the RTCC peripheral. The *rtcc.h* library does offer very useful functions that do simplify the configuration and use of the real-time clock and calendar. For example, the following two functions can quickly enable the secondary oscillator, turn on the RTCC and enable its configuration:

```
RtccInitClock();
RtccWrOn();           // unlock and enable writing to RTCC
```

Setting the date and time of day becomes an equally simple task with the following code:

```
rtccTimeDate TD = { 0x06, 0x20, // year
                    0x01, 0x11, // day, month
                    0x12, 0x03, // hour, weekday
                    0x30, 0x01}; // sec, min
RtccWriteTimeDate( &TD, FALSE);
```

Perhaps even more interesting is how the alarm mask definition is simplified:

```
RtccSetAlarmRpt( RTCC_RPT_YEAR, TRUE);
```

As a pro, consider how, instead of setting the *AMASK* register to a cryptic binary value, we can use a self-explanatory constant: *RTCC_RPT_YEAR* to decide when the alarm match should trigger our next interrupt.

As a con though, consider that the *RTCC_RPT_YEAR* constant is not documented in the device datasheet but only in the *rtcc.h* header file (and simply listed in the help file *.chm*). Once you find it, you will be hard pressed to make sure to match it to the desired pattern of bits in the *AMASK* register as per the device datasheet.

In the next few chapters we will see more such examples of use of the peripheral libraries and in the end I will let you decide how and when you want to take things in your own hands and when you will want to trust the libraries to do things for you.

On the companion website (www.flyingpic24.com) you will find multiple versions of the examples presented in this and many of the following chapters where the peripheral libraries have been used in place of direct access to the registers (i.e. *alarm.c* and *alarm-p.c*). By comparing them side by side, you will be able to judge for yourself the pros and cons of both approaches.

Tips & Tricks

- As of version 3.0 of the MPLAB C compiler, a new *attribute* has been added (required) to the interrupt service routines declaration to specify whether the PSV window pointer save and restore should be managed by the compiler (see the next chapter to learn more about the PSV). Since in all the example applications presented in this book we will not modify the PSV window pointer, we will redefine the *_ISR* and *_ISR_FAST* macros to add the new *no_auto_psv* attribute automatically to all our interrupt service routine declarations.

```
#if __C30_VERSION__ >= 300
#undef _ISR
#define _ISR __attribute__((interrupt,no_auto_psv))
#undef _ISRFAST
#define _ISRFAST __attribute__((interrupt,shadow,no_auto_psv))
#endif
```

We could place this code in the *config.h* header file but, since we will be soon using interrupts in multiple library modules, we will instead include it in the *EX16.h* where

since lesson 3 (Tips & Tricks) we have been assembling all things specific to the Explorer16 demonstration board.

```
/*
**  EX16.h
**
**  Standard definitions for use with the Explorer16 board
*/
#ifndef _EX16
#define _EX16

#include <p24fxxx.h>

#if __C30_VERSION__ >= 300
#undef _ISR
#define _ISR __attribute__((interrupt,no_auto_psv))
#undef _ISRFAST
#define _ISRFAST __attribute__((interrupt,shadow,no_auto_psv))
#endif

#define FCY 16000000UL // instruction clock 16MHz
#define USE_AND_OR // use OR to assemble control register bits
#define _FAR __attribute__((far))

// prototypes
void InitEX16( void); // initialize the Explorer16 board

#endif
```

- On the PIC24 architecture there is no single control bit that disables all interrupts, but there is an instruction (DISI) that can disable interrupts for a limited number of cycles. If there are portions of code that require all interrupts to be temporarily disabled, you can use the following inline assembly command:

```
__asm__ volatile("disi #0x3FFF"); // disable temporarily all interrupts

// your code here
// ...

DISICNT = 0; // re-enable all interrupts
```

Exercises

- Rewrite the RTCC interrupt example to use the *rtcc.h* library and blink an LED at 1 Hz.
- Output time and/or date on a seven-segment LED display using a few I/O ports.

Books

- Brown, G., 2003. *Flying Carpet, The Soul of An Airplane*, Iowa State Press, Ames, IO. Greg has many fun episodes from the real life of a general aviation pilot who uses his plane for recreation as well as a family utility.

- Curtis, K.E., 2006. *Embedded Multitasking*, Newnes, Burlington, MA.
Keith knows multitasking and what it takes to create small and efficient embedded control applications.

Links

- <http://www.aopa.org>
This is the web site of the Aircraft Owners and Pilot Association. Feel free to browse through the web site and access the many magazines and free services offered by the association. You will find a lot of useful and interesting information in here.
- http://www.niell.org/nixie_clock/Nixieclock.html
A PIC-based clock with a retro style, using glowing Nixie tubes.

Taking a Look Under the Hood

Whether you are trying to get a driver's license or a pilot's license, sooner or later you have to start looking under the hood, or the cowlings for pilots. You don't have to understand how each part of the engine works nor how to fix it – mechanics will be happy to do that for you. But a basic understanding of what is going on will help you be a better driver/pilot. If you understand the machine, you can control it better – it's that simple. You can diagnose little problems, and you can do a little maintenance.

Working with a compiler is not that dissimilar – sooner or later you have to start looking under the hood if you want to get the best performance out of it. Since the very first lesson we have been peeking inside the engine compartment. This time we will delve into a little bit more detail.

Flight Plan

In this lesson we will review the basics of string declaration as an excuse to introduce the memory allocation techniques used by the MPLAB[®] C compiler for the PIC24. The Harvard architecture of the PIC24 poses some interesting challenges that require innovative solutions. We will use several tools, including the Embedded Memory window, the Watches window and the Map file, to investigate how the MPLAB C compiler and linker operate in combination to generate the most compact and efficient code.

Preflight Checklist

Similarly to previous lessons we will make use of the MPLAB X IDE, the MPLAB C Compiler for the PIC24, a programmer/debugger of your choice and pretty much any demo board that can host a PIC24FJ128GA010 microcontroller.

Use the **New Project Setup** checklist to create a new project called **6-Strings** and a new source file similarly called **strings.c**.

The Flight

Strings are treated in the C language as simple ASCII character arrays. Every string composed of characters is assumed to be stored sequentially in memory in consecutive 8-bit elements

of the array. After the last character of the string an additional byte containing a value of zero (represented in a character notation with `'\0'`) is added as a termination flag.

Notice, however, that this is just a convention that applies to the standard C string manipulation library *string.h*. It would be entirely possible, for example, to define a new library and store strings in arrays where the first element is used to record the length of the string; in fact, Pascal programmers would be very familiar with this method. Additionally, if you are developing *international* applications, i.e. applications that communicate using languages that require large character sets (like Chinese, Japanese, Korean...), you might want to consider using Unicode, a technology that allocates multiple bytes per character, instead of plain ASCII. The MPLAB C library *stdlib.h* provides basic support for the translation from/to multi-byte strings defined by the ANSI90 standard.

Let's get started by reviewing the declaration of a variable containing a single character:

```
char c;
```

As we have seen from the previous lessons, this is how we declare an 8-bit integer (character), which is treated as a signed value ($-128.. +127$) by default.

We can declare and initialize it with a numerical value:

```
char c = 0x41;
```

Or, we can declare and initialize it with an ASCII value:

```
char c = 'a';
```

Note the use of the single quotes for ASCII character constants. The result is the same, and for the C compiler there is absolutely no distinction between the two declarations – characters *are* numbers.

We can now declare and initialize a string as an array of 8-bit integers (characters):

```
char s[5] = { 'H', 'E', 'L', 'L', '0' };
```

In this example, we initialized the array using the standard notation for numerical arrays. However, we could have also used a far more convenient notation (a shortcut) specifically created for string initializations:

```
char s[5] = "HELLO";
```

To further simplify things, and save you from having to count the number of characters composing the string (thus preventing human errors), you can use the following notation:

```
char s[] = "HELLO";
```

The MPLAB C compiler will automatically determine the number of characters required to store the string, whilst automatically adding a termination character (zero) that will be useful

to the string manipulation routines later to correctly identify the string length. So, the example above is, in truth, equivalent to the following declaration:

```
char s[6] = { 'H', 'E', 'L', 'L', 'O', '\0' };
```

Assigning a value to a char (8-bit integer) variable and performing arithmetic upon it is no different to performing the same operation on any integer type:

```
char c;    // declare c as an 8-bit signed integer
c = 'a';   // assign value 'a' from the ASCII table
c++;       // increment, it will be changed into a 'b'
```

The same operations can be performed upon any element within an array of characters (string). However, there is no simple shortcut, similar to the one used above for the initialization, that can assign a new value to an entire string:

```
char s[15]; // declare s as a string of 15 characters
s = "Hello!"; // Error! This does not work!
```

By including the *string.h* file at the top of your source file, you will gain access to numerous useful functions that will allow you to:

- copy the content of a string into another...
`strcpy(s, "HELLO"); // s : "HELLO"`
- append (or concatenate) two strings...
`strcat(s, " WORLD"); // s : "HELLO WORLD"`
- determine the length of a string...
`i = strlen(s); // i : 11`
- and many more.

Memory Space Allocation

Just like with numerical initializations, every time a string variable is declared and initialized, as in:

```
char s[] = "Flying with the PIC24";
```

three things happen:

1. The MPLAB C linker reserves a contiguous set of memory locations in RAM (data space) to contain the variable: 22 bytes in the example above. This space is part of the *ndata* (near) data section.
2. The MPLAB C linker stores the initialization value in a 22-byte-long table (in program memory). This space is part of the *init* code section.
3. The MPLAB C compiler creates a small routine that will be called before the main program (part of the *crt0* code we mentioned in previous chapters) to copy the values, thereby initializing the variable.

In other words, the string "Flying with the PIC24" ends up using twice the space you would expect, as one copy of it is stored in Flash program memory and space is reserved for it in RAM memory, too. Additionally, you must consider the initialization code and the time spent in the actual copying process. If the string is not supposed to be manipulated during the program, but is only used “as is”, transmitted out over a serial port or sent to a display, then there is no need to waste precious resources. Declaring the string as a *constant* will save RAM space and initialization code/time:

```
const char s[] = "Flying with the PIC24";
```

Now, the MPLAB C linker will only allocate space in program memory, in the *const* code section, where the string will be accessible via the Program Space Visibility window – an advanced feature of the PIC24 architecture that we will review shortly.

The string will be treated by the compiler as a direct pointer into program memory and, as a consequence, there will be no need to waste RAM space.

In the previous examples of this lesson, we saw other strings implicitly defined as constants:

```
strcpy( s, "HELLO");
```

The string "HELLO" was implicitly defined as being of *const char* type, and was similarly assigned to the *const* section in program memory to be accessible via the Program Space Visibility window.

Note that if the same constant string is used multiple times throughout the program, the MPLAB C compiler will automatically store only one copy in the *const* section to optimize memory use, even if all optimization features of the compiler have been turned off.

Program Space Visibility

The PIC24 architecture is somewhat different from most other 16-bit microcontroller architectures you might be familiar with. It was designed for maximum efficiency according to the Harvard model, as opposed to the more common Von Neumann model. The big difference between the two is that in the Harvard model there are two completely separate and independent buses available: one for access to the program memory (Flash) and one for access to the data memory (RAM). The net result is a doubling of bandwidth, since when the data bus is in use during the execution of one instruction, the program memory bus is available to fetch the next instruction code and initiate its decoding. In traditional Von Neumann architectures, the two activities must instead be interleaved, with a consequent penalty in performance. The drawback of this architectural choice is that access to constants and data stored in program memory requires special considerations.

The PIC24 architecture offers two methods to read data from program memory: using special table access instructions (*tblrd*), and through a second mechanism called *Program Space*

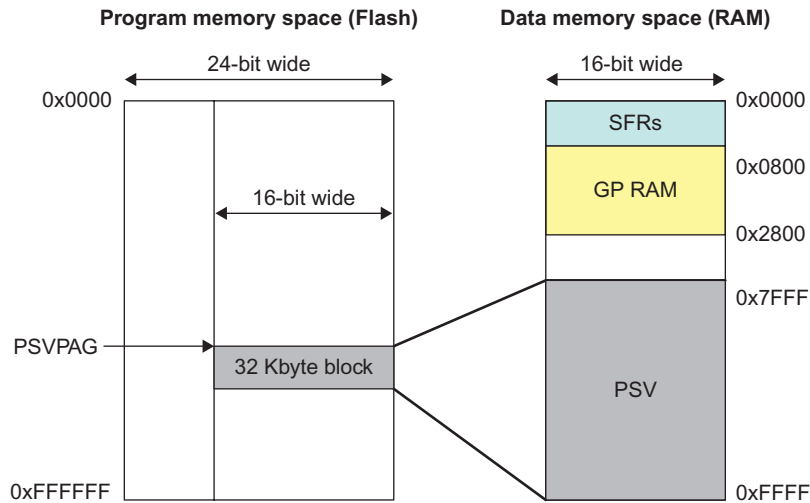


Figure 6.1: PIC24F Program Space Visibility (PSV) window

Visibility or *PSV* (Figure 6.1). This is a window of 32 Kbytes of program memory accessible via the data memory bus. In other words, the PSV is a bridge between the program memory bus and the data memory bus.

Notice that although the PIC24 uses a 24-bit-wide program memory bus, it operates only on a 16-bit-wide data bus. The mismatch between the two buses makes the PSV “bridge” a little more interesting. In practice, the PSV connects only the lower 16 bits of the program memory bus to the data memory bus. The upper portion (8 bits) of each program memory word is not accessible using the PSV window. However, when using the table access instructions, all parts of the program memory word become accessible, but at the cost of having to differentiate the manipulation of data in RAM (using direct addressing) from the manipulation of data in program memory (using the special table access instructions).

The PIC24 programmer can therefore choose between the more convenient, but relatively memory-inefficient, method for transferring data between the two buses of the PSV, or the more memory-efficient, but less transparent, solution offered by the table access instructions.

The designers of the MPLAB C compiler considered the trade-offs and chose to use both mechanisms albeit using them to solve different problems at different times:

- The PSV is used to manage constant arrays (numeric and strings) so that a single type of pointer (to the data memory bus) can be used uniformly for constants and variables.
- The table access mechanism is used to perform the variable initializations (limited to the *crt0* segment) for maximum compactness and efficiency.

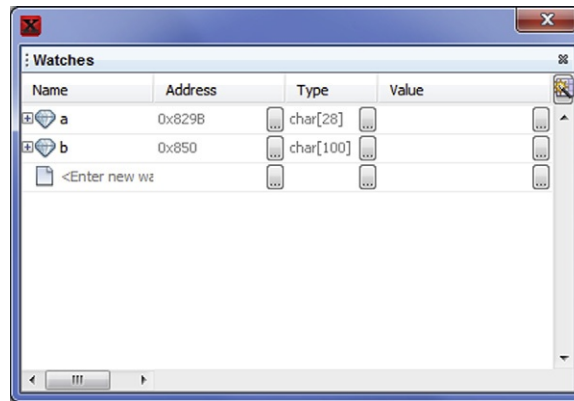


Figure 6.2: The Watches window showing the strings *a* and *b*

Investigating Memory Allocation

We will start investigating these issues with the following short snippet of code:

```
/*
** Strings
*/
#include <config.h>
#include <string.h>

// 1. variable declarations
const char a[] = "Learn to fly with the PIC24";
char b[100] = "";

// 2. main program
main()
{
    strcpy( b, "MPLAB C compiler"); // assign new content to b
} //main
```

Build the project for debugging using the **Debug>Project Debug** command.

Then open the **Watches** window and add to it the two strings *a* and *b* (Figure 6.2):

- You can select **Debug>New Watch ...** from the main menu.
- You can press the **CTRL+SHIFT+F7** keyboard shortcut.
- You can put the cursor in the editor window on each variable and right click with your mouse and select **New Watch ...** from the context menu.

By default, MPLAB X will show each element of the array as a hexadecimal value but you can customize the view by right clicking on the string name (before expanding it) and selecting **Display Value As:** and picking the **Character** option as I do in Figure 6.3.

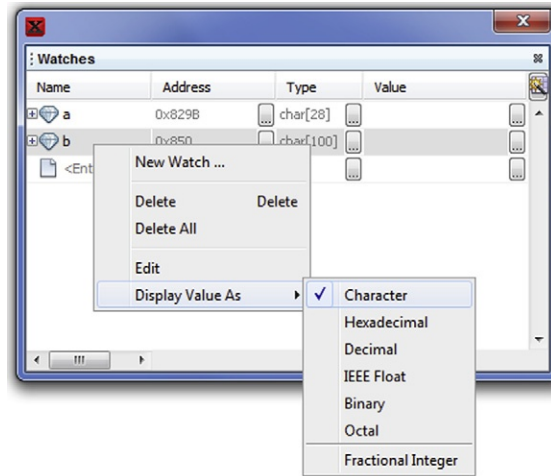
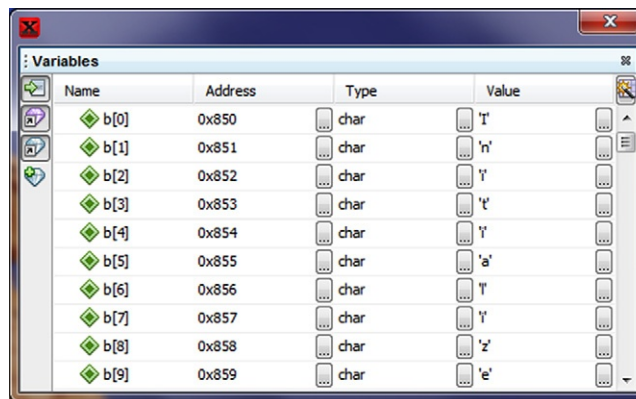


Figure 6.3: The Watches windows context menu

Figure 6.4: The string *b* expanded to show the array contents

You will be presented with the Watches window context menu.

Now by clicking on the little + (enclosed in a box) icon you will be able to expand the view to show each individual element (Figure 6.4).

Back in Figure 6.2, you might have noticed that the address of the string *a* (0x829B) appears to be a much higher value than the address of the string *b* (0x850). This reflects the fact that the constant string *a* is using only the minimum amount of space required in the Flash program memory of the PIC24 and will be accessed through the PSV space (starting at address 0x8000) and no RAM has been assigned to it. In contrast, the address of string *b* is clearly within the address space of the PIC24 RAM memory.

Name	Address	Type	Value
b	0x850	char[100]	
b[0]	0x850	char	'M'
b[1]	0x851	char	'P'
b[2]	0x852	char	'L'
b[3]	0x853	char	'A'
b[4]	0x854	char	'B'
b[5]	0x855	char	' '
b[6]	0x856	char	'C'
b[7]	0x857	char	' '
b[8]	0x858	char	'C'

Figure 6.5: The string *b* contents updated after the call to *strcpy()*

Notice also how the string *b* appears to be already initialized when we begin our debugging session (or after each reset). In reality, MPLAB X is allowing the *crt0* code to execute before starting our debugging session, so we don't have a chance to observe how the array is empty at the very beginning and how its initial value is copied from a location in Flash memory just before the call to the function *main()*.

Note

Only the most curious and patient readers will be able to see how the initialization of the string *b* is performed using the Table Read (*tblrd*) assembly instruction to extract the data from the program memory (Flash) and to store the values in the allocated space in data memory (RAM).

While single stepping (select **Debug>StepOver** from the main menu) through the short *main()*, notice how the contents of the string *b* get overwritten as the *strcpy()* function is called (Figure 6.5).

Note

Although the *string.h* library contains dozens of functions, you will be pleased to know that the MPLAB C linker is wisely appending to our executable code only the functions that are actually being used.

Looking at the Map

Another tool we have at our disposal to help us understand how strings (and in general any array variable) are initialized and allocated in memory is the *map file*. This text file, produced

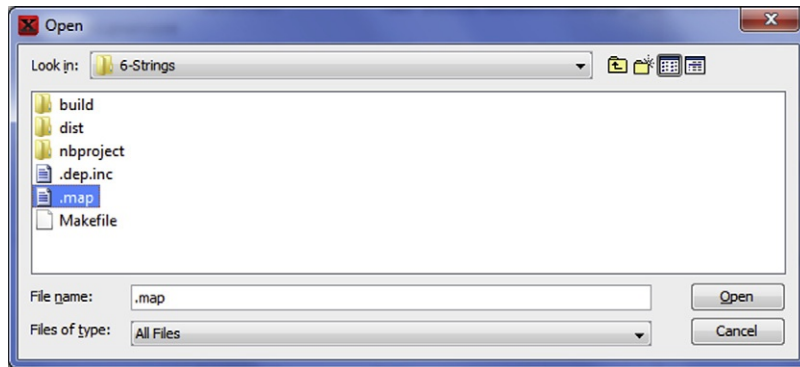


Figure 6.6: The Open dialog box

by the MPLAB C linker, can be easily inspected with the MPLAB X editor and is designed specifically to help you understand and resolve memory allocation issues.

To find this file, look for it in the main project directory where all the project source files are. Select **File>Open File** and then browse until you reach the project directory. There you will find a file called *.map* (no name, just the *.map* extension) (Figure 6.6). Make sure the Open dialog box is currently filtering for *All Files* in the *Files of type* field or you won't be able to see it.

Map files tend to be pretty long and verbose but, by learning to inspect only a few critical sections, you will be able to find a lot of useful data. The *Program Memory Usage* summary, for example, is found among the very first few lines.

section	address	length (PC units)	length (bytes) (dec)
.text	0x200	0x90	0xd8 (216)
.const	0x290	0x38	0x54 (84)
.dinit	0x2c8	0x4c	0x72 (114)
.text	0x314	0x16	0x21 (33)
.isr	0x32a	0x2	0x3 (3)
Total program memory used (bytes):			0x1c2 (450) <1%

This is a list of small sections of code assembled by the MPLAB C linker in a specific order and position (dictated by a *linker script* file).

Most section names are pretty intuitive, other are ... historical:

- *.text* section: where all the code generated from your source files by the MPLAB C compiler will be placed (the name of this section has been used since the original implementation of the very first C compiler).
- *.const* section: where the constants (integers and strings) will be placed for access via the PSV.

- `.dinit` section: where RAM variable's initialization data (used by the `crt0` code) will be found.
- `.isr`: where the Interrupt Service Routine (in this case a default one) will be found.

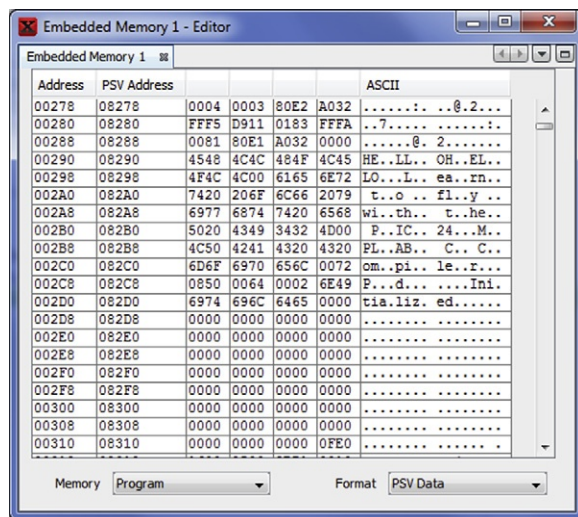


Figure 6.7: Inspecting the `.const` and `.dinit` memory sections

It's in the `.const` section that the `a` constant string as well as the “MPLAB C” (implicit) constant string are stored for access via the PSV window.

You can confirm this by inspecting the **Embedded Memory** window at the address 0x8290, remembering to select the *Memory* mode to **Program** and the *Format* option to **PSV Data**.

Observe the “groups of two” character grouping of the string “MPLAB C Compiler” in Figure 6.7. Remember how the PSV allows us to use only 16 bits of each 24-bit program memory word.

In `.dinit` is where the `b` variable's initialization string is to be found. This section follows immediately after the `.const` section, so you will be able to see it in the same Figure 6.7.

Observe how this string is prepared for access via the table instructions, therefore it uses each and every one of the 24 bits available in each program memory word. Note the character grouping of “groups of three” of the string “Initialized”.

The next part of the map file we might want to inspect is the *Data Memory Usage* (RAM) summary.

section	address	alignment	gaps	total length (dec)
-----	-----	-----	-----	-----
.icd	0x800		0x50	0x50 (80)
.ndata	0x850		0	0x64 (100)
Total data memory used (bytes):				0xb4 (180) 2%

In our simple example, it contains only two sections:

1. *.icd*, a small area of 80 bytes reserved for the in circuit debugger use starting at address 0x800, the first location available in the PIC24 RAM.
2. *.ndata*, containing only one variable: *b*, for which 100 bytes are reserved immediately following *.icd*.

Pointers

Pointers are variables used to refer indirectly (i.e. point) to other variables or part of their content. Pointers and strings go hand in hand in C programming, as they are a powerful mechanism for working on any array data type. So powerful in fact, that they are also one of the most dangerous tools in the programmers hands and a source of a disproportionately large share of programming bugs. Some programming languages, such as Java, have gone to the extreme of completely banning the use of pointers in an effort to make the language more robust and verifiable.

The MPLAB C compiler takes advantage of the PIC24 16-bit architecture to manage with ease large amounts of data memory (up to 32 Kbytes of RAM in GA and GB models). In particular, thanks to the PSV window, the MPLAB C compiler doesn't make any distinction between pointers to data memory objects and *const* objects allocated in program memory space. This allows a single set of standard functions to manipulate variables and/or generic memory blocks as needed from both spaces.

Note

This will come as a big relief to those of you who have previously attempted to program 8-bit PIC® microcontrollers in C.

The following classic program example will compare the use of pointers versus indexing to perform sequential access to an array of integers:

```
int *pi;           // define a pointer to an integer
int i;            // index/counter
int a[10];        // an array of integers

// 1. sequential access using array indexing
for( i=0; i<10; i++)
    a[ i ] = i;

// 2. sequential access using a pointer
pi = a;
for( i=0; i<10; i++)
{
    *pi = i;
    pi++;
}
```

In 1. we performed a simple *for* loop and each time round the loop we used *i* as an index into the array. To perform the assignment the compiler will have to take the value of *i*, multiply it by the size of the array element in bytes (2) and add the resulting offset to the initial address of the array *a*.

In 2. we initialized a pointer to point to the initial address of the array *a*. At each time round the loop we used the pointer (*) to perform the assignment, then we simply incremented the pointer.

Comparing the two cases, we see how, by using the pointer, we can save at least one multiplication step for each time round the loop. If the array element is used more times inside the loop, the performance improvement is going to be proportionally greater.

Pointer syntax can become very “concise” in C, allowing for some pretty effective code to be written, but also opening the door to more bugs.

As a minimum, you should become familiar with the most common contractions. The previous snippet of code is more often reduced to the following:

```
// 2. sequential access to array using pointers
for( i=0, pi=a; i<10; i++)
    *pi++ = i;
```

Also note that an empty pointer, that is, a pointer without a target, is assigned a special value *NULL*, which is implementation-specific and defined in *stddef.h*.

The Heap

One of the advantages offered by the use of pointers is the ability to manipulate objects that are defined dynamically (at run time) in memory. The *heap* is the area of data memory reserved for such use, and a set of functions, part of the standard C library *stdlib.h*, provide the tools to allocate and free the memory blocks. They include as a minimum the two fundamental functions:

```
void *malloc(size_t size);
```

which takes a block of memory of requested size from the heap and returns a pointer to it; and

```
void free(void *ptr);
```

which returns the block of memory pointed to by *ptr* to the heap.

The MPLAB C linker places the heap in the RAM memory space left unused above all project global variables and the reserved stack space. Although the amount of memory left unused is known to the linker and listed in the map file of each project, you will have to explicitly instruct the linker to reserve an exact amount for use by the heap.

Use the **File>Project Properties** menu command to open the *Project Properties* dialog box, select the **pic30-ld** (MPLAB C Linker) tab, and then define the heap size in bytes.

As a general rule, allocate the largest amount of memory possible as this will allow the *malloc()* function to make the most efficient use of the memory available. After all, if it is not assigned to the heap it will remain unused.

MPLAB C Memory Models

The PIC24 architecture allows for a very efficient (compact) instruction encoding for all operations performed on data memory within the first 8 Kbytes of addressing space. This is referred to as the *near* memory area and in the case of the PIC24FJ128GA010 it corresponds to the group of SFRs (within the first 2 Kbytes) and the following 6 Kbytes of general purpose RAM. Only the top 2 Kbytes of RAM are actually outside the near space.

Access to memory beyond the 8-Kbyte limit requires the use of indirect addressing methods (pointers) and could be less efficient if not properly planned for. The stack (and with it all the local variables used by C functions) and the heap (used for dynamic memory allocation) are naturally accessed via pointers and are correspondingly ideal candidates to be placed in the upper RAM space. This is exactly what the linker will attempt to do by default. It will also try to place all the global variables defined in a project in the near memory space for maximum efficiency. If a variable cannot be placed within the near memory space it has to be “manually” declared with a *far* attribute, so that the compiler will generate the appropriate access code. This behavior is referred to as the Small Data Memory model. This is the alternative to the Large Data Memory model, where each variable is assumed to be far unless the *near* attribute is specified.

In practice, while using the PIC24FJ128GA010, you will use almost uniquely the default small memory model and on rare occasions you will find it necessary to identify a variable with the *far* attribute. We will observe one such case in lesson number 12, where a very large array that would otherwise not fit in the near memory space will have to be declared as *far*. As a consequence, not only will the compiler generate the correct addressing instructions, but the linker will also push it to an upper area of RAM, giving priority to the other global variables and allowing them to be accessed in the near space.

Since access to elements of an array (explicitly via pointers or by indexing) is performed via indirect addressing anyway, there will be no performance or code size penalty.

A similar concept applies to the program memory space. In fact, within each compiled module, functions are called by making use of a more compact addressing scheme that relies on a maximum range of ± 32 Kbytes. Program memory models (small and large) define the default behavior of the compiler/linker with regards to the addressing of functions within or outside this 32-Kbyte range.

Post-Flight Briefing

In C language, strings are defined as simple arrays of characters, but the C language standard had no concept of different memory regions (RAM vs Flash) nor of the particular mechanisms required to cross the bridge between different buses in a Harvard architecture. The programmer using the MPLAB C compiler needs a basic understanding of the trade-offs of the various mechanisms available and the allocation strategies adopted to make the most out of the precious resources (RAM especially) available to the embedded control applications.

Notes for the C Experts

The *const* attribute is normally used in C language together with most other variable types only to assist the compiler in catching common parameters usage errors. When a parameter is passed to a function as a *const*, or a variable is declared as a *const*, the compiler can in fact help flag any following attempt to modify it. The MPLAB C use of the PSV only extends this semantic in a very natural way, allowing for a more efficient implementation as we have seen.

Notes for the Assembly Experts

- The *string.h* library contains many useful block manipulation functions that can be useful, via the use of pointers, to perform operations on any type of data array, not just strings, such as *memcpy()*, *memcmp()*, *memset()* and *memmove()*.
- The *ctype.h* library, on the other hand, contains functions that help discriminate individual characters according to their position in the ASCII table, to discriminate lower case from upper case, and/or convert between the two cases.

Notes for the PIC Microcontroller Experts

Since the PIC24 program memory is implemented using Flash technology, programmable with a single supply voltage even at run time and during code execution, it is possible to design boot-loaders, which are applications that automatically update part or all of their own code. It is also possible to utilize sections of the Flash program memory as a non-volatile memory storage area, as long as you stay within some pretty basic limitations. To write to the Flash program memory you will need to utilize the table access methods and exercise extreme caution. The PSV window is a read-only mechanism and, as we have seen before, it gives access only to 16 of the 24 bits of each program memory location.

Also pay notice to the fact that the memory can only be written in complete rows of 64 words each and must be first erased in blocks of eight rows (512 words) each. This can make

frequent updates impractical if single words or, as is more usual, small data structures are being managed.

Tips & Tricks

String manipulation can be fun in C once you realize how to make the zero termination character work for you efficiently. Take for example, the *mycpy()* function below:

```
void mycpy( char *dest, char * src)
{
    while( *dest++ = *src++);
}
```

This is quite a dangerous piece of code, as there is no limit to how many characters could be copied. Additionally, there is no check as to whether the *dest* pointer is pointing to a buffer that is large enough and you can imagine what would happen should the *src* string be missing the termination character. It would be very easy for this code to continue beyond the allocated variable spaces and to corrupt the entire contents of the data RAM, including the all precious SFRs.

As a minimum, you should try to at least verify that pointers passed to your functions have been initialized before use. Compare them with the *NULL* value (declared in *stdlib.h* and/or *stddef.h*) to catch the error.

Add a limit to the number of bytes to be copied. It is reasonable to assume that you will know the size of the strings/arrays used by your program and if you don't, use the *sizeof()* operator. A better implementation of *mycpy()* would be the following:

```
void mycpy( char *dest, char *src, int max)
{
    if ((dest != NULL) && ( src != NULL))
        while (( max-- > 0) && ( *src))
            *dest++ = *src++;
}
```

Exercises

Why not try developing new string manipulation functions to perform the following operations:

- Search for a string in an array of strings, sequentially.
- Implement a binary search.
- Develop a simple hash table management library.

Books

- Wirth, N., 1976. *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, NJ.

With un-paralleled simplicity, Wirth (the father of the Pascal programming language) takes you from the basics of programming all the way up to writing your own compiler.

Links

- http://en.wikipedia.org/wiki/Pointers#Support_in_various_programming_languages
Learn more about pointers and see how they are managed in various programming languages.

Flying “Solo”

Congratulations, you have endured the first few lessons and gained the necessary confidence to perform your first flight without an instructor sitting next to you, you are going to fly solo! As a consequence, in the next group of lessons more is going to be expected of you.

In the second part of this book, we will continue reviewing one by one the fundamental peripherals that allow a PIC24 to interface with the outside world. Since the complexity of the examples will grow a little bit, having an actual demonstration board at hand is recommended so that a practical demonstration can be performed. I will refer often to the standard Microchip Explorer16 demonstration board, but any third-party tool that offers similar features or allows for a small prototyping area can be used just as effectively.

Synchronous Communication

On some of the major airlines, sometimes they make available an additional channel – the “cockpit channel” where you can listen to the actual conversation over the radio between the pilots and the traffic controllers. When you listen to it the first few times, it seems impossible to believe that there is actually any intelligent conversation going on. At first it all sounds like a continuous sequence of seemingly random numbers and unrecognizable acronyms. But, as you listen further and become familiarized with some of the terms used in aviation, it starts to make sense. There is a precise protocol that is followed by both pilots and controllers, selected radio frequencies are used as the media, and there is a whole language that must be learned and practiced to communicate from the cockpit of any airplane.

In embedded control, communication is equally a matter of understanding the protocols as well as the characteristics of the physical media available. In embedded control programming, learning to choose the right communication interface can be as important as knowing how to use it.

Flight Plan

In this lesson we will review a couple of communication peripherals available in all the general purpose devices in the new PIC24 family. In particular we will explore the synchronous serial communication interfaces SPI™ and I²C™, comparing their relative strengths and limitations for use in embedded control applications.

Preflight Checklist

In addition to the usual software tools, including the MPLAB® X IDE and MPLAB C compiler for the PIC24, this lesson will require the use of your in circuit programmer/debugger of choice and the Explorer16 demonstration board or equivalent board featuring an RS232 serial port and a Serial EEPROM using the SPI or I²C interface (possibly both).

Use the **New Project Setup** checklist to create a new project called **7-SPI** and a new source file similarly called **spi2.c**.

The Flight

The PIC24FJ128GA010 offers seven communication peripherals that are designed to assist in all common embedded control applications. As many as six of them are “serial”

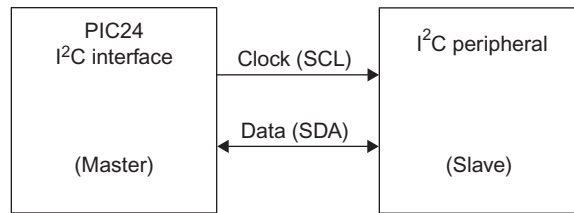


Figure 7.1: I²C Interface block diagram

communication peripherals, as they transmit and receive a single bit of information at a time. They are:

- 2 × the SPI® synchronous serial interfaces
- 2 × the I²C® synchronous serial interfaces
- 2 × the Universal Asynchronous Receiver and Transmitters (UARTs)
- 1 × PMP parallel interface.

The main difference between synchronous interfaces (like the SPI or I²C) and the asynchronous ones (like the UART) is in the way the timing information is passed from transmitter to receiver. Synchronous communication peripherals need a physical line (a wire) to be dedicated to the clock signal, providing synchronization between the two devices.

The Parallel Master Port (PMP) completes the list of basic communication interfaces of the PIC24. The PMP has the ability to transfer up to 8 bits of information at a time, while providing several address lines and featuring the standard control signals: -CS, -RD, -WR that interface directly to most commercially available LCD display modules (alphanumeric and graphic modules with integrated controller) as well as CompactFlash memory cards (or CF-I/O devices), printer ports and an almost infinite number of other basic 8-bit parallel devices available on the market.

In the rest of this lesson we will begin focusing specifically on the use of the synchronous serial interfaces. In the following two chapters we will cover separately the asynchronous serial interfaces (UART) and the PMP.

Synchronous Serial Interfaces

The I²C interface uses two wires (and therefore two pins of the microcontroller), one for the clock (referred to as SCL) and one (bidirectional) for the data (SDA) (Figure 7.1).

The SPI interface instead separates the data line in two, one for the input (SDI) and one for the output (SDO), requiring one extra wire but allowing simultaneous (faster) data transfer in both directions (Figure 7.2).

In order to connect multiple devices to the same serial communication interfaces (bus configuration) the I²C interface requires a 10-bit address to be sent over the data line to select a single device before any actual data is transferred. This slows down the communication but

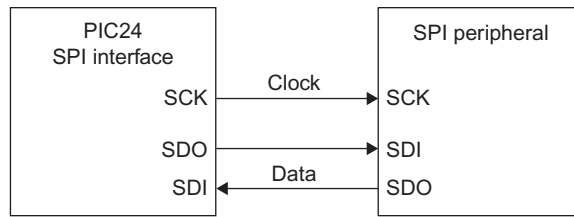


Figure 7.2: SPI Interface block diagram

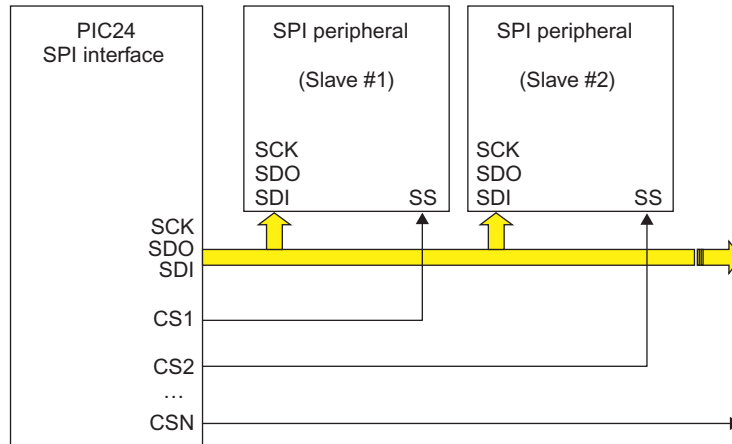


Figure 7.3: SPI Bus block diagram

allows the same two wires (SCL and SDA) to be used for as many as (theoretically) 1024 devices, although in practice it is unlikely you could get anywhere near that number.

In all synchronous serial interfaces the devices that originate the clock signal are typically referred to as the masters, as opposed to the devices that synchronize with it called the slaves. The I²C interface allows for multiple devices to act as masters and share the bus using a simple arbitration protocol. On the other hand the SPI interface requires an additional physical line, the *slave select* (SS), to be connected to each device. In practice this means that when using an SPI bus, as the number of devices connected grows, the number of I/O pins required on the PIC24 grows proportionally with them (Figure 7.3).

Sharing an SPI bus among multiple masters is theoretically possible but practically very rare. The main advantages of the SPI interface are truly its simplicity and its speed, which can be one order of magnitude higher than that of the fastest I²C bus (even without taking into consideration the details of the protocol-specific overhead).

Asynchronous Serial Interfaces

In asynchronous communication interfaces, there is no clock line, while typically two data lines are used: TX and RX respectively for input and output (optionally two more lines

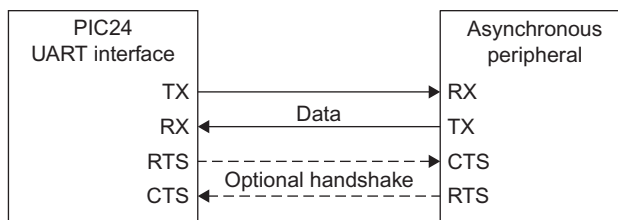


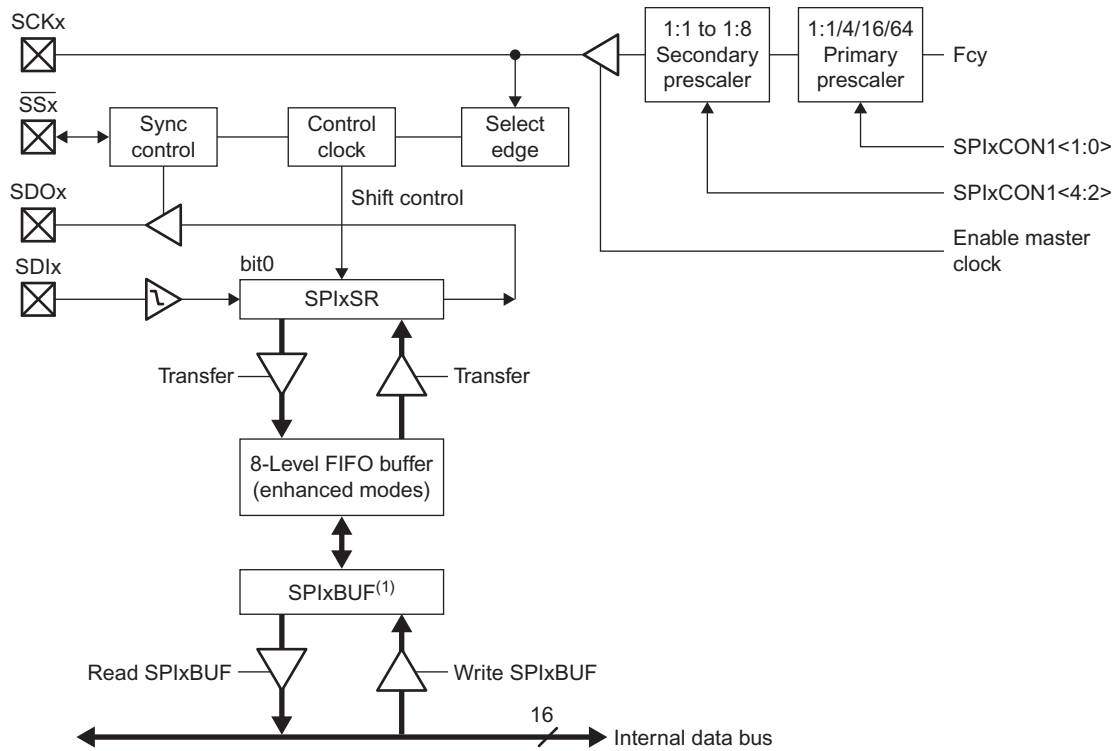
Figure 7.4: Asynchronous serial interface block diagram

Table 7.1: A comparison of basic serial interfaces

Peripheral	Synchronous		Asynchronous
	SPI	I ² C	UART
Max. bit rate	10 Mbit/s	1 Mbit/s	500 Kbit/s
Max. bus size	Limited by no. of pins	128 devices	Point to point (RS232) 256 devices (RS485)
Number of pins	3 + n x CS	2	2 (4 with handshake)
Pros	Simple, low cost, high speed	Small pin count, allows multiple masters	Longer distance, improved noise immunity (requires transceivers)
Cons	Single master, short distance	Slowest, short distance	Requires accurate clock frequency
Typical application	Direct connection to ASICs and other peripherals on same PCB	Bus connection with peripherals on same PCB	Interface with terminals, personal computers and other data acquisition systems
Examples	Serial EEPROMs (25CXXX series), MCP320X A/D converter, ENC28J60 Ethernet controller, MCP251X CAN controller...	Serial EEPROMs (24CXXX series), MCP98XX temperature sensors, MCP322x A/D converters...	RS232, RS422, RS485, LIN bus, MCP2550 IrDA interface...

may be used to provide hardware handshake). The synchronization between transmitter and receiver is obtained by extracting timing information from the data stream itself. Start and stop bits are added to the data and precise formatting (with a fixed baud rate) allows reliable data transfer (Figure 7.4).

Several asynchronous serial interface standards dictate the use of special transceivers to improve the noise immunity, extending the physical distance up to several thousand feet. Each serial communication interface has its advantages and disadvantages. Table 7.1 tries to summarize the most important ones as well as the most common applications.



Note 1: In standard modes, data is transferred directly between SPIxSR and SPIxBUF.

Figure 7.5: The SPI module block diagram

Synchronous Communication Using the SPI Modules

The SPI interface is perhaps the simplest of all the interfaces available, although the PIC24 implementation is particularly rich in options and interesting features.

The SPI interface is essentially composed of an 8-bit shift register; bits are simultaneously shifted in (MSB first) from the SDI line and shifted out from the SDO line in synch with the clock on pin SCK (Figure 7.5).

If the device is configured as a bus master, the clock is generated internally (derived from the peripheral clock after a cascade of two pre-scalers for maximum flexibility) and output on the SCK pin. If the device is a bus slave, the clock is received from the SCK pin.

Beyond this point things get a little messy as, over the years, there has been no consensus on the exact polarity of the SCK signal, the number of bits in a transaction (8 or 16) and where exactly (which edge) the input sampling should happen. The interface is still simple enough so that all cases can be handled by the PIC24 SPI peripherals by setting properly just a small number of configuration bits.

Upper byte:							
U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	DISSCK	DISSDO	MODE16	SMP	CKE
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
SSEN	CKP	MSTEN	SPRE2	SPRE1	SPRE0	PPRE1	PPRE0
bit 7			bit 0				

Figure 7.6: The *SPIxCON1* control register

As for all other peripherals we will encounter, all the essential configuration options have been grouped in a special function register, *SPIxCON1* in this case (Figure 7.6), and additional advanced options are offered in *SPIxCON2*.

To demonstrate the basic functionality of the SPI peripheral we will use the Explorer16 demo board where the PIC24 SPI2 module is connected to a 25LC256 EEPROM device, often referred to as a Serial EEPROM (SEE or sometimes just E^2 – pronounced e-squared). This is a small and inexpensive device that contains 256 Kbits, or 32 Kbytes, of non-volatile, high-endurance memory.

In order to prepare the SPI2 module for communication with this serial memory device we will need to fine tune the peripheral module configuration.

The SEE responds to a short list of 8-bit (*MOD16* = 0) commands that according to the device datasheet must be supplied via the SPI interface with the following setting:

- Clock *IDLE* level is low, clock *ACTIVE* is high (*CKP* = 0)
- Serial output changes on transition from *ACTIVE* to *IDLE* (*CKE* = 1)
- The PIC24 will act as a bus *MASTER* (*MSTEN* = 1) and will produce the clock signal SCK deriving it from the internal clock after prescaling (in this case we will use the default prescalers values 1:64 and 1:8 for a total of 1:512)

The chosen configuration value can be defined as a constant that will be later assigned to the *SPI2CON1* register

```
#define SPI_MASTER 0x0120 // 8-bit master mode, CKE=1, CKP =0
```

To enable the peripheral we will access the *SPI2STAT* register, where, similarly to most other PIC24 peripherals, bit 15 is the main enable control bit, we will define another constant for readability:

```
#define SPI_ENABLE 0x8000 // enable SPI port, clear status
```

Pin 12 of *PORTD* is connected to the memory chip select (*CS*), active low pin, so we will add two more definitions to the program, once more, to make it more readable:

```
#define CSEE    _RD12        // select line for Serial EEPROM
#define TCSEE   _TRISD12     // tris control for CSEE pin
```

We can now write the peripheral initialization part of our demonstration program:

```
// 1. init the SPI peripheral
TCSEE = 0;           // make SSEE pin output
CSEE = 1;            // de-select the Serial EEPROM
SPI2CON1 = SPI_MASTER; // select mode
SPI2STAT = SPI_ENABLE; // enable the peripheral
```

We can now write a small function that will be used to transfer data to and from the Serial EEPROM device:

```
// send one byte of data and receive one back at the same time
int WriteSPI2( int data)
{
    SPI2BUF = data;           // write to buffer for TX
    while( !SPI2STATbits.SPIRBF); // wait transfer completion
    return SPI2BUF;           // read the received value
} // WriteSPI2
```

The function *WriteSPI2()* is truly a bi-directional transfer function. It immediately writes a character to the transmit buffer and then enters a loop to wait for the receive flag to be set to indicate that the transmission was completed as well as data was received back from the device. The data received is then returned as the value of the function.

When communicating with the memory device though, there are situations when a command is sent to the memory, but there is no immediate response. Also there are cases when data is read from the memory device but no further commands need to be sent by the PIC24. In the first case (write command), the return value of the function can be simply ignored. In the second case (read command), a dummy value can be sent to the memory while shifting in data from the device.

The 25LC256 datasheet contains accurate depictions of all seven possible command sequences that can be used to read or write data to and from the memory device. A small table of constants can help encode all such commands:

```
// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1        // write status register
#define SEE_WRITE   2        // write command
#define SEE_READ    3        // read command
#define SEE_WDI     4        // write disable
#define SEE_STAT    5        // read status register
#define SEE_WEN     6        // write enable
```

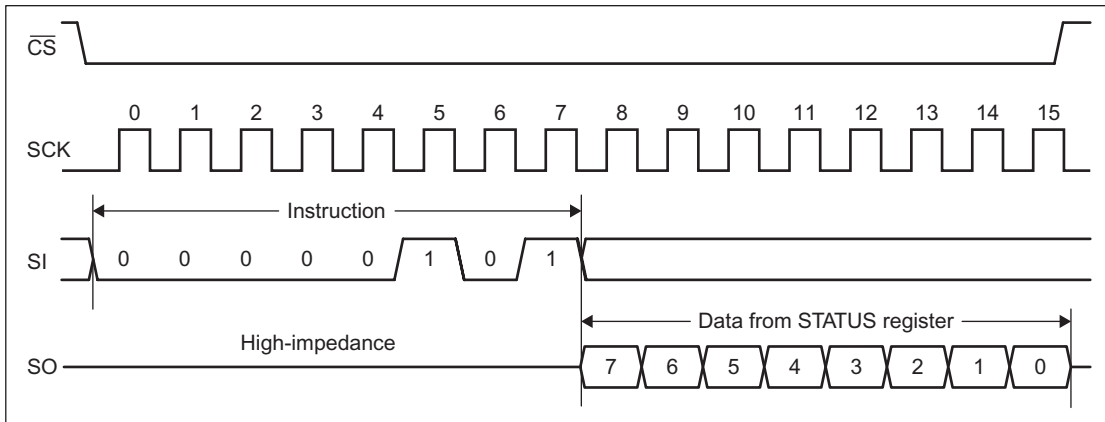


Figure 7.7: The complete Read Status Register command timing sequence

We can now write a small test program to verify that the communication with the device is properly established. For example, using the Read Status Register command we can interrogate the memory device and verify that the SPI is properly configured.

Testing the Read Status Register Command

After sending the appropriate command (*SEE_STAT*), we will need to add an additional call to the *WriteSPI2()* function with a dummy piece of data to capture the response from the memory device (Figure 7.7).

Sending any command to the SEE requires as a minimum the following steps:

1. activate the memory by taking low the CS pin
2. shift out the 8-bit command
3. add one or more additional steps here, depending on the specific command
4. de-activate the memory (taking high the CS pin) to complete the command, after which the memory will go back to a low power consumption stand-by mode

In practice, the following code is required to perform the complete Read Status Register operation:

```
// 2. Check the Serial EEPROM status
CSEE = 0;           // select the Serial EEPROM
writeSPI2( SEE_STAT); // send a READ STATUS COMMAND
i = writeSPI2( 0);   // send/receive
CSEE = 1;           // deselect, terminate command
```

The complete project listing should look like:

```
/*
** SPI2
*/
#include <config.h>
```



```

// I/O definitions
#define CSEE    _RD12        // select line for Serial EEPROM
#define TCSEE    _TRISD12    // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER 0x0120    // select 8-bit master mode, CKE=1, CKP=0
#define SPI_ENABLE 0x8000    // enable SPI port, clear status

// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1        // write status register
#define SEE_WRITE    2        // write command
#define SEE_READ    3        // read command
#define SEE_WDI    4        // write disable
#define SEE_STAT    5        // read status register
#define SEE_WEN    6        // write enable

// send one byte of data and receive one back at the same time
int WriteSPI2( int data)
{
    SPI2BUF = data;           // write to buffer for TX
    while( !SPI2STATbits.SPIRBF); // wait transfer completion
    return SPI2BUF;           // read the received value
} // WriteSPI2

main()
{
    int i;

    // 1. init the SPI peripheral
    TCSEE = 0;                // make SSEE pin output
    CSEE = 1;                  // de-select the Serial EEPROM
    SPI2CON1 = SPI_MASTER;    // select mode
    SPI2STAT = SPI_ENABLE;    // enable the peripheral

    // 2. Check the Serial EEPROM status
    CSEE = 0;                  // select the Serial EEPROM
    WriteSPI2( SEE_STAT);      // send a READ STATUS COMMAND
    i = WriteSPI2( 0);         // send/receive
    CSEE = 1;                  // deselect, terminate command
} // main

```

After connecting the Explorer16 demo board to your programmer/debugger tool, select **Debug>Project** (or press **CTRL+F5**). After a few seconds, the PIC24 should be programmed, verified and the program counter (green bar) will be set on the first line of the main function.

Open the **Watches** window and add the symbol **i** to the list.

Now place the cursor on the last line of code in the main function and **set a break point** (click on the gutter to the left of the line number).

Then start the execution by selecting the **Debugger>Continue (F5)** command.

When the execution terminates, the contents of the 25LC256 memory Status Register should have been transferred to the variable *i*, visible in the watches window.

Table 7.2: The 25LC256 Serial EEPROM status register

7	6	5	4	3	2	1	0
W/R	-	-	-	W/R	W/R	R	R
WPEN	x	x	x	BP1	BP0	WEL	WIP

W/R = Writable/readable.

R = read-only.

Unfortunately, you will be disappointed to learn that the default status of the 25LC256 memory (at power on) is represented by a 0x00 value, since *BP1* and *BP0* are off to indicate no block protection, the write enable latch *WEL* is disabled, and no Write In Progress *WIP* flag should be active.

Not a very telling result for our little test program. So to spice up things a little we could start by setting the Write Enable latch before interrogating the Status Register – it would be great to see bit 1 set (see Table 7.2).

To set the Write Enable latch we will insert the following code before the section 2 that we will promptly renumber to 2.2:

```
// 2.1 send a Write Enable command
CSEE = 0;           // select the Serial EEPROM
WriteSPI2( SEE_WEN); // write enable command
CSEE = 1;           // deselect, terminate command
```

Relaunch the **Debug>Project** command and, with a breakpoint still set on the last line of code in the main program, select **Debug>Continue**.

If everything went well you will see the variable *i* in the watch window updated showing a value of 2.

Now these are the great satisfactions that you can get only by developing code for a 16-bit embedded controller!

More seriously, now that the Write Enable latch has been set, we can add a write command and start modifying the contents of the EEPROM device. We can write a single byte at a time, or we can write a long string, up to a maximum of 64 bytes, all in a single swoop using a so-called Page Write. On the memory device datasheet you will find out more about simple address restrictions that apply to this mode of operation.

Writing to the EEPROM

After sending the write command, two bytes of address (*addr_MSB* and *addr_LSB*) must be supplied before the actual data is shifted out. The following code exemplifies the correct write sequence:

```
// send a Write command
CSEE = 0;           // select the Serial EEPROM
```

```

WriteSPI2( SEE_WRITE);      // write command
WriteSPI2( addr_MSB);       // address MSB first
WriteSPI2( addr_LSB);       // address LSB (word aligned)
WriteSPI2( data);           // send 8-bit of data
// continue writing more data...
CSEE = 1;

```

Notice how the actual EEPROM write cycle initiates only after the CS line is brought high again. Also, it will be necessary to wait for a time (T_{wc}) specified in the memory device datasheet for the cycle to complete before a new command can be issued. There are two methods to make sure that the memory is allowed the right amount of time to complete the write command. The simplest one consists of inserting a fixed delay after the write sequence. The length of such delay should be longer than the maximum cycle time specified in the memory device datasheet ($T_{wc\ max.} = 5\ ms$).

A better method consists of checking the Status Register contents before issuing any further read/write command, waiting for the Write In Progress (WIP) flag to be cleared (this will also coincide with the Write Enable bit being reset).

```

// wait until any write in progress is completed
while ( ReadSR() & 0x1); // check the WIP flag

```

By doing so, we will be waiting only the exact minimum amount of time required by the memory device in the current operating conditions.

Reading the Memory Contents

Reading back the memory contents is even simpler; here is a snippet of code that will perform the necessary sequence:

```

// perform a read sequence
CSEE = 0; // select the Serial EEPROM
WriteSPI2( SEE_READ); // read command
WriteSPI2( addr_MSB); // address MSB first
WriteSPI2( addr_LSB); // address LSB (word aligned)
data = WriteSPI2( 0); // send dummy, read msb
// continue reading a second byte, a third...
CSEE = 1;

```

The read sequence can be indefinitely extended by reading sequentially the entire memory contents if necessary, and upon reaching the last memory address ($0x7FFF$), rolling over and starting from $0x0000$ again.

A Non-Volatile Storage Library

We can now assemble a small library of functions dedicated to accessing the 25LC256 Serial EEPROM. The library will hide all the details of the implementation such as the SPI port

used, specific sequences and timing details. It will expose instead only two basic commands to read and write integer data types to a generic (black box) non-volatile storage device.

Let's create a new project using the new project wizard and the usual checklist. An appropriate name could be **7-NVM**. We can now copy most of the previous example code in a new file that we will call **SEE25.c**, and transfer it to the /lib subdirectory. This will be our first new library module.

```
/*
** SEE25 Access Library Module
*/
#include <p24Fxxx.h>
#include <SEE.h>

// I/O definitions for PIC24 + Explorer16 demo board
#define CSEE    _RD12        // select line for Serial EEPROM
#define TCSEE    _TRISD12    // tris control for CSEE pin

// peripheral configurations
#define SPI_MASTER 0x0122    // select 8-bit master mode, CKE=1, CKP=0, 4MHz
#define SPI_ENABLE 0x8000    // enable SPI port, clear status

// 25LC256 Serial EEPROM commands
#define SEE_WRSR    1        // write status register
#define SEE_WRITE    2        // write command
#define SEE_READ    3        // read command
#define SEE_WDI    4        // write disable
#define SEE_RDSR    5        // read status register
#define SEE_WEN    6        // write enable
```

From the previous example code you can extract the initialization code, the SPI2 write function and the status register read command. Each one will become a separate function.

```
void InitSEE(void)
{
    // init the SPI peripheral
    CSEE = 1;                // de-select the Serial EEPROM
    TCSEE = 0;                // make SSEE pin output
    SPI2CON1 = SPI_MASTER;    // select mode
    SPI2STAT = SPI_ENABLE;    // enable the peripheral
}

// InitSEE

// send one byte of data and receive one back at the same time
int WriteSPI2( int data)
{
    SPI2BUF = data;          // write to buffer for TX
    while( !SPI2STATbits.SPIRBF); // wait transfer completed
    return SPI2BUF;          // read the received value
}

// WriteSPI2
```

```

int ReadSR( void)
{
    // Check the Serial EEPROM status register
    int i;
    CSEE = 0;           // select the Serial EEPROM
    WriteSPI2( SEE_RDSR); // send a READ STATUS COMMAND
    i = WriteSPI2( 0);   // send/receive
    CSEE = 1;           // deselect to terminate command
    return i;
} // ReadSR

```

To create a function that reads a 16-bit integer value from the non-volatile memory, first we verify that any previous command (write) has been correctly terminated by reading the status register. A sequential read of two bytes is used to assemble an integer value.

```

int iReadSEE( int address)
{ // read a 16-bit value starting at an even address
    int lsb, msb;
    // wait until any work in progress is completed
    while ( ReadSR() & 0x1); // check the WIP flag

    // perform a 16-bit read sequence
    CSEE = 0;           // select the Serial EEPROM
    WriteSPI2( SEE_READ); // read command
    WriteSPI2( address>>8); // address MSB first
    WriteSPI2( address & 0xfe); // address LSB (word aligned)
    msb = WriteSPI2( 0); // send dummy, read msb
    wlsb = WriteSPI2( 0); // send dummy, read lsb
    CSEE = 1;
    return ( msb<<8)+ lsb);
} // iReadSEE

```

Finally, the write enable function can be created extracting the short segment of code used to access the Write Enable latch from our previous project and adding a page write sequence.

```

void WriteEnable( void)
{
    // send a Write Enable command
    CSEE = 0;           // select the Serial EEPROM
    WriteSPI2( SEE_WEN); // write enable command
    CSEE = 1;           // deselect to complete the command
} // WriteEnable

void iWriteSEE( int address, int data)
{ // write a 16-bit value starting at an even address
    // wait until any work in progress is completed
    while ( ReadSR() & 0x1); // check the WIP flag

    // Set the Write Enable Latch
    WriteEnable();

    // perform a 16-bit write sequence (2 byte page write)
    CSEE = 0;           // select the Serial EEPROM
    WriteSPI2( SEE_WRITE); // write command

```

```
WriteSPI2( address>>8);    // address MSB first
WriteSPI2( address & 0xfe); // address LSB (word aligned)
WriteSPI2( data >>8);      // send msb
WriteSPI2( data & 0xff);    // send lsb
CSEE = 1;
} // iWriteSEE
```

More functions could be added at this point to access *long* and *long long* types for example, but for our purposes this will suffice.

Note that the *page write* operation requires the address to be aligned on a power of two boundary (in this case any even address will do). The requirement must be extended to the read function for consistency.

Now add the **SEE25.c** file to the project by right-clicking on the project window on the **Source Files** logic folder and choose **Add Existing Item**, then select the **SEE25.c** file from the current project directory.

To make a few selected functions from this module accessible to other applications, create a new file: **SEE.h** file and move it to the **/include** subdirectory:

```
/*
** SEE.h
**
** encapsulates a Serial EEPROM functionality
** exposing only 16-bit integer storage functions
*/
// initialize access to memory device
void InitSEE(void);

// 16-bit integer read and write functions
// NOTE: address must be an even value
// (see page write restrictions on device datasheet)
int iReadSEE ( long address);
void iWriteSEE( long address, int data);
```

This will expose only the initialization function and the integer read/write functions, hiding all other details of the implementation.

Add the **SEE.h** file to the project by right clicking in the project windows on the **Header Files** logic folder and select it from the current project directory.

Testing the New SEE Library Module

To test the functionality of the library module, we can create a small test application that will attempt to fill the memory with a sequence of 16-bit integers. Each value will be written and immediately read back to verify the successful update of each memory location.

```
/*
** SEETest.c
**
```

```

#include <config.h>
#include "SEE.h"

main()
{
    int i, r;

    // initialize the SPI2 port and CS to access the 25LC256
    InitSEE();

    // fill memory and verify
    for( i=0; i<16*1024; i++)
    {
        iWriteSEE( i<<1, i);    // write
        r = iReadSEE( i<<1);    // read back
        if (r!=i)
            break;              // error
    }

    // main loop
    while( 1);
} //main

```

Save this file as **SEETest.c** and add it to the current project too.

Once the **Debug>Project** command is completed successfully, the code will already be programmed on the device and the green cursor will be positioned on the first line of the main function (or possibly inside the first line of the first function invoked by the main function).

- Set a **breakpoint** on the empty main loop at the end of the main function.
- Hit the **Debug>Continue** command and watch the program stop after several seconds.
- Note the value of the variable *i* in the **Watches** window.

If all went well, the variable *i* should have reached the value 0x400, indicating that all 16 Kwords (32 Kbytes) of the device have been written and verified correctly. In the improbable case where you should find *i* to be a smaller value, it would indicate the address of the first memory location where the verify step failed (a damaged memory cell?).

Note

Feel free to modify this test to perform a number of read/write cycles on each memory location, but make sure to always limit this number of write cycles by inserting a counter or a break point in your loops or it will quickly turn into a test of the Serial EEPROM *endurance*. In fact, if left free to loop indefinitely on a single location (at a rate that will be mostly dependent on the actual *T_{wc}* of the device), in the best-case scenario, maximum *T_{wc}* (5 ms), the theoretical endurance limit of the EEPROM (1,000,000 cycles) could be reached in 5,000 seconds, or slightly less than one hour and a half of continuous operation. In the worst-case scenario (a fast EEPROM) it could take less than twenty minutes!

The I²C Interface

The fact that on the Explorer16 demonstration board there are no I²C connected peripherals to experiment with is in no way an indication of a lesser importance of this interface to the embedded control programmer. On the contrary, if anything, the popularity of the I²C interface has been growing in recent years. This is reflected in the continuous introduction of new devices using I²C, but also new advanced application-specific protocols that use I²C at their physical level and as a transport layer.

A few examples of devices recently introduced by Microchip using the I²C interface offer an even broader picture of the many uses for this interface:

- 1-Mbit Serial EEPROMS: 24xx1025
- 18-bit delta sigma ADCs: MCP3421
- 16-bit delta sigma ADCs: MCP3425
- 12-bit SAR ADCs: MCP3221
- 12-bit D/A: MCP4725
- integrated temperature sensor (+/−0.5°C): MCP9803
- I/O expander 8/16-bit: MCP23016/MCP2308.

But many other manufacturers expand the list of applications further to include:

- Battery gauges
- Audio codecs
- GPS receivers
- LCD display controllers
- Card readers.

Examples of application-specific protocols based on the I²C interface are:

- SMBus™, the System Management Bus, used in personal computers (and servers) for low-speed system management applications
- IPMI™, the Intelligent Platform Management Interface, used to monitor the system health and manage systems (mostly servers) remotely
- PMBus™, the Power Management Bus, used by advanced digitally controlled power supply units to monitor, configure and control power.

The hardware requirements of the generic interface are very simple:

- A clock line (SCL), controlled in open drain configuration by the Master(s)
- A data line (SDA), controlled (open drain configuration as well) alternatively by Master and Slave devices
- Two resistors, with values that will vary with selected speed and voltages (typically between 2.2 k ohm and 4.5 k ohm each), are used to pull up the two open-drain output buffers.

That's all! Now you are beginning to understand the reason for the popularity of this interface. Since, as we said from the beginning, I²C is a synchronous serial interface, two distinct types of devices can access the bus: masters and slaves.

Masters can initiate the communication. They get to control the SCL line as they are in charge of the clock generation. They send commands and control the actual transfer speed. The I²C specifications indicate a range of relatively low clock speeds. The original documentation (published by Philips decades ago) indicated a maximum of 100 kbit/s, while the most recent updates (Smart Serial specification) brought the speed up to 400 kbit/s first and, only more recently, up to 1 Mbit/s. This is clearly much lower than the SPI interface clock rates and perhaps within the realm of a software (bit-banged I/O) solution.

When we consider the most common application that sees the PIC[®] microcontroller as the *single master* addressing a handful of devices – typically 24xx Serial EEPROMs – this is definitely an option. The advantage of a software approach is in the freedom it provides to choose any I/O pin we want. The task is further simplified on the PIC24 since all I/Os can easily be configured for open-drain operation (see *ODCx* registers).

Note

In the single master case mentioned above, the SCL line control never changes hands and therefore a common hardware and software simplification includes reducing this line to a standard CMOS output.

In the good old times, (assembly) implementations of single master I²C interface for 8-bit PIC microcontrollers could be coded using as little as **40 instructions!**

Another common trick used in power-constrained applications is that of powering the entire peripheral device directly off one of the I/Os of the PIC microcontroller. In particular, when interfacing a small Serial EEPROM, say a 2-Kbytes-capable 24LC16, you will verify that the power consumption is limited to 3 mA even when operating at 400 kHz. A value that is well within the capabilities of any PIC24 I/O port.

Note

On the Microchip web site you will find several examples of *software* I²C libraries (AN1100 for example) either written for or easily ported to the PIC24.

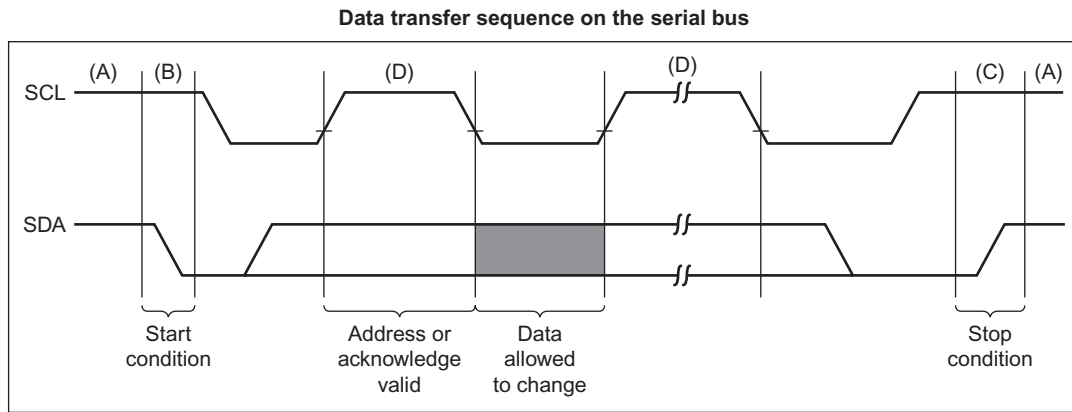


Figure 7.8: Data transfer sequence on the I²C bus

I²C Data Transfer Rules

Contrary to the case of the SPI interface, the I²C interface is based on a single, well-defined set of specifications that leave no ambiguity to the polarity of the signals. At idle, both lines SCL and SDA are supposed to be high for the effect of the pull-up resistors.

Two simple rules dictate how to operate from here:

1. When the SCL line is low, and only at this time, the SDA line can change.
2. When the SCL line is high, the SDA line status indicates the value of a bit.

Two exceptions to rule 1 create special conditions that are used to delimit the beginning and end of each *transaction* between two devices on the bus. When SCL is high:

- A *START* condition is indicated by the SDA line changing from high to low.
- A *STOP* condition is indicated by the SDA line changing from low to high.

Figure 7.8, extracted from a typical device datasheet, is worth a million words.

Since I²C is a *bus*, not just a *point-to-point* connection, multiple devices are allowed to talk to each other over the same two wires. Originally, the standard called for a 7-bit *device address* to be used for identifying individual devices, but this got eventually extended to 10 bits, increasing from 128 to 1024 the number of individual devices that can be theoretically connected on the same pair of wires.

Things are simple for slaves. They cannot initiate transactions, but they have to keep quiet and wait for a master to provide the clock and, only when addressed, respond. In contrast, things can get a bit more complicated for the masters. It can happen that two of them initiate the communication at the same time, generating what is called a *bus collision*.

Collisions are easy to detect: when one of the masters is trying to send a *1* (SDA line in pull up) and it notices that something else is *yanking* the line low sending a *0*, there is the clue!

Arbitration is just a matter of deciding who gets to continue, which, quite obviously, has to be the one sending the *0*. In other words, the master requesting to talk to the lowest address, the one with more zeros in the most significant bits of the address, gets to continue. As long as the *loser* is quick to recognize the situation, there is actually no interruption or disruption of the communication whatsoever and the whole thing plays out quite transparently.

The vast majority of I²C applications in embedded control though rotate around a much simpler scenario where there is only one master (the microcontroller) and one or a handful of slaves, so you won't have to worry about collisions.

Also, the most common type of slave device used happens to be the Serial EEPROM and if 25xxx is the numbering standard for all SPI capable serial EEPROMs, 24xx is the numbering scheme used to denote I²C memory devices.

I²C Serial EEPROMs

I²C has been for years the favorite choice for Serial EEPROM users and manufacturers for two reasons:

- Only two pins (I/Os) are required to communicate with the device, enabling the embedded control designer to use very low pin count (read inexpensive) microcontrollers.
- Just four pads (areas of contacts on the chip) are required for a Serial EEPROM device. Two provide the power supply and the remaining two are the I²C lines. This reduces the silicon area overhead due to the contacts on a typically very small device (hence the high impact/cost of the pads), an important advantage in the eyes of the manufacturer.

In practice, most I²C Serial EEPROMs have at least a fifth contact/pin (*WP*) to protect the device contents by disabling writing; see [Figure 7.9](#).

Note

I²C Serial EEPROMs are offered today in a five-pin **SOT23 package** with capacities from 128 bits up to 16 Kbits (2 Kbytes). Notice how only a few years ago this package was used for individual transistors in surface-mount technology.

Numerous Serial EEPROM models offer three additional pins (bringing the grand total to eight), acting as address select inputs, to allow for banking of up to eight devices on the same bus.

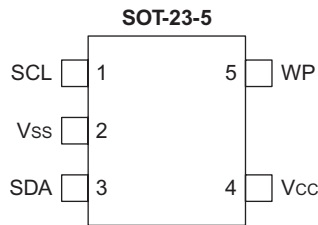


Figure 7.9: A serial EEPROM in a SOT23-5 package

In fact, the original (7-bit) I²C specifications called for the device address to be split in two groups:

- The first four bits (most significant) were fixed and assigned by the standard to each *type* of device: Serial EEPROMs were assigned the value *0101(0x5)*.
- The remaining three bits (least significant) would be matched with three external pins.

As a consequence, in any given I²C bus the original standard allowed only up to eight serial memory devices to coexist, assigning them the addresses: 0x50, 0x51 ... 0x57.

Today when Serial EEPROMs are fitted into a package with fewer than eight pins (and in other special cases we will discuss below) the three least significant bits of the device address are hard-wired to ground internally (see 24LC00 models) or simply re-used for *other* purposes (see 24LC16B models).

Talking to I²C Serial EEPROMs

It's time to start sending and receiving data bytes and to learn how to form commands to communicate with an I²C Serial EEPROM. As we have seen, the I²C bus uses a single wire (SDA) to transfer data bits back and forth between slaves and masters. While this offers a potential advantage over the SPI interface (reducing the number of pins required and therefore the cost) it also means that the communication can only happen in one direction at a time; this is often referred to as *half duplex* communication, and the two devices better agree carefully on the timing for proper control of the line.

Each time a byte is sent (from master to slave) on the I²C bus, the master device has to generate nine clock pulses on the SCL line:

- During the first eight pulses, the master will control the SDA line and send the bits starting from the most significant bit.
- During the ninth pulse, the SDA line will be released by the master and it will be the slave's turn to reply with an acknowledge bit (0 = Acknowledge, 1 = Not Acknowledged) to confirm it is present and ready to execute the upcoming command.

When a byte must be sent from a slave to a master, it is still the master's responsibility to generate the nine clock pulses on the SCL line but this time:

- During the first eight pulses, the master will release the line and let the slave send the data bits starting from the most significant.

- During the ninth clock pulse, the master will regain control of the SDA line and will send back to the slave a feedback in the form of another acknowledge bit (Figure 7.10).

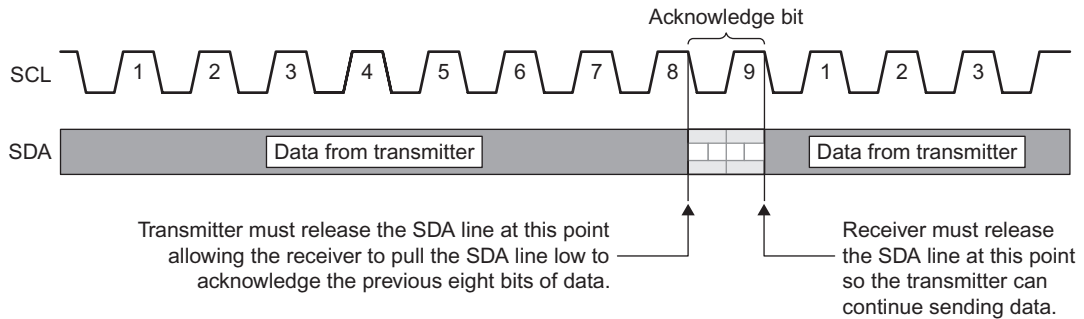


Figure 7.10: Alternating control of the SDA line during the Acknowledge bit

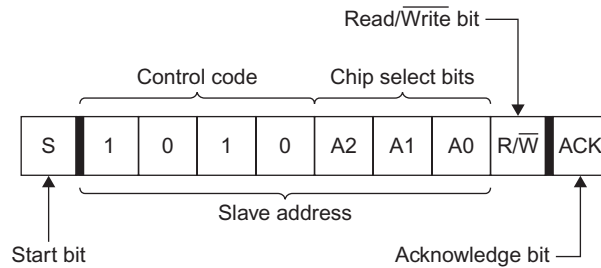


Figure 7.11: The address byte containing the read/write command bit

Forming Commands

There are two fundamental types of commands in I²C: *read* commands and *write* commands. Both are composed of a single byte containing the (7-bit) device address and one last eighth bit used to differentiate between the two operations: 0 for write, 1 for read (Figure 7.11).

Note

The great simplicity of the protocol is also, as you can imagine, the source of one of the greatest concerns for the embedded control designer. That single bit (bit 0 of the command byte) is all that separates an innocuous read operation from a potential memory content corruption! It is so easy for noise to sneak into an I²C bus transaction and, by affecting a single bit, transform a routine read operation into a nightmare. It is for this reason that most I²C SEE devices offer the external Write Protect pin (not just a Write Enable bit in a control register like most SPI SEE do) to disable completely all write operations.

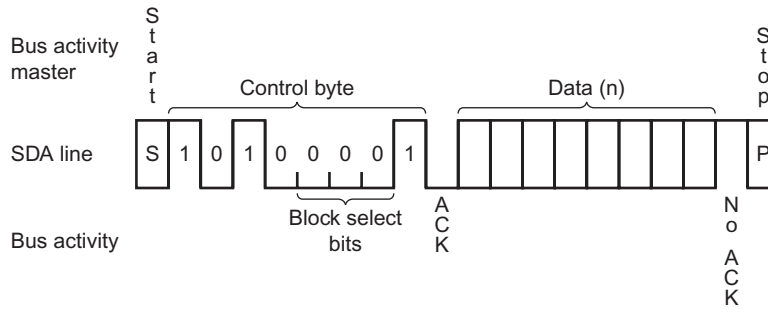


Figure 7.12: A read command

The SEE Grammar

By combining all the basic elements we have reviewed so far, we can compose now any command the I²C protocol can support. Using the following simple notation:

- *S* = START sequence
- *P* = STOP sequence
- *A/N* = Acknowledge bit
- *0xXX* = data byte (in hex notation)
- *0bXXXXXXXX* = data byte (in binary notation).

and using brackets () to indicate parts of the conversation produced by a slave we can represent a typical I²C protocol message in a compact yet readable notation.

Here is an example of a read command sequence for a 24LC00 (128 bit) SEE:

```
S 0b01010001 (A) (Data) NP
```

Or in other words:

- The master sends a Start bit
- Followed by a read command composed of:
 - the SEE type address “0101”
 - three chip select bits “000”
 - the READ command “1”
- The slave acknowledges (if present and the address matches)
- The slave sends back the contents of the current memory location (Data)
- The master sends a NACK (not acknowledge) to communicate that no further data is required
- Followed by the Stop bit to terminate the transaction

Figure 7.12 offers a more graphical representation.

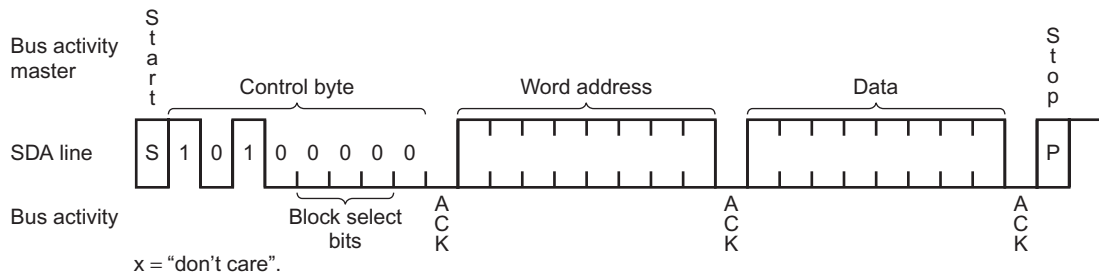


Figure 7.13: A write command

Similarly a write command can be represented with the following compact notation:

S 0b01010000 (A) *ByteAddress* (A) *Data* (A) P

Or in other words:

- The master sends a Start bit
- Followed by a write command composed of:
 - the SEE type address “0101”
 - three chip select bits “000”
 - the WRITE command “0”
- The slave acknowledges (if present and the address matches)
- The master sends the *Byte Address* of the data to be written
- The slave sends an ACK (acknowledge) to confirm the address reception
- The master sends a first byte of *Data*
- The slave sends an ACK (acknowledge) to communicate the data reception
- Followed by the Stop sequence to terminate the transaction

Once more, all of the above can be graphically represented as in [Figure 7.13](#).

Reading and Writing 16-Bit Values

Similarly to what we have done with the SPI interface, we can now prepare a small library module to demonstrate how to encapsulate the operation of a Serial EEPROM device (a 24LC16 to be precise) exposing only a few functions to read and write 16-bit integers.

The two I²C peripheral modules of the PIC24 are controlled by the *I2CxCON* register that allows us to activate pretty much every function and capability of the module ([Figure 7.14](#)).

In practice, you can consider it split into two parts: the top bits (bit 15–bit 5) of the register define the configuration and mode of operation of the peripheral. The bottom five bits (bit 4–bit 0) are triggers that initiate I²C protocol sequences. For example, setting the *SEN* bit initiates a START sequence. Setting the *PEN* bit activates a STOP sequence and so on...

Upper byte:							
R/W-0	U-0	R/W-0	R/W-1 HC	R/W-0	R/W-0	R/W-0	R/W-0
12CEN	—	12CSIDL	SCLREL	IPMIEN	A10M	DISSLW	SMEN
bit 15				bit 8			

Lower byte:							
R/W-0	R/W-0	R/W-0	R/W-0 HC	R/W-0 HC	R/W-0 HC	R/W-0 HC	R/W-0 HC
GCEN	STREN	ACKDT	ACKEN	RCEN	PEN	RSEN	SEN
bit 7				bit 0			

Figure 7.14: The I2CxCON register

When a sequence terminates, the corresponding bit in the control register is automatically cleared.

There is also a status register (*I2CxSTAT*) available to report when errors have been encountered and to report a number of special conditions mostly useful when operating in slave mode.

Also, each I²C module has a dedicated baud-rate register (*I2CxBRG*), essentially a divider of the system clock, that allows us to control the speed of execution of each sequence, relieving us of the need to time each operation manually. Even though in the following, for simplicity, we will wait for each sequence to complete, we could devise the whole program to use interrupts and avoiding any delay.

Note

In the following, I will make use of the *I2C.h* standard peripheral library as it provides just such a thin layer of abstraction to be useful without getting in the way and allowing us to learn to use the hardware peripheral quickly.

Finally, we get to create a new source file, we'll call it **SEE24.c**, starting with the usual PIC24 include file, the *I2C.h* library file, but also including the same *SEE.h* header file we prepared before for the SPI module: the functions implemented will be the same.

```
/*
** SEE24.c
**
** 24LCXX I2C Serial EEPROM access demo
*/
#include <EX16.h>
#include <i2c.h>
#include <SEE.h>
```


Following a pattern similar to the one used in many other exercises before, we will define an initialization function *InitSEE()* that will take care of configuring the *I2C1* module to operate as a (single) master at the relatively low speed of 100 kHz.

```
void InitSEE( long fcy)
// fcy = processor operating frequency in Hz (system clock)
{ // Configure I2C for 7 bit address mode 100kHz

    OpenI2C1( I2C_ON | I2C_IDLE_CON | I2C_7BIT_ADD | I2C_STR_EN
              | I2C_GCALL_DIS | I2C_SM_DIS | I2C_IPMI_DIS,
              (fcy /(2*BUS_FRQ))-1);

    IdleI2C1();
    T1CON=0x8030;
    TMR1=0;
    while( TMR1< 100);
} //InitSEE
```

As you can see, all the heavy lifting is performed by two functions calls: *OpenI2C1()* essentially writes to *I2C1CON* the required (basic) configuration passed as the first parameter, and writes to the *I2C1BRG* the required clock divider value calculated for the given system clock frequency and the desired bus frequency, passed as a second parameter.

```
#define BUS_FRQ 100000L    // 100kHz
```

Note

The first parameters passed to the *OpenI2C1()* function is a good example of the dilemma presented to the programmer by the use of the peripheral libraries. Six symbols/macros are or-ed together to enable/disable features of the I2C module: *I2C_IDLE_CON*, *I2C_7BIT_ADD*, *I2C_STR_EN*, *I2C_GCALL_DIS*, *I2C_SM_DIS* and *I2C_IPMI_DIS*. Are you sure you can match them with the corresponding bits of the *I2CxCON* register?

Also, the peripheral libraries allow the choice between the use of the OR (“|”) function or the AND (“&”) function to assemble those symbols. The choice is more of a matter of style and personal preferences, but confusingly enough the selection is done by defining (or not) another enigmatic symbol: *USE_AND_OR*. Unfortunately, if you get it wrong, you will not receive any error message, your code simply won’t work.

My preference is for the OR style, as you can see. So should I “*#define USE_AND_OR*” would it work with AND or OR? My head is spinning!

This mystery will be revealed in just a few paragraphs.

The other function offered by the *I2C.h* library, *IdleI2C1()*, is quite a handy one, and we will make heavy use of it in the following. It is kind of a catch-all function that makes sure every sequence the I²C peripheral module is working on is completed. A short timed delay

loop, using `Timer1`, completes the initialization sequence and has been added there only to make sure the Serial EEPROM device has time to settle after a power-up sequence before our PIC24 starts tormenting it.

Actually, since we are at it, we can rewrite this last part using the standard peripheral libraries as well. Simply add one more include to the list:

```
#include <timer.h>
```

And then transform the last three lines of the *InitSEE()* function into:

```
OpenTimer1( T1_ON | T1_SOURCE_INT | T1_PS_1_256, -1);
TMR1=0;
while( TMR1< 100);
```

As you can see, the resulting code is a just a little more descriptive of the `Timer1` settings.

One final detail, let's make sure to include the *EX16.h* header file. In there, since chapter 5, we have placed two useful definitions:

- *FCY* – the frequency of the main system clock determined both by the selection of the oscillator (and PLL) and the crystal mounted on the Explorer16 board.
- *USE_AND_OR* – a symbol that, when defined, configures all the peripheral library modules to use the OR function (instead of the default AND) to aggregate bit fields used in the various function parameters.

The iWriteSEE() Function

In the previous section, we saw the sequence required to issue a write command to a Serial EEPROM. Here we will extend that sequence to store a 16-bit value (two bytes) using what is called a *page write* sequence. We can summarize it using the following compact notation:

```
S 0b01010000 (A) ByteAddress (A) DataLSB (A) DataMSB (A) P
```

The first part of this sequence (in bold) is responsible for selecting a specific memory location, the address, where the write operation will begin. It can be conveniently encapsulated in a function on its own as it will return to be useful later.

Here is how we can code it using two new functions: *StartI2C1()* and *MasterWriteI2C1()*, whose respective roles will be obvious:

```
int addressSEE( long add)
// send the address selection command
// repeat if SEE busy
{
    int cmd;

    // 1. Form SEE command + address msb (3)
    cmd= 0xA0|((add>>7)&0xE);
```

```
// 2. WRITE(!) the Address msb
// try send command and repeat until ACK is received
while( 1)
{
    StartI2C1();
    IdleI2C1();

    // send command and address msb(3)
    MasterWriteI2C1( cmd+WRITE_CMD);
    IdleI2C1();

    if (I2C1STATbits.ACKSTAT==0)
        break;

    StopI2C1();
    IdleI2C1();
} // while waiting for ACK

// 3. send byte address
MasterWriteI2C1( add);
IdleI2C1();

// 4. exit returning the cmd byte
return cmd;
} // addressSEE
```

Notice how, after the command is formed, the second part of the function is actually expressed as a loop. In fact, after a write command is sent to a Serial EEPROM, the device might return a NACK indicating that it is busy and not available to take the command at the moment.

Busy doing what?, you will ask. Well, EEPROMs are notoriously slow devices when it comes to write operations; they can take as much as 5 ms. So if write commands are issued in a rapid sequence, a busy message, in the form of a NACK, is to be expected.

The *AddressSEE()* function is smart enough to perform a Stop sequence and repeat the whole Start and command part of the sequence until the Serial EEPROM is ready to accept the command (ACK is received). In [Figure 7.15](#) you can see a logic analyzer capture of the actual

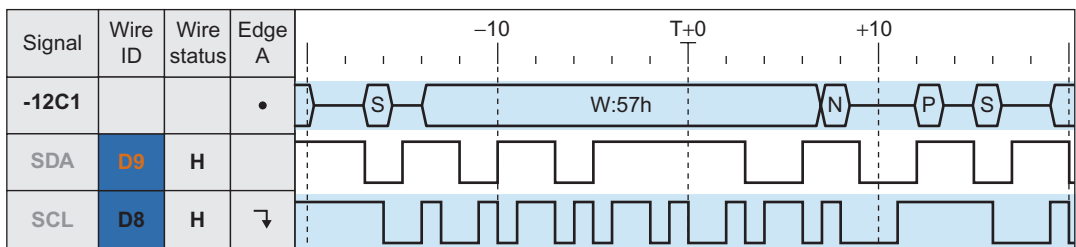


Figure 7.15: Logic analyzer capture of a serial EEPROM busy returning a NACK

event. The tool used (a low-cost 34-channel USB Intronix LogicPort) features a convenient interpreter that provides a more intuitive representation (top line) of the bus operation.

It is only at this point that the *addressSEE()* function sends a second byte containing the ByteAddress and returns.

The *writeSEE()* function can now use the *addressSEE()* function and then complete the sequence passing the two bytes of data (the 16-bit value) that will be written at the selected address and following location.

```
void iWriteSEE( long add, int v)
// SEE write command sequence
{
    int cmd;

    // 1. select address
    cmd = AddressSEE( add);

    // 2. stream data out
    MasterWriteI2C1( v&0xFF);
    IdleI2C1();

    MasterWriteI2C1( v>>8);
    IdleI2C1();

    // 3. terminate the command sequence
    StopI2C1();
    IdleI2C1();
} // iWriteSEE
```

Notice that the Serial EEPROM will begin the actual write operation only after the Stop sequence is completed by the master. Different Serial EEPROM models might accept a different number of bytes and require a specific address alignment to perform the page write. The 24LC16 type memory, which we assume will be used in this example, can in fact accept as many as 16 bytes. It can store them in an internal buffer and write them simultaneously (in a single write cycle) upon receiving the Stop sequence.

The iReadSEE() Function

If the simplest read sequence for a Serial EEPROM is the *current address read*, expressed in the previous posting in the compact notation:

```
S 0b1010001 (A) (Data) NP
```

extending it to read sequentially two bytes (to obtain a 16-bit value) is simple enough if after reading the first byte we respond to the Serial EEPROM with an ACK, inviting the device to continue streaming out the content of the next memory location:

```
S 0b1010001 (A) (DataLSB) [A] (DataMSB) NP
```

A more generic and useful read command, a *random read*, is obtained by combining the write command address selection with the current address read sequence:

```
S 0b01010000 (A)ByteAddress(A) P
S 0b01010001 (A) (DataLSB) [A] (Data MSB) NP
```

Since we have already created the *addressSEE()* function providing the first part, completing the *readSEE()* function is a trivial exercise now:

```
int iReadSEE( long add)
// random access read command sequence
{
    int cmd, r;

    // 1. select address
    cmd = AddressSEE( add);

    StopI2C1();
    IdleI2C1();

    // 2. read command
    StartI2C1(); IdleI2C1();
    MasterWriteI2C1( cmd+READ_CMD);
    IdleI2C1();

    // 3. stream data in (will continue until NACK is sent)
    r= MasterReadI2C1( );

    AckI2C1(); IdleI2C1();
    r|= (MasterReadI2C1())<<8;

    // 4. terminate read sequence (send NACK then STOP)
    NotAckI2C1(); IdleI2C1();
    StopI2C1(); IdleI2C1();

    return r;
} // iReadSEE
```

This completes the *SEE24.c* module, implementing the same routines we had previously defined in the *SEE.h* module when using the 25LCXXX type SPI memories.

Let's write now **SEE24test.c**, a short test program that we can use to check the newly created library module.

A simple *for* loop in the *main()* function of our project will test our ability to write (and read back) to every and each location of the 24LC16 device.

Since we have designed the routines so as to virtualize the device as a 16-bit integer storage service, we will be able to store up to 1024 integers using all the capacity (16Kbits) of the little 24LC16 Serial EEPROM. Notice that each integer address will need to be multiplied by two to obtain the byte address (unless we modify *AddressSEE()* to perform the operation automatically for us)...

```
/*
** SEE24 Library test
*/
#include <config.h>
#include <SEE.h>

int i, r;

main()
{
    // init the I2C peripheral
    InitSEE();

    // fill the memory and verify
    for( i=0; i<1024; i++)
    {
        iWriteSEE( i<<1, i);
        r = iReadSEE( i<<1);
        if (r!=i)
            break;
    }

    // main loop
    while( 1);
}
```

To test the library you will need to solder an eight-pin socket in the prototyping area of the Explorer16 board and connect two pull-up resistors to the 3.3 V rail. You will also have to connect the pins: *56-SDA1-RG3* and *57-SCL1-RG2* of the PIC24FJ128GA010, to the *SDA(5)* and *SCL(6)* pins of a 24LC16B Serial EEPROM.

To provide power to the device and to enable writing commands, connect the *WP(7)* and *Vcc* pins to the 3.3 V rail and the *Vss(4)* pin to *GND*.

After successfully assembling all the parts, you will be able to verify the correct operation of our little project: the loop will terminate with *i==1024* only after all memory locations have been written to and verified.

The Big I²C SEE Table

Not all I²C Serial EEPROMs are created equal. In fact there is a lot of variety out there: as the memory size increases the addressing schemes change, and the number and use of the device select pins change as well. The size increase reflects more or less closely the passing of time (years) and with it the evolution of the market of Serial EEPROMs within and beyond the confines of the original specifications of the bus.

To be able to adapt the code developed in the previous, you will need to pay close attention to the details. To simplify the task, I have tried to collect in one big table the key elements that differentiate each device ([Table 7.3](#)).

Table 7.3: I2C Serial EEPROM summary

Model	Size (bytes)	A0	A1	A2	WP	Address (bytes)	Page Size	Comments
24xx00	16	NC	NC	NC	No	1	NA	Available in SOT23-5 package
24xx01B	128	NC	NC	NC	Yes	1	8	Available in SOT23-5 package
24C01C	128	A0	A1	A2	No	1	16	Just like the -01B but with active device select lines
24xx014	128	A0	A1	A2	Yes	1	16	Just like the -01C but adds functional WP pin
24xx02B	256	NC	NC	NC	Yes	1	8	.
24C02C	256	A0	A1	A2	Yes	1	16	.
24xx04B	512	B0	NC	NC	Yes	1	16	B0 is the address 9th bit
24xx08B	1 k	B0	B1	NC	Yes	1	16	B0 and B1 are the address 9th and 10th bit
24xx16B	2 k	B0	B1	B2	Yes	1	16	B0/B1/B2 are the address 9-10-11th bit
24xx32A	4 k	A0	A1	A2	Yes	2	32	.
24xx64	8 k	A0	A1	A2	Yes	2	32	.
24xx65	8 k	A0	A1	A2	Yes	2	8	.
24xx128	16 k	A0	A1	A2	Yes	2	64	.
24xx256	32 k	A0	A1	A2	Yes	2	64	.
24xx512	64 k	A0	A1	A2	Yes	2	128	.
24xx515	64 k	A0	A1	B0	Yes	2	128	B0 replaces the 15th bit (MSB) of the address

Note

Table 7.3 is based on information from the original datasheets published by Microchip Technology Inc.

I used a bold font to highlight the main irregularities in the table. I also used the “xx” notation where you will find available “AA”, “LC” and sometimes “FC” options indicating respectively: low-voltage (1.8 V) devices, standard devices (2.5–5 V) and fast devices (1 MHz max. bus speed rated).

To try and make sense of the data, I can offer a simplified summary in a few bullets:

- To reduce the die area (and therefore the price), small devices (below 256 byte capacity) do not support the device address pins (only one device can be connected to the bus at all times). Although if proper addressing is required, a -C model offering such functionality is available.

- To keep the bus overhead low (using only one byte of address after the command byte), devices with capacity between 512 bytes and 2 Kbytes (–04 to –16) use three bits of the command byte (in violation of the standard) to carry from one to three bits of the address (marked as B0/B1 and B2). As a consequence, the device addressing is incomplete and only one device can be connected to the bus.
- Memory sizes at or above 4 Kbytes require a second address byte following the command byte.
- The write buffer (page) size grows somewhat linearly with the size of the array, but there are notable exceptions, watch out!

The SEE24.c module presented recently was designed for the 24xx16B type of Serial EEPROM, a device that you will find right in the middle of the table and, as you can see, perhaps not representing the most generic case. Depending on the memory size required by your application you will need to:

- Apply minor changes to the addressSEE() function (adding a second address byte and moving the msb address bits there)

or in case you are planning to use the smallest devices available (24xx00) that do not offer a write (page) buffer:

- Modify the writeSEE() function to separate the two data byte write commands in two independent complete write sequences.

As a final note, it is possible to write code for a microcontroller to detect automatically the type and size of the Serial EEPROM connected to the bus, but this is far from being a universal algorithm. It is possible, but only within the confines of a single manufacturer and possibly a subset of the entire product range: see AN690 for one of my early attempts...

Post-Flight Briefing

In this lesson we have seen briefly how to use the SPI peripheral module, in its simplest configuration, to gain access to a 25LC256 Serial EEPROM memory. Shortly after, we have repeated the experiment but with an I²C interface and a 24LC16B type of non-volatile memory device. The small interchangeable library modules developed in the examples will hopefully be useful to you to provide non-volatile *mass* storage to your future applications.

Notes for the C Experts

The C programmer used to developing code for large workstations and personal computers will be tempted to develop the library further to include the most flexible and comprehensive set of functions. My word of advice is to resist, hold your breath and count to ten, especially before you start adding any new parameter to the library functions. In the embedded-control

world passing more parameters means using up more stack space, spending more time copying data to and from the stack and in general producing a larger output code. Keep the libraries simple and therefore easy to test and maintain. This does not mean that proper object-oriented programming practices should not be followed. On the contrary, the example above can be considered already an example of object encapsulation, as all the details of the SPI interface and Serial EEPROM internal workings can be completely hidden from the user, who is provided with a simple interface to a generic storage device.

Notes for the Experts

In developing the SPI code examples above, we have ignored any access speed consideration and simply configured the SPI module for the slowest possible operation. The PIC24 SPI peripheral module operates off the peripheral clock system, which can be ticking as fast as 16 MHz in the current production models. Few peripherals can operate at such speeds at 3 V. Specifically, the 25LC256 series Serial EEPROMs operate with a maximum clock rate of 5 MHz when the power supply is in the 2.5 V to 4.5 V range. This means that the fastest SPI port configuration compatible with the memory device can be obtained with a primary prescaler configured for a 4:1 ratio and a secondary prescaler configured for 1:1 operation ($16\text{ MHz}/4 = 4\text{ MHz}$). A sequential read command could therefore provide a maximum throughput of 4 Mbits per second or 512 Kbytes per second. At such a rate the CPU would still be able to execute 32 instructions between each new byte of data received, not enough to perform complex calculations, but most probably sufficient for simple data transfer tasks.

Notes for the PIC Microcontroller Experts

In addition to the SPI options available on most PIC microcontrollers (offered by the SSP and MSSP modules), such as:

- Selectable clock polarity
- Selectable clock edge
- Master or slave mode operation

the PIC24 SPI interface module adds several new capabilities, including:

- A 16-bit transfer mode
- Data input sampling phase selection
- Framed transmission mode
- Frame synch pulse control (polarity and edge selectable)
- Enhanced mode (eight deep transmit and receive FIFOs)

In particular, the 16-bit transfer mode could be used during sequential read and/or page write operation to improve the efficiency and increase the amount of cycles available to the CPU between accesses to the SPI buffers (doubling it). But it is the enhanced mode, with the

eight-levels-deep FIFOs, that can truly free up a considerable amount of CPU time. Up to eight words of data (16 bytes) can be written or retrieved from the SPI buffers in short bursts, leaving much more time to the CPU to process the data in between the successive bursts.

Tips & Tricks

If you store important data in an external non-volatile memory, you might want to put some additional safety measures in place (both hardware and software). From a hardware perspective make sure that:

- Adequate power supply decoupling (a capacitor) is provided close to the memory device.
- A pull-up resistor (10kohm) is provided on the Chip Select line, to avoid floating during the microcontroller power-up and reset.
- An additional pull-down resistor (10kohm) can be provided on the SCK clock line to avoid clocking of the peripheral during boundary scan and other board testing procedures.
- Clean and fast power-up and down slope are provided to the microcontroller to guarantee reliable Power On Reset operation. If necessary, add an external voltage supervisor (see MCP809 devices for example).

A number of software methods can then be employed to prevent even the most remote possibility that a program bug or the proverbial cosmic ray might trigger the write routine. Here are some suggestions:

- Avoid reading and especially updating the NVM content right after power-up. Allow a few milliseconds for the power supply to stabilize (application dependent).
- Add a software write-enable flag, and demand that the calling application set the flag before calling the write routine, possibly after verifying some essential application-specific entry condition.
- Add a stack level counter; each function in the stack of calls implemented by the library should increment the counter upon entry and decrement it on exit. The write routine should refuse to perform if the counter is not at the expected level.
- Some users refuse to use the NVM memory locations corresponding to the first address (0x0000) and/or the last address (0xffff), believing they could be statistically more likely to be the subject to corruption.
- More seriously, store two copies of each essential piece of data, performing two separate calls to the write routine. If each copy contains even a simple checksum, it will be easy, when reading it back, to discard the corrupted one and recover.

Exercises

- Several functions in the library are performing locking loops that could reduce the overall application performance. By using the SPI port interrupts implement a non-blocking version of the library.
- Enable the new SPI 16-bit mode to accelerate basic read and write operations.
- Develop (circular) buffered versions of the read and write functions.

Books

- Eady, F., 2004. *Networking and Internetworking with Microcontrollers*, Newnes, Burlington, MA.
An entertaining introduction to serial communication in embedded control.
- Buck, R., 1997. *Flight of Passage, A Memoir*, Hyperion, New York, NY.
A grand adventure, two teenagers fly coast to coast in an aviation rite of passage.

Links

- http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en010003
Use the link above or search on Microchip web site for a free tool called “Total Endurance Software”. It will help you estimate the endurance to expect from a given NVM device in your actual application conditions. It will give you an indication of the total number of erase/write cycles or the number of expected years of your application life before a certain target failure rate is reached.

Asynchronous Communication

If you have any experience with radio communication, whether it be with a walkie-talkie or a proper CB radio, you know how different it is from talking on a cell phone. For one, it's a half-duplex system, meaning you cannot talk while somebody else is talking. You have to patiently listen, wait for your turn and then speak up, trying to be as concise as possible to give others the possibility of joining the conversation too. A simple verbal handshake system is used to prevent conflicts and misunderstanding.

This is exactly how it works in aviation. There is a precise protocol, a set of rules that dictates who should talk at any given point in time, what they should say and how. There are specific roles, such as the air traffic controllers, the pilots, the flight stations and the towers; and they all share the media in a coordinated and efficient way.

This works well as an introduction to the many asynchronous serial protocols. Some are full-duplex, other are just half-duplex, some are multipoint, others are point-to-point, but they all require coordination and adherence to basic rules (standards) that make communication possible and allow for efficient use of the media.

Flight Plan

In this lesson we will review the PIC24 asynchronous serial communication interface modules – UART1 and UART2. We will develop a basic console library that will be handy in future projects for interface and debugging purposes.

Preflight Checklist

In addition to the usual software tools, including the MPLAB[®] X IDE, the MPLAB C compiler for the PIC24 and an in circuit debugger and programmer such as the MPLAB ICD3, this lesson will require the use of the Explorer16 demonstration board, or a similar board with an RS232 port (transceiver + connector), and a PC with an RS232 serial port (or a serial to USB adapter). You will also need a terminal emulation program; if you are using Microsoft Windows XP operating system, the HyperTerminal application will suffice (**Start>Programs>Accessories>Communication>HyperTerminal**). Windows 7 users will have to download a third-party terminal application (such as TeraTerm).

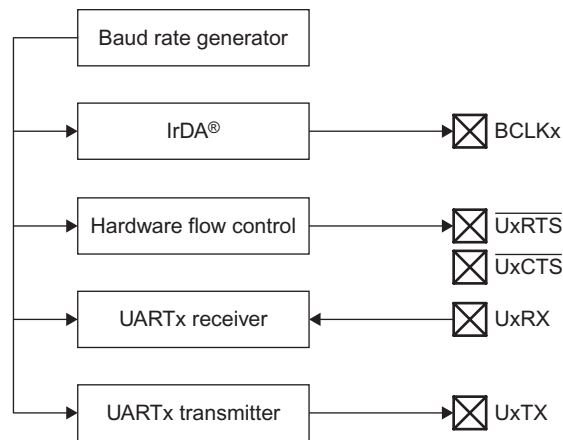


Figure 8.1: Simplified UART modules block diagram

The Flight

The UART interface is perhaps the oldest interface used in the embedded-control world. Some of its features were dictated by the need for compatibility with the first mechanical teletypewriters; this means that at least some of its technology has century-old roots.

On the other hand, nowadays finding an asynchronous serial port on a new computer (and especially on a laptop) is becoming a challenge. The serial port has been declared a *legacy interface* and, for several years now, strong pressure has been placed on computer manufacturers to replace it with the USB interface. Despite the decline in the popularity of the serial port, and the clearly superior performance and characteristics of the USB interface, in the world of embedded applications asynchronous serial interfaces are strenuously resisting this trend because of their great simplicity and extremely low cost of implementation.

Four main classes of asynchronous serial applications are still being used:

- RS232 point-to-point connection: often simply referred to as “the serial port”, is used by terminals, modems and personal computers, using +12V/−12V transceivers.
- RS485 (EIA-485) multi-point serial connection: is used in industrial applications. It uses a 9-bit word for addressing individual instruments and special half-duplex transceivers.
- LIN bus: a low-cost, low-voltage bus designed for non-critical automotive applications. It requires a UART capable of baud rate auto-detection.
- Infrared wireless communication (IrDA): requires a 38–40 kHz signal modulation and an inexpensive optical transceiver.

The PIC24’s UART modules can support all four major application classes and packs in a few more interesting features too.

To demonstrate the basic functionality of a UART peripheral (Figure 8.1), we will use the Explorer16 demo board where the UART2 module is connected to an RS232 transceiver

device and to a standard DB9 female connector. This can be connected to any PC serial port or, in absence of the “legacy interface” as mentioned above, to an RS232 to USB converter device. In both cases the Windows HyperTerminal program will be able to exchange data with the Explorer16 board with a basic configuration setting.

The first step is the definition of the transmission parameters; the options include:

- Baud rate
- Number of data bits
- Parity bit, if present
- Number of stop bits
- Handshake protocol

For our demo we will choose the fast and convenient configuration: “115200, 8, N, 1, CTS/RTS”, that is:

- 115,200 baud
- 8 data bit
- No parity
- 1 stop bit
- Hardware handshake using the CTS and RTS lines

UART Configuration

Use the *New Project Setup* checklist to create a new project called **8-Serial** and a new source file similarly called **serial.c**. We will start by adding a few useful I/O definitions to help us control the hardware handshake lines:

```
/*
** Serial
** UART2 RS232 asynchronous communication demonstration
*/
#include <config.h>

// I/O definitions for the Explorer16
#define CTS    _RF12        // Clear To Send, in, handshake
#define RTS    _RF13        // Request To Send, out, handshake
#define TRTS    TRISFbits.TRISF13 // tris control for RTS pin
```

The hardware handshake is especially necessary when communicating with a Windows terminal application, since Windows is a multitasking operating system and its applications can sometimes experience long delays that would otherwise cause significant loss of data. We will use one I/O pin as an input (*RF12* on the Explorer16 board) to sense when the terminal is ready to receive a new character (Clear To Send), and one I/O pin as an output (*RF13* on the Explorer16 board) to advise the terminal when our application is ready to receive a character (Request To Send).

Note

Watch out when shopping for serial cables! There are inexpensive three-wire cables that do not support hardware handshaking as they only contain RX, TX and GND.

To set the baud-rate, we get to play with the Baud-Rate Generator (*BREG2*), a 16-bit counter that feeds on the peripheral clock circuit. From the device datasheet we learn that in the normal mode of operation (*BREGH=0*) it operates off a 1:16 divider versus a high-speed mode (*BREGH=1*) where its clock operates off a 1:4 divider. A simple formula, published on the datasheet, allows us to calculate the ideal setting for our configuration.

```
BREG2 = (Fosc / 8 / baudrate) - 1; for BREGH=1
```

In our case, this translates to the following expression:

```
BREG2 = (Fosc / 8 / 115,200) - 1 = 33.7 where Fosc = 32MHz.
```

To decide how to best round out the result, we need a 16-bit integer after all; we will use the reverse formula to calculate the actual baud-rate and determine the percentage error:

```
Error = ((Fosc/ 8 / (BREG2 + 1)) - baudrate) / baudrate %
```

Rounding up to a value of 34 we obtain an actual baud-rate of 114,285 baud with an error of just 0.7%, well within acceptable tolerance. Rounding down to a value of 33 we obtain 117,647 baud or a 2.1% error, outside the acceptable tolerance range (+/-2%) for a standard RS232 port.

We therefore define the constant *BRATE*:

```
#define BRATE 34 // 115200 Baud (BREGH=1)
```

Two more constants will help us define the initialization values for the UART2 main control registers called *U2MODE* and *U2STA*.

The initialization value for *U2MODE* (Figure 8.2) will include the *BREGH* bit, the number of stop bits and the parity bit settings.

```
#define U_ENABLE 0x8008 // enable the UART peripheral
```

Upper byte:							
R/W-0	U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0 ⁽¹⁾	R/W-0 ⁽¹⁾
UARTEN	—	USIDL	IREN	RTSMD	—	UEN1	UEN0
bit 15				bit 8			

Lower byte:							
R/W-0 HC	R/W-0	R/W-0 HC	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	RXINV	BRGH	PDSEL1	PDSEL0	STSEL
bit 7				bit 0			

Figure 8.2: The *UxMODE* control registers

Upper byte:							
R/W-0	R/W-0	R/W-0	U-0	R/W-0 HC	R/W-0	R-0	R-1
UTXISEL1	UTXINV ⁽¹⁾	UTXISEL0	—	UTXBRK	UTXEN	UTXBF	TRMT
bit 15				bit 8			

Lower byte:							
R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/C-0	R-0
URXISEL1	URXISEL0	ADDEN	RIDLE	PERR	FERR	OERR	URXDA
bit 7				bit 0			

Figure 8.3: The UxSTA control registers

The initialization for *U2STA* (Figure 8.3) will enable the transmitter and clear the error flags.

```
#define U_TX      0x0400 // enable transmission
```

We will create a new function, by using the constants defined above, to initialize the control registers of the UART2, the baud-rate generator and the I/O pins used for the handshake:

```
void InitU2( void)
{
    U2BRG   = BRATE;
    U2MODE  = U_ENABLE;
    U2STA   = U_TX;
    RTS     = 1;           // set RTS default status
    TRTS    = 0;           // make RTS output
} // InitU2
```

Sending and Receiving Data

Sending a character to the serial port is a three-step procedure:

1. Make sure that the terminal (PC running Windows HyperTerminal) is ready. Check the Clear to Send (CTS) line. CTS is an active low signal; that is, while it is high, we had better wait patiently.
2. Make sure that the UART is not still busy sending some previous data. PIC24 UARTs have a four-level-deep FIFO buffer, so all we need to do is wait until at least the top level frees up, or, in other words, we need to check for the transmit buffer full flag UTXBF to be clear.
3. Finally, transfer the new character to the UART transmit buffer (FIFO).

All of the above can be nicely packaged in one short function:

```
int putU2( int c)
{
    while ( CTS);           // wait for !CTS, clear to send
    while ( U2STAbits.UTXBF); // wait while Tx buffer full
    U2TXREG = c;
    return c;
} // putU2
```


To receive a character from the serial port, we follow a very similar sequence:

1. Alert the terminal that we are ready to receive by asserting the RTS signal (active low).
2. Patiently wait for a character to arrive in the receive buffer, checking the URXDA flag.
3. Fetch the character from the receive buffer (FIFO).

Again, all of the above steps can be nicely packaged in one last function:

```
char getU2( void)
{
    RTS = 0;                // assert Request To Send !RTS
    while ( !U2STAbits.URXDA); // wait
    RTS = 1;
    return U2RXREG;          // read from the receive buffer
} // getU2
```

Testing the Serial Communication Routines

To test our serial port control routines, we can now write a small program that will initialize the serial port, send a prompt, and let us type on the terminal keyboard while echoing each character back to the terminal screen:

```
main()
{
    char c;
    // 1. init the UART2 serial port
    InitU2();

    // 2. prompt
    putU2( '>');

    // 3. main loop
    while ( 1)
    {
        // 3.1 wait for a character
        c = getU2();
        // 3.2 echo the character
        putU2( c);
    } // main loop
} // main
```

- Connect the serial cable to the PC (directly or via a Serial to USB converter) and configure HyperTerminal for the same communication parameters: 115200, 8, N, 1, and for flow control select Hardware (RTS/CTS) on the available COM port.
- Click on HyperTerminal **Connect** button to start the terminal emulation.
- Follow the standard checklist to connect the In Circuit Debugger.
- Build the project for debugging by selecting **Debug>Project**. This will automatically program the Explorer16 demo board and execute the demonstration program.

Note

I recommend, for now, you do not attempt to single step, use breakpoints, or Run To Cursor when using the UART! See the Tips & Tricks section at the end of the chapter for a detailed explanation.

Note also that, if HyperTerminal is already set to provide an echo for each character sent, you will see double ... literally! To disable this functionality, first hit the **Disconnect** button on HyperTerminal. Then select **File>Properties** and in the Properties dialog box select the **Settings** pane. This will be a good opportunity to set a couple more options that will come handy in the rest of the lesson.

Select the **VT100 Terminal Emulation** mode so that a number of commands (activated by special *escape* strings) will become available and will give us more control of the cursor position on the terminal screen (Figure 8.4).

Select **ASCII Setup** to complete the configuration (Figure 8.5). In particular, make sure that the *Echo typed characters locally* function is **NOT** checked (this will immediately improve your ... ddoouubbllee vviissiiioonn!!).

Also check the *Append line feeds to incoming line ends* option, this will make sure that every time an ASCII *carriage return* ('r') character is received an additional *line feed* ('\n') character is inserted automatically.

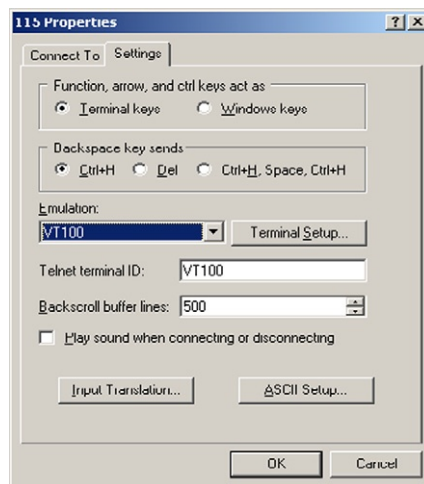


Figure 8.4: HyperTerminal Properties dialog box, Settings pane

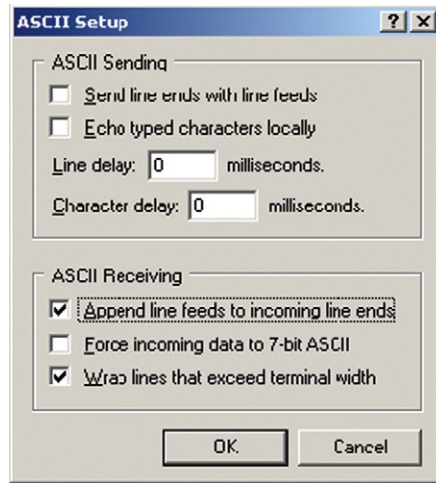


Figure 8.5: ASCII Setup dialog box

Building a Simple Console Library

To transform our demo project in a proper terminal console library that could become handy in future projects, we need only a couple more functions that will complete the puzzle: a function to print an entire (zero-terminated) string and a function to input a full text line. Printing a string is, as you can imagine, the simple part:

```
void putsU2( char *s)
{
    while( *s)           // loop until *s == '\0' the end of the string
        putU2( *s++);    // send the character and point to the next one
    putU2( '\r');        // terminate with a cr / line feed
    putU2( '\n');
} // putsU2
```

It is just a loop that keeps calling the *putU2()* function to send, one after the other, each character in the string to the serial port.

Reading a text string from the terminal (console) into a string buffer can be equally simple, but we have to make sure that the size of the buffer is not exceeded (should the user type a really long string) and we have to convert the carriage return character at the end of the line in a proper ‘\0’ character for the string termination.

```
char *getsnU2( char *s, int len)
{
    char *p = s;          // copy the buffer pointer
    do{
        *s = getU2();      // wait for a new character
```

```

        if ( *s=='\r')        // end of line, end loop
            break;
        s++;                  // increment buffer pointer
        len--;
    } while ( len>1 );        // until buffer full
    *s = '\0';                // null terminate the string
    return p;                 // return buffer pointer
} // getsnU2

```

In practice, the function, as presented, would prove very hard to use. There is no echo of what is being typed and the user has no room for errors. Make only the smallest typo and the entire line must be retyped. If you are like me, you do make a lot of typos ... all of the time, and the most battered key on your keyboard is the backspace key. A better version of the *getsnU2()* function must include character echo and at least provisions for the backspace key to perform basic editing. It takes really only a couple more lines of code. The echo is quickly added after each character is received. The backspace character (identified by the ASCII code *0x8*) is decoded to move the buffer pointer one character backward (as long as we are not at the beginning of the line already). We must also output a specific sequence of characters to visually remove the previous character from the terminal screen.

```

char *getsnU2( char *s, int len)
{
    char *p = s;              // copy the buffer pointer
    do{
        *s = getU2();          // wait for a new character
        putU2( *s);            // echo character

        if (( *s==BACKSPACE)&&( s>p))
        {
            putU2( ' ');       // overwrite the last character
            putU2( BACKSPACE);
            len++;
            s--;                // back the pointer
            continue;
        }

        if ( *s=='\n')         // line feed, ignore it
            continue;
        if ( *s=='\r')         // end of line, end loop
            break;
        s++;                   // increment buffer pointer
        len--;
    } while ( len>1 );        // until buffer full
    *s = '\0';                // null terminate the string
    return p;                 // return buffer pointer
} // getsnU2

```

Put all the functions in a separate file that we will call **CONU2.c**. Then create a small header file **CONU2.h**, to decide which functions (prototypes) and which constants to publish and make visible to the outside world.

```
/*
** CONU2.h
** console I/O library for Explorer16 board
*/

// I/O definitions for the Explorer16
#define CTS      _RF12    // Clear To Send, in, HW handshake
#define RTS      _RF13    // Request To Send, out, HW handshake
#define BACKSPACE 0x08    // ASCII backspace character code

// init the serial port (UART2, 115200, 8, N, 1, CTS/RTS)
void InitU2( void);

// send a character to the serial port
int putU2( int c);

// wait for a new character to arrive to the serial port
char getU2( void);

// send a null terminated string to the serial port
void putsU2( char *s);

// receive a null terminated string in a buffer of len char
char * getsnU2( char *s, int n);
```

Since we have enabled the VT100 terminal emulation mode (see HyperTerminal settings above), we have now a few commands available to better control the terminal screen and cursor position, such as:

- `Clrscr()`, to clear the terminal screen.
- `Home()`, to move the cursor to the home position in the upper left corner of the screen.

These commands are performed by sending so-called *escape sequences* (defined in the ECMA-48 standard (ISO/IEC 6429 and ANSI X3.64), also referred to as *ANSI escape codes*. They all start with the characters *ESC* (ASCII *0x1b*) and the character '[' (left squared bracket).

```
// useful macros
#define Clrscr() putsU2( "\x1b[2J")           // Clear the screen
#define Home()  putsU2( "\x1b[1;1H")         // return cursor home
#define pcr()   putU2( '\r' ); putU2( '\n')   // carriage return
```

As we did in previous chapters, we can now copy the Console module into the library:

- Copy the **CONU2.c** file into the **/lib** subdirectory.
- Copy the **CONU2.h** file into the **/include** subdirectory.

From now on, using a terminal connected via the serial port will be as simple as including the *CONU2.h* file at the top of your application and adding the *CONU2.c* module to the list of source files in your project.

Testing a VT100 Terminal

In order to test the console library we can now write up a small program that will:

1. Initialize the serial port
2. Clear the terminal screen

3. Send a welcome message/banner
4. Send a prompt character
5. Read a full line of text
6. Print the text on a new line

Save the following code in a new file that we will call **CONU2test.c**.

```
/*
** CONU2Test.c
** UART2 RS232 asynchronous communication demonstration
*/
#include <stdio.h>
#include <config.h>
#include <CONU2.h>

#define BUF_SIZE 128

main()
{
    char s[ BUF_SIZE];

    // 1. init the console serial port
    InitU2();

    // 2. text prompt
    Clrscr();
    Home();
    putsU2( "Learn to fly with the PIC24!\r\n");
    sprintf( s, "Learn to fly the PIC24! %d\r\n", 17);
    putsU2( s);

    // 3. main loop
    while ( 1)
    {
        // 3.1 read a full line of text
        getsnU2( s, sizeof(s));
        // 3.2 send a string to the serial port
        putsU2( s);
        // 3.3 send a carriage return
        pcr();
    } // main loop
} // main
```

Using the *New Project* checklist, create a new project called **8-Console**. Add all three source files: **CONU2.h**, **CONU2.c** and **CONU2test.c** to it.

Use the **Build for Debugging** checklist to connect the In Circuit Debugger/Programmer, build the project and program the Explorer16 board.

Test the editing capabilities of the new console library you have just completed by typing and *editing* a few lines of text using the HyperTerminal program.

Using the Serial Port as a Debugging Tool

Once you have a small library of functions to send and receive data to a console through the serial port, you have a new powerful debugging tool available. You can strategically position calls to print functions to present the content of critical variables and other diagnostic information on the terminal. You can easily format the output to be in the most convenient format for you to read. You can add input functions to set parameters that can help better test your code or you can use the input function to simply pause the execution and give you time to read the diagnostic output when required. This is one of the oldest debugging tools, effectively used since the first computer was invented.

The Matrix

To finish this lesson on a more entertaining note, let's develop a new demo project that we will call **8-Matrix**. The intent is that of testing the speed of the serial port and the PC terminal emulation by sending large quantities of text to the terminal and clocking its performance. The only problem is that we don't (yet) have access to a large storage device from where to extract some meaningful content to send to the terminal. So the next best option is that of *generating* some content using a pseudo-random number generator. The *stdlib.h* library offers a convenient *rand()* function that returns a positive integer between 0 and *MAX RAND* (a constant defined in the *limits.h* file that in the MPLAB C compiler implementation can be verified to be equal to 32,767).

Using the *remainder* (modulo) operator (%) we can reduce its output to any smaller integer range and produce only a subset of printable character values from the ASCII set. The following statement for example will produce only characters in the range from 33 to 127.

```
putU2(33 + (rand()%94));
```

To generate a more appealing and entertaining output, especially if you happened to watch the movie *The Matrix*, we will present the (random) content by columns instead of rows. We will use the pseudo-random number generator to change the content and the *length* of each column as we continuously refresh the screen.

```
/*
** The Matrix
**
*/
#include <config.h>
#include <CONU2.h>
#include <stdlib.h>

#define COL      40
#define ROW      23
#define DELAY    3000
```

```

main()
{
    int v[40]; // vector containing length of each string
    int i,j,k;

    // 1. initializations
    T1CON = 0x8030; // TMR1 on, prescale 256, Tcy/2
    InitU2(); // initialize the console
    Clrscr(); // clear the terminal (VT100 emulation)
    getU2(); // wait for character to randomize sequence
    srand( TMR1);

    // 2. init each column length
    for( j =0; j<COL; j++)
        v[j] = rand()%ROW;

    // 3. main loop
    while( 1)
    {
        Home();

        // 3.1 refresh the screen with random columns
        for( i=0; i<ROW; i++)
        {
            // refresh one row at a time
            for( j=0; j<COL; j++)
            {
                // print random characters for each column length
                if ( i < v[j])
                    putU2( 33 + (rand()%94));
                else
                    putU2(' ');
                putU2( ' ');
            } // for j
            pcr();
        } // for i

        // 3.2 randomly increase or reduce each column length
        for( j=0; j<COL; j++)
        {
            switch ( rand()%3)
            {
                case 0: // increase length
                    v[j]++;
                    if (v[j]>ROW)
                        v[j]=ROW;
                    break;

                case 1: // decrease length
                    v[j]--;
                    if (v[j]<1)
                        v[j]=1;
                    break;
            }
        }
    }
}

```



```
        default:// unchanged
            break;
    } // switch
} // for
} // main loop
} // main
```

Forget the performance, watching this code run is fun. In fact you will have to add a small delay loop (inside the *for* loop in 3.1) to make it more pleasant on the eye or it will run too fast:

```
// 3.1.1 delay to slow down the screen update
TMR1 =0;
while( TMR1<DELAY);
```

Note

Remember to take the blue pill the next time!

Post-Flight Briefing

In this lesson we have developed a small console I/O library while reviewing the basic functionality of the UART module for operation as an RS232 serial port. We connected the Explorer16 board to a VT100 (emulated) terminal (Windows HyperTerminal). We will take advantage of this library in the next few lessons to provide us with a new debugging tool and possibly as a user interface for more advanced flights/projects.

Notes for the C Experts

I am sure at this point you are wondering about the possibility of using the more advanced library functions defined in the *stdio.h* library (such as *printf*) to direct the output to the UART2 peripheral. In fact this is possible simply by replacing one of the essential library functions: *write.c*.

```
/*
** write.c
** replaces stdio lib write function with UART2
*/
#include <p24fxxx.h>
#include <stdio.h>
#include <CONU2.h>

int write(int handle, void *buffer, unsigned int len)
{
    int i;
    switch (handle)
    {
```

```

    case 0:
    case 1:
    case 2:
        i = len;
        while( i-- )
            putU2( *(char*)buffer++ );
        break;
    default:
        break;
    }
    return(len);
}

```

Save this code in a file called **write.c** in your project directory and add it to the list of source files for the project.

From this moment on, the linker will perform the connection and any call to one of the *stdio.h* library functions producing output on one of the standard streams (*stdin*, *stdout*, *stderr*) will be re-directed to the UART2.

Notice that you will be still responsible for the proper UART initialization and the *CONU2.c* file will have to be included in the project sources as well.

Notes for the PIC[®] Microcontroller Experts

Sooner or later, every embedded-control designer will have to come to terms with the USB bus. Even if, for now, a small *dongle* (converting the serial port to a USB port) is a reasonable solution, eventually you are going to find opportunities and designs that will actually benefit from the superior performance and compatibility of the USB bus. Several PIC24 microcontroller models (the PIC24GB1 series for example) already incorporate a USB Serial Interface Engine (SIE) as a standard communication interface and since they are mostly pin-to-pin compatible with the PIC24GA0 series used in this book, you can replace the PIM on the Explorer16 demonstration board (or replace the processor on your development board of choice) and run the same examples unmodified. In the notes at the end of chapter 3, you will find an extension of the *config.h* file that will take care of reconciling the differences found in the device configuration bits.

Unfortunately, the USB interface is not something that I could dare introducing in a couple of paragraphs in this or later chapters. To give you an idea of the complexity of the subject, Jan Axelson, in her seminal book *USB Complete*, spends 120 pages just introducing the fundamental concepts of the interface.

Microchip offers the free USB software Framework (part of the huge *Microchip Application Library, or MAL*) with drivers and ready-to-use solutions for the most common classes of applications. But even with all this help, it would be a grave injustice to my readers if I

attempted to cover this complex subject with a single application example. My most basic advice for those of you who are in a hurry and cannot spend the time to read Jan's excellent book is to look straight at one class of USB applications known as the Communication Device Class (or CDC). This makes the USB connection look completely transparent to the PC application by creating virtual serial ports so that even HyperTerminal cannot tell the difference. Most importantly, you will not need to write and/or install any special Windows drivers. When writing your application in C, you won't even notice the difference, if not for the absence of a need to specify any communication parameter other than the virtual com port number. In USB (CDC) there is no baud-rate to set, no parity to calculate, while the communication speed is so much higher...

Tips & Tricks

As we mentioned during one of the early exercises presented in this lesson, single stepping through a routine that enables and uses the UART to transmit and receive data from the HyperTerminal program is a bad idea. You will be frustrated seeing the HyperTerminal program misbehave and/or simply lock up and ignore any data sent to it without any apparent reason. In order to understand the problems you need to know more about how the MPLAB ICD3 in circuit debugger operates. After executing each instruction, when in single-step mode or upon encountering a breakpoint, the ICD3 debugger not only stops the CPU execution, but also *freezes* all the peripherals. It freezes them as in dead-cold-ice all of a sudden; not a single clock pulse is transmitted through their digital veins. When this happens to a UART peripheral that is busy in the middle of a transmission, the output serial line (TX) is also frozen in the current state. If a bit was being shifted out in that precise instant, and specifically if it was a 1, the TX line will be held in the *break* state (low) indeterminately.

The HyperTerminal program, on the other side, would sense this permanent *break* condition and interpret it as a *line error*. It will assume the connection is lost and it will disconnect. Since HyperTerminal is a pretty basic program, it will not bother letting you know what is going on... it will not send a beep, not an error message, nothing, it will just lock up!

If you are aware of the potential problem, this is not a big deal. When you restart your program with the ICD3, you will have just to remember to hit the HyperTerminal **Disconnect** button first and then the **Connect** button again. All operations will resume normally.

Exercises

- Write a console library with buffered I/O (using interrupts) to minimize the impact on program execution (and debugging).

Books

- Axelson, J., 2007. Serial Port Complete: Com Ports, USB virtual Com ports and ports for Embedded Systems, 2nd ed., Lakeview Research, Madison, WI.

- Axelson, J., 1999, *USB Complete*, 3rd ed, Lakeview Research, Madison, WI.
Jan's book has reached the 3rd edition already. She has added more material at every step and still managed to keep things very simple.
- Eady, F., 2005. *Implementing 802.11 with Microcontrollers: Wireless Networking for Embedded Systems Designers*, Newnes, Burlington, MA.
Fred brings his humor and experience in embedded programming to make even wireless networking easy.

Links

- http://en.wikipedia.org/wiki/ANSI_escape_code
This is a link to the complete table of ANSI escape codes as implemented by the VT100 HyperTerminal emulation.
- <http://www.microchip.com/mal>
This is a link to the Microchip Applications Library (MAL) containing the complete USB Framework for the PIC18, PIC24 and PIC32. This is a treasure trove of examples for all device and host applications you might want to implement with the PIC24FJxxGB1xx series microcontrollers.

Glass = Bliss

In the old days big round instruments that looked like steam gauges populated the cockpit of every airplane – from the smallest single-engine Cessna to the ultrasonic Concord. Being so ubiquitous, the six principal instruments, always placed in the same order, had gained the affectionate nickname of the six-pack. But the next time you get on a commercial plane, peek into the cabin if you can. Sure, there are still plenty knobs and switches, but right in front of the pilots you will notice there has been a big change. There is a large and flat piece of glass (or two). And “glass” is what the pilots call this revolution, although there is much more silicon behind it than most of them would suspect. It is the digital revolution of the cockpit, and it has happened in only the last few years.

Numerous powerful microprocessors work hard behind that glass to cram as much information as possible in a very simple, intuitive and possibly pleasing interface. Global Positioning System (GPS) technology has been the driving force behind this innovation, and every airplane manufacturer today offers several advanced glass cockpit options for new models. Some are even speculating that the more recent increase in sales of new airplanes, and the stimulus to the entire industry that followed, might have to be attributed to the excitement generated by the new “glass cockpit”.

Unfortunately these are not exactly the type of airplanes that you, as a student pilot, might be flying for the first few lessons. It might take a little while for modern, new airplanes to hit the school fleets, but it is just a matter of time now: glass bliss is on the horizon.

The embedded world also makes copious use of glass with LCD displays. Let’s explore the basics of LCD interfaces...

Flight Plan

In this lesson, we will learn how to interface with a small and inexpensive LCD display module. This project will be a good excuse for us to learn about and use the Parallel Master Port (PMP) – a flexible parallel interface available on the PIC24 microcontrollers.

Preflight Checklist

In addition to the usual software tools, including the MPLAB[®] X IDE and the MPLAB C compiler for the PIC24, this lesson will require only the use of the Explorer16 demonstration

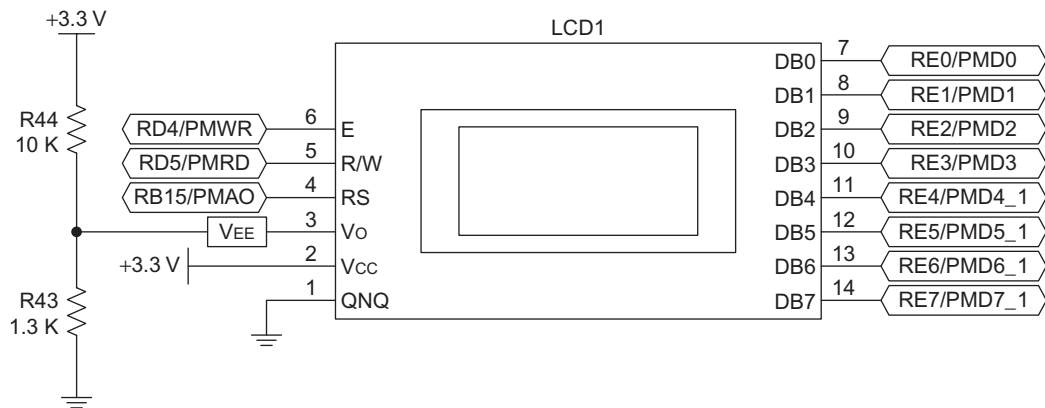


Figure 9.1: Default alphanumeric LCD module connections

board or other board offering an alphanumeric LCD display and the MPLAB ICD3 or equivalent In Circuit Debugger.

The Flight

The Explorer16 board can accommodate three different types of dot-matrix, alphanumeric LCD display modules and one type of graphic LCD display module. By default, it comes with a simple “2-row by 16-character” display, a 3 V alphanumeric LCD module (Tianma TM162JCAWG1) compatible with the industry-standard HD44780 controllers. These LCD modules are complete display systems composed of the LCD glass, column and row multiplexing drivers, power supply circuitry and an intelligent controller, all assembled together into the so-called Chip On Glass (COG) technology. Thanks to this high level of integration, the circuitry required to control the dot-matrix display is greatly simplified. Instead of the hundreds of pins required by the column and row drivers to directly control each pixel, we can interface to the module with a simple 8-bit parallel bus using just 11 I/Os (Figure 9.1).

On alphanumeric modules, we can directly place ASCII character codes into the LCD module controller RAM buffer (DDRAM). The output image is produced by an integrated character generator (a table) using a five by seven grid of pixels to represent each character. The table typically contains an extended ASCII character set, in the sense that it has been somewhat merged with a small subset of Japanese Kanji characters as well some symbols of common use (Figure 9.2). While the character generator table is mostly implemented in the display controller ROM, various display models offer the possibility to extend the character set by modifying/creating new user-defined characters (up to eight on some models) by accessing a second small internal RAM buffer (CGRAM).

Char. code		0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0000	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	0001	0	0	0	1	1	1	1	0	1	1	1	1	1	1
	0010	0	1	1	0	0	1	1	1	1	0	0	1	1	1
	0011	0	0	1	0	1	0	1	0	1	0	1	0	1	1
xxxx0000															
xxxx0001		!	1	A	Q	a	q	.	7	f	4	s	q		
xxxx0010		"	2	B	R	b	r	"	7	z	x	B			
xxxx0011		#	3	C	S	c	s	"	u	t	e	s	w		
xxxx0100		\$	4	O	T	d	t	.	E	t	P	u	Q		
xxxx0101		%	5	E	U	e	u	.	a	a	1	s	U		
xxxx0110		&	6	F	U	f	u	"	h	a	c	o	p	z	
xxxx0111		'	7	G	W	w	w	"	f	z	u	q	π		
xxxx1000		(8	H	X	h	x	"	i	o	z	u	j	z	
xxxx1001)	9	I	Y	i	y	"	t	j	u	"	q		
xxxx1010		*	:	J	Z	j	z	"	E	o	n	v	1	f	
xxxx1011		+	;	K	Z	k	z	"	a	a	E	o	*	a	
xxxx1100		,	<	L	#	l	#	"	i	i	z	z	z	a	
xxxx1101		-	=	M	J	m	j	"	u	z	z	z	z	z	
xxxx1110		.	>	N	^	n	^	"	a	a	a	a	a	a	
xxxx1111		/	?	0	_	o	_	"	u	u	u	u	u	u	

Figure 9.2: Character Generator table used by HD44780-compatible controllers

HD44780 Controller Compatibility

As mentioned above, the 2×16 LCD module used in the Explorer16 board is one among a vast selection of LCD display modules available on the market, in configurations ranging from one to four lines of 8, 16, 20, 32 and up to 40 characters each, that are compatible with the original HD44780 chip-set, today considered an industry standard.

The HD44780 compatibility means that the integrated controller contains just two registers, separately addressable: one for ASCII data and one for commands. The following standard set of commands (Table 9.1) can be used to set up and control the display as shown in Table 9.2.

Thanks to this commonality, any code we will develop to drive the LCD on the Explorer16 board will be immediately available for use with any of the other HD44780-compatible alphanumeric LCD display modules.

The Parallel Master Port

The simplicity of the 8-bit bus shared by all these display modules is remarkable. Beside the eight bi-directional data lines, there is:

- An Enable strobe line (E)
- A Read/Write selection line (R/W)
- An address line (RS) for the register selection

This gives us a total of just 11 pins.

Table 9.1: The HD44780 instruction set

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Clear display	0	0	0	0	0	0	0	0	0	1	Clears display and returns cursor to the home position (address 0)	1.64 ms
Cursor home	0	0	0	0	0	0	0	0	1	*	Returns cursor to home position (address 0). Also returns display being shifted to the original position. DDRAM contents remain unchanged	1.64 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction (I/D), specifies to shift the display (S). These operations are performed during data read/write	40 μ s
Display On/Off control	0	0	0	0	0	0	1	D	C	B	Sets On/Off of all display (D), cursor On/Off (C) and blink of cursor position character (B)	40 μ s
Cursor/display shift	0	0	0	0	0	1	S/C	R/L	*	*	Sets cursor-move or display-shift (S/C), shift direction (R/L). DDRAM contents remain unchanged	40 μ s
Function set	0	0	0	0	1	DL	N	F	*	*	Sets interface data length (DL), number of display line (N) and character font (F)	40 μ s
Set CGRAM address	0	0	0	1	CGRAM address						Sets the CGRAM address. CGRAM data is sent and received after this setting	40 μ s
Set DDRAM address	0	0	1		DDRAM address						Sets the DDRAM address. DDRAM data is sent and received after this setting	40 μ s

(Continued)

Table 9.1: The HD44780 instruction set (Continued)

Instruction	Code										Description	Execution time
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Read busy-flag and address counter	0	1	BF		CGRAM / DDRAM address						Reads busy flag (BF) indicating internal operation is being performed and reads CGRAM or DDRAM address counter contents (depending on previous instruction)	0 μ s
Write to CGRAM or DDRAM	1	0	write data								Writes data to CGRAM or DDRAM	40 μ s
Read from CGRAM or DDRAM	1	1	read data								Reads data from CGRAM or DDRAM	40 μ s

Note

For further I/O saving, the DL control bit can be used to enable a special *nibble* mode, reducing further the total number of I/Os by four to a mere seven.

It would be simple enough to control such a small number of I/Os manually (bit-banging) via the individual PORTE and PORTD pins to implement each bus sequence, but we will take this opportunity instead to explore the capabilities of a new peripheral introduced with the PIC24 architecture: the *Parallel Master Port* (PMP). The designers of the PIC24 family have created this new addressable parallel port to automate and accelerate access to a large number of external parallel devices of common use ranging from analog to digital converters, RAM buffers, ISA bus compatible interfaces, LCD display modules and even hard disks and CompactFlash® cards.

You can think of the PMP as a sort of flexible I/O bus added to the PIC24 architecture that does not interfere with (or slow down) the operation of the 24-bit-wide program memory bus, nor the 16-bit data memory bus. The PMP offers:

- 8- or 16-bit bi-directional data path
- Up to 64k of addressing space (16 address lines)
- Six additional strobe/control lines including:
- Enable
- Address Latch
- Read

Table 9.2: HD44780 command bits

Bit Name	Setting / Status	
I/D	0 = Decrement cursor position	1 = Increment cursor position
S	0 = No display shift	1 = Display shift
D	0 = Display off	1 = Display on
C	0 = Cursor off	1 = Cursor on
B	0 = Cursor blink off	1 = Cursor blink on
S/C	0 = Move cursor	1 = Shift display
R/L	0 = Shift left	1 = Shift right
DL	0 = 4-bit interface	1 = 8-bit interface
N	0 = 1/8 or 1/11 Duty (1 line)	1 = 1/16 Duty (2 lines)
F	0 = 5 x 7 dots	1 = 5 x 10 dots
BF	0 = Can accept instruction	1 = Internal operation in progress

- Write
- Two Chip Select lines

The PMP can also be configured to operate in slave mode, enabling the micro to attach as an addressable peripheral to a larger microprocessor/microcontroller system.

Both bus read and bus write sequences are fully programmable so that not only can the polarity and choice of control signals be configured to match the target bus, but also the timing can be finely tuned to adapt to the speed of the peripherals we interface to.

Configuring the PMP for LCD Module Control

As in all other PIC24 peripherals, there is a set of control registers dedicated to the PMP configuration. The first one is *PMCON*, and you will recognize the familiar sequence of control bits common to all the module's *xxCON* registers (Figure 9.3).

But the list of control registers that we will need to initialize is a bit longer this time and also includes: *PMMODE*, *PMADDR*, *PMSTAT*, *PMAEN* and possibly *PADCFG1*. They are packed with powerful options and they all require your careful consideration. Instead of proceeding through a lengthy review of each and every one of them, I will list only the key choices required specifically by the LCD module interface:

- PMP enabled
- Fully de-multiplexed interface (separate data and address lines will be used)
- Enable strobe signal enabled (*RD4*)
- Read signal enabled (*RD5*)
- Enable strobe active high
- Read active high, Write active low
- Master mode with Read and Write signals on the same pin (*RD5*)
- 8-bit bus interface (using *PORTE* pins).

Upper byte:							
R/W-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PMPEN	—	PSIDL	ADRMUX1	ADRMUX0	PTBEEN	PTWREN	PTRDEN
bit 15				bit 8			

Lower byte:							
R/W-0	R/W-0	R/W-0 ⁽¹⁾	R/W-0 ⁽¹⁾	R/W-0 ⁽¹⁾	R/W-0	R/W-0	R/W-0
CSF1	CSF0	ALP	CS2P	CS1P	BEP	WRSP	RDSP
bit 7				bit 0			

Figure 9.3: *PMCON* Control register

Only one address bit is required, so we will choose the minimum configuration including *PMA0 (RB15)* and *PMA1*.

Also, considering that the typical LCD module is an extremely slow device, we had better select the most generous timing, adding the maximum number of wait states allowed at each phase of a read or write sequence:

- A $4 \times T_{cy}$ wait for data set up before Read/Write
- A $15 \times T_{cy}$ wait between R/W and Enable
- A $4 \times T_{cy}$ wait for data set up after Enable.

A Small Library of Functions to Access an LCD Display

Create a new project called **9-LCD** using the **New Project Setup** checklist and a new source file called **Glass.c**. We will start writing the LCD initialization routine first. It is natural to start with the initialization of the PMP port key control registers.

```
void InitLCD( void)
{
    // PMP initialization
    PMCON = 0x83BF;           // Enable the PMP, long waits
    PMMODE = 0x3FF;          // Master Mode 1
    PMAEN = 0x0001;          // PMA0 enabled
```

After these steps we are able to communicate with the LCD module for the first time and we can follow a standard LCD initialization sequence recommended by the manufacturer. The initialization sequence must be timed precisely (see the HD44780 instruction set for the details) and cannot be initiated before at least 30 ms have been granted to the LCD module to proceed with its own internal initialization (power on reset) sequence. For simplicity and safety, we will hard code a delay in the LCD module initialization function and we will use TMR1 to obtain simple but precise timing loops for all subsequent steps.

```
// init TMR1
T1CON = 0x8030;           // Fosc/2, 1:256, 16us/tick
// wait for >30ms
TMR1 = 0; while( TMR1<2000); // 2000 x 16us = 32ms
```

For our convenience we will also define a couple of constants that will help us, hopefully, make the following code more readable.

```
#define LCDDATA 1
#define LCDCMD 0
#define PMDATA PMDIN1
```

To send each command to the LCD module, we will select the command register (setting the address $PMA0 = RS = 0$) first. Then we will start a PMP write sequence by depositing the desired command byte in the PMP data output buffer.

```
PMADDR = LCDCMD;           // command register
PMDATA = 0b00111000;       // 8-bit, 2 lines, 5x7
```

The PMP will perform the complete bus write sequence as listed below:

1. The address will be published on the PMP address bus (PMA0).
2. The content of PMDATA will be published on the PMP data bus (PMD0-PMD7).
3. After $4 \times T_{cy}$ the R/W signal will be asserted low (RD5).
4. After $15 \times T_{cy}$ the Enable strobe will be asserted high (RD4).
5. After $4 \times T_{cy}$ the Enable strobe will lowered and PMDATA removed from the bus.

Notice how this sequence is quite long as it extends for more than $20 \times T_{cy}$, or more than $1.25 \mu s$, after the PIC24 initiated it. In other words, the PMP will still be busy executing part of this sequence while the PIC24 will have already executed another twenty instructions. Since we are going to wait for a considerably longer amount of time anyway ($>40 \mu s$) to allow the LCD module to execute the command, we will not worry about the time required by the PMP to complete the command at this time.

```
TMR1 = 0; while( TMR1<3); // 3 x 16us = 48us
```

We will then proceed similarly with the remaining steps of the LCD module initialization sequence.

```
PMDATA = 0b00001100;       // ON, cursor off, blink off
TMR1 = 0; while( TMR1<3); // 3 x 16us = 48us

PMDATA = 0b00000001;       // clear display
TMR1 = 0; while( TMR1<100); // 100 x 16us = 1.6ms

PMDATA = 0b00000110;       // increment cursor, no shift
TMR1 = 0; while( TMR1<100); // 100 x 16us = 1.6ms
```

After the LCD module initialization, things will get a little easier. The timing loops will not be necessary any more as we will be able to use the LCD module *read busy flag* command. This will tell us if the integrated LCD module controller has completed the last command given and is ready to receive and process a new one. In order to read the LCD status register containing the *busy flag*, we will need to instruct the PMP to execute a bus read sequence. This is a two-step process: we initiate the read sequence by reading (and discarding) the contents of the PMP data buffer a first time. When the PMP sequence is completed, the data buffer will contain the actual value read from the bus, and we will read its contents from the PMP data buffer again. But how can we tell when the PMP read sequence is complete? Simple, we can check the PMP busy flag in the *PMSTAT* control register.

In summary, to check the *LCD module busy flag* we will need to check first the *PMP busy flag*, issue a read command, wait for the *PMP busy flag* again, and finally we will gain access to the actual LCD module status register contents including the *LCD busy flag*.

By passing the register address as a parameter to the read function, we will obtain a more generic function that will be able to read the LCD status register or the data register as in the following code:

```
char ReadLCD( int addr)
{
    int dummy;
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    PMADDR = addr;              // select the command address
    dummy = PMDATA;             // initiate a read cycle, dummy
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    return( PMDATA);            // read the status register
} // ReadLCD
```

The LCD module status register contains two pieces of information: the LCD busy flag and the LCD RAM pointer current value. We can use two simple macros to split the two pieces: *BusyLCD()* and *AddrLCD()*, and a third one to access the data register: *getLCD()*.

```
#define BusyLCD() ReadLCD( LCDCMD) & 0x80
#define AddrLCD() ReadLCD( LCDCMD) & 0x7F
#define getLCD() ReadLCD( LCDDATA)
```

Using the *BusyLCD()* function we can create a function to write data or commands to the LCD module:

```
void WriteLCD( int addr, char c)
{
    while( BusyLCD());
    while( PMMODEbits.BUSY);    // wait for PMP to be available
    PMADDR = addr;
    PMDATA = c;
} // WriteLCD
```

A few additional macros will help complete the library:

- *putLCD()* will send ASCII data to the LCD module

```
#define putLCD( d) WriteLCD( LCDDATA, (d))
```
- *CmdLCD()* will send generic commands to the LCD module

```
#define CmdLCD( c) WriteLCD( LCDCMD, (c))
```
- *HomeLCD()* will reposition the cursor on the first character of the first row

```
#define HomeLCD() WriteLCD( LCDCMD, 2)
```
- *ClrLCD()* will clear the entire contents of the display

```
#define ClrLCD() WriteLCD( LCDCMD, 1)
```

And finally, for our convenience, we might want to add *putsLCD()*, a function that will send an entire null-terminated string to the display module.

```
void putsLCD( char *s)
{
    while( *s) putLCD( *s++);
} // putsLCD
```

Let's put all of the above to work adding a short main function:

```
main( void)
{
    // initializations
    InitLCD();

    // put a title on the first line
    putsLCD( "Flying the PIC24");

    // main loop, empty for now
    while ( 1)
    {
    }
} // main
```

If all went well after building the project and programming the Explorer16 board with the ICD3 debugger (using the **Debug>Project** command) you will now have the great satisfaction of seeing the title string published on the first row of the LCD display.

Advanced LCD Control

If you felt that all of the above was not too complex, and certainly not rewarding enough, here I have some more interesting stuff, and a new challenge for you to consider.

When introducing the HD44780-compatible alphanumeric LCD modules we mentioned how the display content was generated by the module controller by using a table, the character generator, located in ROM. But we also mentioned the possibility of extending the character set using an

additional RAM buffer known as the CGRAM. Writing to the CGRAM it is possible to create new five-by-seven character patterns to create new symbols, and possibly small graphic elements.

How about adding a small airplane to the character set of the Explorer16 LCD module display?

We will need a function to set the LCD module RAM buffer pointer to the beginning of the CGRAM area using the “Set CGRAM Address” command, or better a macro that uses the *WriteLCD()* function:

```
#define SetLCDG( a) WriteLCD( LCDCMD, (a & 0x3F) | 0x40)
```

To generate two new 5×7 -character patterns, one for the nose of the plane and one for the tail, we will use the *putLCD()* function. Each byte of data will contribute 5 bits (LSB) to define one row of the pattern. After the last row of each character an eighth ‘0’ byte of data will be inserted to align for the next character block.

```
// generate two new characters
SetLCDG(0);
putLCD( 0b00010);
putLCD( 0b00010);
putLCD( 0b00110);
putLCD( 0b11111);
putLCD( 0b00110);
putLCD( 0b00010);
putLCD( 0b00010);
putLCD( 0);      // alignment

putLCD( 0b00000);
putLCD( 0b00100);
putLCD( 0b01100);
putLCD( 0b11100);
putLCD( 0b00000);
putLCD( 0b00000);
putLCD( 0b00000);
putLCD( 0);      // alignment
```

The two new symbols will now be accessible with the codes *0* and *1* respectively of the character generator table.

To reposition the buffer pointer back to the data RAM buffer (*DDRAM*), use the following macro:

```
#define SetLCD( a) WriteLCD( LCDCMD, (a & 0x7F) | 0x80)
```

Notice that while the first line of the display corresponds to addresses from *0* to *0xf* of the *DDRAM* buffer, the second line is always found at addresses from *0x40* to *0x4f* independent of the display size – the number of characters that compose each line of the actual display.

Also, a simple delay mechanism (based once more on *TMR1*) will be necessary to make sure that our airplane flies on time and stays visible. LCD displays are slow; if the display is updated too fast the image tends to disappear like a ghost.

```
#define TFLY 9000          // 9000 x 16us = 144ms
#define DELAY() TMR1=0; while( TMR1<TFLY)
```

It is time to devise a simple algorithm to make the little airplane fly in the main loop; here it is:

```
// main loop
while( 1)
{
    // the entire plane appears at the right margin
    SetLDC(0x40+14);
    putLCD( 0); putLCD( 1);
    DELAY();

    // fly fly fly (right to left)
    for( i=13; i>=0; i--)
    {
        SetLDC(0x40+i);          // cursor to next position
        putLCD(0); putLCD(1);    // new airplane
        putLCD(' ');            // erase the previous tail
        DELAY();
    }

    // the tip disappears off the left margin,
    // only the tail is visible
    SetLDC(0x40);
    putLCD( 1); putLCD(' ');
    DELAY();

    // erase the tail
    SetLDC(0x40);          // point to left margin 2nd line
    putLCD(' ');

    // and draw just the tip appearing from the right
    SetLDC(0x40+15);      // point to right margin 2nd line
    putLCD( 0);
    DELAY();
} // repeat the main loop
```

Have fun flying the PIC24!

Creating an LCD Library Module

As we did in previous chapters, we can separate the LCD control functions from the example application to create a simple library module:

- Move all the LCD support functions, everything but the *main()* function, into a new *LCD.c* file. Save it in the */lib* subdirectory for later use.
- Move all the constants and macro definitions to a new *LCD.h* file. Save it in the */include* subdirectory.

To complete the *LCD.h* file, you will need to create three additional prototypes:


```
void InitLCD( void);  
char ReadLCD( int addr);  
void WriteLCD( int addr, char c);
```

From now on, using the LCD display will be as simple as including the *LCD.h* file at the top of your application and including the *LCD.c* module in the list of source files in your project.

Post-Flight Briefing

In this lesson we learned how to use the parallel master port to drive an LCD display module. In reality we have just started scratching the surface. Also, since the LCD display module is a relatively slow peripheral, it might seem that there has been little or no significant advantage in using the PMP instead of a traditional bit-banged I/O solution. In reality, even when accessing such simple and slow peripherals the use of the PMP can provide two important benefits:

- The timing, sequencing and multiplexing of the control signals are always guaranteed to match the configuration parameters, eliminating the risk of dangerous bus collisions and/or unreliable operation as a consequence of coding errors and/or unexpected execution and timing conditions (interrupts, bugs ...).
- The MCU is completely free from tending the peripheral bus, allowing simultaneous execution of any number of higher-priority tasks.

Notes for User Interface Experts

The family of PIC24 microcontrollers has expanded considerably in the last few years and now includes a number of models that offer two important new features:

- The Charge Time Measurement Unit (CTMU), which allows very precise control of capacitive inputs as required in touch screen applications.
- An integrated graphic controller, including three hardware acceleration units that allow for very effective interface to C-STN and TFT (color) graphic displays.

PIC24 microcontroller models such as the PIC24FJxxDA2xx family, offering both, can be used to implement very sophisticated graphical user interfaces with touch-sensing input without requiring any additional external component (except for additional RAM memory should the internal 96 Kbytes not be sufficient for the application!).

For all PIC24 microcontrollers that offer the basic PMP port there is also the possibility of interfacing directly to mobile displays that integrate both the graphic controller and all the RAM required (several hundred Kbytes) to maintain a 16-bit color QVGA or larger image. At the time of writing this second edition, the cost of such devices is rapidly approaching that of traditional TFT QVGA displays, in sizes up to 3.2 inches, and closing the gap with the basic monochrome alphanumeric LCD display used in the past.

The complexity of developing any advanced graphical user interface is greatly reduced by the availability of a complete (object-oriented) graphic library as part of the Microchip Application Library (MAL) but is outside the scope of this introductory book.

Notes for the C Experts

As we did in the previous lesson, when using the asynchronous serial interfaces, it is possible to replace the low-level I/O routines defined in the *stdio.h* library, and in particular *write.c*, to re-direct the output to the LCD display as in the example code below:

```
#include <p24fxxx.h>
#include <stdio.h>
#include <LCD.h>

int write(int handle, void *buffer, unsigned int len)
{
    int i, *p;
    const char *pf;

    switch (handle)
    {
        case 0:
        case 1:
        case 2:
            for (i = len; i; --i)
                putLCD( *(char*)buffer++);
            break;
        default:
            break;
    }
    return(len);
}
```

In an alternate scheme, you might want to redirect the *stdout* stream to the LCD display as the main output of the application, and the *stderr* stream to the serial port for debugging purposes.

Also, it is most likely at this point that you will want to modify the *putLCD()* function so as to interpret special characters such as ‘\n’, to advance to the next line, or even introducing a partial decoding for a few ANSI escape codes so as to be able to position the cursor and clear the screen (using the macros defined in this lesson) just like on a terminal console.

Tips & Tricks

Since the LCD display is a slow peripheral, waiting for its commands to be completed in tight (locking) loops as in the examples provided in this lesson could constitute an unacceptable waste of MCU cycles in some applications. A better scheme would require LCD commands to be cached in a FIFO buffer and having an interrupt mechanism to periodically schedule their execution. In other words, interrupts would be used to perform multitasking of a slow process in the background of the application execution.

An example of such a mechanism is provided in the *LCD.c* example code provided with the Explorer16 demonstration board.

Exercises

Enhance the *putLCD()* function to interpret correctly the following characters:

- ‘\n’: advance to the next line
- ‘\r’: re-position cursor to the beginning of current line
- ‘\t’: advance to a fixed tabulation position
- ‘\b’: backspace with overwrite of preceding character.

Enhance the *putLCD()* function to interpret the following ANSI escape codes:

- ‘\x1b[2J’: clear entire screen
- ‘\x1b[1;1H’: home cursor
- ‘\x1b[n;mH’: position the cursor at row ‘n’, column ‘m’.

Books

- Galitz, Wilbert O., 2007, *The Essential Guide to User Interface Design*, Wiley, Indianapolis, IN.
Designing user interfaces (graphical or not) is a science, a little planning can go a long way in making your application more usable and effective.

Links

- <http://www.microchip.com/graphics>
This is the design center dedicated to graphic displays interfaces.
- <http://www.microchip.com/mal>
This is the Microchip Applications Library page. It contains the Graphic Library, which offers an object-oriented library for the creation of graphical user interfaces using 16-bit color displays and touch-sensing inputs.
- <http://www.microchip.com/GDD>
This the stand-alone Microchip Graphic Display Designer tool, soon to be integrated in MPLAB X, that allows the rapid prototyping of complex user interfaces based on the Microchip Graphics Library.
- <http://www.microchip.com/PICTailPlus>
This is the catalog of expansion boards available for the Explorer16. Among them the Graphics PICTail Plus (AC164127) will allow you to quickly evaluate color graphic applications.

It's an Analog World

There are certain things that, no matter how many times you practice, never seem to come out the same way twice. Landings are a good example. Even the most experienced airline captains will have a bad day and will screw it up. I'm sure you have happened to notice it, when they “bounce” a landing. What is wrong with landings, why are they so difficult?

The fact is that, no matter how hard you try, the conditions affecting a landing are never really exactly the same. The wind speed and direction change continuously, the performance of the engine, and even the wings, changes with the slightest change in the air temperature. Additionally, the pilot's reflexes (and alertness) change. It all combines to create an infinite number of unpredictable conditions that make for an infinite number of possible ways to get it wrong.

We live in an analog world. All the pilot's input variables are analog: temperature, wind speed and direction. All of our sensory system inputs are analog: sight, sounds, pressure. The output is analog, such as is the movement of the pilot's muscles to control the plane. With time, we learn to interpret (or should I say convert) all the analog inputs from the world around us and make the best decisions we can. Practice can make us perfect, almost!

In embedded control, the information from the analog world must first be converted to digital to be usable within our applications. The analog-to-digital converter module is one of the key interfaces to the “real” world.

Flight Plan

The PIC24 family was designed with embedded control applications in mind and therefore is ideally prepared to deal with the analog nature of this world. A fast analog-to-digital converter (ADC), capable of 500,000 conversions per second, is available on all models. Also included with the ADC is an input multiplexer that allows you to select from a number of analog inputs quickly and with high resolution. In this lesson we will learn how to use the 10-bit ADC module available on the PIC24FJ128GA010 family to perform two simple measurements on the Explorer16 board: first we will read a voltage input from a potentiometer first and later a voltage input from a temperature sensor.

Preflight Checklist

In addition to the usual software tools, including the MPLAB[®] X IDE and MPLAB C compiler, this lesson will require the use of the Explorer16 demonstration board and an in circuit debugger/programmer of your choice.

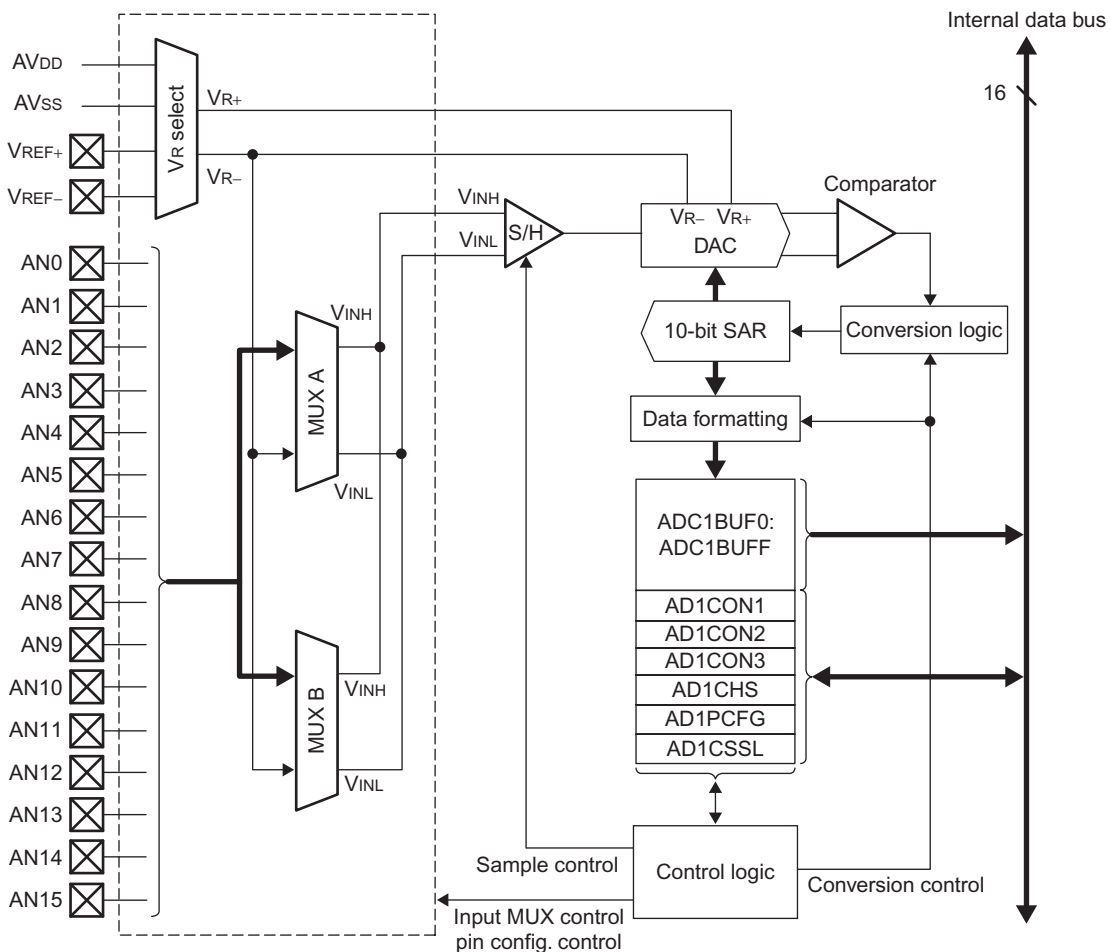


Figure 10.1: ADC module block diagram

The Flight

The first step in using the analog-to-digital converter, like any other peripheral module inside the PIC24, is to familiarize yourself with the module building blocks and the key control registers. Yes, this means reading the datasheet once more, and even the Explorer16 User Guide to find out the schematics.

We can start by looking at the ADC module block diagram in [Figure 10.1](#).

This is a pretty sophisticated structure that offers many interesting capabilities:

- Up to 16 input pins can be used to receive the analog inputs
- Two input multiplexers can be used to select different input analog channels and different reference sources for each

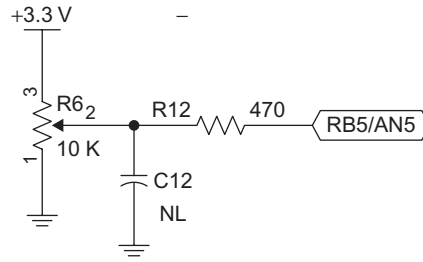


Figure 10.2: Detail of the Explorer16 demonstration board, R6 potentiometer

- The output of the 10-bit converter can be formatted for integer or fixed-point arithmetic, signed or unsigned 16-bit output
- The control logic allows for many possible automated conversion sequences to synchronize the process to the activity of other related modules and inputs
- The conversion output is stored in a 16-bit-wide, 16-word-deep buffer that can be configured for sequential scanning or simple FIFO buffering

All these capabilities require a number of control registers to be properly configured and I understand how, especially at the beginning, the number of options available and decisions to take could make you a bit dizzy. So we will start by taking the shortest and simplest approach with the simplest example application: reading the position of the R6 potentiometer on the Explorer16 board (Figure 10.2).

The 10kohm potentiometer is connected directly to the power supply rails so that its output can span the entire range of values from 3.3V to the ground reference. It is connected to the RB5 pin that corresponds to the analog input AN5 of the ADC input multiplexer.

After creating a new project using the appropriate checklist we can create a new source file **pot.c**, including the usual header file and adding the definition of a couple useful constants. The first one, POT, defines the input channel assigned to the potentiometer and the second one, AINPUTS, is a mask that will help us define which inputs should be treated as analog and which ones as digital.

```
/*
** It's an analog world
** Converting the analog signal from a potentiometer
*/
#include <config.h>

#define POT_CH 5 // 10k potentiometer on pin AN5
#define POTMASK 0xffdf // AN5 = analog inputs
```

The actual initialization of all the ADC control registers can be best performed by a short function, *initADC()*, that will produce the desired initial configuration:

1. *ADIPCFG* will be passed the mask selecting the analog input channels: 0s will mark the analog inputs, 1s will configure the respective pins as digital inputs.

2. *ADICON1* will set the conversion to start automatically, triggered by the completion of the auto-timed sampling phase; also, the output will be formatted for a simple unsigned, right-aligned (integer) value.
3. *ADICSSL* will be cleared, as no scanning function will be used (only one input).
4. *ADICON2* will select the use of MUXA and will connect the ADC reference inputs to the analog input rails: AVdd and AVss.
5. *ADICON3* will select the conversion clock source and divider.
6. Finally, setting *ADON*, the entire ADC peripheral will be activated and ready for use.

```
void InitADC( int amask)
{
    AD1PCFG = amask;    // select analog input pins
    ADICON1 = 0;        // manual conversion control
    ADICSSL = 0;        // no scanning required
    ADICON2 = 0;        // use MUXA, AVss and AVdd used as Vref
    ADICON3 = 0x1F01;   // Tad=2xTcy=125ns
    AD1CON1bits.ADON = 1; // turn on the ADC
} // InitADC
```

Passing *amask* as a parameter to the initialization routine, we make it flexible so it can accept multiple input channels in future applications.

Note

There is a published *errata* for the Explorer16 board. Depending on the specific model of LCD display installed there is a potential leak through pin RB15 that can impact the performance of the A/D converter. The simplest work around consists of making sure that such pin is configured as an output so that the voltage at the pin is limited to one of the supply rails: 3V or ground. This can be accomplished adding the following two lines of code to the *InitADC()* function:

```
_LATB15 = 0;
_TRISB15 = 0;
```

The First Conversion

The actual analog-to-digital conversion is a two-step process. First we need to take a sample of the input voltage signal; only then we can disconnect the input and perform the actual conversion of the sampled voltage into a numerical value. The two distinct phases are controlled by two separate control bits in the *ADICON1* register: *SAMP* and *DONE*. The timing of the two phases is important to provide the necessary accuracy of the measurement.

During the sampling phase the external signal is connected to an internal capacitor that needs to be charged up to the input voltage. Enough time must be provided for the capacitor to track the input voltage and this time is mainly proportional to the impedance of the input signal source (in our case known to be less than 5 kohm) as well as the internal capacitor value. In

general, the longer the sampling time, the better is the result compatibility with the input signal frequency (not an issue in our case).

The conversion phase timing depends on the selected ADC clock source. This is typically derived by the main CPU clock signal via a divider or alternatively by an independent RC oscillator. The RC option is appealing for its simplicity, and is a good choice when a conversion needs to be performed during a sleep (low-power mode) phase, when the CPU clock is turned off. The oscillator clock divider is a better option in more general cases, like this one, since it provides synchronous operation with the CPU and therefore a better rejection of the noise internally produced by it. The conversion clock should be the fastest possible compatible with the specifications of the ADC module. In our case T_{ad} is required to be longer than 75 ns, requiring a minimum clock divider by two.

Here is a basic conversion routine:

```
int ReadADC( int ch)
{
    AD1CHS = ch;           // 1. select analog input channel
    AD1CON1bits.SAMP = 1;   // 2. start sampling
    TMR1 = 0;              // 3. wait for sampling time
    while ( TMR1<100);
    AD1CON1bits.SAMP = 0;   // 4. start conversion
    while (!AD1CON1bits.DONE); // 5. wait for conversion to complete
    AD1CON1bits.DONE = 0;   // 6. clear flag
    return ADC1BUF0;        // 7. read the conversion result
} // ReadADC
```

Automatic Sampling

As you can see, using this basic method, we have been responsible for providing the exact timing of the sampling phase, dedicating a timer to this task and performing two wait loops. But on the PIC24 there is a new option that allows for a more automatic process. The sampling phase can be self-timed, provided the input source impedance is small enough to require a maximum sampling time of $32 \times T_{ad}$ ($32 \times 120 \text{ ns} = 3.8 \mu\text{s}$ in our case). This can be achieved by setting the *SSRC* bits in the *AD1CON1* register to the *0b111* configuration, upon termination of the self-timed sampling period automatically enables the start of the conversion. The period itself is selected by the *AD1CON3* register *SAM* bits. Here is a new and improved example that uses the self-timed sampling and conversion trigger.

```
void InitADC( int amask)
{
    AD1PCFG = amask;       // select analog input pins
    AD1CON1 = 0x00E0;      // auto convert after end of sampling
    AD1CSSL = 0;           // no scanning required
    AD1CON3 = 0x1F01;      // sample time=31Tad, Tad=2xTcy=125ns
    AD1CON2 = 0;           // use MUXA, AVss and AVdd used as Vref
    AD1CON1bits.ADON = 1;  // turn on the ADC
}
```



```

    // Explorer 16 Development Board Errata (work around 2)
    // RB15 should always be a digital output
    _LATB15 = 0;
    _TRISB15 = 0;
} // InitADC

```

Notice how having the conversion-start trigger automatically by the completion of the self-timed sampling phase gives us two advantages:

- Proper timing of the sampling phase is guaranteed without requiring us to use any timed delay loop and/or other resource.
- One command (start of the sample phase) suffices to complete the entire sampling and conversion sequence.

With the ADC so configured, starting a conversion and reading the output is a trivial matter:

- *AD1CHS* selects the input channel for MUXA.
- Setting the *SAMP* bit in *AD1CON1* starts the timed sampling phase that will be immediately followed by the conversion.
- The *DONE* bit will be set in the *AD1CON1* register as soon as the entire sequence is completed and a result is ready. You are responsible for manually cleaning it.
- Reading the *ADC1BUF0* register will immediately return the desired conversion result.

```

int ReadADC( int ch)
{
    AD1CHS = ch;           // select analog input channel
    // start sampling, automatic conversion will follow
    AD1CON1bits.SAMP = 1;
    while (!AD1CON1bits.DONE); // wait to complete conversion
    AD1CON1bits.DONE = 0;
    return ADC1BUF0;       // read the conversion result
} // ReadADC

```

Developing a Demo

All that remains is to figure out an entertaining way to put the converted value to use on the Explorer16 demo board. The LEDs connected to PORTA are an obvious choice, but instead of simply providing a binary output, publishing the eight most significant bits of the 10-bit result, why not jazz things up a little and provide a visual feedback more reminiscent of the analog nature of our input? We could turn on one LED at a time, using it as an index on a mechanical dial. Here is the main routine we will use to test our analog-to-digital functions:

```

main ()
{
    int pot, pos;

    // initializations
    InitADC( POTMASK); // initialize the ADC and analog inputs
    TRISA = 0xff00;    // all PORTA pins as outputs

```

```

// main loop
while( 1)
{
    pot = ReadADC( POT_CH); // select the POT input, convert

    // reduce the 10-bit result to a 3 bit value (0..7)
    // (divide by 128 or shift right 7 times
    pos = pot >> 7;

    // turn on only the corresponding LED
    // 0 -> leftmost LED.... 7-> rightmost LED
    PORTA = (0x80 >> pos);
} // main loop
} // main

```

After the call to the initialization routine (to which we provide a mask that defines bit 5 as analog input), we initialize the *TRISA* register to make the pins connected to the LED bar digital outputs. Then in the main loop we perform the conversion on AN5 and we reformat the output to fit our special display requirements. As configured, the 10-bit conversion output will be returned as a right-aligned integer in a range of values between 0 and 1023. By dividing that value by 128 (or in other words shifting it right seven times) we can reduce the range to a zero to seven value. The final output though requires one more transformation to produce the desired one of eight LED configurations. Note that the LED corresponding to the MSB is located to the left of the bar and to maintain the correspondence between the potentiometer movement clockwise and the index LED moving to the right we need to start with a *0b10000000* pattern and shift it right as required.

Use the **Debug>Project** command to build the project for ICD debugging and then program the Explorer16 board. If all goes well, you will be able to play with the potentiometer, moving it from side to side while observing the LED index moving left and right correspondingly.

Expanding the EX16 Library

The two simple functions that we have created to access the A/D converter will be useful in the future, but they are just too small to claim a library module of their own. I suggest we include them in the *EX16.h* and *EX16.c* library module that we have been assembling since lesson 3.

While we are at it, as you might have noticed, we have continued to use Timer1 in the past several lessons to provide accurate delays. Let's add once and for all a simple function that can provide us with a delay of an exact number of milliseconds:

```

void Delaysms( unsigned t)
{
    T1CON = 0x8000;    // enable tmr1, Tcy, 1:1
    while (t-->0)      // wait for t (msec)

```

```
{
    TMR1 = 0;
    while ( TMR1 < (FCY/1000)); // wait 1ms
}
} // Delaysms
```

Developing a Game

OK, I will admit it, the first example ADC application was not too exciting. After all, we have been using a 16MIPS-capable 16-bit machine to perform an analog-to-digital conversion roughly 200,000 times a second (32 T_{ad} sampling + 12 T_{ad} conversion, where $T_{ad} = 125$ ns, you do the math) only to discard all but three bits of the result and watch a single LED light up. How about making it a bit more challenging and playful instead? How about developing a little “Whac-A-Mole” game, just a mono-dimensional version?

Let’s turn on a second LED (the mole), controlled by the PIC24 and distinguishable from the user-controlled LED (the mallet) by having it be somewhat dimmer. Move the mallet (bright LED) by rotating the potentiometer until you reach the mole (dim LED) and you will get to “whack it”! A new mole, in a different random position, will immediately appear and the game will continue.

The pseudo-random number generator function *rand()* (defined in *stdlib.h*) will be helpful here, as all (computer) games need a certain degree of unpredictability. We will use it to randomly determine where to place each new mole.

Save the source file from the first project with a new name **POTgame.c** and create an entire new project: **10-POTgame**. Then let’s update the *main()* function to include just a few more lines of code:

```
#include <EX16.h>
#include <config.h>

main ()
{
    int pot, pos, mole, mallet, c;

    // 1. initializations
    TRISA = 0xff00;    // select all PORTA pins as outputs

    // initialize ADC and analog inputs
    InitADC( POTMASK);

    // 2. use the first reading to randomize the number generator
    srand( ReadADC( POT_CH));
    // generate the first random position
    mole = 0x80 >> (rand() & 0x7);
    // init the counter
    c = 0;
```

```

// 3. main loop
while( 1)
{
    // 3.1 select the POT input and convert
    pot = ReadADC( POT_CH);

    // reduce the 10-bit result to a 3-bit value (0..7)
    // 3.2 divide by 128 or shift right 7 times
    pos = pot >> 7;

    // 3.3 turn on only the corresponding LED
    // 0 -> leftmost LED.... 7-> rightmost LED
    mallet = (0x80 >> pos);

    // 3.4 when the cursor hits the mole LED, generate new one
    while (mallet == mole )
        mole = 0x80 >> (rand() & 0x7);

    // 3.5 display the user LED and 50% dim random LED
    if ((c & 0xf) == 0)
        PORTA = mallet + mole;
    else
        PORTA = mallet ;

    // 3.6 counter to alternate loops
    c++;
} // main loop
} // main

```

In 1. we perform the usual initialization of the analog-to-digital converter module and the PORTA I/Os connected to the bar of LEDs.

In 2. we read the potentiometer value for the first time and we use its value as the seed for the random number generator. This makes the game experience truly unique provided the game does not start with the potentiometer always found in the leftmost or rightmost position. That would provide a seed value of 0 or 1023 respectively and therefore would make the game quite repetitive as the pseudo-random sequence would proceed through the same steps at any game restart.

In 3. the main loop begins, as in the previous example, reading an integer 10-bit value (3.1).

In 3.2 the conversion into an LED position (*pos*) is performed just as before, but it is in 3.4 that things get interesting. If the position of the mallet LED is overlapping the mole LED position, a new random position is immediately calculated. The operation needs to be repeated as a *while* loop because each time a new random value for *mole* is calculated there is a chance (exactly one in eight if our pseudo-random generator is a good one) that the new value could be the same. In other words we could be creating a new mole right under the mallet. And that would not be very challenging or sporty. Don't you agree?

Steps 3.5 and 3.6 are all about displaying and differentiating the two LEDs. To show both LEDs on the display bar we could simply *add* the two binary patterns *mallet* and *mole* but it would be very hard for the player to tell who is who. To represent the mole LED with a dimmer light, we can alternate cycles of the main loop where we present both LEDs and cycles where only the mallet LED is visible. Since the main loop is executed hundreds of thousands of times per second, our eye will perceive the mole LED as dimmer, proportionally to the number of cycles it is missing. For example: if we add the mole LED only once every 16 cycles, its apparent brightness will be only one-sixteenth of that of the mallet LED.

The counter *c*, constantly incremented in 3.6, helps us to implement this mechanism.

In 3.5 we look only at the 4 lsb of the counter (0..15) and we add the mole LED to the display only when their value is 0b0000. For the remaining 15 loops, only the mallet LED will be added to the display.

Build the project and download it to the Explorer16 board. You will have to admit it, it's much more entertaining now!

Measuring Temperature

Moving on to more serious things, there is a temperature sensor mounted on the Explorer16 board and it happens to be a Microchip TC1047A, an integrated temperature sensing device with a nice linear voltage output. This device is very small as it is offered in a SOT-23 (three pin, surface mount) package. The power consumption is limited to 35 μ A (typ.) while the power supply can cover the entire range from 2.5 V to 5.5 V. The output voltage is independent of the power supply and is an extremely linear function of the temperature, typically within 0.5 $^{\circ}$ C, with a slope of exactly 10 mV/ $^{\circ}$ C. The offset is adjusted to provide an absolute temperature indication according to the function shown in [Figure 10.3](#).

We can apply our newly acquired abilities to convert the voltage output of the temperature sensor to digital information once more using the analog-to-digital converter of the PIC24. The temperature sensor is directly connected to the AN4 analog input channel as per the Explorer16 board schematic in [Figure 10.4](#).

Using the new project wizard, let's create a new project called **10-TempSense** and a new source file: **Temp.c**. We can include the *EX16.h* library module to reuse the ADC control routines developed earlier in this lesson. Let's also define two new constants: *TSENS* for the ADC input channel assigned to the temperature sensor and *AINPUTS* to configure the corresponding pin as an analog input.

```
/*  
** Temp.c  
** Converting analog signal from a TC1047 temperature sensor  
*/  
#include <EX16.h>
```

```
#include <config.h>
```

```
#define TEMP_CH 4 // ch 4 = TC1047 temperature sensor
```

```
#define TEMPMASK 0xffef // AN4 as analog input
```

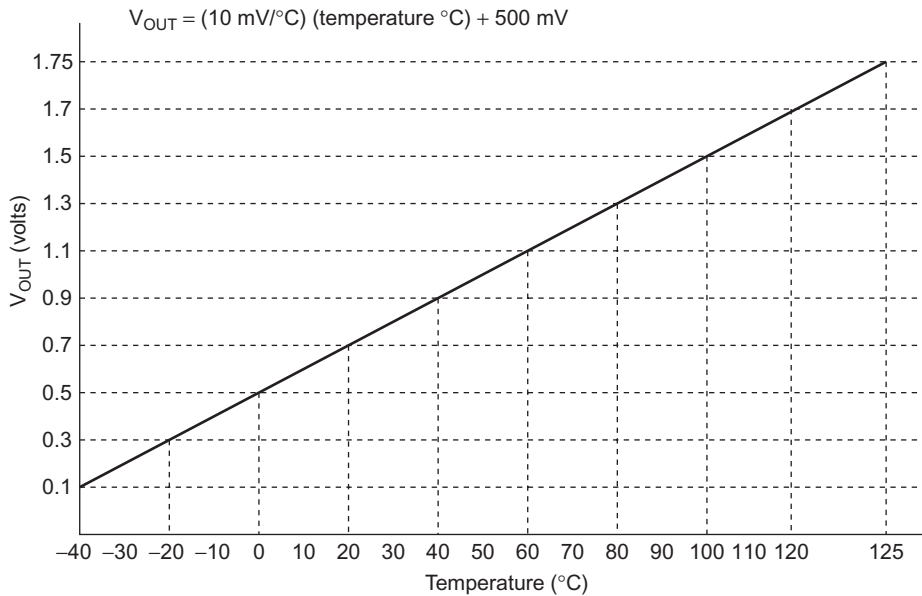


Figure 10.3: TC1047 output voltage vs temperature characteristics

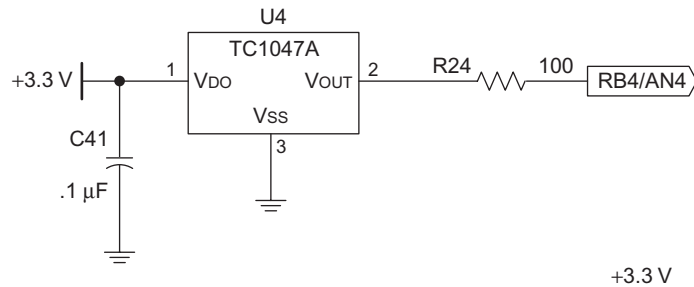


Figure 10.4: Detail of the Explorer16 demonstration board, TC1047A temperature sensor

Presenting the result on the LED bar might be a little tricky though. Temperature sensors provide a certain level of noise and to give a more stable reading it is common to perform a little filtering. Taking groups of 16 samples and performing an average will give us a cleaner value to work with.

```
temp = 0;
for ( j= 16; j >0; j--)
    temp += ReadADC( TEMP_CH); // read the temperature
ref = temp >> 4;
```

But how could we display the result using only the bar of LEDs?

We could pick the most significant bits of the conversion result and publish them in binary or BCD, but once more it would not be fun. How about providing instead a relative temperature indication using a similar (single LED) index moving along the LED bar?

We will sample the initial temperature value just before the main loop and use it as the offset for the center bar position. In the main loop we will update the dot position, moving it to the right as the sensed temperature increases or to the left as the sensed temperature decreases. Here is the complete code for the new temperature sensing example:

```
main ()
{
    int temp, ref, pos, j;

    // 1. initializations
    InitADC( TEMPMASK); // initialize ADC Explorer16 inputs
    TRISA = 0xff00;     // select PORTA pins as outputs

    // 2. get the central bar reference
    temp = 0;
    for ( j= 0; j <16; j++)
        temp += ReadADC( TEMP_CH); // read the temperature
    ref = temp >> 4;

    // 3.0 main loop
    while( 1)
    {
        // 3.1 get a new measurement
        temp = 0;
        for ( j= 0; j<16; j++)
        {
            temp += ReadADC( TEMP_CH); // read the temperature
        }
        temp >>= 4;                // averaged over 16 values
        Delayms( 500);             // 1/2 second

        // 3.2 compare with initial reading, move bar 1 dot/C
        pos = 3 + (temp - ref);
        // 3.3 keep result in value range 0..7, keep bar visible
        if ( pos > 7)
            pos = 7;
        if ( pos < 0)
            pos = 0;

        // 3.4 turn on the corresponding LED
        PORTA = ( 0x80 >> pos);
    } // main loop
} // main
```

In 3.2 we determine the difference between the initial reading *ref* and the new averaged reading *temp*. The result (*pos*) is centered so that a central LED is lit up when the difference is zero.

In 3.3 the result is checked against the boundaries of the available display space. Once the difference becomes negative and more than three bits wide, the display must simply indicate the leftmost position. When the difference is positive and more than four bits wide, the rightmost LED must be activated.

In 3.4 we just publish this result as in the previous example.

To complete the exercise, and give you a more aesthetically pleasing experience, I recommend that you introduce also an additional delay after each new temperature reading. This will slow things down quite a bit, reducing the update rate of the display (and eventually the entire main loop cycle) to a period of half a second. A faster update rate would produce only an annoying flicker when the temperature readings are too close to the intermediate values between two contiguous dot positions.

Launch the project with the **Run>Run Project** command. After identifying the temperature sensor on the board (hint: it is close to the lower left corner of the PIC24 processor module and it looks like a surface-mount transistor), observe how small temperature variations, obtained by touching or blowing hot/cold air on the sensor, move the cursor around.

Another Game

To have a bit more fun with the temperature sensor, we can now merge the last two exercises in one new project: **10-TEMPgame**. The idea is to whack the *mole* (dim) LED by controlling the *mallet* using the temperature sensor. Heat the sensor up with some hot air to move it to the right, blow cold air on it to move it to left. We could call it the *Breath-alizer* game!

```
/*
** TEMPgame.c
** Playing with a temperature sensor
*/
#include <EX16.h>
#include <config.h>

main ()
{
    int temp, ref, pos, mallet, mole, j;

    // 1. initializations
    InitADC( TEMPMASK); // initialize the ADC and analog inputs
    TRISA = 0xff00;      // all PORTA pins as outputs

    // 2. use the first reading to randomize a number generator
    srand( ReadADC( TEMP_CH));
    // generate the first random position
    mole = 0x80 >> (rand() & 0x7);
```



```
// 3. compute the average value for the initial reference
temp = 0;
for ( j= 0; j<16; j++)
    temp += ReadADC( TEMP_CH); // read the temperature
ref = temp >> 4;

// 4. main loop
while( 1)
{
    // 4.1 take the average temperature
    temp = 0;
    for ( j=0; j<16; j++)
    {
        temp += ReadADC( TEMP_CH); // read the temperature
    }
    temp >>= 4; // averaged over 16 values

    // 4.2 compare with the initial reading
    // and move the bar 1 pos. per C
    pos = 3 + (temp - ref);

    // 4.3 keep result in value range 0..7, keep bar visible
    if ( pos > 7)
        pos = 7;
    if ( pos < 0)
        pos = 0;

    // 4.4 turn on the corresponding LED
    mallet = ( 0x80 >> pos);

    // display result for 1/2 second
    for( j=0; j<128; j++)
    { // display the user LED and dim random LED
        if ((j & 0xf) == 0)
            PORTA = mallet + mole;
        else
            PORTA = mallet ;
        Delayms( 5);
    }

    // 4.5 when the cursor hits mole LED, generate new one
    while ( mallet == mole )
        mole = 0x80 >> (rand() & 0x7);
    } // main loop
} // main
```

Post-Flight Briefing

In this lesson we have just started scratching the surface and exploring the possibilities offered by the analog-to-digital converter module of the PIC24. We have used one simple

configuration of the many possible and only a few of the advanced features available. We have tested our newly acquired capabilities with two types of analog inputs available on the Explorer16 board and hopefully we had some fun in the process.

Notes for the C Experts

Even if the PIC24 has a fast divide instruction, there is no reason to waste any processor cycles. In embedded control, *every* processor cycle is precious. If the divisor is a power of two, the integer division can be best performed as a simple shift right by an appropriate number of positions with a computational cost that is at least an order of magnitude smaller than a regular division. If the divider is not a power of two, consider changing it if the application allows. In our last example, we could have opted for averaging 10 temperature samples, or 15 as well as 20, but we chose 16 because this made the division a simple matter of shifting the sum by 4 bits to the right (in a single-cycle PIC24 instruction).

Tips & Tricks

If the sampling time required is longer than the maximum available option ($32 \times T_{ad}$) you can try and extend T_{ad} first or, better, swap things around and enable the automatic sampling start (at the end of the conversion). This way the sampling circuit is always open, charging, whenever the conversion is not occurring. Manually clearing the *SAMP* bit will trigger the actual conversion start.

Further, it is possible to have Timer3 periodically clear the *SAMP* control bit (one of the options for the *SSRC* bits in *ADICON1*), and enabling the ADC end of conversion interrupt will provide the widest choice of sampling periods possible for the least amount of MCU overhead possible: no waiting loops, only a periodic interrupt when the results are available and ready to be fetched.

Exercises

- Use the ADC FIFO buffer to collect conversion results, set up Timer3 for automatic conversion and the interrupt mechanism so that a call is performed only once the buffer is full and temperature values are ready to be averaged.

Books

- Baker, B., 2005. *A Baker's Dozen: Real Analog Solutions for Digital Designers*, Newnes, Burlington, MA.
For proper feed and care of an analog-to-digital converter look no further than Bonnie's cookbook.

Links

- <http://www.microchip.com/thermal>
Temperature sensors are available in many flavors and a choice of interface options including direct I²C™ or SPI™ digital output.

Whac-a-Mole is a trademark of Bob's Space Racers Inc.

Cross-Country Flying

Congratulations! You have endured a few more lessons and gained the ability to complete more complex flights. You are going to enter now the third and last part of your training where you'll practice cross-country flying. No more pattern work around the airport, no more landings and take offs, or maneuvers in the practice area: you will finally get to go somewhere!

In the third part of this book we will start developing new projects that will require you to master several peripheral modules at once. Since the complexity of the examples will grow a little bit more, not only is having an actual demonstration board (the Explorer16) at hand recommended, but having the ability to perform small modifications and using the prototyping area to add new functionality to the demonstration board will be necessary. Simple schematics and component part numbers will be offered in the following chapters as required. On the companion website www.flyingpic24.com you will find additional expansion boards and prototyping options to help you enjoy even the most advanced projects.

Capturing Inputs

As we were saying in a previous chapter, advanced electronics is rapidly gaining space in the cockpits of all but the smallest airplanes. While the glass (LCD) displays are supplanting the old steam gauges, GPS satellite receivers are plotting in real time the airplane position on colorful maps depicting terrain elevations and, with additional equipment, up-to-the-minute satellite weather information too. Pilots can enter an entire flight plan into the navigation system and then follow their path on the moving map just like into a video game. The interaction with these new instruments, though, is becoming the next big challenge. Just like computer applications, each instrument is controlled by a different menu system and a set of knobs and buttons to allow the pilot to provide the inputs quickly and (hopefully) intuitively. But the limited space in the cockpit has so far imposed serious limitations on the type and number of such input devices that, for the most part, at least in the first generations, have been mimicking the knobs and buttons of primitive VHF radios.

If you have a GPS navigation system in your car and you have tried to dial in a street address in a foreign city (say “Bahnhofstrasse, 17, Munich”) by twisting and turning that little knob while driving on a highway... well you know exactly the type of challenge I am talking about. Keyboards are the logical next level of input interface for several advanced avionics (aviation electronics) systems. They are already common in the business jet cockpits, but they are starting to make their appearance in the smaller general aviation airplanes too. How about a keyboard in your next car?

Flight Plan

With the advent of the USB interface, computers have been finally freed from a number of *legacy* interfaces that had been in use for decades, since the introduction of the first IBM PC. The PS/2 mouse and keyboard interface is one of them. The result of this transition is that a large number of the *old* keyboards are now flooding the surplus market and even new PS/2 keyboards are selling for very low prices. This creates the opportunity to give our future PIC24 projects a powerful input capability. It will also give us the motivation to investigate a few alternative interface methods and their trade-offs. We'll implement software state machines, refresh our experience using interrupts and possibly learn to use new peripherals.

The Flight

The physical PS/2 port uses a Five-pin DIN or a Six-pin mini-DIN connector. The first was common on the original IBM PC-XT and AT series but has not been in use for a while now. The smaller Six-pin version has been more common in recent years. Once the different pin-outs are taken in consideration, the two are electrically identical (Figures 11.1 and 11.2).

Electrical Interface

The host must provide a 5V power supply. The current consumption will vary with the keyboard model and year, but you can expect values between 50 and 100 mA (the original specifications used to call for up to 275 mA max.).

The data and clock lines are both open-collector with pull-up resistors (1–10 kohm) to allow for two-way communication. In the normal mode of operation, it is the keyboard that drives both lines in order to send data to the personal computer. When it is necessary though, the computer can take control to configure the keyboard and to change the status LEDs (“Caps” and “Num Lock”).

The PS/2 Communication Protocol

At idle, both the data and clock lines are held high by the pull-ups (located inside the keyboard). In this condition, the keyboard is enabled and can start sending data as soon as a key has been pressed. If the host holds the clock line low for more than 100 μ s, any further

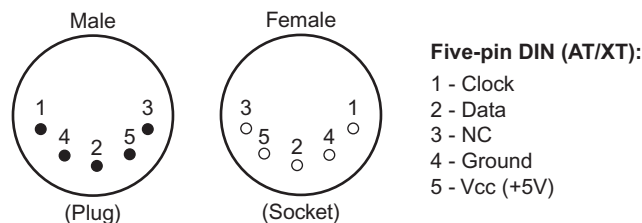


Figure 11.1: Five-pin AT/XT DIN connector

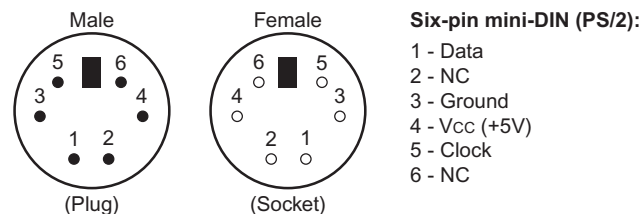


Figure 11.2: Six-pin mini-DIN connector

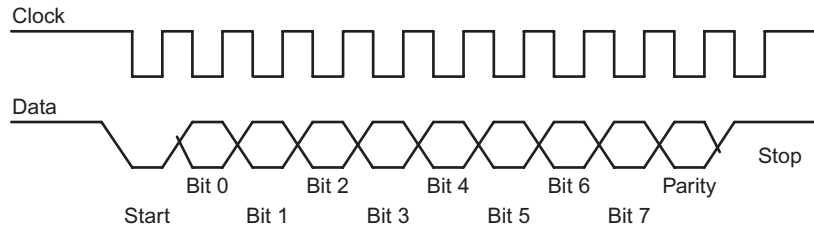


Figure 11.3: Keyboard to host communication waveform

keyboard transmissions are suspended. If the host holds the data line low and then releases the clock line, this is interpreted as a request for the host to send a command (Figure 11.3).

The protocol is a curious mix of synchronous and asynchronous communication protocols we have seen in previous chapters. It is synchronous since a clock line is provided, but it is similar to an asynchronous protocol since a start, stop and parity bit are used to bracket the actual 8-bit data packet. Unfortunately, the baud-rate used is not a standard value and can change from unit to unit, over time, with temperature and the phase of the moon. In fact, typical values range from 10 to 16 Kbits per second. The Data line is allowed to change during the clock-high state. The Data line is sampled when the clock line is low. Whether data is flowing from the host to the keyboard or vice versa, it is the keyboard that always generates the clock signal.

Note

The USB bus reverses the roles as it makes each peripheral a synchronous slave of the host. This simplifies things enormously for a non-real-time, non-pre-emptive, multi-tasking operating system such as Windows. The serial port and the parallel port were similarly asynchronous interfaces, and probably for the same reason, both became legacy with the introduction of the USB bus specification.

Interfacing a PIC24 to the PS/2

The unique peculiarities of the protocol make interfacing to a PS/2 keyboard an interesting challenge, as neither the PIC24 SPI™ interface, nor the UART interface can be used. In fact, the SPI interface does not accept 11-bit words (8-bit or 16-bit words are the only options), while the PIC24 UART would require the periodic transmission of special break characters to make use of the powerful automatic baud-rate detection capabilities. Also notice that the PS/2 protocol is based on 5V level signals. This requires care in choosing which pins can be directly connected to the PIC24. Only the 5V-tolerant digital input pins can be used, which excludes the I/O pins that are multiplexed with the analog-to-digital converter and the comparators inputs.

Input Capture

The first idea that comes to mind is to implement in software a PS/2 serial interface peripheral using the input capture mechanism (Figure 11.4).

There are five input capture modules available on the PIC24FJ128GA010 connected respectively to the IC1–IC5 pins multiplexed on PORTD pins 8, 9, 10, 11 and 12.

Each input capture module is controlled by a single corresponding control register *ICxCON* and works in combination with one of two timers, either Timer2 or Timer3.

One of several possible events can trigger the input capture:

- Rising edge
- Falling edge
- Rising and falling edge
- 4th rising edge
- 16th rising edge.

The current value of the selected timer is recorded and stored in a FIFO buffer to be retrieved by reading the corresponding *ICxBUF* register.

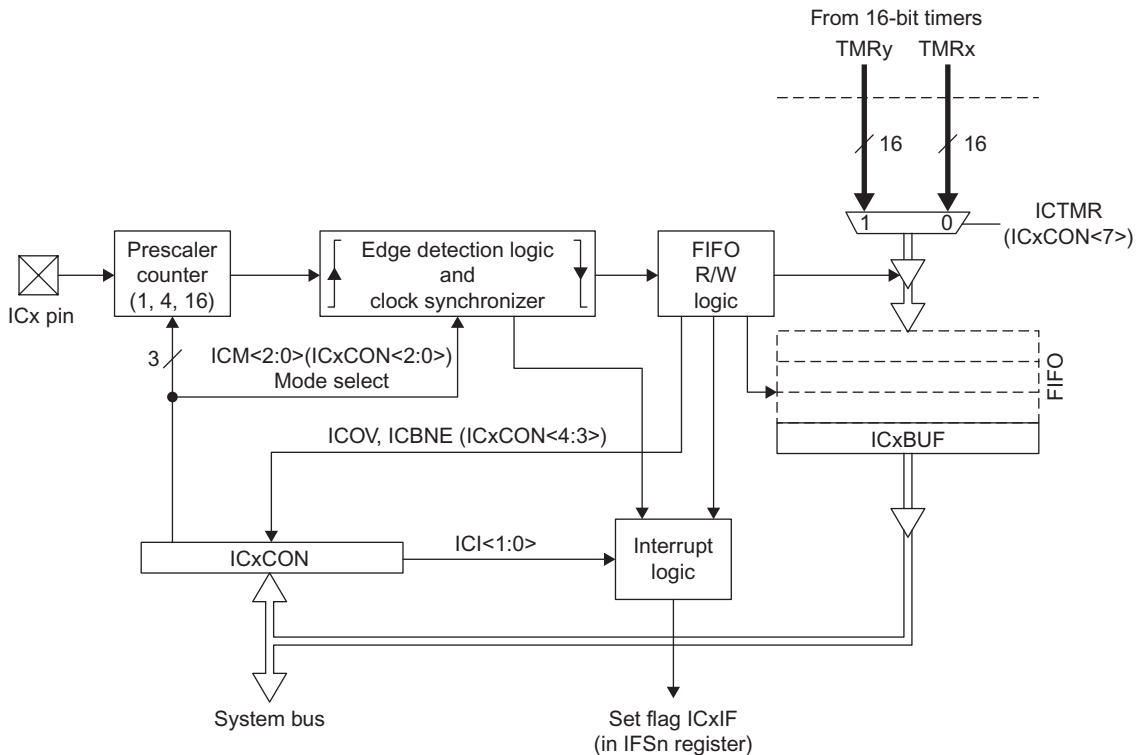


Figure 11.4: Input capture module block diagram

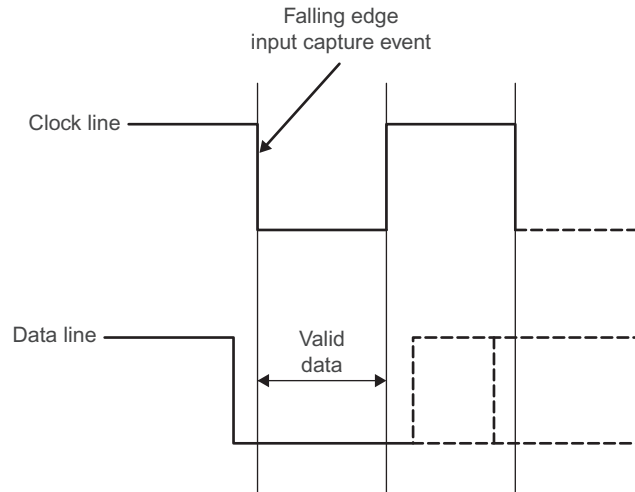


Figure 11.5: PS/2 interface bit timing and the input capture trigger event

In addition to the capture event, an interrupt can be generated after a programmable number of events: each time, every second time, every third time or every fourth time.

To put the input capture peripheral to use and receive the data stream from a PS/2 keyboard we can connect the IC1 input to the clock line and configure the peripheral to generate an interrupt on each and every falling edge of the clock ([Figure 11.5](#)).

After creating a new project, which we will call **11-PS2IC**, and applying our usual template, we can start adding the following initialization code:

```
#define PS2DAT _RG12      // PS2 Data input pin
#define PS2CLK _RD8      // PS2 Clock input pin

void InitKBD( void)
{
    // init I/Os
    _TRISD8 = 1;          // make RD8, IC1 an input pin, clock
    _TRISG12 = 1;         // make RG12 an input pin, data

    // clear the flag
    KBDReady = 0;

    IC1CON = 0x0082;      // TMR2, int. every capt, falling edge
    _IC1IF = 0;           // clear the interrupt flag
    _IC1IE = 1;           // enable the IC1 interrupt

    _T2IF = 0;            // clear the timer interrupt flag
    _T2IE = 1;            // interrupt on (TMR2 not active yet)
} // Init KBD
```

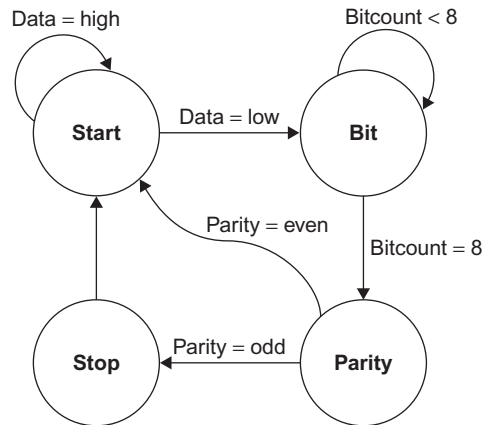


Figure 11.6: The PS/2 receive state machine diagram

We will also need to create an interrupt service routine for the IC1 interrupt vector. This routine will have to operate as a state machine (Figure 11.6) and perform in a sequence the following steps:

1. Verify the presence of a start bit (data line low)
2. Shift in 8 bits of data and compute a parity
3. Verify a valid parity bit
4. Verify the presence of a stop bit (data line high)

If any of the above checks fails, the state machine must reset and return to the start condition. When a valid byte of data is received, we will store it in a buffer – think of it as a mailbox – and a flag will be raised so that the main program or any other “consumer” routine will know a valid key code has been received and is ready to be retrieved. To fetch the code, it will suffice to copy it from the mailbox first and then to clear the flag.

The state machine requires only four states and a counter. All the transitions can be summarized in Table 11.1.

Theoretically, I suppose we should consider this an 11-state machine, counting each time the Bit state is entered with a different *Bitcount* value as a distinct state. But the four-state model works best for an efficient C language implementation. Let’s define a few constants and variables that we will use to maintain the state machine:

```
// definition of the keyboard PS/2 state machine
#define PS2START    0
#define PS2BIT      1
#define PS2PARITY   2
#define PS2STOP     3
```

```
// PS2 KBD state machine and buffers
int PS2State;
int KBDBuf, KCount, KParity;

// mailbox
volatile char KBDReady;
volatile char KBDCode;
```

Table 11.1: PS/2 receive state machine transitions table

State	Conditions	Effect
Start	Data = low	Init Bitcount, Init Parity, Transition to Bit state
Bit	Bitcount < 8	Shift in key code, LSB first (shift right), Update Parity, Increment Bitcount
	Bitcount = 8	Transition to Parity state
Parity	Parity = even	Error. Transition back to Start
	Parity = odd	Transition to Stop
Stop	Data = low	Error. Transition back to Start
	Data = high	Save the key code in buffer, Set flag, Transition to Start

Note

The keyword *volatile* is used as a modifier in a variable declaration to alert the compiler that the content of the variable could change unpredictably as a consequence of an interrupt or other hardware mechanism. We use it here to prevent the compiler from applying any optimization technique (loop extraction, procedural abstraction...) whenever these two variables are used. Admittedly, we could have omitted the detail in this code example, (after all, all optimizations are supposed to be turned off during debugging), only to find ourselves with a big headache in the future, when using this code in a more complex project and trying to squeeze it to get the highest possible performance.

Notice also that *KBDReady* and *KBDcode* are the *only* two variables used in both the interrupt service routine and the main interface code.

The interrupt service routine for the input capture IC1 module can be implemented using a simple switch statement that performs the entire state machine.

```
void _ISR_IC1Interrupt( void)
{ // input capture interrupt service routine
```

```

switch( PS2State){
default:
case PS2START:
    if ( !PS2DAT)
    {
        KCount = 8;           // init bit counter
        KParity = 0;          // init parity check
        PR2 = TMAX*16;         // init timer period
        T2CON = 0x8000;        // enable TMR2
        PS2State = PS2BIT;
    }
    break;

case PS2BIT:
    KBDBuf >>=1;              // shift in data bit
    if ( PS2DAT)
        KBDBuf |= 0x80;
    KParity ^= KBDBuf;         // update parity
    if ( --KCount == 0)        // if all bits read, move on
        PS2State = PS2PARITY;
    break;

case PS2PARITY:
    if ( PS2DAT)
        KParity ^= 0x80;
    if ( KParity & 0x80)        // if parity is odd, next
        PS2State = PS2STOP;
    else
        PS2State = PS2START;  // else restart
    break;

case PS2STOP:
    if ( PS2DAT)               // verify stop bit
    {
        KBDCode = KBDBuf;      // save key code in mailbox
        KBDReady = 1;          // set flag, key available
        T2CON = 0;             // stop the timer
    }
    PS2State = PS2START;
    break;

} // switch state machine

// clear interrupt flag
_IC1IF = 0;

} // IC1 Interrupt

```

Testing the PS/2 Receive Routines

Before we get to test and debug this code, we have to complete the project with a few final touches. Let's package the PS/2 receive routines as a module that we might want to call

PS2IC.c. Remember to include the file in the project; right click in the project window while your mouse is over the Source Files logical folder, and select **Add Existing Item...**

Let's also prepare an include file to publish the accessible function: *initKBD()*, the flag *KBDReady* and the buffer for the received key code *KBDCCode*.

```
/*
** PS2.h
**
** PS/2 keyboard input library using input capture
*/

extern volatile char KBDReady;
extern volatile char KBDCCode;

void InitKBD( void);
```

Note that there is no reason to publish any other detail of the inner workings of the PS2 receiver implementation. This will give us the freedom later to try a few different methods without changing the interface. Save this file as **PS2.h** in the **/include** subdirectory and add it to the project **Header Files** logical folder.

Let's also create a new file, **PS2Test.c**, that will contain the main routine and will use the PS2IC module to test its functionality.

```
/*
** PS2 KBD Test
**
**
*/
#include <config.h>
#include <PS2.h>

main()
{
    InitKBD();           // init state machine
    LATA = 0;
    TRISA = 0;           // PORTA LED outputs

    while (1)
    {
        if ( KBDReady)   // wait for the flag
        {
            PORTA = KBDCCode; // fetch the key code
            KBDReady = 0;     // clear the flag
        }
    } // main loop
} //main
```

Calling the PS/2 keyboard initialization routine will initialize the chosen input capture pins and the state machine, and will enable the interrupt on input capture.

The main loop will wait for the interrupt routine to raise the flag signaling a key code is available. It will fetch the key code and publish it on the row of eight LEDs. Finally, it will clear the flag to signal we are ready to receive a new character.

Now remember to add the file to the project and launch the application by selecting **Run>Run Project** from the main menu. Pressing a key on the keyboard, you will see the corresponding *key code* output in binary on the LED bar.

Where I would like to draw your attention now is to the robustness aspect of the application. What would happen if an additional pulse was received on the clock line (or if one was missed) during the normal operation of the keyboard? In other words, we need to make sure that not only is our state machine capable of handling an ideal sequence, but also that it is able to handle abnormal conditions.

The simplest way to handle possible abnormal conditions involves the introduction of a timeout mechanism, so that if the connection with the keyboard is lost half way through a key code transmission or any number of clock pulses is lost, the state machine does not remain stuck out of sync but instead resets after a specified amount of time elapses.

The code below is meant to replace the `_IC1Interrupt()` service routine core. The few lines of code that have been added are now highlighted in bold.

```
void _ISR _IC1Interrupt( void)
{ // input capture interrupt service routine

    // reset timer on every edge
    TMR2 = 0;

    switch( PS2State){
    default:
    case PS2START:
        if ( !PS2DAT)
        {
            KCount = 8;           // init bit counter
            KParity = 0;          // init parity check
            PR2 = TMAX*16;        // init timer period
            T2CON = 0x8000;       // enable TMR2
            PS2State = PS2BIT;
        }
        break;

    case PS2BIT:
        KBDBuf >>=1;              // shift in data bit
        if ( PS2DAT)
            KBDBuf |= 0x80;
        KParity ^= KBDBuf;        // update parity
        if ( --KCount == 0)       // if all bit read, move on
            PS2State = PS2PARITY;
        break;
    }
```

```

case PS2PARITY:
    if ( PS2DAT)
        KParity ^= 0x80;
    if ( KParity & 0x80)          // if parity is odd, next
        PS2State = PS2STOP;
    else
        PS2State = PS2START;    // else restart
    break;

case PS2STOP:
    if ( PS2DAT)                // verify stop bit
    {
        KBDCODE = KBDBuf;      // save key code in mailbox
        KBDReady = 1;          // set flag, key available
        T2CON = 0;              // stop the timer
    }
    PS2State = PS2START;
    break;

} // switch state machine

// clear interrupt flag
_IC1IF = 0;

} // IC1 Interrupt

```

An additional interrupt service routine must be added to the project to ensure that when the timeout limit is reached the state machine is forced back to the *PS2START* state.

```

void _ISR _T2Interrupt( void)
{ // timeout
    // reset the state machine
    PS2State = PS2START;

    // stop the timer
    T2CON = 0;

    // clear flag and exit
    _T2IF = 0;

} // T2 Interrupt

```

This in turn requires two additional lines of code to be added to the *InitPS2()* function to ensure that an interrupt is generated when too much time passes after the last clock transition without a STOP condition being received.

You should be able to quickly verify that such enhancements do not interfere with the normal operation of the state machine, but verifying that we have properly taken care of the abnormal conditions might require a more elaborate hardware set-up, possibly involving the development of an ad hoc tool or a simulator.

Another Method – Change Notification

While the input capture technique worked all right, there are other options that we might be curious to explore in order to interface efficiently with a PS/2 keyboard. In particular, there is another interesting peripheral available in the PIC24 that could offer an alternative method to implement a PS/2 interface: the *Change Notification (CN)* module. There are as many as 22 I/O pins connected with this module and this can give us some freedom in choosing the ideal input pins for the PS/2 interface while making sure they don't conflict with other functions required in our project or already in use by the Explorer16 board. There are only four control registers associated with the CN module (Figure 11.7). The *CNEN1* and *CNEN2* registers contain the interrupt enable control bits for each of the CN input pins. Setting any of these bits enables a CN interrupt for the corresponding pins. Note that only one interrupt vector is available for the entire CN module, therefore it will be the responsibility of the interrupt service routine to determine which one of the enabled inputs has actually changed.

Each CN pin has a weak pull-up connected to it. The pull-ups act as current sources that are connected to the pin, and eliminate the need for external pull-up resistors when push button or keypad devices are connected. The pull-ups are enabled separately using the *CNPU1* and *CNPU2* registers, which contain the control bits for each of the CN pins. Setting any of the control bits enables the weak pull-ups for the corresponding pins.

In practice, all we need to support the PS/2 interface is only one of the CN inputs connected to the PS2 clock line. The PIC24 weak pull-up will not be necessary in this case as it is already provided by the keyboard.

There are 22 pins to choose from, and we will need to find a CN input that is not shared with the analog-to-digital converter (remember we need a 5V-tolerant input) and is not overlapping with some other peripheral used on the Explorer16 board. This takes a little studying between the device datasheet and the Explorer16 User Guide. But once the input pin is chosen, say CN11 (multiplexed with PortG pin 9, the SS line of the SPI2 module and the PMP module Address line 2), a new initialization routine can be written in just a couple of lines.

```
#define PS2DAT  _RG12          // PS2 Data input pin
#define PS2CLK  _RG9          // PS2 Clock input pin

void InitKBD( void)
{
    // init I/Os
    _TRISG9 = 1;              // make RG9 an input pin
    _TRISG12 = 1;            // make RG12 an input pin

    // clear the flag
    KBDReady = 0;

    CNEN1 = 0x0800;          // enable CN11 - RG9 pin
    _CNIF = 0;               // clear the interrupt flag
    _CNIE = 1;               // enable change notification int
} // Init KBD
```


File name	Addr	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All resets
CNEN1	0060	CN15IE	CN14IE	CN13IE	CN12IE	CN11IE	CN10IE	CN9IE	CN8IE	CN7IE	CN6IE	CN5IE	CN4IE	CN3IE	CN2IE	CN1IE	CN0IE	0000
CNEN2	0062	—	—	—	—	—	—	—	—	—	—	CN21IE	CN20IE	CN19IE	CN18IE	CN17IE	CN16IE	0000
CNPU1	0068	CN15PUE	CN14PUE	CN13PUE	CN12PUE	CN11PUE	CN10PUE	CN9PUE	CN8PUE	CN7PUE	CN6PUE	CN5PUE	CN4PUE	CN3PUE	CN2PUE	CN1PUE	CN0PUE	0000
CNPU2	006A	—	—	—	—	—	—	—	—	—	—	CN21PUE	CN20PUE	CN19PUE	CN18PUE	CN17PUE	CN16PUE	0000

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Figure 11.7: The CN control registers table

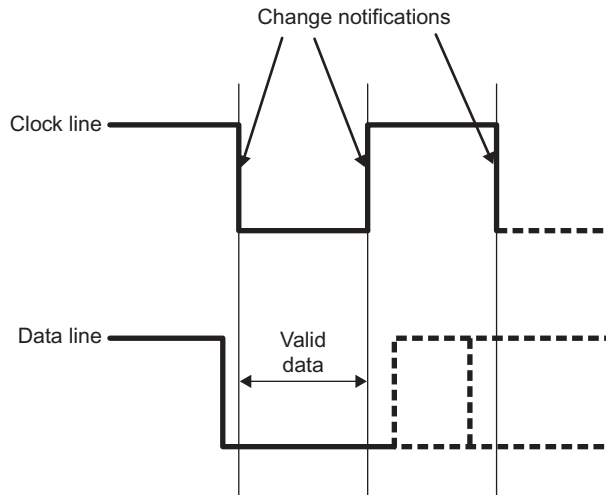


Figure 11.8: PS/2 bit timing and the change notification trigger events

As per the interrupt service routine, we can use exactly the same state machine used in the previous example, adding only a couple of lines of code to make sure that we are looking at a falling edge of the clock line (Figure 11.8).

In fact, when using the input capture module, we could only choose to receive an interrupt on the desired clock edge, while the change notification module will generate an interrupt on both the falling and rising edges. A simple check of the status of the clock line immediately after entering the interrupt service routine will help us to tell the two edges apart.

```
void _ISR_CNInterrupt( void)
{ // change notification interrupt service routine

    // make sure it was a falling edge
    if ( PS2CLK == 0)
    {
        switch( PS2State){
        default:
        case PS2START:
            if ( ! PS2DAT)
            {
```

```
        KCount = 8;           // init bit counter
        KParity = 0;          // init parity check
        PS2State = PS2BIT;
    }
    break;

case PS2BIT:
    KBDBuf >>=1;              // shift in data bit
    if ( PS2DAT)
        KBDBuf |= 0x80;
    KParity ^= KBDBuf;         // update parity
    if ( --KCount == 0)        // if all bits read
        PS2State = PS2PARITY; // move on
    break;

case PS2PARITY:
    if ( PS2DAT)
        KParity ^= 0x80;
    if ( KParity & 0x80)        // if parity is odd
        PS2State = PS2STOP;    // move on
    else
        PS2State = PS2START;
    break;

case PS2STOP:
    if ( PS2DAT)               // verify stop bit
    {
        KBDCode = KBDBuf;      // keycode in mailbox
        KBDReady = 1;          // set flag, available
    }
    PS2State = PS2START;
    break;

    } // switch state machine
} // if falling edge

// clear interrupt flag
_CNIF = 0;

} // CN Interrupt
```

Add the constant and variable declarations already used in the previous example.

```
// definition of the keyboard PS/2 state machine
#define PS2START    0
#define PS2BIT      1
#define PS2PARITY    2
#define PS2STOP     3

// PS2 KBD state machine and buffer
int PS2State;
int KBDBuf, KCount, KParity;
```

```
// mailbox  
volatile char KBDReady;  
volatile char KBDCode;
```

Package it all together in a file that we will call **PS2CN.c**.

The include file is the same was used in the previous example (*PS2.h*) since we are going to offer the same interface. Similar considerations apply to the test program; *PS2Test.c* can be re-used in this new instance.

So all we have to do is to create a new project, called **11-PS2CN**, and add the same source and header files used in *11-PS2IC* while replacing only the *PS2IC.c* file with the new *PS2CN.c* implementation.

Launch the project using the **Run>Run Project** command and verify that the same key codes (seen in the previous exercise) appear on the row of LEDs connected to PORTA.

Note

Similarly to the previous case, you might want to add a timeout mechanism to handle abnormal situations. The modifications to the state machine will be exactly identical to those applied in the previous exercise.

Evaluating Cost

Changing from the *input capture* to the *change notification* method was almost too easy. The two peripherals are extremely powerful and although designed for different purposes, when applied to the task at hand they performed almost identically. In the embedded world though, you should constantly ask yourself whether you could solve a problem using fewer resources, even when, as in this case, there seems to be an abundance. Let's evaluate the real cost of each solution by counting the resources used and their relative scarcity. When using the *input capture* method, we are in fact using one of five IC modules available in the PIC24FJ128GA010 model. These peripherals are designed to operate in conjunction with a timer (Timer2 or Timer3) although we have not used the actual timing information in our application but only the interrupt mechanism associated with the input edge trigger. When using the *change notification* method, we are using only 1 of 22 possible inputs, but we are also taking control of the sole interrupt vector available to this peripheral. In other words, should we need any other input pin to be controlled by the change notification peripheral, we will have to share the interrupt vector, adding latency and complexity to the solution. I would call this a tie.

A Third Method – I/O Polling

There is one more method that we could explore to interface to a PS/2 keyboard. It is the most basic one and it requires the use of a timer, set for a periodic interrupt. Its inputs can be any of the (5V-tolerant) I/O pins of the microcontroller. So, in a way, from a configuration and layout point of view, this method is the most flexible. It is also the most generic as any microcontroller model, even the smallest and most inexpensive, will offer at least one timer module suitable for our purpose. The theory of operation is pretty simple. At regular intervals an interrupt will be generated, set by the value of the period register associated with the chosen timer (Figure 11.9).

We will use Timer4 this time since we never used it before, hence *PR4* will be the corresponding period register. The interrupt service routine (*_T4Interrupt()*) will sample the status of the PS/2 clock line and it will determine whether a falling edge has occurred on the PS/2 clock line over the previous period. When a falling edge is detected, the data line status is read to receive the key code. In order to determine how frequently we should perform the sampling, and therefore identify the optimal value of the *PR4* register, we should look at the shortest amount of time allowed between two edges on the PS/2 clock line. This is determined by the maximum bit-rate specified for the PS/2 interface, which, according to the documentation in our possession, corresponds to about 16 Kbit/s. At that rate, the clock signal can be represented by a square wave with approximately 50% duty cycle, and a period of approximately 62.5 μ s. In other words, the clock line will stay low for a little more than 30 μ s each time a data bit is presented on the data line, and will stay high for approximately the same amount of time, during which the next bit will be shifted out. By setting *PR4* to a value that will make the interrupt period shorter than 30 μ s (say 25 μ s), we can guarantee that the clock line will

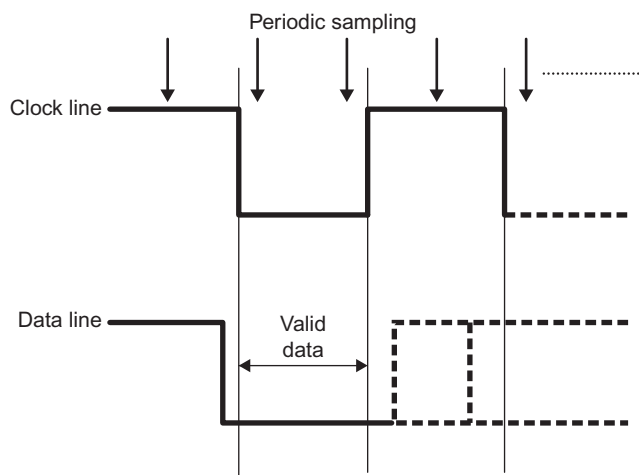


Figure 11.9: PS/2 bit timing, I/O polling sample points

be sampled at least once between two consecutive edges. The keyboard transmission bit rate though could be as slow as 10Kbit/s, giving a maximum distance between edges of about $50\mu\text{s}$. In that case we would be sampling the clock and data lines twice and possibly up to three times between each clock edge. In other words, we will have to build a new state machine to detect the actual occurrence of a falling edge and to keep track properly of the PS/2 clock signal.

This state machine (Figure 11.10) requires only two states and all the transitions can be summarized in Table 11.2.

When a falling edge is detected we can still use the same state machine developed in the previous projects to read the data line (we will refer to it as the *data* state machine). It is important to notice that in this case the value of the data line is not guaranteed to be sampled right after the actual falling edge of the clock line has occurred. To avoid the possibility of reading the data line outside the valid period, it is imperative to sample simultaneously both the clock and the data line. By definition (PS/2 specifications), if the clock line is low, the data can be considered valid. The requirement translates in practice into the necessity to assign both the data and clock inputs to pins of the same port. In our example we will choose to use RG12 (again) for the clock line, and RG13 for the data line. In this way, as soon as we enter the interrupt service routine we can copy PORTG contents to a temporary variable. This will give us an atomic action and perfectly simultaneous sampling of the two lines. Here is the simplest implementation of the Clock state machine illustrated above.

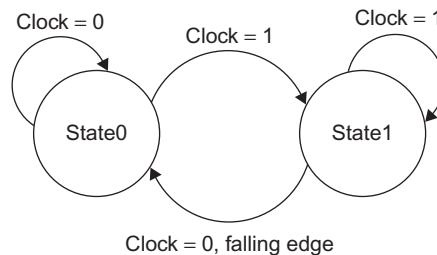


Figure 11.10: Clock-polling state machine graph

Table 11.2: Clock-polling state machine transitions table

State	Conditions	Effect
State0	Clock = 0	Remain in State0
	Clock = 1	Rising edge, Transition to State1
State1	Clock = 1	Remain in State1
	Clock = 0	Falling edge detected Execute the Data state machine Transition to State0

```
#define PS2DAT  _RG12    // PS2 Data input pin
#define PS2CLK  _RG13    // PS2 Clock input pin, Open Drain
#define CLKMASK 0x2000    // mask to detect the clock line bit 13
#define DATMASK 0x1000    // mask to detect the data line bit 12

int KBDBuf, KState;

// mailbox
volatile char KBDReady;
volatile char KBDCode;

void _ISR _T4Interrupt( void)
{
    int PS2IN;

    // sample the inputs, clock and data, at the same time
    PS2IN = PORTG;

    // Keyboard clock state machine
    if ( KState)
    { // previous time clock was high, State1
        if ( !(PS2IN & CLKMASK))        // PS2CLK = 0
        {
            // falling edge detected
            KState = 0;                // transition to State0

            <<<... Insert Data state machine here!>>>

        } // falling edge

        else
        { // clock still high, remain in State1

            } // clock still high
        // State 1
    }
    else
    { // State 0
        if ( PS2IN & CLKMASK)        // PS2CLK = 1
        { // rising edge detected
            KState = 1;                // transition to State1

        } // rising edge

        else
        { // clock still low, remain in State0

            } // clock still low
        // Kstate 0
    }

    // clear the interrupt flag
    _T4IF = 0;
} // T4 Interrupt
```

Thanks to the periodic nature of the polling mechanism we have just developed, we can add robustness to the routine with minimal effort. We can add a counter (*KTimer*) to count

idle loops of both states of the clock state machine. This way we will be able to create a timeout and detect abnormal conditions should the PS/2 keyboard be disconnected during a transmission or should the receive routine lose synchronization for any other reason.

In [Table 11.3](#) the transition table is quickly updated to include the timeout counter.

The new transition table adds only a few instructions to our interrupt service routine, highlighted below in bold:

```
void _ISR _T4Interrupt( void)
{
    int PS2IN;

    // sample the inputs, clock and data, at the same time
    PS2IN = PORTG;

    // Keyboard clock state machine
    if ( KState)
    { // previous time clock was high, State1
        if ( !(PS2IN & CLKMASK)) // PS2CLK = 0
        {
            // falling edge detected
            KState = 0; // transition to State0
            KTimer = KMAX; // restart the counter

            <<<... Insert Data state machine here!>>>
        } // falling edge

        else
        { // clock still high, remain in State1
            KTimer--;
            if ( KTimer ==0) // timeout!
                PS2State = PS2START; // reset state machine
        } // clock still high
    } // State 1

    else
    { // State 0
        if ( PS2IN & CLKMASK) // PS2CLK = 1
        { // rising edge detected
            KState = 1; // transition to State1
        } // rising edge

        else
        { // clock still low, remain in State0
            KTimer--;
            if ( KTimer == 0) // timeout!
                PS2State = PS2START; // reset state machine
        } // clock still low
    } // Kstate 0

    // clear the interrupt flag
    _T4IF = 0;
} // T4 Interrupt
```

Table 11.3: Clock-polling (with timeout), state machine transitions table

State	Conditions	Effect
State0	Clock = 0	Remain in State0 Decrement KTimer If KTimer = 0, error Reset the data state machine
	Clock = 1	Rising edge, Transition to State1
State1	Clock = 1	Remain in State1 Decrement KTimer If KTimer = 0, error Reset the data state machine
	Clock = 0	Falling edge detected Execute the Data state machine Transition to State0 Restart KTimer

Testing the I/O Polling Method

Let's now insert the data state machine from the previous projects, modified to operate on the value sampled in PS2IN at the interrupt service routine entry.

```
switch( PS2State){
default:
case PS2START:
    if ( !(PS2IN & DATMASK))
    {
        KCount = 8;           // init bit counter
        KParity = 0;          // init parity check
        PS2State = PS2BIT;
    }
    break;

case PS2BIT:
    KBDBuf >>=1;              // shift in data bit
    if ( PS2IN & DATMASK)      // PS2DAT
        KBDBuf |= 0x80;
    KParity ^= KBDBuf;         // calculate parity
    if ( --KCount == 0)        // if all bit read
        PS2State = PS2PARITY;
    break;

case PS2PARITY:
    if ( PS2IN & DATMASK)
        KParity ^= 0x80;
    if ( KParity & 0x80)        // if parity is odd
        PS2State = PS2STOP;
```



```

        else
            PS2State = PS2START;
            break;
    case PS2STOP:
        if ( PS2IN & DATMASK)    // verify stop bit
        {
            KBDCode = KBDBuf;    // write in the buffer
            KBDReady = 1;        // set flag
        }
        PS2State = PS2START;
        break;

    } // switch

```

Let's complete this third module with a proper initialization routine.

```

void InitKBD( void)
{
    // init I/Os
    _ODG13 = 1;    // make RG13 open drain output
    _TRISG13 = 1;  // make RG13 an input pin (for now)
    _TRISG12 = 1;  // make RG12 an input pin

    // clear the flag
    KBDReady = 0;

    PR4 = 25 * 16; // 25 us
    T4CON = 0x8000; // T4 on, prescaler 1:1
    _T4IP = 3;      // lower priority
    _T4IF = 0;      // clear interrupt flag
    _T4IE = 1;      // enable interrupt
} // Init KBD

```

This is quite straightforward. Let's save it all in a module we can call **PS2T4.c**. We can then create a new project (**11-PS2T4**), by replacing only the *PS2T4.c* file while keeping the *PS2.h* and *PS2Test.c* files used in the previous two exercises.

Launch the project using the **Run>Run Project** command and verify that we are still correctly receiving all the key codes as in the previous two examples!

Cost and Efficiency of the Solution

Comparing the cost of this solution to the previous two, we realize that the I/O polling approach is the one that gives us the most freedom in choosing the input pins and uses only one resource, a timer and its interrupt vector. The periodic interrupt can also be seamlessly shared with other tasks to form a common time base. The timeout feature was also obtained with minimal effort. In order to implement it in the previous examples we had to use a second timer and its interrupt service routine in addition to the input capture and/or change

notification peripherals and associated interrupts. Looking at the efficiency of the solutions (in terms of CPU overhead), the input capture and the change notification methods appear to have an advantage as interrupts are generated only when an edge is detected on the clock line. The input capture is the best method from this point of view, as we can select precisely the one type of edge we are interested in, that is, the falling edge of the clock line. The I/O polling method appears to require the longest interrupt routine, but the number of lines does not reflect the actual weight of the interrupt service routine. In fact, of the two nested state machines that compose the I/O polling interrupt service routine, only a few instructions are executed at every call, resulting in a very short execution time and minimal overhead.

To verify the actual software overhead imposed by the interrupt service routines we can perform one simple test on each one of the three implementations of the PS/2 interface. I will use the last one as an example. We can allocate one of the I/O pins (one of the LED outputs would be a logical choice) to help us visualize when the microcontroller is inside an interrupt service routine. We can set the pin on entry, and reset it right before exit.

```
void _ISR _T4Interrupt( void)
{
    _RA0 = 1;          // flag up, inside the ISR
    ...
    <<< Interrupt service routine here >>>
    ...
    _RA0 = 0;          // flag down, back to the main
}
```

In [Figure 11.11](#) you can observe the sequence of interrupts captured using a low-cost (USB) logic analyzer (an oscilloscope would do just as well).

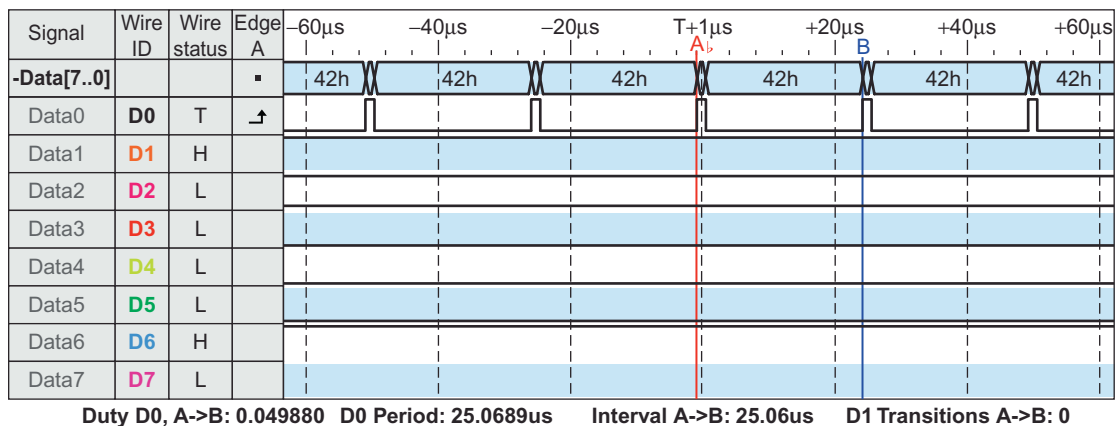


Figure 11.11: Logic analyzer view, measuring the I/O polling period

The logic analyzer I used already provided two automatic measurements:

- The period of the RA0 (*D0*) signal: 25 μ s, as expected.
- The duty cycle of the RA0 (*D0*) signal: <5%.

This last value gives us a direct indication of the computing power absorbed by the PS/2 interface using the polling mechanism; in our case that turns out to be a very small fraction of the total available.

Completing the Interface, Adding a FIFO Buffer

Independently of the solution you will choose out of the three we have explored so far, there are a few more details we need to take care of before we can claim to have completed the interface to the PS/2 keyboard. First of all, we need to add a FIFO buffering mechanism between the PS/2 interface routines and the *consumer* or the main application. So far we have provided only a simple mailbox mechanism that can store only the last key code received. If you investigate further how the PS/2 keyboard protocol works, you will discover that when a single key is pressed and released, a minimum of three (and a maximum of five) key codes are sent to the host. If you consider shift, control and alt key combinations, things get a little more complicated and you realize immediately that the single-byte mailbox is not going to be sufficient. My suggestion is to add at least a 16-byte FIFO buffer. The input to the buffer can be easily integrated with the receiver interrupt service routines so that when a new key code is received it is immediately inserted in the FIFO. The buffer can be declared as an array of characters and two pointers will keep track of the head and tail of the buffer in a circular scheme (Figure 11.12).

```
// circular buffer
char KCB[ KB_SIZE];

// head and tail or write and read pointers
volatile int KBR, KBW;
```

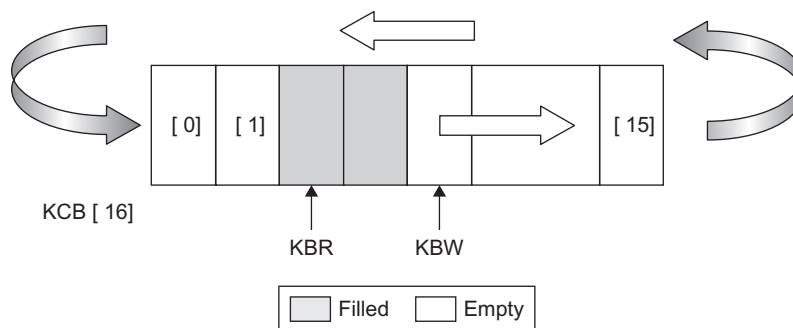


Figure 11.12: Circular buffer FIFO

By following a few simple rules we can keep track of the buffer content:

- The write pointer KBW (or head), marks the first empty location that will receive the next key code
- The read pointer KBR (or tail) marks the first filled location
- When the buffer is empty, KBR and KBW are pointing at the same location
- When the buffer is full, KBW points to the location before KBR
- After reading or writing a character to/from the buffer, the corresponding pointer is incremented
- Upon reaching the end of the array, each pointer will wrap around to the first element of the array

Insert the following snippet of code into the initialization routine:

```
// init the circular buffer pointers
KBR = 0;
KBW = 0;
```

Then update the interrupt routine state machine STOP state:

```
case PS2STOP:
    if ( PS2IN & DATMASK)        // verify stop bit
    {
        KCB[ KBW] = KBDBuf;      // write in buffer
        if ( (KBW+1)%KB_SIZE != KBR)// if full
            KBW++;                // else increment
        KBW %= KB_SIZE;          // wrap around
    }
    PS2State = PS2START;
    break;
```

Notice the use of the `%` operator to give us the remainder of the division by the buffer size. This allows us to keep the pointers wrapping around the circular buffer.

A few considerations are required for fetching key codes from the FIFO buffer. In particular, if we choose the input capture or the change notification methods, we will need to make a new function available (*getKeyCode()*) to replace the mailbox/flag mechanism. The function will return *FALSE* if there are no key codes available in the buffer and *TRUE* if there is at least one key code in the buffer, returning the code itself via a pointer.

```
int getKeyCode( char *c)
{
    if ( KBR == KBW)            // buffer empty
        return FALSE;

    // buffer contains at least one key code
    *c = KCB[ KBR++];           // extract the first key code
    KBR %= KB_SIZE;             // wrap around the pointer
```

```

    return TRUE;
} // getKeyCode

```

Notice that the extraction routine modifies only the read pointer *KBR*, therefore it is safe to perform this operation when the interrupts are enabled. Should an interrupt occur during the extraction, there are two possible scenarios:

1. The buffer was empty: a new key code will be added, but the *getKeyCode()* function will *notice* the available character only at the next call.
2. The buffer was not empty: the interrupt routine will add a new character to the buffer tail, if there is enough room.

In both cases there are no particular concerns of conflicts or dangerous consequences.

If we choose the polling technique, there is one more option we might want to explore. Since the timer interrupt is constantly active, we can use it to perform one more task for us. The idea is to maintain the simple mailbox and flag mechanism for delivering key codes as the interface to the receive routine, and have the interrupt constantly checking the mailbox ready to replenish it with the content from the FIFO. This way we can confine the entire FIFO management to the interrupt service routine, making the buffering task completely transparent and maintaining the simplicity of the mailbox delivery interface. The new and complete interrupt service routine for the polling I/O mechanism is presented below.

```

void _ISR _T4Interrupt( void)
{
    int PS2IN;
    // check if buffer available
    if ( !KBDReady && ( KBR!=KBW))
    {
        KBDCode = KCB[ KBR++];
        KBR %= KB_SIZE;
        KBDReady = 1;           // signal character available
    }
    // _RA0 = 1;
    // sample the inputs clock and data at the same time
    PS2IN = PORTG;

    // Keyboard state machine
    if ( KState)
    {
        // previous time clock was high KState 1
        if ( !(PS2IN & CLKMASK)) // PS2CLK = 0
        {
            // falling edge detected,
            KState = 0;           // transition to State0
            KTimer = KMAX;       // restart the counter

            switch( PS2State){
            default:
            case PS2START:
                if ( !(PS2IN & DATMASK))

```

```
        {
            KCount = 8;           // init bit counter
            KParity = 0;          // init parity check
            PS2State = PS2BIT;
        }
        break;

    case PS2BIT:
        KBDBuf >>=1;             // shift in data bit
        if ( PS2IN & DATMASK)     // PS2DAT
            KBDBuf |= 0x80;
        KParity ^= KBDBuf;        // calculate parity
        if ( --KCount == 0)       // if all bits read
            PS2State = PS2PARITY;
        break;

    case PS2PARITY:
        if ( PS2IN & DATMASK)
            KParity ^= 0x80;
        if ( KParity & 0x80)      // if parity is odd
            PS2State = PS2STOP;
        else
            PS2State = PS2START;
        break;

    case PS2STOP:
        if ( PS2IN & DATMASK)     // verify stop bit
        {
            KCB[ KBW] = KBDBuf;   // write in buffer
            if ( (KBW+1)%KB_SIZE != KBR)// if full
                KBW++;            // else increment
            KBW %= KB_SIZE;       // wrap around
        }
        PS2State = PS2START;
        break;

    } // switch
} // falling edge
else
{ // clock still high, remain in State1
    KTimer--;
    if ( KTimer ==0)
        PS2State = PS2START;
} // clock still high
} // Kstate 1
else
{ // Kstate 0
    if ( PS2IN & CLKMASK)        // PS2CLK = 1
    { // rising edge, transition to State1
        KState = 1;
    } // rising edge
    else
```

```

    { // clock still low, remain in State0
      KTimer--;
      if ( KTimer == 0)
        PS2State = PS2START;
    } // clock still low
  } // Kstate 0

  // clear the interrupt flag
  _T4IF = 0;
  // _RA0 = 0;
} // T4 Interrupt

```

Completing the Interface, Performing Key Codes Decoding

So far we have been talking generically about *key codes* and you might have assumed that they match the ASCII codes for each key: say if you press the “A” key on the keyboard you would expect the corresponding ASCII code (*0x41*) to be sent. In reality, things are a little more complex. In order to provide the flexibility required to support the many international (language-specific) keyboard layouts, each key is assigned a numerical value or *scan code*. These values are still related to the original implementation of the keyboard firmware (which used an 8048 microcontroller) on the first IBM PC keyboard circa 1980. The translation from *scan codes* to specific ASCII characters happens at a higher level (performed by Windows keyboard drivers). For historical reasons, there are at least three different and partially compatible *scan code sets*. Fortunately, by default, all keyboards support the scan code set #2, which is the one we will focus on in the following.

Each time a key is pressed (any key, including a shift or control key) the scan code associated to it is sent to the host; this is called the *make code*. But also, as soon as the same key is released, a new sequence of scan codes is sent to the host; this is called the *break code*. The break code is typically composed of the same scan code but prefixed with the numeric value *0xF0*. Some keys can have a 2-byte-long make code (typically the Ctrl, Alt and arrows) and consequently the break code is 3 bytes long (see [Table 11.4](#)).

Table 11.4: Examples of make and break codes used in scan code set 2 (default)

Key	Make Code	Break Code
“A”	1C	F0,1C
“5”	2E	F0,2E
“F10”	09	F0,09
Right Arrow	E0, 74	E0, F0, 74
Right “Ctrl”	E0, 14	E0, F0, 14

In the following, in order to translate the scan codes into proper ASCII characters, we will use a table mapping the scan codes of a basic US English keyboard:

```
// PS2 keyboard codes (standard set #2)
const char keyCodes[128]={
    0,  F9,  0,  F5,  F3,  F1,  F2, F12,  //00
    0, F10,  F8,  F6,  F4, TAB, '`',  0,  //08
    0,  0, L_SHFT, 0, L_CTRL, 'q', '1',  0,  //10
    0,  0, 'z', 's', 'a', 'w', '2',  0,  //18
    0, 'c', 'x', 'd', 'e', '4', '3',  0,  //20
    0, ' ', 'v', 'f', 't', 'r', '5',  0,  //28
    0, 'n', 'b', 'h', 'g', 'y', '6',  0,  //30
    0,  0, 'm', 'j', 'u', '7', '8',  0,  //38
    0, ' ', 'k', 'i', 'o', '0', '9',  0,  //40
    0, ' ', '/', 'l', ';', 'p', '-',  0,  //48
    0,  0, '\\',  0, '[', '=',  0,  0,  //50
    CAPS, R_SHFT, ENTER, ']',  0, 0x5c,  0,  0,  //58
    0,  0,  0,  0,  0,  0, BKSP,  0,  //60
    0, '1',  0, '4', '7',  0,  0,  0,  //68
    0, ' ', '2', '5', '6', '8', ESC, NUM,  //70
    F11, '1', '3', '-', '*', '9',  0,  0  //78
};
```

Notice that the array has been declared as *const* so that it will be allocated in program memory space to save precious RAM space.

It will also be convenient to have available a similar table for the shift function of each key.

```
const char keySCodes[128] = {
    0,  F9,  0,  F5,  F3,  F1,  F2, F12,  //00
    0, F10,  F8,  F6,  F4, TAB, '~',  0,  //08
    0,  0, L_SHFT, 0, L_CTRL, 'Q', '!',  0,  //10
    0,  0, 'Z', 'S', 'A', 'W', '@',  0,  //18
    0, 'C', 'X', 'D', 'E', '$', '#',  0,  //20
    0, ' ', 'V', 'F', 'T', 'R', '%',  0,  //28
    0, 'N', 'B', 'H', 'G', 'Y', '^',  0,  //30
    0,  0, 'M', 'J', 'U', '&', '*',  0,  //38
    0, '<', 'K', 'I', 'O', ')', '(',  0,  //40
    0, '>', '?', 'L', ':', 'P', '_',  0,  //48
    0,  0, '\\',  0, '{', '1',  0,  0,  //50
    CAPS, R_SHFT, ENTER, '}',  0, '|',  0,  0,  //58
    0,  0,  0,  0,  0,  0, BKSP,  0,  //60
    0, '1',  0, '4', '7',  0,  0,  0,  //68
    0, ' ', '2', '5', '6', '8', ESC, NUM,  //70
    F11, '1', '3', '-', '*', '9',  0,  0  //78
};
```

For all the ASCII characters, the translation is straightforward, but we will have to assign special values to the function, shift and control keys. Only a few of them will find a corresponding code in the ASCII set.


```
// special function characters
#define TAB      0x9
#define BKSP     0x8
#define ENTER    0xd
#define ESC      0x1b
```

For all the others we will have to create our own conventions, or until we have a use for them we might just ignore them and assign them a common code (0).

```
#define L_SHFT   0x12
#define R_SHFT   0x12
#define CAPS     0x58
#define L_CTRL   0x0
#define NUM      0x0
#define F1       0x0
#define F2       0x0
#define F3       0x0
#define F4       0x0
#define F5       0x0
#define F6       0x0
#define F7       0x0
#define F8       0x0
#define F9       0x0
#define F10      0x0
#define F11      0x0
#define F12      0x0
```

You might want to add these definitions to the **PS2.h** file.

The *getC()* function (below) can be added to the **PS2T4.c** (or any of the previous implementations) to perform the basic translations of scan codes into ASCII, automatically taking care of the shift and caps key status.

```
int CapsFlag=0;

char getC( void)
{
    unsigned char c;

    while( 1)
    {
        while( !KBDReady);      // wait for a key to be pressed
        // check if it is a break code
        while (KBDCODE == (char) 0xf0)
        { // consume the break code
            KBDReady = 0;
            // wait for a new key code
            while ( !KBDReady);
            // check if the shift button is released
            if ( KBDCODE == L_SHFT)
```

```
        CapsFlag = 0;
        // and discard it
        KBDReady = 0;
        // wait for the next key
        while ( !KBDReady);
    }
    // check for special keys
    if ( KBDCODE == L_SHFT)
    {
        CapsFlag = 1;
        KBDReady = 0;
    }
    else if ( KBDCODE == CAPS)
    {
        CapsFlag = !CapsFlag;
        KBDReady = 0;
    }

    else // translate into an ASCII code
    {
        if ( CapsFlag)
            c = keySCodes[KBDCODE%128];
        else
            c = keyCodes[KBDCODE%128];
        break;
    }
}
// consume the current character
KBDReady = 0;

return ( c);
} // getC
```

Post-Flight Briefing

In this lesson we have learned how to interface to a PS/2 computer keyboard, exploring as many as three alternative methods. This gave us the perfect opportunity to exercise two new peripheral modules: the input capture and the change notification. We also discussed methods to implement a FIFO buffer and polished a little our interrupt management skills. Throughout the entire lesson our focus has been constantly on balancing the use of resources and the performance offered by each solution.

Tips & Tricks

Each PS/2 keyboard has an internal FIFO buffer 16 key codes deep. This allows the keyboard to accumulate the user input even when the host is not ready to receive. The host, as we mentioned at the very beginning of this chapter, has the option to stall the communication by pulling low the clock line at any given point in time (for at least 100 μ s) and can hold it low for the desired period of time. When the clock line is released, the keyboard will resume

transmissions. It will also re-transmit the last key code, if it had been interrupted, and will offload its FIFO buffer.

To exercise our right to stall the keyboard transmissions as a host, we have to control the clock line with an output using an open drain driver. Fortunately this is easy with the PIC24 thanks to its configurable I/O PORT modules. In fact, each I/O port (*PORT_x*) has an associated control register (*ODCx*) that can individually configure each pin output driver to operate in open-drain mode.

Note that this feature is extremely useful in general to interface PIC24 outputs to any 5V device.

In our example turning the PS/2 clock line into an open-drain output would require only a few lines of code.

```
_ODG13 = 1;    // configure PORTG pin 13 output open-drain
_LATG13 = 1;   // initially let the output in pull up
_TRISG13 = 0;  // enable the output driver
```

Note that, as usual for all PIC[®] microcontrollers, even if a pin is configured as an output its current status can still be read as an input. So there is no reason to switch continuously between input and output when we alternate between sending commands to and receiving characters from the keyboard.

Exercises

- Add a function to send commands to the keyboard to control the status LEDs and set the key repeat rate.
- Replace the *stdio.h* library input function *read()* to redirect the keyboard input from the *stdin* stream.
- Add support for a PS/2 mouse interface.

Books

- Anderson F., 2003. *Flying the Mountains*, McGraw-Hill, New York, NY.
Flying the mountains requires extra caution and preparation. This could be the next challenge after you have completed your private pilot license.

Links

- <http://www.computer-engineering.org/>
This is an excellent website where you will find concise documentation on the PS/2 keyboard and mouse interface.

The Dark Screen

I have always liked driving the car at night. Generally, there is less traffic, the air is always cooler and, unless I am really tired, the lights of the vehicles in the other direction never really bothered me much. But when my instructor proposed a first cross-country flight at night, I got a little worried. The idea of staring at a windshield filled with pitch-black void ... was a little frightening, I have to admit. The actual experience a week later converted me forever. Sure, night flying is a bit more serious stuff than the usual around-the-pattern practice, there is more careful planning involved, but it is just so rewarding. Flying over an uninhabited area fills the screen with so many stars that a city boy, like me, has hardly ever seen – it feels like flying a starship to another solar system. Flying over or near a large city transforms the grey and uniform spread of concrete of alternating parking lots and housing developments into a wonderful show of lights – it's like Christmas as far as the eye can see. Turns out, the screen is never really dark, it's a big show and it is on every night.

Flight Plan

In this lesson we will consider techniques to interface to a TV screen or, for that matter, any display that can accept a standard composite video signal. It will be a good excuse to use new features of several peripheral modules of the PIC24 and review new programming techniques. Our first project objective will be to get a nice dark screen (a well-synchronized video frame), but we will soon see it fill up with several entertaining graphical applications.

The Flight

There are many different formats and standards today in use in the world of video, but perhaps the oldest and most common one is the so called “composite” video format. This is what was originally used by the very first TV sets ever to appear in the consumer market. Today it represents the minimum common denominator of every video display whether this is a modern high-definition, flat-screen TV of the latest generation, a DVD player, or a VHS tape recorder. All video devices are based on the same basic concept; that is, the image is “painted” one line at a time, starting from the top left corner of the screen and moving horizontally to the right edge, then quickly and invisibly jumping back to the left edge at a lower position and painting a second line, and so on and on, in a zig-zag motion until the entire screen has been scanned ([Figure 12.1](#)). Then the process repeats and the entire image is

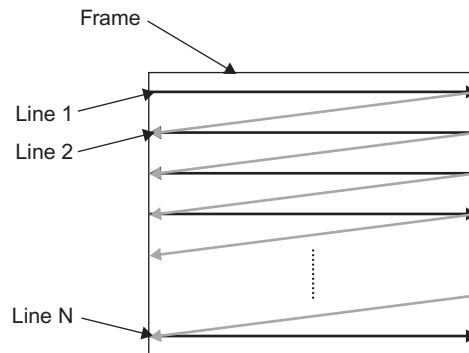


Figure 12.1: Video image scanning

Table 12.1: International video standards examples

	USA	Europe, Asia	France and Others
Standard	NTSC	PAL	SECAM
Frames per second	29.97*	25	25
Number of lines	525	625	625

Note: NTSC used to be 30 frames per second, but the introduction of the new color standard changed it to 29.97, to accommodate a specific frequency used by the *color sub-carrier* crystal oscillator.

refreshed fast enough for our eyes to be tricked into believing that the entire image is present at the same time and, if there is motion, it is fluid and continuous.

In different parts of the world, slightly different and therefore incompatible systems have been developed over the years, but the basic mechanism remains the same. What changes eventually is the number of lines composing the image, the refreshing frequency, and the way the color information is encoded.

Table 12.1 illustrates three of the most commonly used video standards adopted in the USA, Europe and Asia. All those standards encode the “luminance” information (that is, the underlying black and white image) together with synchronization information in a similarly defined composite signal.

The name *composite* is used to describe the fact that this video signal combines and transmits three different pieces of information in one: the actual luminance information and both horizontal and vertical synchronization information (Figure 12.2).

The horizontal line signal is composed of:

- The horizontal synchronization pulse, used by the display to identify the beginning of each line
- The so-called “back porch”, which creates a dark frame around the image

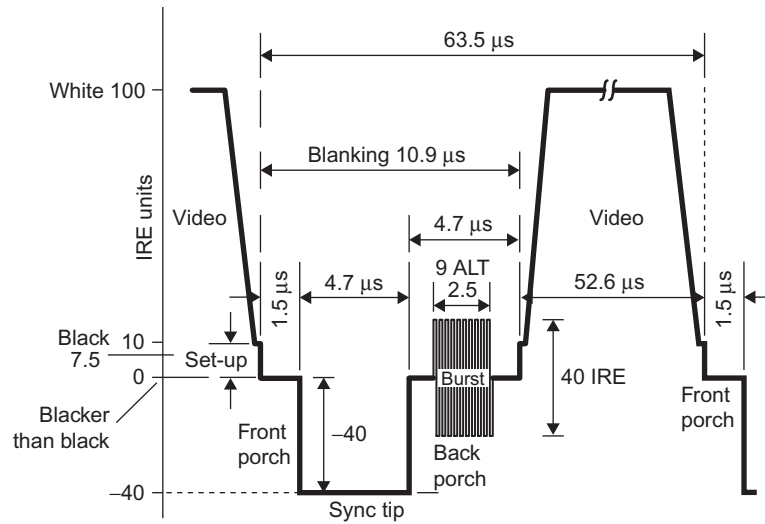


Figure 12.2: NTSC composite signal, horizontal line detail

- The actual line luminosity signal: the higher the voltage the more luminous the point
- The so called “front porch”, producing the right edge of the image

The color information is transmitted separately, modulated on a high-frequency sub-carrier. The three main standards differ significantly in the way they encode the color information but, for our purposes, it will be easy to ignore the problem altogether to obtain a simple black and white display output.

All the above standard systems use a technique called *interlacing* to provide a (relatively) high-resolution output while requiring a reduced bandwidth. In practice, only half the lines are transmitted and painted on the screen in each frame. Alternate frames present only the odd or the even lines composing the picture so that the entire image content is effectively updated only at half the refresh rate (25 Hz and 30 Hz respectively). This is effective for typical TV broadcasting but can produce an annoying flicker when text and especially horizontal lines are displayed, as is often the case in computer monitor applications. For this reason, modern computer displays use *progressive* instead of interlaced scanning. Most modern TV sets, and especially those using LCD and plasma technologies, perform a de-interlacing of the received broadcast image. In our project we will avoid interlacing as well, sacrificing half of the image resolution in favor of a more stable and readable display output. In other words, we will transmit frames of 262 lines (for NTSC) at the double rate of 60 frames per second. Readers who have easier access to PAL or SECAM TV sets/monitors will find it relatively easy to modify the project for a 312-line resolution with a refresh rate of 50 frames per second. A complete video frame signal is represented in [Figure 12.3](#).

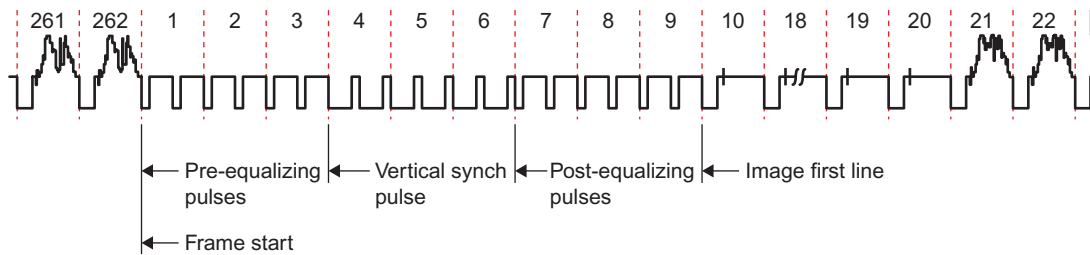


Figure 12.3: A complete video frame signal

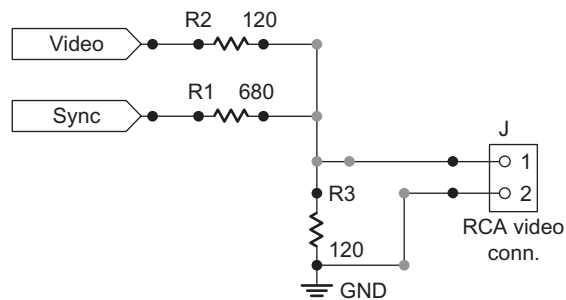


Figure 12.4: Simple HW interface for NTSC video output

Notice how, out of the total number of lines composing each frame, three line periods are filled by prolonged synchronization pulses to provide the vertical synchronization information, identifying the beginning of each new frame. They are preceded and followed by groups of three additional lines, referred to as the pre- and post-equalization lines.

Generating the Composite Video Signal

If we limit the scope of the project to generating a simple black and white image (no gray shades, no color) and a non-interlaced image as well, we can simplify the hardware and software requirements of our project considerably. In particular, the hardware interface can be reduced to just three resistors of appropriate value connected to two digital I/O pins ([Figure 12.4](#)). One of the I/O pins will generate the synchronization pulses and the other I/O pin will produce the actual luminance signal.

The values of the three resistors must be selected so that the relative amplitudes of the luminance and synchronization signals are close to the standard NTSC/PAL specifications, the signal total amplitude is close to 1 V peak to peak and the output impedance of the circuit is approximately 75 ohms. With the standard resistor values shown in the previous picture, we can satisfy such requirements and generate the three basic signal levels required to produce a black and white image, shown in [Table 12.2](#) and [Figure 12.5](#).

Table 12.2: Synch and video signal composition

Signal feature	Sync	Video
SYNCH PULSE	0	0
BLACK LEVEL	1	0
WHITE LEVEL	1	1

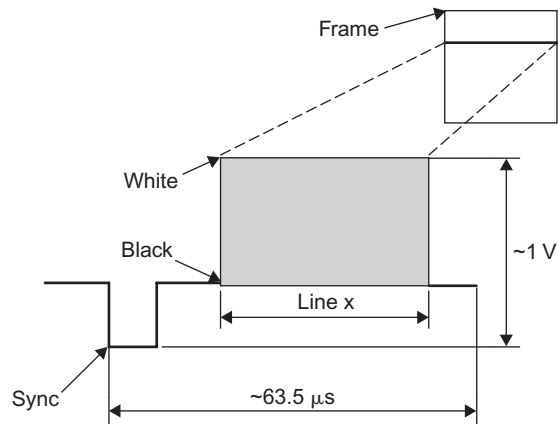


Figure 12.5: Simplified NTSC composite signal

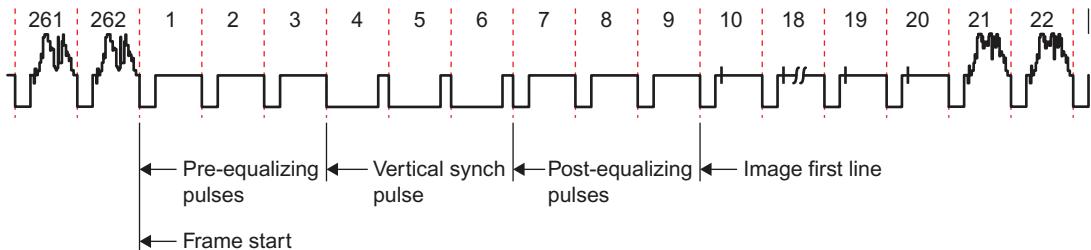


Figure 12.6: Simplified NTSC video frame (non-interlaced)

Since we are not going to use the interlacing technique, we can also simplify the pre-equalization, vertical synchronization and post-equalization pulses by producing a single horizontal synchronization pulse per each period as illustrated in [Figure 12.6](#).

The problem of generating a complete video output signal can now be reduced to (once more) a simple state machine that can be driven by a fixed period time base produced by a single timer interrupt. The state machine will be quite trivial as each state will be associated to one type of line composing the frame, and it will repeat for a fixed number of times before transitioning to the next state ([Figure 12.7](#)).

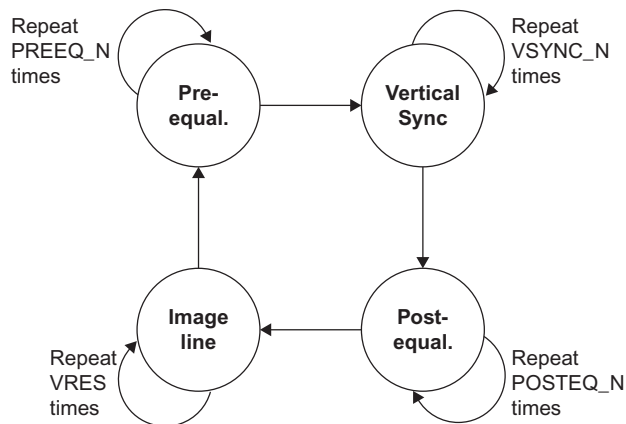


Figure 12.7: Video state machine graph

Table 12.3: Video state machine transitions table

State	Repeat	Transition to
Pre-equal.	PREEQ_N times	Vertical sync
Vertical sync	3 times	Post-equal.
Post-equal.	POSTEQ_N times	Image line
Image line	VRES times	Pre-equal.

A simple table such as [Table 12.3](#) will help describe the transitions from each state.

While the number of vertical synchronization lines is fixed and prescribed by the NTSC/PAL video standard, the number of lines effectively composing the image inside each frame is up to us to define (within limits of course). Although in theory we could use all of the lines available to display the largest possible amount of data on the screen, we will have to consider some practical limitations, and in particular the RAM memory available to store the video image inside the PIC24FJ128GA010 microcontroller. These limitations will dictate a specific number (*VRES*) of lines to be used for the image while all the remaining (up to the NTSC/PAL standard line count) will be left blank.

Note

The PIC24F GA1 and GB1 series of microcontrollers expand the RAM memory available to 16 Kbytes. The PIC24F DA2 and GB2 series expand the RAM memory to a whopping 96 Kbytes. For those devices, the number of lines is limited only by the standard (NTSC or PAL) used.

In practice, if *V_LINES* is the total number of lines composing a standard video frame and *VRES* is the desired vertical resolution, we will determine a value for *PREEQ_N* and *POSTEQ_N* as follows:

```
#define V_LINES 262           // number of lines composing a frame
#define VSYNC_N 3            // V sync lines

// count the number of remaining black lines top+bottom
#define VBLANK_N (V_LINES - VRES - VSYNC_N)

#define PREEQ_N VBLANK_N / 2    // pre equalization + bottom blank lines
#define POSTEQ_N VBLANK_N - PREEQ_N // post equ + top blank lines
```

If we choose Timer3 to generate the time base we can initialize its period register, *PR3*, to produce an interrupt with the prescribed period and create an interrupt service routine where we will place the state machine. Here is a skeleton of the interrupt service routine on which we will grow the complete video generator logic.

```
// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N, POSTEQ_N, VRES, PREEQ_N};

void _ISRFAST_T3Interrupt( void)
{
    // advance the state machine
    if ( --VCount == 0)
    {
        VCount = VC[ VState];
        VState = VS[ VState];
    }

    // vertical state machine
    switch ( VState) {

        case SV_PREEQ:
            // prepare for the new frame
            ...
            break;

        case SV_SYNC:
            // vertical sync pulse
            ...
            break;

        case SV_POSTEQ:
            // horizontal sync pulse
            ...
            break;

        default:
            case SV_LINE:
                ...
    } //switch
```

```
// clear the interrupt flag
_T3IF = 0;

} // T3Interrupt
```

To generate the sync pulse there are several options; we can use:

- short delay loops using a counter
- a second timer, and associated interrupt service routine
- an output compare module and the associated interrupt service routines

The first solution is probably the simplest to code, but has the clear disadvantage of wasting a large number of processor cycles ($4.5\mu\text{s} \times 16 \text{ cycles per second} = 72 \text{ cycles}$). If repeated each horizontal line period ($63.5\mu\text{s}$ or about 1018 cycles) this would add up to as much as 7% of the total processing power available.

The second solution is clearly more efficient, and by now we have ample experience in using timer interrupts and their interrupt service routines to execute small state machines.

The third solution involves the use of a new peripheral we have not yet explored in the previous chapters and deserves a little more attention.

Using the Output Compare Modules

The PIC24FJ128GA010 microcontroller has five output compare peripheral modules (Figure 12.8) that can be used for a variety of applications, including single pulse generation, continuous pulse generation and pulse width modulation (PWM). Each module can be associated with one of two 16-bit timers (Timer2 or Timer3) and has one output pin that can be configured to toggle and produce rising or falling edges if necessary. Most importantly, each module has an associated and independent interrupt vector.

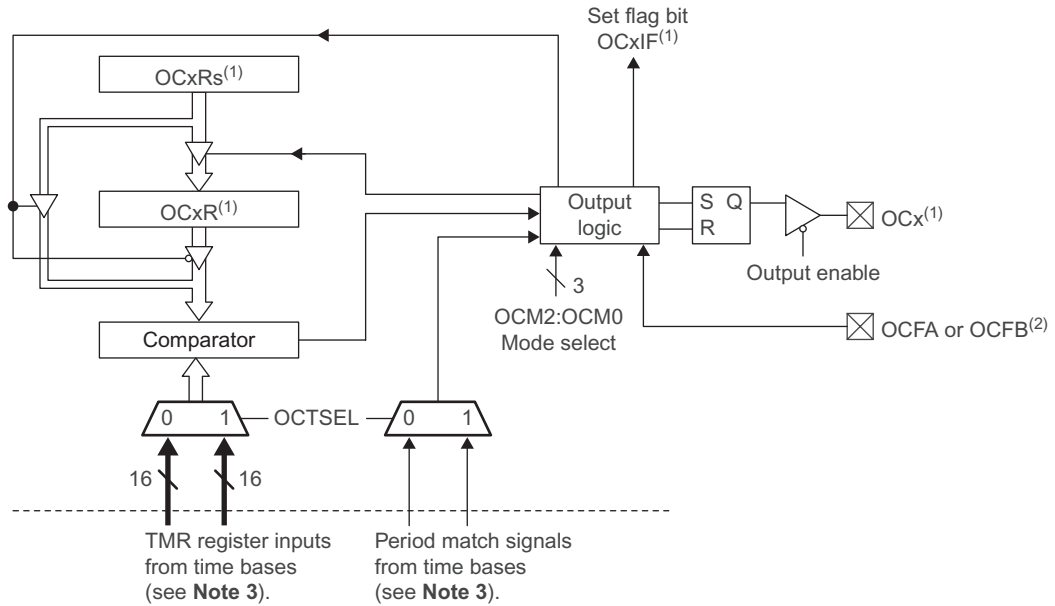
When used in continuous pulse mode specifically, the *OCxR* register can be used to determine the instant (relative to the value of the selected timer) when the interrupt event will be triggered and, if desired, an output pin will be set/reset or toggled as required.

The *OCxCON* register (Figure 12.9) is the only configuration register required to control each of the output compare modules.

In our application the output compare mechanism can be quite useful as there are two precise instants where we need to take action: the end of the horizontal synchronization pulse, when generating a pre-/post-equalization or a vertical synchronization line (Figure 12.10), and the end of the back porch, where the actual image begins.

We can use one of the output compare modules (*OC3*) to produce the complete waveform. This way the output pin (*RD2*) can be connected directly to the sync output signal.

The *OC3CON* control register will be set so to activate the output compare module in the continuous pulse mode (*OCM = 101*) and to use Timer3 as the reference time base (*OCTSEL = 1*).



- Note 1:** Where “x” is shown, reference is made to the registers associated with the respective output compare channels.
Note 2: OCFA pin controls OC1–OC4 channels.
Note 3: Each output compare channel can use one of two selectable time bases. Refer to the device data sheet for the time bases associated with the module.

Figure 12.8: Output compare module block diagram

Upper byte:							
U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
—	—	OCSIDL	—	—	—	—	—
bit 15							bit 8
Lower byte:							
U-0	U-0	U-0	R-0 HC	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	OCFLT	OCTSEL	OCM2	OCM1	OCM0
bit 7							bit 0

Figure 12.9: Register OCxCON Detail

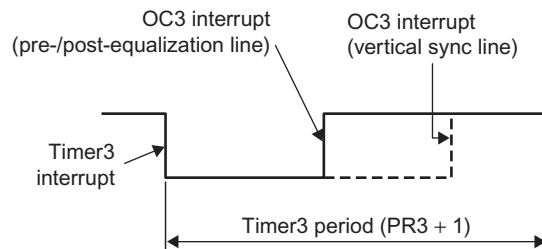


Figure 12.10: Interrupt sequence for a synchronization line

We will also initialize the *OC3R* register with the selected timing value depending on the type of line (state of the state machine) as follows:

```
// vertical state machine
switch ( VState) {
    case SV_PREEQ:
        // prepare for the new frame
        ...
        break;

    case SV_SYNC:
        // vertical sync pulse
        OC3R = LINE_T - HSYNC_T - BPORCH_T;
        break;

    case SV_POSTEQ:
        // horizontal sync pulse
        OC3R = HSYNC_T;
        break;

    ...
}
```

When generating a video line, we will use a second output compare module (OC4), in single pulse mode, to mark the end of the back porch and the corresponding interrupt service routine will be used to initiate the streaming of the actual image line (Figure 12.11).

```
case SV_LINE:
    // activate OC4 for the SPI loading
    OC4R = HSYNC_T + BPORCH_T;
    OC4CON = 0x0009;    // single pulse mode
    ...
    break;
```

Memory Allocation

So far we have been working on the generation of the synchronization signal of the two composing the video waveform: synch and video. The latter gets active only once we begin generating one of the lines containing the actual image. Toggling the video I/O, we can

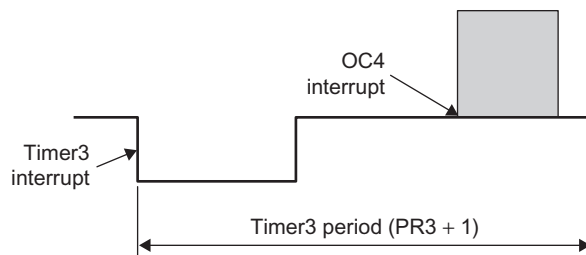


Figure 12.11: Interrupt sequence for a video line

alternate segments of the line that will be painted in white (1) or black (0). Since the NTSC/PAL standards specify a maximum video (luminance) signal bandwidth of 4.2 MHz, and the space between front and back porch is 52 μ s wide, it follows that the maximum number of alternate segments (cycles) of black and white we can display is 218 (52×4.2). In other words, our maximum theoretical horizontal resolution is 436 pixels per line, assuming the screen is completely used from side to side. The maximum vertical resolution is given by the total number of lines composing each standard frame minus the minimum number of equalization and vertical synchronization lines. If we were to generate the largest possible image (PAL gives the worst-case scenario) it would be composed of an array of 300×436 pixels or 130,800 pixels. If one bit is used to represent each pixel that would require us to allocate an array of 16,350 bytes, way too large to fit in the 8 Kbytes available within the PIC24FJ128GA010 RAM. In practice, while it is nice to be able to generate a high-resolution output, we need to make sure that the image will fit in the available RAM memory and possibly leave enough space for an actual application to run comfortably, allowing for adequate room for the stack and application variables. While there is an almost infinite number of possible combinations of the horizontal and vertical resolution values that will give an acceptable memory size, there are two considerations that we will use to pick the perfect numbers: making the horizontal resolution a multiple of 16 will make the math involved in determining the position of a pixel in the memory map easier assuming we will use an array of integers. Also, making the two resolution values in an approximate ratio of 4:3 will avoid image geometrical distortions, in other words, circles drawn on the screen will look like circles rather than ovals.

Choosing a horizontal resolution of 256 pixels (HRES) and a vertical resolution of 192 lines (VRES) we obtain an image memory requirement of 6,144 bytes ($256 \times 192/8$), leaving as much as 2,048 bytes for stack and application variables.

Using the MPLAB[®] C compiler for the PIC24 we can easily allocate a single array of integers (grouping 16 pixels at a time in each word) to contain the entire image memory map. But we need to make sure that the entire contents of the array are addressable and this is not possible if we declare it simply as a *near* variable (the default when using the small memory model). Near variables must be found within the first 8 Kbytes of the data addressing space but this space also includes the special function registers area. The best way to avoid an allocation error message is to explicitly declare the video memory map with a *far* attribute.

```
#define _FAR __attribute__(( far))
int _FAR VMap[VRES * (HRES/16)];
```

Note

`_FAR` is a useful macro that we will need in several future projects. It deserves its place inside the `EX16.h` include file.

This ensures that access to the array elements is performed via pointers, something we would have done anyway both when reading and when writing to the array.

Image Serialization

If each image line is represented in memory in the *VMap* array by a row of 16 integers, we will need to serially output each bit (pixel) in a timely fashion in the short amount of time (52 μ s) between the back and the front porch part of the composite video waveform.

In other words, we will need to set or reset the chosen video output pin with a new pixel value every 200ns or better. This would translate into about three machine cycles between pixels, way too fast for a simple shift loop even if we plan on coding it directly in assembly. Worse, even assuming we managed to squeeze the loop in so tight, we would end up using an enormous percentage of the processing power for the video generation, leaving very few processor cycles for the main application (<18% in the best case). Fortunately, there is one peripheral of the PIC24 that can help us efficiently serialize the image data: it's the SPI synchronous serial communication module.

In a previous chapter we used the SPI2 port to communicate with a serial EEPROM memory. In that chapter we noted how the SPI module is composed of a simple shift register that can be clocked by an external clock signal (when in slave mode) or by an internal clock (when in master mode). In our new project we can use the SPI1 module as a master connecting only the SDO (serial data output) directly to the video pin of the hardware interface. We can leave the SDI (data input) unused and disable the SCK (clock output) and SS (slave select) pins. Among the many advanced features of the PIC24 SPI module there are two that fit particularly well with our video application: the ability to operate in 16-bit mode and a powerful eight-level-deep FIFO buffer. Operating in 16-bit mode we can practically double the transfer speed of data between the image memory map and the SPI module. Enabling the eight-level-deep FIFO buffer we can load up to 128 pixels (8 words \times 16 bits) at a time into the SPI buffer and quickly return from the interrupt service routine only to return 25 μ s later for a second final load, maximizing the efficiency of the video generator by requiring only two short bursts of activity for each image line.

We can now write the interrupt service routine for the OC4 module to produce the actual image line output.

```
void _ISRFAST _OC4Interrupt( void)
{
    // load SPI FIFO with 8 x 16-bit words = 128 pixels
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
```

```

    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;
    SPI1BUF = *VPtr++;

    if ( --HCount > 0)
    {
        // activate again in time for the next SPI load
        OC4R += ( PIX_T * 7 * 16);
        OC4CON = 0x0009;    // single pulse mode
    }

    // clear the interrupt flag
    _OC4IF = 0;
} // OC4Interrupt

```

Notice how the interrupt service routine reconfigures the OC4 module for a second burst (the second half of the image line) after loading the first 128 pixel data in the SPI buffer.

Now that we have identified all the pieces of the puzzle we can write the complete initialization routine for all the modules required by the video generator.

```

void InitVideo( void)
{
    // 1. set the priority levels
    _T3IP = 4;           // this is the default value anyway
    _OC4IP = 4;

    // 2. configure Timer3 module
    TMR3 = 0;           // clear the timer
    PR3 = LINE_T;       // set the period register to video line
    T3CON = 0x8000;     // enabled, prescaler 1:1, internal clock

    // 3. configure OC3 in continuous pulse mode
    OC3R = HSYNC_T;
    OC3RS = 0;
    OC3CON = 0x000d;

    // 4. enable Timer3/OC4 interrupts, clear the flags
    _OC4IF = 0; _OC4IE = 1;
    _T3IF = 0; _T3IE = 1;

    // 5. init the SPI1
    // Master, 16 bit, disable SCK, disable SS
    if ( PIX_T == 2)
        SPI1CON1 = 0x143B; // prescaler 1:3
    else
        SPI1CON1 = 0x1437; // prescaler 1:2
    SPI1CON2 = 0x0001;     // Enhanced mode, 8x FIFO
    SPI1STAT = 0x8000;     // enable SPI port

    // 6. init the vertical sync state machine
    VState = SV_PREEQ;
    VCount = PREEQ_N;
} // InitVideo

```


Notice how the parameter *PIX_T* can be used to select different SPI clock pre-scaling values so to adapt to different horizontal resolution requirements. Setting *PIX_T* = 3 will provide each pixel three clock cycles for a total of 187.5 ns, very close to the 200 ns value previously calculated for the 256 pixel horizontal resolution, ideal in a 4:3 ratio. Alternatively a setting of *PIX_T* = 2 will provide the optimal setting for a screen with a 16:9 ratio.

Building the Video Module

We can now complete the coding of the entire video state machine, adding all the definitions and pin assignments necessary, including provisions for the selection of the target video standard NTSC or PAL using a conditional preprocessor directive *#ifdef*, *#else*, *#endif*.

```
/*
** graphic.c
** NTSC/PAL Video Generator using:
**     T3      Interrupt for main timing
**     OC3     Horizontal Synchronization pulse
**     OC4     SPI buffer reload timing
**     SPI1    pixel serialization (8x16-bit FIFO)
**
** Pin assignments:
**     SD01 - RF8 - VIDEO
**     OC3  - RD2 - SYNC32 (requires AV16/32 board)
*/
#include <EX16.h>
#include <font.h>
#include <graphic.h>

// timing definitions for video vertical state machine
#ifdef PAL
    #define V_LINES 312    // total number of lines composing a frame
    #define LINE_T 1024    // total number of Tcy in a line

#else // default NTSC configuration
    #define V_LINES 252    // total number of lines composing a frame
    #define LINE_T 1016    // total number of Tcy in a line
#endif

#define VSYNC_N 3        // V sync lines

// count the number of remaining black lines top+bottom
#define VBLANK_N (V_LINES - VRES - VSYNC_N)

#define PREEQ_N VBLANK_N / 2        // pre equalization + bottom blank lines
#define POSTEQ_N VBLANK_N - PREEQ_N // post equalization + top blank lines

// definition of the vertical sync state machine
#define SV_PREEQ 0
#define SV_SYNC 1
#define SV_POSTEQ 2
#define SV_LINE 3
```

```

// timing definitions for video horizontal state machine
#define HSYNC_T 90      // Tcy in a horizontal synch pulse
#define BPORCH_T 90     // Tcy in a back porch
// Tcy in each pixel, valid values are only 2 or 3
#define PIX_T 2         // Aspect Ratio 2 = 16:9, 3 = 4:3

int _FAR VMap[VRES * (HRES/16)];

volatile int *VPtr;
volatile int HCount, VCount, VState, HState;

// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N, POSTEQ_N, VRES, PREEQ_N};

void _ISRFAST _T3Interrupt( void)
{
    // advance the state machine
    if ( --VCount == 0)
    {
        VCount = VC[ VState];
        VState = VS[ VState];
    }

    // vertical state machine
    switch ( VState) {
        case SV_PREEQ: // 0
            // prepare for the new frame
            VPtr = VMap;
            break;

        case SV_SYNC: // 1
            // vertical sync pulse
            OC3R = LINE_T - HSYNC_T - BPORCH_T;
            break;

        case SV_POSTEQ: // 2
            // horizontal sync pulse
            OC3R = HSYNC_T;
            break;

        default:
        case SV_LINE: // 3
            // activate OC4 for the SPI loading
            OC4R = HSYNC_T + BPORCH_T;
            OC4CON = 0x0009; // single pulse mode
            HCount = HRES/128; // loads 8x16 bits at a time
            break;
    } //switch

    // clear the interrupt flag
    _T3IF = 0;
} // T3Interrupt

```

To make it a complete library module we will need to add the interrupt service routine presented for the OC4 output compare module illustrated in the previous section of this chapter as well as a couple of additional accessory functions.

```
void ClearScreen( void)
{
    int i, j;
    int *v;

    v = (int *)&VMap[0];

    // clear the screen
    for ( i=0; i < (VRES*( HRES/16)); i++)
        *v++ = 0;
} // ClearScreen

void HaltVideo( void)
{
    T3CONbits.TON = 0;    // turn off the vertical state machine
} // HaltVideo

void SynchV( void)
{
    while ( VState != SV_PREEQ);
} // SynchV
```

In particular, *ClearScreen()* will be useful to initialize the image memory map, the *VMap* array. The function *HaltVideo()* will be useful to suspend the video generation should an important task/application require 100% of the PIC24 processing power. The *SynchV()* function can be used to synchronize a task to the video generator. This function will return only when the video generator has finished *painting* the last line of the screen. This can be useful for graphic applications to minimize flicker and/or provide more fluid scrolling and motion.

Save all of the above functions in a file called **graphic.c** in the **/lib** subdirectory, and add this file to a new project called **12-Video**.

Then create a new file and add the following definitions:

```
/*
** graphic.h
**
** Composite Video Graphic library
*/
#define VRES    192    // desired vertical resolution
#define HRES    256    // desired horizontal resolution (pixel)

void InitVideo( void);
void HaltVideo( void);
void ClearScreen( void);
void SynchV( void);

extern int VMap[HRES/16*VRES];
```

Save this file as **graphic.h** in the **/include** subdirectory and add it to the same project.

Notice how the horizontal resolution and vertical resolution values are the only two parameters exposed. Within reasonable limits, due to timing constraints and the many considerations exposed in the previous sections, they can be changed to adapt to specific application needs. The state machine and all other mechanisms of the video generator module will adapt their timing as a consequence.

Testing the Video Generator

In order to test the video generator module we have just completed we need only a few lines of code for our *main()* function.

```
/*
** GraphicTest.c
**
** Testing the basic graphic module
*/
#include <config.h>
#include <graphic.h>

main()
{
    // initializations
    ClearScreen(); // init the video map
    InitVideo();   // start the video state machine

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

Save the file as **GraphicTest.c** and add it to the project.

If you have an Explorer16 board, this is the time to take out the soldering iron and connect the three resistors and a standard RCA video jack to the small prototyping area in the top right corner of the board. Alternatively, if you feel your electronic hobbyist skills are up to the task, you could develop a small PCB for a daughter board that would fit in the expansion connectors of the Explorer16.

For your convenience I have added the simple video circuit to the AV16/32 PICTail Plus board that you can purchase as a kit (including all discrete components and connectors) from the companion website www.flyingpic24.com.

Connecting an oscilloscope or a low-cost logic analyzer to the RD2 pin (Sync), you will be able to observe the output of the OC3 capture and compare module. When setting the appropriate time base, 50µs per division, and using the automatic trigger, you should be able

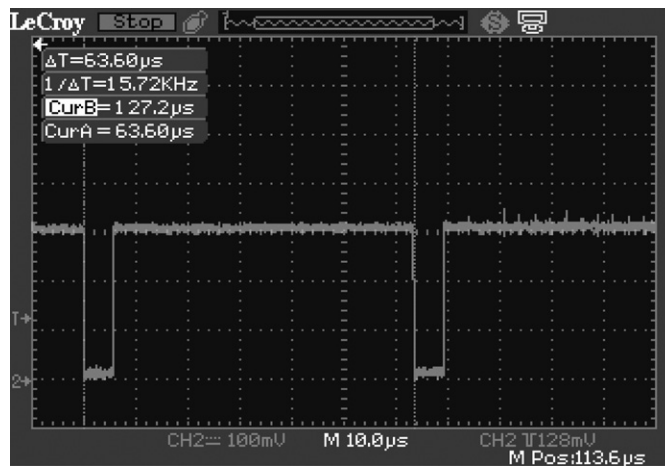


Figure 12.12: Zoomed view of a single pre-equalization line

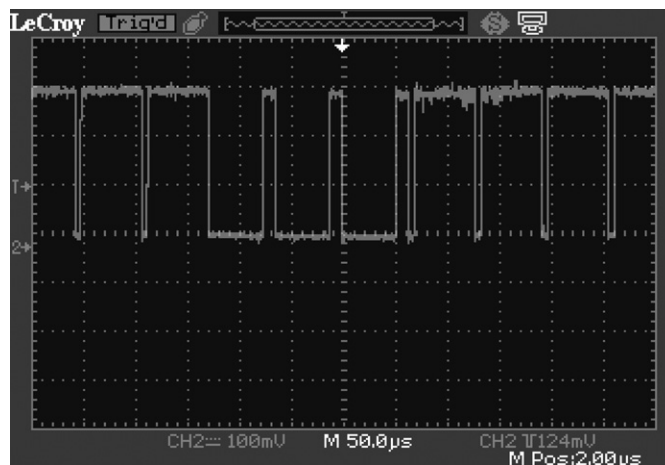


Figure 12.13: Vertical sync pulses

to capture a number of horizontal synchronization pulses. Zooming to $10\mu\text{s}$ per division you will verify that each pulse is approximately $63.5\mu\text{s}$ long in NTSC mode or $64\mu\text{s}$ long if in PAL mode (Figure 12.12).

In order to capture the vertical synchronization pulses though, you will need to set the trigger to detect a falling edge and filter only for pulses that are larger than $40\mu\text{s}$ or so (Figure 12.13).

Measuring Performance

Since the video generator module uses two different sources of interrupt and a state machine with four states, it might be interesting to get an idea of the actual processor overhead

involved. Expanding on the technique used in the previous lesson, we will add a few lines of code to turn a pin of PORTA, *RA0*, into a flag to indicate when we are executing inside the Timer3 interrupt service routine and *RA1* into a flag to indicate when we are executing inside the OC4 interrupt service routine.

```
void _ISRFAST _T3Interrupt( void)
{
    _RA0=1;
    ...
    _RA0=0;
} // T3Interrupt

void _ISRFAST _OC4Interrupt( void)
{
    _RA1=1;
    ...
    _RA1=0;
} // OC4Interrupt

main()
{
    TRISA = 0;
    ...
}
```

After recompiling and connecting a low-cost logic analyzer tool to pins RA0 and RA1, we can zoom in a single horizontal line period (Figure 12.14). Using two cursors to isolate the desired interval, we can measure its exact length ($63.57\mu\text{s}$) and the duty cycle of the two signals that when added together will give us a good estimate of the CPU overhead. In the case of an image line, the worst case where three interrupts occur in each period, this value turns out to be approx 21% ($0.019 + 0.191 \times 100$) of the processor time, a remarkably small amount.

The Dark Screen

Playing with the oscilloscope and a logic analyzer can be entertaining for a little while, but I am sure at this point you will feel an itch for the real thing! Test the video interface on a real TV screen or any other device capable of receiving a composite video signal.

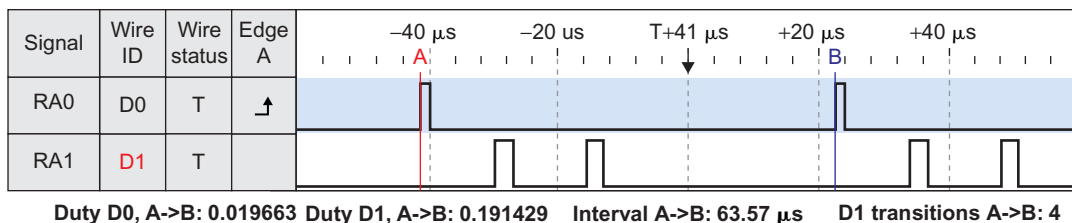


Figure 12.14: Logic analyzer capture, measuring performance



Figure 12.15: The dark screen

The experience will be breathtaking!

Or not! (Figure 12.15.) In fact, if you wire all the connections just right, when you power up the Explorer16 board what you are going to be staring at is just a blank (or I should say black) screen. Sure, this is an achievement; in fact, this already means that a lot of things are working right as both the horizontal and vertical synchronization signals are being decoded correctly by the TV set and a nice and uniform black background is being displayed.

A Test Pattern

To spice things up we should start filling that video array with something worth looking at, possibly something simple that can give us an immediate visual feedback on the proper functioning of the video generator.

Let's create a new test program: **GraphicTest2.c**, as follows:

```
/*
** GraphicTest2.c
**
** Testing the basic graphic module
** A pattern
*/
#include <config.h>
#include <graphic.h>

main()
{
    int x, y;

    // 1. fill the video memory map with a pattern
    for( y=0; y<VRES; y++)
```

```

    for (x=0; x<HRES/16; x++)
        VMap[y*16 + x]= y;

// 2. start the video state machine
InitVideo();

// 3. main loop
while( 1)
{
    } // main loop
} // main

```

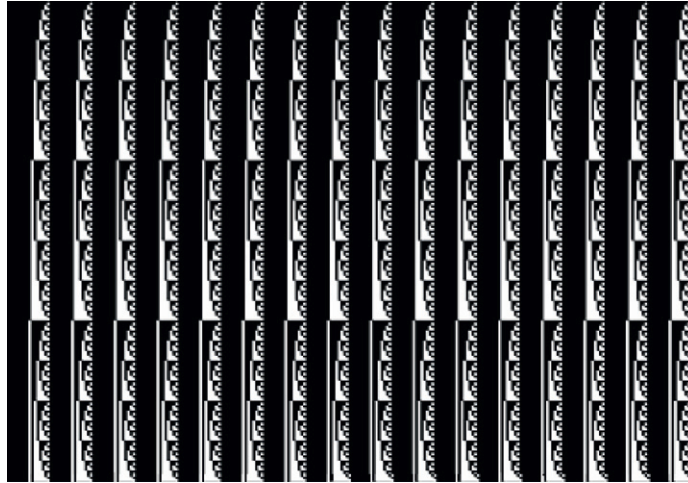


Figure 12.16: A screen capture of the video output generated with the test pattern

Instead of calling the *ClearScreen()* function, this time we will use two nested *for* loops to initialize the *VMap* array. The external *y* loop counts the vertical lines, the internal *x* loop moves horizontally, filling the 16 words (each containing 16 bits) with the same value: the line counter *y*. In other words, on the first line each 16-bit word will be assigned the value 0, on the second line each word will be assigned the value 1 and so on until the last line (192), where each word will be assigned the value 191 (0xBF in hexadecimal).

If you launch the new project, you should be able to see the pattern in [Figure 12.16](#).

Even in its simplicity, there is a lot we can learn from observing the test pattern. First of all, we notice that each word is visually represented on the screen in binary with the most significant bit presented on the left. This is a consequence of the order used by the SPI module to shift out bits: that is in fact MSB first. Secondly, we can verify that the last row contains the expected pattern: 0xBF. So we know that all rows of the memory map are being displayed. Finally we can appreciate the detail of the image. Different output devices (TV sets, projectors, LCD panels...) will be able to lock the image more or less effectively

and/or will be able to present a sharper image depending on the actual display resolution and their input stages bandwidth. In general, you should be able to appreciate how the PIC24 can generate effectively straight vertical lines. This is not a trivial achievement. In fact, for each pixel to align properly, row after row in a straight vertical line, there must be an absolutely jitter-less (deterministic) response to the timer interrupts, a notable characteristic of all PIC[®] microcontroller architectures.

This does not mean that on the largest screens you will not be able to notice small imperfections here and there, as small echoes and possibly minor visual artifacts in the output image. Realistically, the simple three-resistor interface can only take us so far.

Ultimately, the entire composite video signal interface could be blamed for a lower-quality output. As you might know, S-video, VGA and most other video interfaces keep luminance and synchronization signals separate to provide a more stable and clean picture.

Plotting

Now that we are reassured about the proper functioning of the graphic display module, we can start focusing more on generating the actual images onto the memory map. The first natural step is to develop a function that allows us to light up one pixel at a precise coordinate pair (x,y) on the screen. The first thing to do is derive from the y coordinate the line number. If the x and y coordinates are based on the traditional Cartesian plane representation with the origin located in the bottom left corner of the screen, we need to invert the address before accessing the memory map, so that the first row in the memory map corresponds to the y maximum coordinate, $VRES-1$ or 191, while the last row in the memory map corresponds to the y coordinate, 0. Also, since our memory map is organized in rows of 16 words, we will need to multiply the resulting line number by 16 to obtain the address of the first word on the given line. This can be obtained with the following expression:

```
VMap[ (VRES-1 -y) *16]
```

Pixels are grouped in 16-bit words, so to resolve the x coordinate we need first to identify the word that will contain the desired pixel. A simple division by 16 will give us the word offset on the line. Adding the offset to the line address as calculated above will provide us with the complete word address inside the memory map:

```
VMap[ (VRES-1 -y) *16 + (x/16)]
```

In order to optimize the address calculation we can make use of shift operations to perform the multiplication and divisions as follows:

```
VMap[ (VRES-1 -y) << 4 + (x>>4)]
```

To identify the bit position inside the word corresponding to the required pixel, we can use the remainder of the division of x by 16, or more efficiently we can mask out the lower 4

bits of the x coordinate. Since we want to turn the pixel on, we will need to perform a binary OR operation with an appropriate mask that has a single bit set in the corresponding pixel position. Remembering that the display puts the MSB of each word to the left (the SPI module shifts bits MSb first) we can build the mask with the following expression:

```
( 0x8000 >> ( x & 0xf))
```

Putting it all together we obtain the core *Plot()* function:

```
VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
```

As a final touch we can add *clipping*, this is a simple safety check to make sure that the coordinates we are given are in fact valid and within the current screen map limits.

```
void Plot( unsigned x, unsigned y)
{
    if ((x<HRES) && (y<VRES) )
        VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
} // Plot
```

By defining the *x* and *y* parameters as *unsigned* integers we guarantee that negative values will be discarded too as they will be considered large integers outside the screen resolution.

A Starry Night

To test the newly developed *Plot()* function we will modify the test program to become the new **GraphicTest3.c** source file. This time, we will use the *pseudo-random* number generator functions, available in the standard C library *stdlib.h*, to produce random *x* and *y* coordinate pairs for a thousand points. This way we will test both the *Plot()* function and, in a way, the random generator itself with the following simple code:

```
/*
** GraphicTest3.c
**
** Testing the basic graphic module
** Starry Night
*/
#include <config.h>
#include <graphic.h>
#include <stdlib.h>

void Plot( unsigned x, unsigned y)
{
    if ((x<HRES) && (y<VRES) )
        VMap[ ((VRES-1-y)<<4) + (x>>4)] |= (0x8000 >> (x & 0xf));
} // Plot

main()
{
    int i;
```

```
// initializations
ClearScreen(); // init the video map
InitVideo();   // start the video state machine
srand(13);     // seed the pseudo random number generator

for( i=0; i<1000; i++)
{
    plot( rand()%HRES, rand()%VRES);
}

// main loop
while( 1);

} // main
```

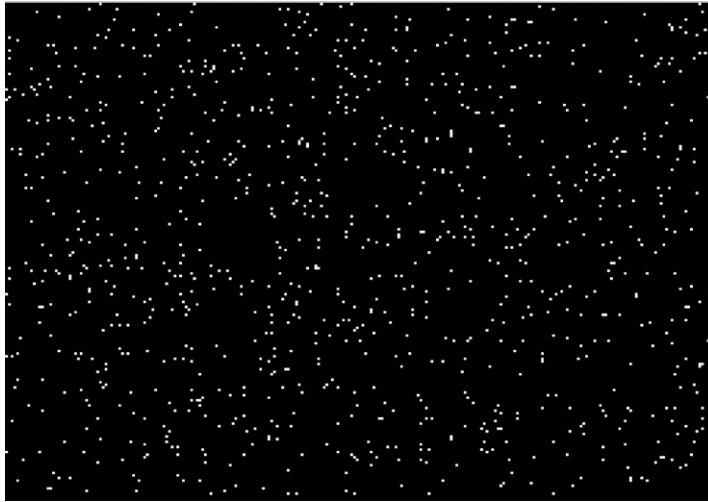


Figure 12.17: Screen capture, plotting a starry night

The output on your video display should look like a nice starry night as in the screen shot captured in [Figure 12.17](#).

A starry night it is, but not a realistic one you will notice, as there is no recognizable trace of any increased density of stars around a belt, in other words there is no Milky Way! This is a good thing! This is a simple proof that our pseudo-random number generator is in fact doing the job it is supposed to.

We can now add the plot function to the *graphic.c* module. Also remember to add the prototype to the *graphic.h* module so that in the following exercises we will be able to use it.

```
void Plot( unsigned, unsigned);
```

Line Drawing

The next obvious step is drawing lines, or it would be better to say line segments. Granted, horizontal and vertical line segments are not a problem, for a simple *for* loop can take care

of them. Drawing oblique lines is a completely different thing. We could start with the basic formula for the line between two points that you may remember from the old school days:

$$y = y_0 + (y_1 - y_0) / (x_1 - x_0) * (x - x_0)$$

where (x_0, y_0) and (x_1, y_1) are the coordinates respectively of two generic points that belong to the line.

This formula gives us, for any given value of x , a corresponding y coordinate, so we might be tempted to use it in a loop for each discrete value of x between the starting and ending point of the line, as in the following example:

```
/*
** LineTest.c
**
** Testing the basic graphic module
** Drawing lines
*/
#include <config.h>
#include <graphic.h>

main()
{
    int x;
    float x0 = 10, y0 = 20, x1 = 200, y1 = 150, x2 = 20, y2 = 150;

    // initializations
    ClearScreen(); // init the video map
    InitVideo();   // start the video state machine

    // draw an oblique line (x0,y0) - (x1,y1)
    for( x=x0; x<x1; x++)
        Plot( x, y0+(y1-y0)/(x1-x0)* (x-x0));

    // draw a second (steeper) line (x0,y0) - (x2,y2)
    for( x=x0; x<x2; x++)
        Plot( x, y0+(y2-y0)/(x2-x0)* (x-x0));

    // main loop
    while( 1);
} // main
```

The output produced is an acceptably continuous segment only for the first (shallower) line where the horizontal distance $(x_1 - x_0)$ is greater than the vertical distance $(y_1 - y_0)$. In the second and much steeper line the dots appear disconnected and we are clearly unhappy with the result ([Figure 12.18](#)). Also, we had to perform floating-point arithmetic, a computationally expensive proposition compared to integer arithmetic, as we have seen in Chapter 4.

Bresenham Algorithm

Back in 1962, when working at IBM in the San Jose development lab, Jack E. Bresenham developed a line-drawing algorithm that uses exclusively integer arithmetic and is today

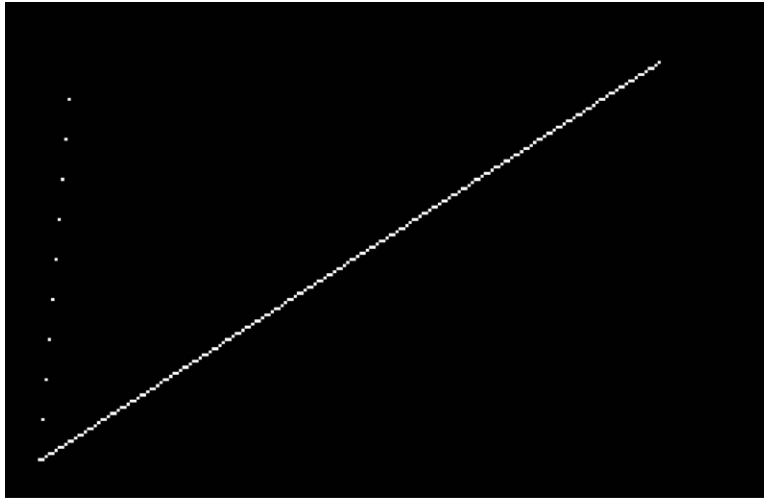


Figure 12.18: Screen capture, drawing oblique lines

considered the foundation of any computer graphic program. Its approach is based on three optimization *tricks*:

1. Reduction of the drawing direction to a single case (left to right)
2. Reduction of the line steepness to the single case where the horizontal distance is the greatest
3. Multiplying both side of the equation by the horizontal distance (*deltax*) to obtain only integer quantities

The resulting line-drawing code is compact and extremely efficient; here is an adaptation for our video module:

```
#define abs( a)      (((a)> 0) ? (a) : -(a))

void Line( int x0, int y0, int x1, int y1)
{
    int steep, t ;
    int deltax, deltay, error;
    int x, y;
    int ystep;

    steep = ( abs(y1 - y0) > abs(x1 - x0));

    if ( steep )
    { // swap x and y
        t = x0; x0 = y0; y0 = t;
        t = x1; x1 = y1; y1 = t;
    }
    if (x0 > x1)
    { // swap ends
        t = x0; x0 = x1; x1 = t;
        t = y0; y0 = y1; y1 = t;
    }
}
```

```

deltax = x1 - x0;
deltay = abs(y1 - y0);
error = 0;
y = y0;
if (y0 < y1) ystep = 1; else ystep = -1;
for(x = x0; x < x1; x++)
{
    if ( steep) Plot(y,x); else Plot(x,y);
    error += deltay;
    if ( (error<<1) >= deltax)
    {
        y += ystep;
        error -= deltax;
    } // if
} // for
} // Line

```

We can add this function to the *graphic.c* module and its prototype to the include file *graphic.h*.

To test the efficiency of the Bresenham algorithm, we can replace the main source file in the *12-Video* project with the new **Bresenham.c**. This example code will first draw a frame around the screen and then it will exercise the line-drawing routine to produce one a hundred lines from randomly generated coordinate pairs. The main loop also contains a check for the S3 button (the leftmost button on the bottom of the Explorer16 demo board) to be pressed before the screen is cleared again and a new set of random lines is drawn on the screen (Figure 12.19).

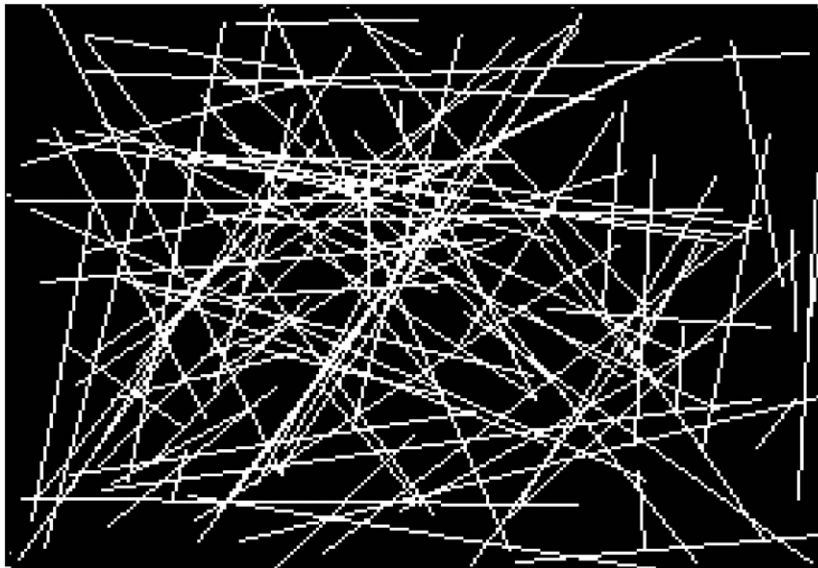


Figure 12.19: Screen capture, Bresenham line-drawing algorithm

```
/*
** Bresenham.c
**
** Testing the graphic module
** Bresenham line drawing algorithm
*/
#include <config.h>
#include <graphic.h>
#include <stdlib.h>

main()
{
    int i;

    // initializations
    InitVideo();    // start the state machines
    srand( 12);

    // main loop
    while( 1)
    {
        ClearScreen();
        Line( 0, 0, 0, VRES-1);
        Line( 0, VRES-1, HRES-1, VRES-1);
        Line( HRES-1, VRES-1, HRES-1, 0);
        Line( 0, 0, HRES-1, 0);

        for( i = 0; i<100; i++)
            Line( rand()%HRES, rand()%VRES, rand()%HRES, rand()%VRES);
        while( 1)
        {
            if ( !_RD6)
                break;
        } // wait for a key

    } // main loop
} // main
```

You will be impressed by the speed of the line-drawing algorithm. Even when increasing the number of lines drawn to batches of one thousand the performance of the PIC24 will be apparent.

Plotting Math Functions

With the graphic module completed we can now start exploring some interesting applications that can take full advantage of its visualization capabilities. One classic application could be plotting a graph based on data logged from a sensor or, more simply for our demonstration purposes, calculated on the fly from a given math function.

For example, let's assume the function is a sinusoid (with a twist) as in the following:

$$y(x) = x * \sin(x)$$

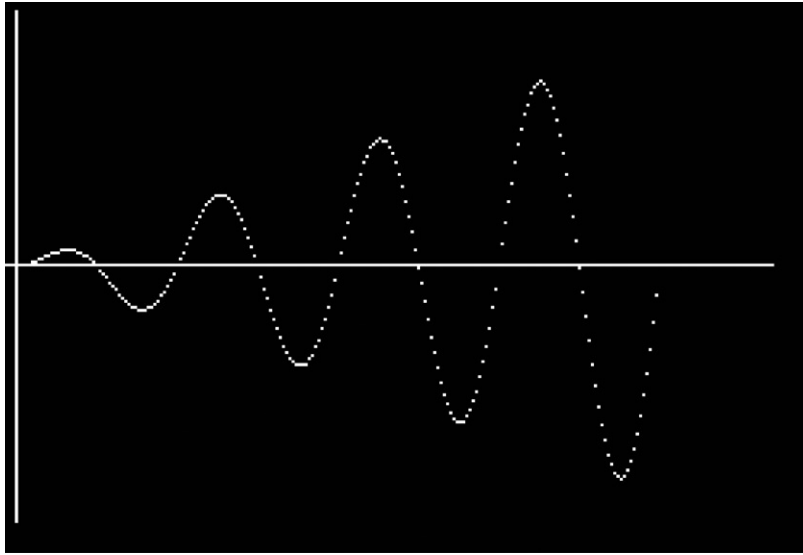


Figure 12.20: Screen capture, a sinusoidal function graph

Let's also assume we want to plot its graph for values of x between 0 and $8 \times \text{PI}$.

With minor manipulations we can scale the function to fit our screen, remapping the input range from 0 to 200 and the output range to the $+75/-75$ values range (Figure 12.20).

The program example below will plot the function after tracing the x and y axes.

```
/*
** Plotting a 1D function graph
**
*/
#include <config.h>
#include <graphic.h>
#include <math.h>

#define X0 10
#define Y0 (VRES/2)
#define PI 3.141592654f

main( void)
{
    int x, y;
    float xf, yf;

    // initializations
    ClearScreen();
    InitVideo();

    // draw the x and y axes crossing in (X0,Y0)
    Line( X0, 10, X0, VRES-10);    // y axis
    Line( X0-5, Y0, HRES-10, Y0);  // x axis
```



```

// plot the graph of the function for
for( x=0; x<200; x++)
{
    xf = (8 * PI / 200) * (float) x;
    yf = 75.0 / (8 * PI) * xf * sin( xf);
    Plot( x+X0, yf+Y0);
}

// main loop
while( 1);
} // main

```

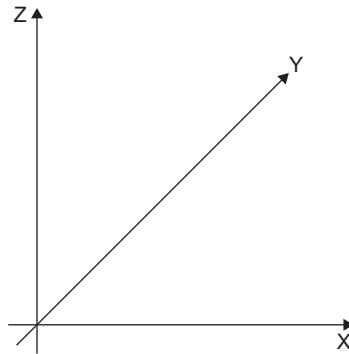


Figure 12.21: Isometric projection

Should the points on the graph become too sparse, we have the option of using the line-drawing algorithm to connect each point to the previous.

Two-Dimensional Function Visualization

More interesting and perhaps entertaining could be plotting graphs of two-dimensional functions. This adds the thrill of managing the perspective distortion and the challenge of connecting the calculated points to form a visually pleasant grid.

The simplest method to squeeze the third axis into a two-dimensional image is to use what is commonly known as an isometric projection, a method that requires minimal computational resources while providing a small visual distortion. The following formulas applied to the x , y and z coordinates of a point in a three-dimensional space produce the px and py coordinates of the projection on a two-dimensional space (our video screen) (Figure 12.21).

```

px = x + y/2;
py = z + y/2;

```

In order to plot the three-dimensional graph of a given function: $z = f(x,y)$ we proceed with a grid of points equally spaced in the x and y plane using two nested *for* loops. For each point we compute the function to obtain the z coordinate and we apply the isometric projection to

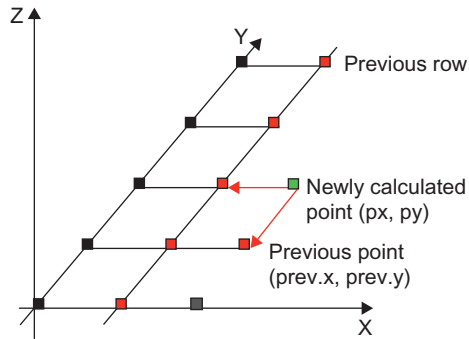


Figure 12.22: Drawing a grid to enhance a two-dimensional graph visualization

obtain a (px, py) coordinate pair. Then we connect the newly calculated point with a segment to the previous point on the same row (previous column). A second segment needs to be drawn to connect the point to the previously computed point in the same column and the previous row (Figure 12.22).

While it is trivial to keep track of the coordinates of the previously computed point on the same row, recording the coordinates of the points on each previous row might require significant memory space. If for example we are using a grid of 20×20 points, we would need to store the coordinates of up to 400 points. Requiring two integers each, that would add up to 800 words or 1,600 bytes of precious RAM memory. In reality, as should be evident from Figure 12.22, all we really need is the coordinates of the points on the *edge* of the grid as painted so far. Therefore, with a little care, we can reduce the memory requirement to just 20 coordinate pairs by maintaining a small rolling buffer.

The example code below visualizes the graph of the function:

$$Z(x, y) = 1 / \sqrt{x^2 + y^2} * \cos(\sqrt{x^2 + y^2})$$

for values of x and y in the range $-3 \times \text{PI}$ to $+3 \times \text{PI}$

```
/*
** Plotting a 2D function graph
**
**
*/
#include <config.h>
#include <math.h>
#include <graphic.h>

#define X0      10
#define Y0      10
#define PI      3.141592654f
#define NODES   20
#define SIDE    10
```

```
typedef struct {
    int x;
    int y;
} point;

point edge[NODES], prev;

main( void)
{
    int i, j, x, y, z;
    float xf, yf, zf, sf;
    int px, py;

    // initializations
    ClearScreen();
    InitVideo();

    // draw the x, y and z axes crossing in (X0,Y0)
    Line( X0, 10, X0, VRES-50);          // z axis
    Line( X0-5, Y0, HRES-10, Y0);        // x axis
    Line( X0-2, Y0-2, X0+120, Y0+120);    // y axis

    // init the array of previous egde points
    for( j = 0; j<NODES; j++)
    {
        edge[j].x = X0+ j*SIDE/2;
        edge[j].y = Y0+ j*SIDE/2;
    }

    // plot the graph of the function for
    for( i=0; i<NODES; i++)
    {
        // transform the x coordinate range to 0..200 offset 100
        x = i * SIDE;
        xf = (6 * PI / 200) * (float)(x-100);
        prev.y = Y0;
        prev.x = X0 + x;

        for( j=0; j<NODES; j++)
        {
            // transform the y coordinate range to 0..200 offset 100
            y = j * SIDE;
            yf = (6 * PI / 200) * (float)(y-100);

            // compute the function
            sf = sqrt( xf * xf + yf * yf);
            zf = 1/(1+ sf) * cos( sf );

            // scale the output
            z = zf * 75;

            // apply isometric perspective and offset
            px = X0 + x + y/2;
```

```

    py = Y0 + z + y/2;

    // plot the point
    Plot( px, py);

    // draw connecting lines to visualize the grid
    Line( px, py, prev.x, prev.y); // connect to prev point on same x
    Line( px, py, edge[j].x, edge[j].y);

    // update the previous points
    prev.x = px;
    prev.y = py;
    edge[j].x = px;
    edge[j].y = py;
  } // for j
} // for i

// main loop
while( 1);

} // main

```

After building the project and connecting to a display you will notice how quickly the PIC24 will produce the output graph, although significant floating-point math is required as the function is applied sequentially to 400 points and as many as 800 line segments are drawn on the video memory ([Figure 12.23](#)).

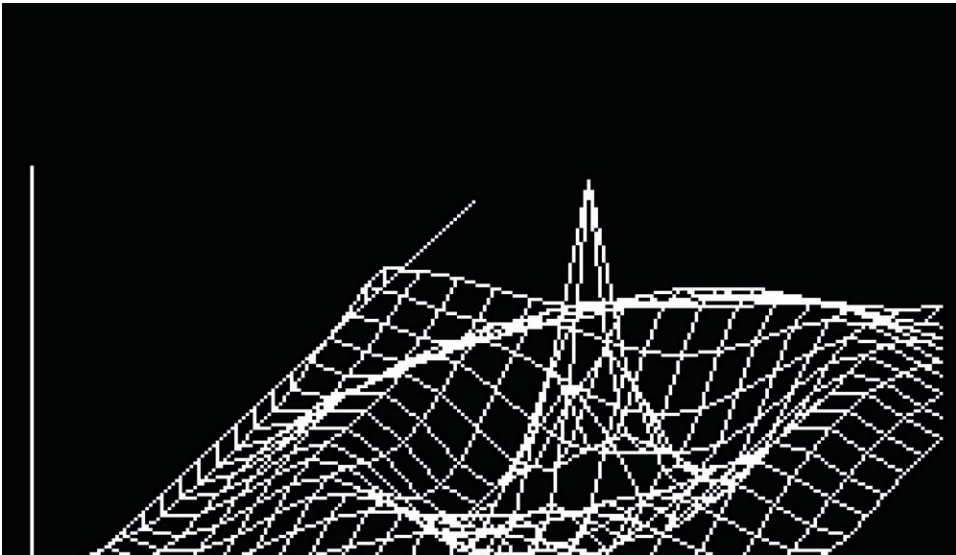


Figure 12.23: Screen capture, graph of a two-dimensional function

Fractals

Fractals is a term coined for the first time by Benoit Mandelbrot, a mathematician (and fellow researcher at the IBM Pacific Northwest Labs) back in 1975, to denote a large set of mathematical objects which present an interesting property: that of appearing self-similar at all scales of magnification as if constructed recursively with an infinite level of detail. There are many examples of fractals in nature, although their self-similarity property is typically extended over a finite scale range. Examples include clouds, snowflakes, mountains, river networks and even the blood vessels in our body.

Because it lends itself to impressive computer visualizations, perhaps the most popular example of a mathematical fractal object is the Mandelbrot set. It's defined as a subset of the complex plane where the quadratic function $z^2 + c$ is iterated. By exclusion, points, c , of the complex plane for which the iteration does not *diverge* are considered to be part of the set. Since it is easy to prove that once the modulus of z is greater than two the iteration is bound to diverge (hence the given point is not part of the set), we can proceed by elimination. The problem is that as long as the modulus of z remains smaller than two we have no way of telling when to stop the iteration and declare the point part of the set. So typically, computer algorithms that depict the Mandelbrot set use an approximation, by setting an arbitrary maximum number of iterations past which a point is simply assumed to part of the set.

Here is an example of how the inner iteration can be coded in C:

```
// initialization
x = x0;
y = y0;
k = 0;

// core iteration
do {
    x2 = x*x;
    y2 = y*y;
    y = 2*x*y + y0;
    x = x2 - y2 + x0;
    k++;
} while ( (x2 + y2 < 4) && ( k < MAXIT));

// check if the point belongs to the Mandelbrot set
if ( k == MAXIT) Plot( x0, y0);
```

With $x0$ and $y0$ the coordinates in the complex space of the point c .

We can repeat this iteration for each point of a squared subset of the complex plane to obtain an image of the entire Mandelbrot set. From the literature we learn that the entire set is included in a disc of radius 2 around the origin, so we can develop a first program that will scan the complex plan in a grid of 192×192 points (to use the maximum screen resolution as defined by our video module) overlapping such a disc.

```

/*
**
** Mandelbrot Set graphic demo
**
*/
#include <config.h>
#include <graphic.h>

#define SIZE VRES
#define MAXIT 64

void Mandelbrot( float xx0, float yy0, float w)
{
    float x, y, d, x0, y0, x2, y2;
    int i, j, k;

    // calculate increments
    d = w/SIZE;

    // repeat on each screen pixel
    y0 = yy0;
    for(i=0; i<SIZE; i++)
    {
        x0 = xx0;
        for(j=0; j<SIZE; j++)
        {
            // initialization
            x = x0;
            y = y0;
            k = 0;

            // core iteration
            do {
                x2 = x*x;
                y2 = y*y;
                y = 2*x*y + y0;
                x = x2 - y2 + x0;
                k++;
            } while ( (x2 + y2 < 4) && ( k < MAXIT));

            // check if the point belongs to the Mandelbrot set
            if ( k == MAXIT) Plot( j, i);

            // compute next point x0
            x0 += d;
        } // for j
        // compute next y0
        y0 += d;
    } // for i
} // Mandelbrot

main()
{
    float x, y, w;

```

```
// initializations
InitVideo();           // start the state machines

// initial coordinates lower left corner of the grid
x = -2.0;
y = -2.0;
// initial grid side
w = 4.0;

ClearScreen(); // clear the screen
Mandelbrot( x, y, w);

// main loop
while( 1);

} // main
```

With the maximum number of iterations set to 64, the PIC24 will produce the complete image in [Figure 12.24](#), the so-called Mandelbrot cardioid, in approximately 30 seconds.

I will confess that since, as a kid, I bought my first personal computer, actually *home computer* was the term used back then (a Sinclair ZX Spectrum), I have been playing with fractal programs. So I have a vivid memory of the long hours I used to spend staring at the computer screen waiting for the old trusty Z80 processor (running at the whopping speed of 3.5 MHz) to paint this same image. A few years later my first IBM PC, an XT clone (running on an 8088 processor at a not much higher clock speed, 4 MHz) was not faring much better, and although the screen resolution of my monochrome Hercules graphic card was higher, I would still launch programs in the evening to watch the results the following morning after what amounted sometimes to up to eight hours processing. Clearly, the amount of computation required to paint a fractal image varies enormously with the chosen area and the

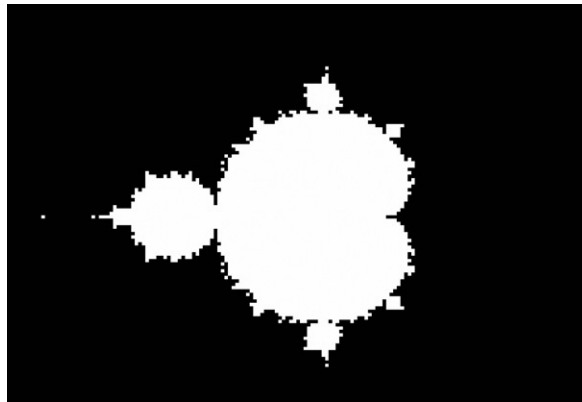


Figure 12.24: Screen capture, Mandelbrot set

number of maximum iterations allowed, but the first time I ran this program I could not avoid being amazed by the speed with which I saw the PIC24 paint the *cardiod* under my eyes.

But the real fun has just begun. The most interesting parts of the Mandelbrot set are at the fringes, where we can increase the magnification and zoom in to discover an infinitely complex world of details. By visualizing not just the points that belong to the set, but also the ones at its edges that diverge, and assigning a color that depends on how fast they did in fact diverge, we can further improve the resulting image. Since we have only a monochrome display, we will simply use alternate bands of black and white assigned to each point according to the number of iterations it took before it either reached the maximum modulus or the maximum number of iterations. Simply enough, this means we will have to modify just one line of code from our previous example:

```
...
    // check if the point belongs to the Mandelbrot set
    if ( k & 1) Plot( j, i);
...
```

Also, since the best way to play with Mandelbrot set images is to explore them by selecting new areas and zooming in on the details, we can transform the main program loop by adding a simple user interface, by means of the four buttons of the Explorer16 board. We can imagine splitting the image into four quadrants ([Figure 12.25](#)). Each quadrant will have a corresponding button, and by pressing it, we will zoom in, doubling the resolution and halving the grid dimension w .

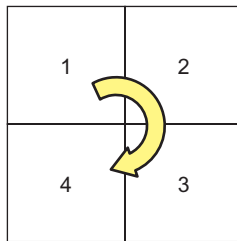


Figure 12.25: Splitting the screen into four quadrants

```
main()
{
    float x, y, w;

    // initializations
    InitVideo();    // start the state machines

    // intial coordinates lower left corner of the grid
    x = -2.0;
    y = -2.0;
```



```
// initial grid size
w = 4.0;

while( 1)
{
    ClearScreen();           // clear the screen
    Mandelbrot( x, y, w);    // draw new image

    // wait for a button to be pressed
    while (1)
    // wait for a key pressed
        if ( !_RD6)
        { // first quadrant
            w/= 2;
            y += w;
            break;
        }
        if ( !_RD7)
        { // second quadrant
            w/= 2;
            y += w;
            x += w;
            break;
        }
        if ( !_RA7)
        { // third quadrant
            w/= 2;
            x += w;
            break;
        }
        if ( !_RD13)
        { // fourth quadrant
            w/= 2;
            break;
        }
    } // wait for a key

} // main loop

} // main
```

Figure 12.26 shows a little selection of interesting areas you will be able to explore with a little patience.

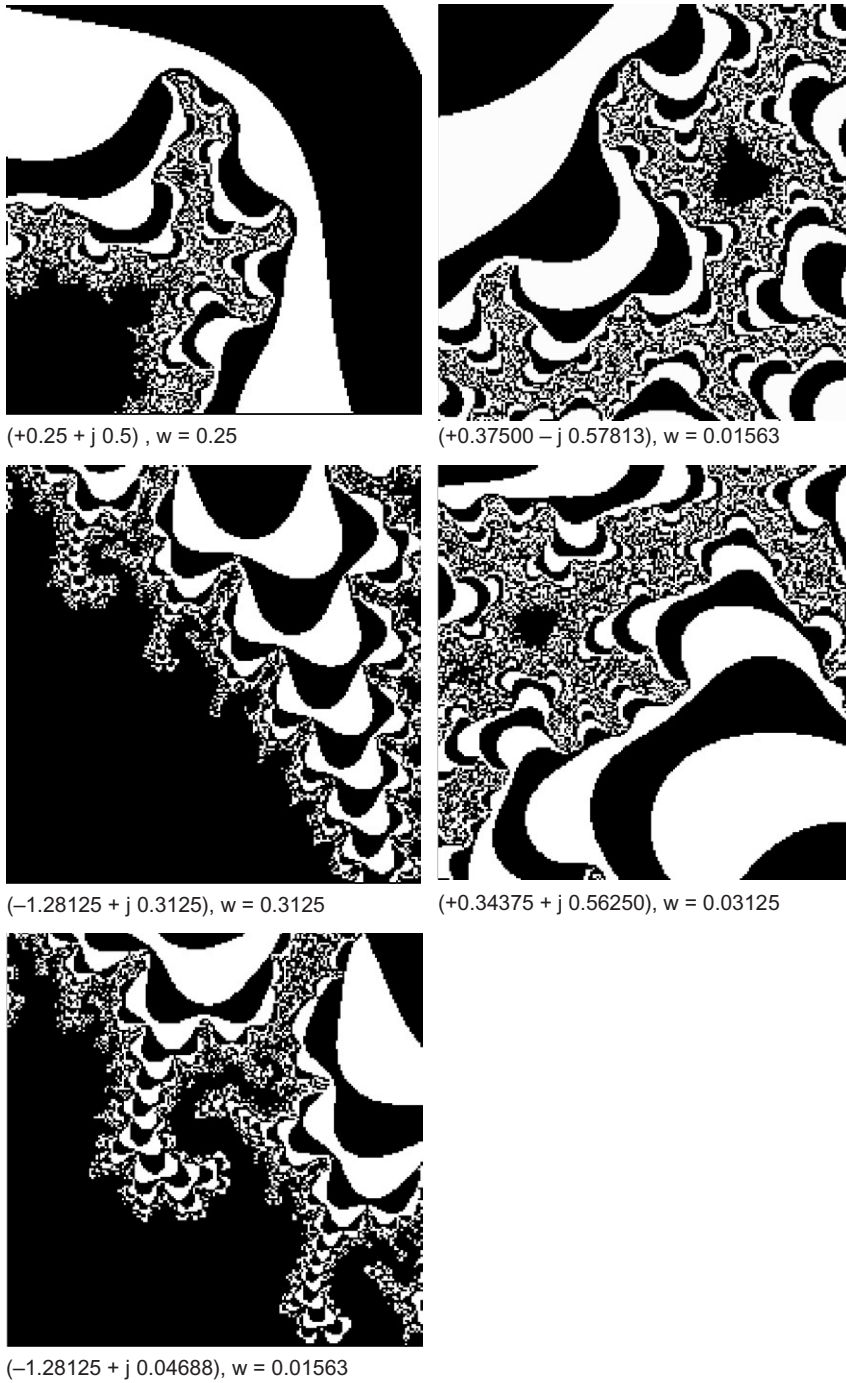


Figure 12.26: Interesting areas to explore

0	0	0	1	1	1	0	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	1	1	1	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	1	0	0	0	1	0
0	0	0	0	0	0	0	0

So far we have been focusing heavily on graphical visualizations, but on more than one occasion you might have felt the desire to actually augment the information presented on the screen with some text. Writing text on the video memory is no different than plotting points or drawing lines, and in fact it can be achieved with a variety of methods, including using the plotting and line-drawing functions we have already developed. But for greater performance, and in order to require the smallest possible amount of code, the easiest way to get text on our graphic display is by developing and using an 8×8 font array (Figure 12.27). Each character can be drawn in an 8×8 pixel box, 1 byte will encode each row, 8 bytes will encode the entire character. We can then assemble the 96 base alphabetical, numerical and punctuation characters in the order and position in which they are presented in the ASCII character set in a single array and save it as an include file.

To save space, we don't need to create the first 32 codes defined in the ASCII set as they correspond mostly to commands and legacy special synchronization codes used by teletypewriters and modems of the old times.

[illegible]

```
// 1 - !
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b00011000,
0b00000000,
0b00011000,
0b00000000,
...
```

Notice that the *Font8x8[]* array is defined with the attribute *const* so that its contents are allocated in the program memory space (FLASH memory of the PIC24) to save precious RAM memory space.

A complete listing of the *font.h* file would waste several pages, so we will omit it here, but you will be able to find it in the companion CD-ROM ready to be copied to the */include* subdirectory.

Of course, as it is a matter of personal taste, you are welcome to modify the *Font8x8[]* array contents to fit your preferences.

Printing a character on the screen then is a matter of copying one byte at a time from the font array to the desired screen position. In the simplest case, characters can be aligned to the words that compose the *VMap* (video memory) array defined by the graphics module. In this way the character positions would be limited to 32 characters per line (256/8) and a maximum of 24 rows of text could be displayed (192/8). A more advanced solution would call for absolute freedom in positioning each character at any given pixel coordinate. This would require a type of manipulation, often referred to as *BitBLT* (an acronym that stands for Bit BLock Transfer), that is common in computer graphics but particularly in video games design. In the following though we will stick to the simpler approach of looking for the solution that requires the smallest amount of resources to get the job done.

We can write then a simple function that prints one ASCII character at a time on the screen at the current cursor position as follows:

```
void putcV( int a)
{
    int i, *p;
    const char *pf;

    // 1. check if char in range
    a -= F_OFFS;
    if ( a < 0)          a = 0;
    if ( a >= F_SIZE)    a = F_SIZE-1;

    // 2. check page boundaries
    if ( cx >= HRES/8)    // wrap around x
    {
```

```
        cx = 0;
        cy++;
    }
    if ( cy >= VRES/8)           // wrap around y
        cy = 0;

    // 3. set pointer to word in the video map
    p = &VMap[ cy * 8 * HRES/16 + cx/2];
    // set pointer to first row of the character in the font array
    pf = &Font8x8[ a << 3];

    // 4. copy one by one each line of the character on the screen
    for ( i=0; i<8; i++)
    {
        if ( cx & 1)
        {
            *p &= 0xff00;
            *p |= *pf++;
        }
        else
        {
            *p &= 0xff;
            *p |= (*pf++)<<8;
        }
        // point to next row
        p += HRES/16;
    } // for

    // increment cursor position
    cx++;
} // putcV
```

In the very first few lines of the function (1.) we verify that the character passed to the function is part of the subset of the ASCII character set currently defined in our font. If not, we change it into either the first character defined or the last one. An alternative strategy available to the reader would have been to ignore the character altogether and exit the routine immediately in such a case.

The second part of the function (2.) deals with positioning the cursor (*cy, cy*), making sure that if we reach the right edge of the screen we wrap around onto the next line as a typewriter would. A similar action is taken when we reach the bottom right extreme of the screen by wrapping around to the top of the screen. The alternative here would have been to implement a scrolling feature that would move the entire contents of the screen up by one line to make room for a whole new line of text.

In the third part (3.) a pointer to the screen memory map is computed based on the cursor coordinates, and a pointer into the font array is computed based on the ASCII character code. Finally (4.), a loop takes care of copying line by line the font image into the video array. Since the video array (*VMap*) is organized in words (and the MSB is displayed first) a little attention must be paid in transferring each byte to the proper position within each 16-bit word. If the cursor position is even, the MSB of the selected word is replaced by the font

data. If the cursor position is odd, the LSB of the selected word is replaced by the font data. At each step in the loop, the pointer inside the video map (*p*) is incremented by 16 words (*HRES/16*) to point to the same position on the following line, while the pointer inside the font array (*pf*) is incremented by one to obtain the next byte composing the character image.

For our convenience we can now create a function that will print an entire NULL terminated ASCII string on the screen:

```
void putsV( unsigned char *s)
{
    while (*s)
        putcV( *s++);
} // putsV
```

Add these two functions to the **graphic.c** library module in the **/lib** subdirectory. Remember also to include the font at the top of the file:

```
#include <config.h>
#include <font.h>
#include <graphic.h>
```

And to define a couple integers to keep track of the cursor position:

```
int cx, cy;
```

Finally, let's add the function prototypes and a couple of useful macros to the **graphic.h** header file in the **/include** subdirectory:

```
/*
** Text on Graphic Page
*/

extern int cx, cy;

void putcV( int a);

void putsV( unsigned char *s);

#define Home()      { cx=0; cy=0;}
#define Clrscr()    { ClearScreen(); Home();}
#define AT( x, y)   { cx = (x); cy = (y);}
```

- *Home()* will simply position the cursor at the upper left corner of the screen.
- *Clrscr()* will clear the screen by invoking the function defined in the graphic module.
- *AT()* will position the cursor as required for the next *putcV* and/or *putsV* command.

Notice how, differently from the graphic coordinate system, the text cursor coordinate system is defined with the origin located in the home position on the upper left corner of the screen and increasing vertical coordinates refer to lines further down the page.

Testing the Text Functions

In order to quickly test the effectiveness of the new text module, we can now create a small program **TextTest.c**, which, after printing a small banner on the first line of the screen, will print out each character defined in the 8×8 font.

```
/*
** TextTest.c
**
*/
#include <config.h>
#include <graphic.h>

main( void)
{
    int i;

    // initializations
    InitVideo();    // start the state machines

    Clrscr();
    AT( 0, 0);
    putsV( "FLYING THE PIC24!");

    AT( 0, 2);
    for( i=32; i<128; i++)
        putcV( i);

    while (1);
} // main
```

Create a new project called **12-Text**. Make sure that all the required modules are added to the project, including *graphic.c*, *graphic.h*, *font.h* and *TextTest.c*.

Finally, launch the project using the **Run>Run Project** command (Figure 12.28).

Text Page Video

With the newly developed text functions we have acquired the capability to display both text and graphics on the video screen. The system in its entirety requires 6,080 bytes of RAM for the video map, which is a significant portion of the total amount of RAM available inside the PIC24FJ128GA010, but only a minuscule portion of the program memory available.

```
Total program memory used (bytes): 0xba3 (2979) 2%
...
Total data memory used (bytes): 0x189e (6302) 76%
```

If our application was going to need the video output only to display text this would have been an extremely inefficient solution. Turning it into a figure only has the effect of detaching it from its context. In fact by using an 8×8 font we can only display 32 characters per line and a maximum of 24 lines for a grand total of 768 characters. In other words, if our application uses the video as a pure text display we are wasting as much as 5,244 bytes of precious RAM. In the early computer days (including the first IBM PC) this was a serious



Figure 12.28: Screen capture, text on the graphic page

(economic) problem that demanded a custom hardware solution. Therefore all early personal computer systems had a *text page*: that is a video mode where the display could display ONLY text, with the advantage of reducing considerably the RAM requirements (to a fraction of those of a graphic page) while also increasing considerably the screen manipulation performance. In a text page character ASCII codes are stored directly in the video memory and they are converted on the fly to the graphical font representation by a hardware device (known as the font generator) intimately connected to the video scanning and timing logic. In this way, the amount of memory required to maintain a page of 768 characters (as in our previous project) would have been only and exactly 768 bytes, that is approximately only 10% of the memory required by our graphic display solution.

This sounds to me like an interesting new challenge. In this the next project we will develop a more RAM-efficient video solution targeting pure text display applications. This will force us to go back to the initial definition of the state machine at the heart of the graphic video module. Fortunately, we can keep most of its structure intact and proceed to optimize only a few critical areas; all the elements that compose the horizontal and vertical synchronization signals will remain unchanged. Also, the construction of horizontal lines remains untouched up to the point where we start sending data to the SPI1 module to serialize. Where in the graphic display we take each word of the memory map as is and we push it on to the SPI buffer, in a text page video application we will need to operate on a byte at a time and

interpose a conversion step. The *Font8x8[]* array will act as a look-up table that will be used to convert on the fly the ASCII code from the text page (now *VMap* will be defined as a byte array) into an image that will be sent to the SPI buffer for serialization. In generic terms, we can express this translation with the following expression:

```
lookup = Font8x8[ *VPtr * 8 + RCount];
```

where *VPtr* is the pointer to the current character inside the text page array, and *RCount* is a counter from zero to seven that keeps track of each video line forming one row of text (there are eight video lines for each row of text).

In practice, things are a little more complicated: since the SPI module must be fed with 16 bits of data at a time, so we need to assemble two characters in one word after performing two look-ups, one after the other.

```
lookup1 = Font8x8[ *VPtr++ * 8 + RCount];
lookup2 = Font8x8[ *VPtr++ * 8 + RCount];
SPI1BUF = ( 256 * lookup1 + lookup2);
```

Repeat this eight times to fill the entire SPI buffer.

Now this is a lot of work to perform in the few microseconds available to the OC4 interrupt service routine. Even if we were to enable the highest level of optimization of the compiler (and in this book we chose to actually never enable any optimization) the possibility that we would fit it in in the time available (less than 25 μ s) is pretty slim. There are simply too many multiplications and additions to perform when working with the look-up table. Fortunately, this is something we can change. In fact, we can rearrange the way the font array is built. While it is convenient to initialize the array proceeding sequentially, filling in all eight rows of each character and then proceeding to the next, in order to simplify the look-up expression it would be best if the array was organized the other way around. In other words, we should fill the array starting with the first byte of each character in the font; followed by the second byte of each character and so on. We could re-write the expressions above with the new rearranged font *RFont* as follows:

```
lookup1 = RFont[ (RCount * F_SIZE) + *VPtr++ ];
lookup2 = RFont[ (RCount * F_SIZE) + *VPtr++ ];
SPI1BUF = ( 256 * lookup1 + lookup2);
```

The great advantage lies in the fact that now (*RCount*F_SIZE*) is a constant offset and we can even obtain a pointer inside the font that already takes care of such offset with the following expression:

```
FPtr = &RFont[ RCount * F_SIZE];
```

This can be precalculated (inside the Timer3 interrupt service routine) at the beginning of each line for a significant saving.

The new look-up expressions are now simplified to:

```
lookup1 = FPtr[ *VPtr++ ];
lookup2 = FPtr[ *VPtr++ ];
SPI1BUF = ( lookup1 << 8 + lookup2);
```

We can now modify the OC4 interrupt service routine to make full use of our highly optimized font table look-up:

```
void _ISRFAST _OC4Interrupt( void)
{
    unsigned i,k;

    // load SPI buffers indirectly from font
    i=8;
    do{
        k = FPtr[ *VPtr++];
        SPI1BUF = ( k << 8 ) | FPtr[ *VPtr++];;
    } while( --i >0);

    if ( --HCount > 0)
    { // activate again in time for the next SPI load
        OC4R += ( PIX_T * 8 * 16);
        OC4CON = 0x0009;    // single event
    }

    // clear the interrupt flag
    _OC4IF = 0;
} // OC4Interrupt
```

As we said before, the modifications to the Timer3 interrupt service routine are minor, as only a couple of pointers need to be prepared for the text lines to be properly sequenced and for the font offset to be pre-calculated:

```
void _ISRFAST _T3Interrupt( void)
{
    // advance the state machine
    if ( --VCount == 0)
    {
        VCount = VC[ VState];
        VState = VS[ VState];
    }

    // vertical state machine
    switch ( VState) {
        case SV_PREEQ: // 0
            LPtr = VMap;
            RCount = 0;
            break;

        case SV_SYNC:
            // vertical sync pulse
```

```
        OC3R = LINE_T - HSYNC_T - BPORCH_T;
        break;

    case SV_POSTEQ: // 2
        // horizontal sync pulse
        OC3R = HSYNC_T;
        break;

    default:
    case SV_LINE: // 3
        // activate OC4 for the SPI loading
        OC4R = HSYNC_T + BPORCH_T;
        OC4CON = 0x0009; // single event
        HCount = HRES/128; // reload counter

        // prepare the font pointer
        FPtr = &RFont[RCount * F_SIZE];
        // prepare the line pointer
        VPtr = LPtr;

        // Advance the RCount
        if ( ++RCount == 8 )
        {
            RCount = 0;
            LPtr += COLS;
        }
    } //switch

    // clear the interrupt flag
    _T3IF = 0;
} // T3Interrupt
```

The video initialization routine will now require one more step as the font array needs to be rearranged as discussed above:

```
// prepare a reversed font table
for (i=0; i<8; i++)
{
    p = Font8x8 + i;
    for (j=0; j<F_SIZE; j++)
    {
        *r++ = *p;
        p+=8;
    } // for j
} // for i
```

While for simplicity we implement this as a second array allocated in RAM where we copy things in the new order, the ultimate solution is to rearrange the *font.h* file definition, so that the *Font8x8* array is already defined in the new and optimal order, there is no RAM waste, nor is processing time used during the video initialization to perform the translation.

Back when working on the graphical interface we found that a 256×192 pixel screen was an acceptable compromise between screen resolution and memory usage as it would leave 2 Kbytes of RAM available for the application to use. Now the balance is considerably changed: with a 24-lines by 32-column display, only 768 bytes are used by the video module and we can in fact afford to expand the resolution a bit. The horizontal resolution is the one most in need of an upgrade. Most video terminals in fact use a 25×80 format while the average printed document has no less than 60 characters per line. While we could afford the RAM ($25 \text{ rows} \times 80 \text{ columns} = 2,000 \text{ characters}$), this time it is the NTSC video specifications that are going to dictate the ultimate limit. As we observed at the very beginning of this chapter, the maximum signal bandwidth for an NTSC video composite signal is fixed at 4.2 MHz, while the portion of the waveform producing the visible line image is $52 \mu\text{s}$ wide. This determines a maximum theoretical horizontal resolution of 436 pixels, which in the case of an 8×8 font would imply a maximum number of 54 columns. In practice, it would be better to choose a smaller value and, to make the best use of the SPI FIFO mechanism that we have been using with success so far, it would be better to choose a number that is a multiple of 16. While in the graphic module we used two successive blocks of 128 pixels each to fill the SPI FIFO buffers, for the text page module we can now add a third block, bringing the total horizontal resolution up to 48 characters. Note how this will require the SPI clock prescaler to be switched to the higher-frequency mode ($\text{PIX}_T = 2$).

For the vertical resolution we have considerable freedom since the NTSC standard specifies 262 lines, of which theoretically up to 253 could be used for the actual image; there is no difficulty in making 25 rows of text (adding up to 200 lines) fit.

Overall, our text page module will produce a 25 rows by 48 columns display using a total of just 1,200 bytes. This will represent a considerable improvement in readability with respect to the text on the graphic page approach, with a significant reduction in the RAM memory usage as well.

This is the new set of constants and definitions that completes the new *Text* video module:

```
/*
** text.c
** NTSC/PAL Video Generator using:
**     T3      Interrupt for main timing
**     OC3     Horizontal synchronization pulse
**     OC4     SPI buffer reload timing
**     SPI1    pixel serialization (8x16-bit FIFO)
**
** Pin assignments:
**     SD01 - RF8 - VIDEO
**     OC3  - RD2 - SYNC (requires AV16/32 board)
*/
#include <EX16.h>
#include <text.h>
```

```
#include <font.h>

#define VRES (ROWS*8)
#define HRES (COLS*8)

// timing definitions for video vertical state machine
#ifdef PAL
    #define V_LINES 312    // total number of lines in frame
    #define LINE_T 1024    // total number of Tcy in a line
#else // default NTSC configuration
    #define V_LINES 262    // total number of lines in frame
    #define LINE_T 1016    // total number of Tcy in a line
#endif

#define VSYNC_N 3        // V sync lines

// count the number of remaining black lines top+bottom
#define VBLANK_N (V_LINES - VRES - VSYNC_N)

#define PREEQ_N VBLANK_N / 2        // pre equ + bottom blank
#define POSTEQ_N VBLANK_N - PREEQ_N // post eq + top blank

// definition of the vertical sync state machine
#define SV_PREEQ 0
#define SV_SYNC 1
#define SV_POSTEQ 2
#define SV_LINE 3

// timing definitions for video horizontal state machine
#define HSYNC_T 85    // Tcy in a horizontal synch pulse
#define BPORCH_T 90    // Tcy in a back porch
// Tcy in each pixel, valid values are only 2 or 3
#define PIX_T 2    // Aspect Ratio 2 = 16:9, 3 = 4:3
// use 2 (16:9) to allow for up to 48 columns of text

// Text Page array
char VMap[ COLS * ROWS];
char *VPtr, *LPtr;

// reordered Font
unsigned char RFont[ F_SIZE*8];
unsigned char *FPtr;

int HCount, VCount, RCount, VState;

// next state table
int VS[4] = { SV_SYNC, SV_POSTEQ, SV_LINE, SV_PREEQ};
// next counter table
int VC[4] = { VSYNC_N, POSTEQ_N, VRES, PREEQ_N};
```

The same routines we developed for the TextOnGPage project can now be added directly to this project.

```
void ClearScreen( void)
{
    int i, j;
    char *v;
```

```

    v = VMap;

    // clear the screen
    for ( i=0; i < (ROWS); i++)
        for ( j=0; j < (COLS); j++)
            *v++ = ' '-F_OFFSETS;
} // ClearScreen

void HaltVideo( void)
{
    T3CONbits.TON = 0;    // turn off the vertical state machine
} // HaltVideo

int cx, cy;

void putcV( int a)
{
    // check if char in font range
    a -= F_OFFSETS;
    if ( a < 0)          a = 0;
    if ( a >= F_SIZE)    a = F_SIZE-1;

    // check page boundaries
    if ( cx >= COLS)      // wrap around x
    {
        cx = 0;
        cy++;
    }
    cy %= ROWS;          // wrap around y

    // find first row in the video map
    VMap[ cy * COLS + cx] = a;

    // increment cursor position
    cx++;
} // putcV

void putsV( unsigned char *s)
{
    while (*s)
        putcV( *s++);
} // putsV

void pcr( void)
{
    cx = 0;
    cy++;
    cy %= ROWS;
} // pcr

```

We can save the new file as **text.c** and save it in the **/lib** subdirectory. The corresponding header file **text.h** will go to the **/include** subdirectory as usual.

```

/*
** text.h
**

```

```
** Text Page Video Module
**
*/
// #define PAL
#define ROWS    25      // rows of text
#define COLS    48      // columns of text

// Text Page array
extern char VMap[ COLS * ROWS];

// initializes the video output
void InitVideo( void);

// stops the video output
void HaltVideo( void);

// clears the video map
void ClearScreen( void);

// cursor
extern int cx, cy;

void putcV( int a);

void putsV( unsigned char *s);

void pcr( void);

#define Home()      { cx=0; cy=0;}
#define Clrscr()    { ClearScreen(); Home();}
#define AT( x, y)   { cx = (x); cy = (y);}
```

Testing the Text Page Performance

In order to test the new text page video module we could try and modify an example seen in Chapter 8: the Matrix demo. Back then we were using the asynchronous serial communication module (UART1) to communicate with a VT100 computer terminal or, more likely, a PC running the HyperTerminal program configured for emulation of the historical DEC terminal VT100 protocol. Now we can replace the *putcU2()* function calls used to send a character to the serial port, with *putcV()* calls directed at our video interface.

Let's create a new project called 12-***Matrix*** and let's add all the necessary modules to it including the **text.c**, **text.h** and finally a new main module that we will call ***matrix2.c*** or ***the-matrix-reloaded.c*** if you prefer.

```
/*
** The Matrix Reloaded
**
*/
#include <config.h>
#include <text.h>
```

```

#define COL      COLS
#define ROW      ROWS

// define an alien character (sub)set
const char ALIEN[] = "{}[]!@#^*)(+-\\|`~_-= ";

main()
{
    int v[ COL]; // vector containing length of each string
    int i,j,k;

    // 1. initializations
    InitVideo();
    Clrscr();      // clear the screen
    srand( 12);    // start the random number sequence

    // 2. init each column length
    for( j =0; j<COL; j++)
        v[j] = rand()%ROW;

    Home();

    // 3. refresh the screen with random columns
    for( i=0; i<ROW; i++)
    {
        // refresh one row at a time
        for( j=0; j<COL; j++)
        {
            // print a random character down to each column length
            if ( i < v[j])
                putcV( ALIEN[ rand()%20]);
            else
                putcV(' ');
        }
        putcV( ' ');
    } // for j
    pcr();
} // for i

// 4. main loop
while( 1)
{
    // 4.1 delay to slow down the screen update
    Delayms( 200);

    // 4.2 randomly increase or reduce each column length
    for( j=0; j<COL; j++)
    {
        switch ( rand()%3)
        {
            case 0: // increase length
                v[j]++;
                if (v[j] > ROW) v[j] = ROW;
                // add new random character

```



```
        AT( j,v[j]-1);
        putcV( ALIEN[ rand()%20]);
        break;

    case 1: // decrease length
        // remove character
        if (v[j]>0)
        {
            AT( j,v[j]-1);
            putcV( ' ');
            v[j]--;
        }
        break;

    default:// unchanged
        break;
} // switch
} // for
} // main loop
} // main
```

After saving it, launch the project on the Explorer16 connected to your video device of choice. You will notice how much faster the screen updates can be, as the program now has direct access to the video memory and there is no serial connection limiting the information transfers. Also, since every character placed in the video memory can now be retrieved and manipulated in place, new tricks are possible to make the video resemble more closely the movie characteristics and the somewhat alien scrolling effect. Speaking of aliens, the string named *ALIEN* is used in this demo to assemble an alien-looking set of characters (composed mostly of punctuation), but ultimately we should have defined a new font file with proper alien symbology.

Besides the visual impression though, we are now interested in measuring the actual processor overhead imposed by the text mode video routines that perform the on-the-fly font translation. As we did in the previous chapters, we can use a couple of the PORTA pins, RA0 and RA1, to signal when we are executing code inside one of the interrupt service routines.

```
void _ISRFAST _T3Interrupt( void)
{
    _RA0=1;
    ...
    _RA0=0;
} // T3Interrupt

void _ISRFAST _OC4Interrupt( void)
{
    _RA1=1;
    ...
    _RA1=0;
} // OC4Interrupt
```

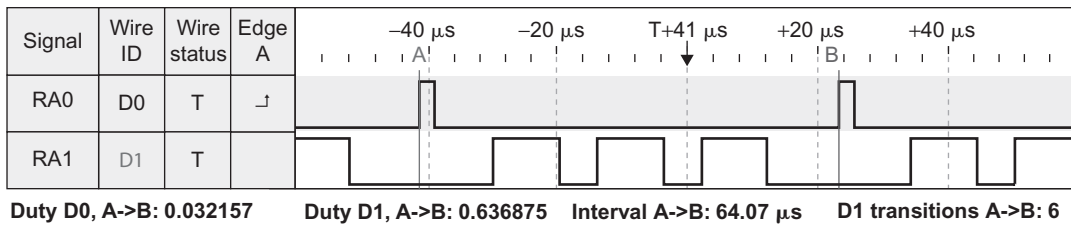


Figure 12.29: Measuring text mode video overhead

Remember to add the initialization of the *TRISA* register inside the at the top of the *main()* function to enable *RA0* and *RA1* as outputs. Then connect the two to a low-cost logic analyzer or an oscilloscope and capture one full video line.

Figure 12.29 shows that the combined duty cycle of the two interrupt routines, adding up to 66%, is telling us that the CPU overhead is almost triple that of the graphic video module. This is the penalty we have to pay for the great reduction in RAM memory requirements and the increased resolution we gain for all those applications that require a pure text output (or simple tiled graphics!).

Post-Flight Briefing

In this lesson we have explored the possibility of producing a video output using a minimal hardware interface composed in practice of only three resistors. We learned to use four peripheral modules together to build the complex mechanism required to produce a properly formatted composite video signal. A 16-bit timer was used to generate the fundamental horizontal synchronization period. Two output compare modules provided intermediate timing references and finally the SPI module was used in enhanced mode to serialize the video data using the new eight-level-deep \times 16-bit FIFO. After developing basic graphic functions to plot individual pixels first and efficiently draw lines, we explored some of the possibilities offered by the availability of a graphic video output, including uni-dimensional and two-dimensional function graphing. After briefly exploring the world of fractals we changed gear to look at the problem of displaying text. First we developed routines to add text to the graphic page, later we developed a new video mode specifically optimized for text display.

Tips & Tricks

The final touch, to complete our brief excursion in the world of graphics, would be to add some animation to our video output libraries. To make the motion fluid and avoid an annoying continuous flicker of the image on the screen, we would need to adopt a technique known as *double buffering*. This requires us to have two image buffers in use at any point

in time. One is the *active* buffer and its contents are shown on the screen, while the other, *hidden*, is being drawn. When the second buffer drawing is complete, the two are swapped. The first buffer, not visible any more, is cleared and the drawing process starts again. The only limitation with implementing this technique in our case is represented by the RAM memory size requirements. To make two image buffers fit in the 8 Kbytes of memory of the PIC24FJ128GA010, while leaving some space for variables and stack, we would need to reduce the image resolution. A pair of image buffers of 160×160 , for example, would fit as each would require only 3,200 bytes.

```
int _FAR V1Map[VRES * (HRES/16)];
int _FAR V2Map[VRES * (HRES/16)];
```

Note

Users of the PIC24F GA1, GB1, GB2 and DA2 families will be able to use full-sized buffers, benefitting from the much larger amount of RAM memory available.

The only other changes required to the project would be:

- Replace direct references to the *VMap[]* array with references to pointers.
- Make the interrupt-driven state machine that refreshes the screen use a pointer to the active buffer:

```
int *VA;
```

- Make the plotting and drawing functions use a pointer to the hidden buffer:

```
int *VH;
```

- The swap between the two buffers can then be performed swapping only two pointers:

```
void swapV( void)
{
    int * V;
    while ( VCount != 1);           // wait until the end of the frame
    V = VA; VA = VH; VH = V;       // at the next VSync it will swap the screen
} //swapV
```

Notice that care must be taken not to perform the swap in the middle of a frame, but synchronized with the end of a frame and the beginning of the next.

Exercises

- Replace the *write.c* function to redirect the *stdio.h* library functions output to the text/graphic screen.
- Add the PS/2 keyboard input support to provide a complete console.

Books

- Koster, R., 2004. *A Theory of Fun for Game Design*, Paraglyph Press.
You must take game design seriously. Or maybe not?

Links

- <http://www.microchip.com/graphics>
This is the design center dedicated to graphics solutions.
- http://en.wikipedia.org/wiki/Zx_spectrum
The Sinclair ZX Spectrum was one of the first personal computers (home computers they used to be called) launched in the early 1980s. Its graphic capabilities were very similar to those of the graphic libraries we developed in this project. Although it used several custom logic devices to provide the video output, its processing power was less than a quarter that of the PIC24. Still, the limited ability to produce color (only 16 colors with a resolution of a block of 8×8 pixels) enticed many programmers to create thousands of challenging and creative video games.

Mass Storage

The relationship between weight (mass) and performance of an airplane is generally well understood by most pilots and non-pilots too. Try and put too much weight on those wings and the take-off is going to be longer, much longer, or actually so long that there is not enough runway to continue and there is no take-off at all, ouch!

The more common problem seems to be understanding how much all that stuff you want (or your significant other wants) to bring along actually weighs. Packing the airplane for a trip with friends or family is just like packing your backpack for an excursion in the outdoors. The fact that everything seemed to fit in does not mean you will be able to lift it up. As a pilot you won't be allowed to guess it, you will have to compile a weight and balance sheet and, if necessary, use a scale to determine the exact numbers and decide what to sacrifice: some of the load or maybe some of the fuel. One thing that I can strongly discourage you from doing, though, is to ask the significant other to step on the scale. He or she won't approve of being considered part of the problem, nor of providing proof of the exact mass.

Flight Plan

In many embedded control applications you might find a need for a larger non-volatile data storage space, well beyond the capabilities of the common Serial EEPROM devices we interfaced to in previous chapters and certainly larger than the Flash program memory available inside the microcontroller itself. You might be looking for orders of magnitude more, hundreds of megabytes and possibly gigabytes. If you own a digital camera, an MP3 player or even just a cell phone, you have probably become familiar with the storage requirements of consumer multimedia applications and with the available mass storage technologies. Hard disk drives have become smaller and less power-thirsty, but also a multitude of solid-state solutions (based once more on Flash technologies such as CompactFlash™, SmartMedia™, Secure Digital (SD™), Memory Stick™ and others...) have flooded the market and, because of the volumes absorbed by the consumer market, the price range has been reduced to a point where it is possible if not convenient to integrate these devices into embedded control applications.

In this lesson we will learn how to interface one of the most common and inexpensive mass storage device types to a PIC24 microcontroller using the smallest amount of processor resources.

The Flight

Each one of the many competing mass storage technologies has its strengths and weaknesses, as each one was designed for a somewhat different target application. We will choose the mass storage media according to the following criteria:

- Wide availability of the memory and required connectors
- Small pin count required by the physical interface (serial)
- Large memory capacity
- Open specifications available
- Ease of implementation
- Low cost of the memory and the required connectors

The Secure Digital standard compares favorably in all these aspects as it is today one of the most commonly adopted mass storage media for digital cameras and many other multimedia consumer applications. The SD card specifications represent an evolution of a previous technology known as MultiMediaCard or MMC, with which they are still partially (forward) compatible both electrically and mechanically. The Secure Digital Card Association (SDCA) owns and controls the technical specification standards for SD memory cards and they require all companies who plan to actively engage in the design, development, manufacture or sale of products which utilize the SD specifications to become members of the association. As of this writing a general SDCA membership will cost you \$2,000 in annual fees. The MultiMediaCard Association (MMCA), on the other hand, does not require implementers to necessarily become members, but makes copies of the MMC specifications available for sale starting at \$500. So both technologies are far from free, nor open. Fortunately, there is a subset of the SD specifications that has been released to the public by the SDCA in the form of a “simplified physical specification”. This information is all we need to develop a basic understanding of the SD/MMC memory technology and get started designing a PIC24 mass storage interface.

The SD/MMC Physical Interface

SD cards require only nine electrical contacts and an SD/MMC-compatible connector, which can be purchased from most online catalogs for less than a couple of dollars, and requires only a couple of pins more to account for insertion detection and write protection switch sensing (Figure 13.1). There are two main modes of communication available: the first one (known as the SD bus) is original to the SD/MMC standard and it requires a nibble (4-bit) wide bus interface; the second mode is serial and is based on the popular SPI™ bus standard. It is this second mode that makes the SD/MMC mass storage devices particularly appealing for all embedded control applications, as most microcontrollers will either have a hardware SPI interface available or will be able to easily emulate one (bit-banging) with a reduced number of I/Os. Finally, the physical specifications of the SD/MMC cards indicate

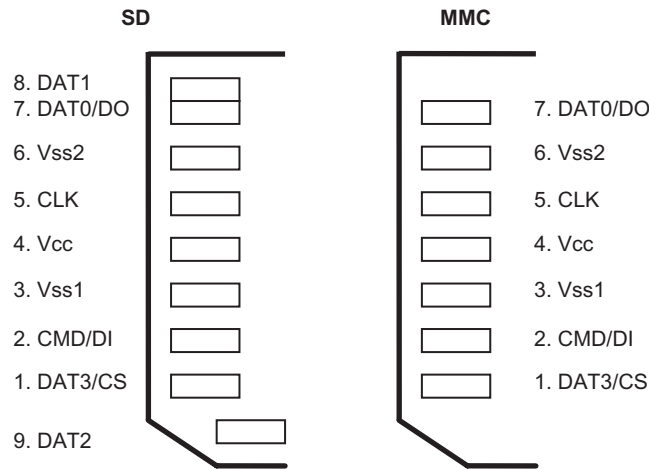


Figure 13.1: SD card and MMC card connectors pin-out

an operating voltage range of 2.0V to 3.6V, which is ideally suited for all applications with modern microcontrollers implemented in advanced CMOS processes, as is the case of the PIC24 family.

Interfacing to the Explorer16 Board

Unfortunately, although the number of electrical connections required for the SPI interface is very low, all SD/MMC card connectors available on the market are designed for surface-mount applications only, which makes it almost impossible to use the prototyping area of the Explorer16 demonstration board. To facilitate this lesson, and the following lessons that will make use of mass storage devices, complete schematics and PCB layout information for an expansion board, the AV16/32, are available on the companion website www.flyingpic24.com. The expansion board offers several other interfaces that will be used to follow the experiments presented in this book's chapters 11 through 15 and is offered as a bare PCB or a complete kit.

Long after this book was first published, Microchip has made available an official SD/MMC PICtail Plus expansion board for the Explorer16, identified by the part number AC164122. The differences are small but, in order to support both, in the following we will take extra care to generalize the interface using simple macros (*#define*) for the I/O access.

Since in the previous chapters we have used the first SPI peripheral module to produce a video output, an application that does not allow for sharing of the resource, we will share instead the second SPI module (SPI2) between the SD card interface and the EEPROM interface (presented in chapter 7) by using separate *Chip Select* signals for the two.

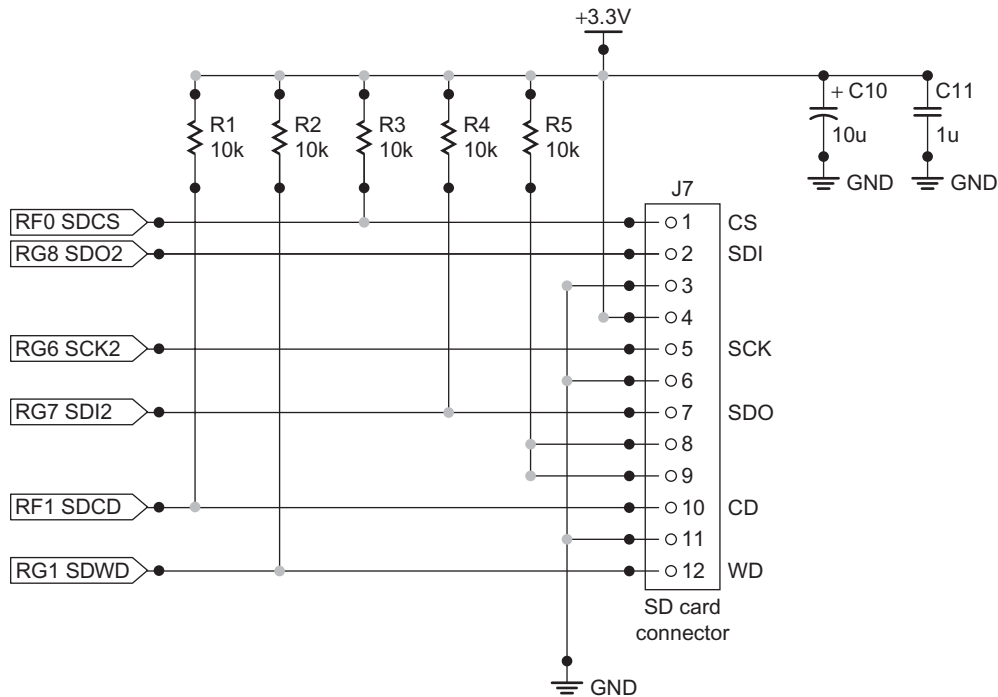


Figure 13.2: SD/MMC card interface to Explorer16 demo board

Note

The AC164122 unfortunately works only with the SPI1 module and therefore cannot be used in conjunction with a composite video application.

In addition to the SCK, SDI and SDO pins, we will provide pull-ups for the unused pins (reserved for the 4-bit-wide SD bus interface) of the SD/MMC connector and for two more pins that will be dedicated to the Card Detect and Write Protect signals (Figure 13.2).

Starting a New Project

Using the **New Project** checklist, let's create a new project called **13-SDMMC** and a new source file called **SDMMC.c**. We will start by writing the basic initialization routines for all the necessary I/Os and configuring the SPI module.

```
/*
** SD/MMC card interface
**
*/
#include <EX16.h>
```



```

#ifdef AC164122          // PICTail Plus, top slot
    // I/O definitions
    #define SDWP    _RF1    // Write Protect input
    #define SDCD    _RF0    // Card Detect input
    #define SDCS    _RB1    // Card Select output

    // Card Select TRIS control
    #define TRISCS() _PCFG1 = 1; _TRISB1=0

    // SPI port selection
    #define SPIBUF   SPI1BUF
    #define SPICON   SPI1CON1
    #define SPISTAT   SPI1STAT
    #define SPIRFUL   SPI1STATbits.SPIRBF
#else // default AV16/32 configuration
    // I/O definitions
    #define SDWP    _RG1    // Write Protect input
    #define SDCD    _RF1    // Card Detect input
    #define SDCS    _RF0    // Card Select output

    // Card Select TRIS control
    #define TRISCS() _TRISF0=0

    // SPI port selection
    #define SPIBUF   SPI2BUF
    #define SPICON   SPI2CON1
    #define SPISTAT   SPI2STAT
    #define SPIRFUL   SPI2STATbits.SPIRBF
#endif

```

Note how, with a simple conditional definition, we have established a default I/O assignment for the AV16/32 I/Os and an alternative one available in case the *AC164122* macro is defined. By defining generic register names, such as *SPIBUF*, *SPICON* and *SPISTAT*, we have also automated the selection of the corresponding SPI port.

```

void InitSD( void)
{
    SDCS = 1;
    TRISCS();          // make Card select an output pin

    // init the spi module for a slow (safe) clock speed first
    SPICON = 0x013c;    // CKE=1; CKP=0, sample middle, 1:64

    SPISTAT = 0x8000;    // enable the peripheral
} // InitSD

```

In particular, in the *SPIxCON1* register we need to configure the SPI module to operate in master mode with the proper clock polarity, clock edge, input sampling point and initial clock frequency. The clock output *SCK* must be enabled and set low when idle. The sampling point for the *SDI* input must be centered. The frequency is controlled by means of two prescalers (primary and secondary) that divide the main processor cycle clock *Tcy* to generate the SPI

clock signal. After power up and until the SD card is properly initialized, we will have to reduce the clock speed to a safe setting below 400 kHz. We will use the primary prescaler setting 1:64 to obtain a 250 kHz clock signal. This is just a temporary arrangement though; after sending only the first few commands, we will be able to speed up the communication considerably.

Notice how only the *Card Select* signal (a generic I/O) needs to be manually configured as an output pin while the SCK and SDO pins are automatically configured as outputs as soon as we enable the SPI peripheral.

Selecting the SPI Mode of Operation

When an SD/MMC card is inserted in the connector and powered up, it is in the *default* mode of communication: the *SD bus mode*. In order to inform the card that we intend to communicate using the alternative *SPI mode*, all we need to do is to select the card (sending the SDCS pin low) and send the first *RESET* command. We can stay assured that, once having entered the SPI mode, the card will not be able to change back to the SD bus mode unless the power supply is cycled. This means though that, if the card is removed from the slot without our knowledge and then reinserted, we will have to make sure that the initialization routine or at least the *RESET* command are repeated in order to get back to the SPI mode. We can detect the card's presence at any time by checking the status of the *Card Detect (CD)* line.

Sending Commands in SPI Mode

In SPI mode, commands are sent to an SD/MMC card as packets of 6 bytes and all responses from the SD card are provided with multiple-byte data blocks of variable length. So, all we need to communicate with the memory card is the usual basic SPI routine to send and receive (the two operations are really the same, as we have seen in the previous chapter) a byte at a time.

```
// send one byte of data and receive one back at the same time
unsigned char WriteSPI( unsigned char b)
{
    SPIBUF = b;                      // write to buffer for TX
    while( !SPIRFUL);               // wait transfer complete
    return SPIBUF;                  // read the received value
} // WriteSPI
```

For improved code readability and convenience we will also define two more macros that will mask the same *WriteSPI()* function as a pure *ReadSPI()*, or just as a clock output function *ClockSPI()*. Both these macros will still need to send a dummy byte of data (*0xFF*).

```
#define ReadSPI()    WriteSPI( 0xFF)
#define ClockSPI()   WriteSPI( 0xFF)
```

To send a command we will start selecting the card (SDCS low) and sending through the SPI port a packet composed of three parts ([Figure 13.3](#)).

Byte 1		Byte 2		Byte 3		Byte 4		Byte 5		Byte 6							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0		
0	1	Command										Address				CRC	

Figure 13.3: SPI-mode SD/MMC card command format

The first part is a single byte containing a *command index*. The following definitions will cover all the commands we will be using for this project.

```
// SD card commands
#define RESET                0 // a.k.a. GO_IDLE (CMD0)
#define INIT                 1 // a.k.a. SEND_OP_COND (CMD1)
#define READ_SINGLE         17 // read a block of data
#define WRITE_SINGLE        24 // write a block of data
```

The command index is followed by a 32-bit memory address. It is an *unsigned long* integer value that must be sent MSB first. For convenience we will define a new type to represent such large address fields that we will call *LBA*, borrowing from a term used in other mass storage applications, to represent a very large address to a generic block of data.

```
typedef unsigned long LBA; // logic block address, 32 bit wide
```

Finally, the command packet is completed by a 1-byte CRC. This Cyclic Redundancy Check (CRC) feature is always used in SD bus mode to make sure that every command and every block of data transmitted on the bus is free from errors. But as soon as we switch to the SPI mode (after sending the RESET command) the CRC protection will be automatically disabled as the card will assume that a direct and reliable connection to the host, the PIC24 in our case, is available. By taking advantage of this default behavior, we can simplify considerably our code, replacing the CRC calculation with a precomputed value. This will be the CRC code of the *RESET* command, and it will be ignored for all the subsequent commands for which the CRC field will be a *don't care*. Here is the first part of the *SendSDCmd()* function:

```
int SendSDCmd( unsigned char c, LBA a)
// sends a 6 byte command block to the card and leaves SDCS active
{
    int i, r;

    // enable SD card
    EnableSD();

    // send a command packet (6 bytes)
    WriteSPI( c | 0x40); // send command + frame bit
    WriteSPI( a>>24);    // MSB of the address
    WriteSPI( a>>16);
    WriteSPI( a>>8);
    WriteSPI( a);        // LSB
    WriteSPI( 0x95);     // send CMD0 (RESET) CRC
```

After sending all 6 bytes to the card, we wait in a loop for a response byte. We will in fact keep sending dummy data, continuously clocking the SPI port. The response will be *0xFF*; basically, the SDI line will be kept high until the card is ready to provide us with a proper response code. The specifications indicate that up to 64 clock pulses (8 bytes) might be necessary before a proper response is received. Should we exceed this limit we would have to assume a major malfunctioning of the card and abort communication.

```
// now wait for a response (allow up to 8 bytes delay)
i = 9;
do {
    r = ReadSPI();          // check if ready
    if ( r != 0xFF) break;
} while ( --i > 0);

return ( r);

/* return response
   FF - timeout, no answer
   00 - command accepted
   01 - command received, card in idle state (after RESET)
   other errors
*/
} // SendSDCmd
```

If we receive a response code though, each bit, if set, will provide us with an indication of a possible problem:

- bit 0 = idle state
- bit 1 = erase Reset
- bit 2 = illegal command
- bit 3 = communication CRC error
- bit 4 = erase sequence error
- bit 5 = address error
- bit 6 = parameter error
- bit 7 = always 0.

Notice that on return the *SendSDCmd()* function leaves the SD card still selected (*SDCS* low) so that commands such as *Block Write* and *Block Read*, which require additional data to be sent or received from the card, will be able to proceed. In all other commands that do not require additional data transfers, we will have to remember to deselect the card (set *SDCS* high) immediately after the function call. Furthermore, since we want to share the SPI port with other peripherals, for example the Serial EEPROM mounted on the Explorer16 board, we need to make sure that the SD/MMC card receives a few more clock cycles (eight will suffice) immediately after the rising edge of the chip select line (*SDCS*). According to the SD/MMC specifications, this will allow the card to complete a few important housekeeping

chores, including the proper release of the SDO line, essential to allow other devices on the same bus to communicate properly.

A pair of macros will help us perform this consistently:

```
#define DisableSD() SDCS = 1; ClockSPI()
#define EnableSD() SDCS = 0
```

Completing the SD/MMC Card Initialization

Before the card can be effectively used for mass storage applications there is a well-defined sequence of commands that need to be completed. This sequence is defined in the original MMC card specifications and has been modified only slightly by the SD card specifications. Since we are not planning on using any of the advanced features specific to the SD card standard, we will use the basic sequence as defined for MMC cards for maximum compatibility. There are five parts of this sequence ([Figure 13.4](#)):

1. The card is inserted into the connector and powered up.
2. The CS line is initially kept high (card not selected).
3. More than 74 clock pulses must be provided before the card becomes capable of receiving commands.
4. The card must then be selected and a *RESET(CMD0)* command provided: the card should respond, entering the Idle state and activating the SPI mode.
5. An *INIT(CMD1)* command is provided and will be repeated until the card exits the Idle state.

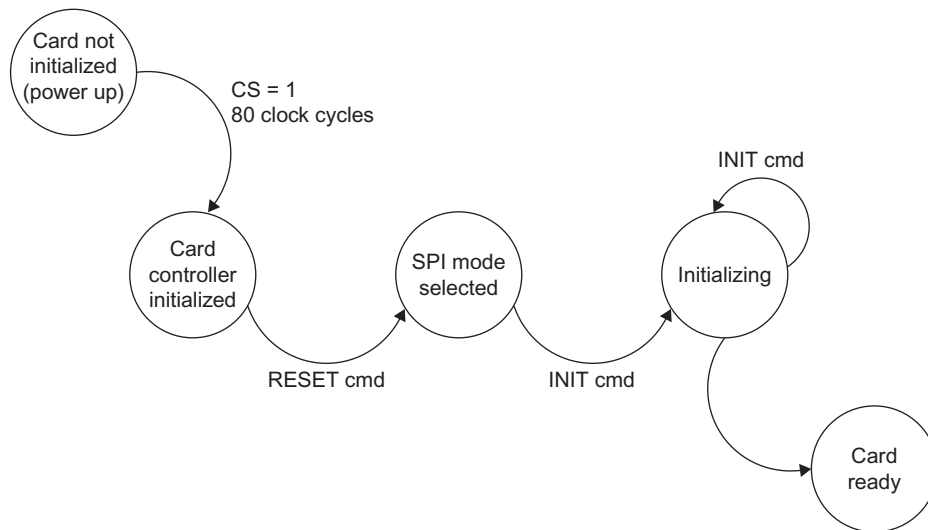


Figure 13.4: SD/MMC card initialization sequence

The following segment of the function *InitMedia()* will perform these initial five steps:

```
int InitMedia( void)
{
    int i, r;

    // 1. while the card is not selected
    DisableSD();

    // 2. send 80 clock cycles to start up
    for ( i=0; i<10; i++)
        ClockSPI();

    // 3. then select the card
    EnableSD();

    // 4. send a reset command to enter SPI mode
    r = SendSDCmd( RESET, 0); DisableSD();
    if ( r != 1)
        return 0x84;

    // 5. send repeatedly INIT
    i = 10000;    // allow for up to 0.3s before timeout
    do {
        r = SendSDCmd( INIT, 0); DisableSD();
        if ( !r) break;
    } while( --i > 0);
    if ( i==0)
        return 0x85;    // timed out
}
```

The initialization command can require quite some time, depending on the size and type of memory card, normally measured in several tenths of a second. Since we are operating at 250kbits/s, each byte sent will require 32 μ s. Accounting for 6 bytes for every command retry and using a timeout count of 10,000 will provide us with a generous timeout limit of approximately two seconds.

It is only upon successful completion of the above sequence that we will be allowed to finally switch gear and increase the clock speed to the highest possible value supported by our hardware. With minimal experimentation you will find that an Explorer16 board, with a properly designed daughter board providing the SD/MMC connector, can easily sustain a clock rate as high as 8MHz. This value can be obtained by reconfiguring the SPI primary prescaler for a 1:1 ratio and using the secondary prescaler for a 1:2 ratio. We can now complete the *InitMedia()* function with the last segment:

```
    // 6. increase speed
    SPISTAT = 0;           // disable momentarily the SPI module
    SPICON1 = 0x013b;      // change prescaler to 1:2
    SPISTAT = 0x8000;      // re-enable the SPI module

    return 0;
} // init media
```

Reading Data From an SD/MMC Card

SD/MMC cards are solid-state devices containing typically large arrays of Flash memory, so we would expect to be able read and write any amount of data (within the card capacity limits) at any desired address. In reality, compatibility considerations with many previous (legacy) mass storage technologies have imposed a number of constraints on how we can access the memory. In fact, all operations are defined in blocks of a fixed size that by default are 512 bytes. It is not a coincidence that 512 bytes is the exact standard size of a data *sector* of a typical personal computer hard disk. Although this can be changed with an appropriate command, we will maintain the default setting to take advantage of this compatibility. We will develop a set of routines that will allow us in the following chapter to implement a complete file system compatible with the most common PC operating systems. This way we will be able to access files written on the card by a personal computer and, vice versa, a personal computer will be able to access files written by our applications.

The *READ_SINGLE (CMD17)* is all we need to initiate a transfer of a single *sector* from a given address in memory. The command takes as an argument a 32-bit *byte address* though, to avoid confusion, in the following we will use uniformly only *LBAs* or block addresses and we will obtain an actual byte address by multiplying the *LBA* value by 512 just before passing the parameter to the *READ_SINGLE* command.

We can use the *SendSDCmd()* function developed above to initiate the read sequence (it will select the card and leave it selected), and after checking the returned response code for errors (there should be none) we will wait for the memory card to send a specific token: *DATA_START*. This uniquely identifies the beginning of the block of data. Again here, as during the initialization phases, it is important to impose a timeout, although we can be generous. Since only the *ReadSPI()* function is called repeatedly (sending/receiving only one byte at a time) while waiting for the data token, a timeout counter of 10,000 will provide an effective time limit of approximately 0.32 seconds (extremely generous).

Once the token is identified, we can confidently read in a rapid sequence all 512 bytes composing the requested block of data. They will be followed by a 16-bit CRC value that we should read although we will have no use for it ([Figure 13.5](#)).

It is only at this point that we will deselect the memory card and terminate the entire read command sequence.

The routine *ReadSECTOR()* below performs the entire sequence in a few lines of code.

```
// SD card responses
#define DATA_START                0xFE

int ReadSECTOR( LBA a, unsigned char *p)
// a      LBA requested
// p      pointer to data buffer
// returns TRUE if successful
```

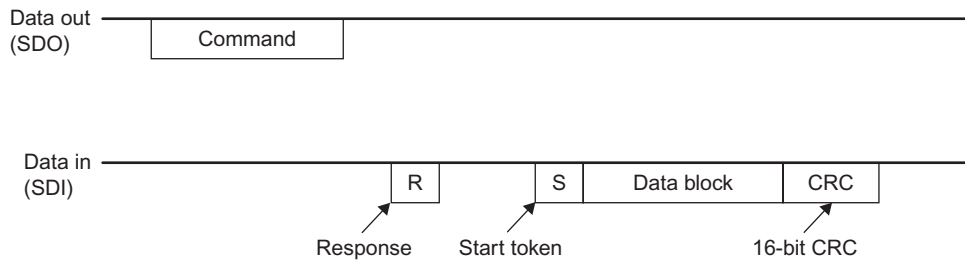


Figure 13.5: Data transfer during a *READ_SINGLE* command

```

{
    int r, i;
    READ_LED = 1;

    r = SendSDCmd( READ_SINGLE, ( a << 9));
    if ( r == 0)    // check if command was accepted
    {
        // wait for a response
        i = 25000;
        do{
            r = ReadSPI();
            if ( r == DATA_START) break;
        }while( --i>0);

        // if it did not timeout, read a 512 byte sector of data
        if ( i)
        {
            for( i=0; i<512; i++)
                *p++ = ReadSPI();

            // ignore CRC
            ReadSPI();
            ReadSPI();

        } // data arrived
    } // command accepted

    // remember to disable the card
    DisableSD();
    READ_LED = 0;

    return ( r == DATA_START);    // return TRUE if successful
} // ReadSECTOR

```

To provide a visual indication of activity on the memory card similarly to hard drives and floppy disk drives, we have assigned one of the LEDs available on the Explorer16 board as the “read” LED hoping this will help prevent a user from removing the card while in use. The LED is turned on before each read command and turned off at the end. Other strategies

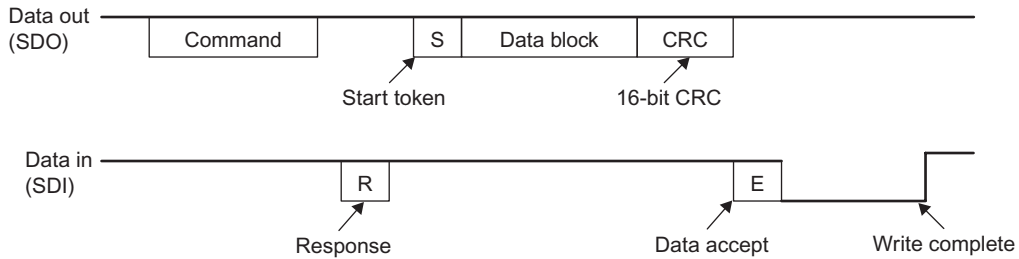


Figure 13.6: Data transfer during a *WRITE_SINGLE* command

are possible though; for example, similarly to what is common practice on USB Flash drives, an LED could be turned on as soon as the card is initialized, regardless of whether an actual command is performed on it at any given point in time. Only calling a de-initialization routine would turn the LED off and indicate to the user that the card can be removed.

Writing Data to an SD/MMC Card

Based on the same consideration we made for the read functions, we will develop a write function that will be similarly constrained to operate on “sectors”; that is, blocks of 512 bytes of data. The write sequence will use, as you would expect, the *WRITE_SINGLE* command, but this time the data transfer will be in the opposite direction. Once we make sure that the command is accepted, we will start immediately sending the *DATA_START* token and, right after it, all 512 bytes of data, followed by two more bytes for the 16-bit CRC (any dummy value will do). At this point we will pause and check that a new token, *DATA_ACCEPT*, is sent by the card. It will confirm that the entire block of data has been received and the write operation has started. While the card is busy writing it will keep the SDO line low. Waiting for the completion of the write command will require a new loop where we will wait for the SDO line to return high. Once more, a timeout must be imposed to limit the amount of time allowed to the card to complete the operation (Figure 13.6). Since all SD/MMC memories are based on Flash memory technology we can expect the time typically required for a write operation to be considerably longer than that required for a read operation. A timeout value of 10,000 would provide us again with a 0.3 s limit, which is more than sufficient to accommodate even the slowest memory card on the market.

```
#define DATA_ACCEPT                0x05

int WriteSECTOR ( LBA a, unsigned char *p)
// a      LBA of sector requested
// p      pointer to sector buffer
// returns TRUE if successful
{
    unsigned r, i;

    WRITE_LED = 1;
```

```
r = SendSDCmd( WRITE_SINGLE, ( a << 9));
if ( r == 0)    // check if command was accepted
{
    WriteSPI( DATA_START);

    for( i=0; i<512; i++)
        WriteSPI( *p++);

    // send dummy CRC
    ClockSPI();
    ClockSPI();

    // check if data accepted
    if ( (r = ReadSPI() & 0xf) == DATA_ACCEPT)
    {
        for( i=65000 ; i>0; i--)
        { // wait for end of write operation
            if ( r = ReadSPI())
                break;
        }
    } // accepted
    else
        r = FAIL;

} // command accepted

// to disable the card and return
DisableSD();
WRITE_LED = 0;

return ( r);    // return TRUE if successful
} // WriteSECTOR
```

Similarly to the read routine, a second LED has been assigned to indicate when a write operation is being performed and potentially alert the user. If the card is removed during the write sequence, data will most likely be lost.

Then let's add a couple of functions for detecting the card presence and the position of the write protect switch.

```
int DetectSD( void)
{
    return ( !SDCD);
} // Detect SD

int DetectWP( void)
{
    return ( !SDWP);
} // Detect WP
```

Notice that the *WP* switch is just providing an indication. It is not connected to a hardware mechanism that would prevent an actual write operation from being performed on the card.

It is your responsibility to decide where and when to check for the presence of the WP and to respect it.

Finally, let's create a new include file called ***SDMMC.h*** to provide the prototypes and basic definitions used in the SD/MMC interface module.

```
/*
** SDMMC.h
**
** SD/MMC card interface
*/
#define FALSE 0
#define TRUE  !FALSE
#define FAIL  0

// IO definitions
#define SD_LED      _RA0
#define READ_LED    _RA1
#define WRITE_LED   _RA2

typedef unsigned long LBA;      // logic block address, 32 bit wide

void InitSD( void);            // initializes the I/O pins
int InitMedia( void);          // initializes the SD/MMC device

int DetectSD( void);           // detects the card presence
int DetectWP( void);           // detects the write protect switch

int ReadSECTOR ( LBA, unsigned char *); // reads a block of data
int WriteSECTOR ( LBA, unsigned char *); // writes a block of data
```

Using the SD/MMC Interface Module

Whether you believe it or not, the six minuscule routines we have just developed are all we need to gain access to the seemingly unlimited amount of storage space offered by the SD/MMC memory cards. For example, a 512 Mbyte card would provide us with approximately 1,000,000 (yes that is one million) individually addressable memory blocks (sectors) each 512 bytes large. Note that, as of this writing, SD/MMC cards of this capacity are normally offered for retail in the USA for less than \$5!

Let's develop a small test program to demonstrate the use of the SD/MMC module. The idea is to simulate a somewhat typical application that is required to save some large amount of data on the SD/MMC memory card. A fixed number of blocks of data will be written in a predetermined range of addresses and then read back to verify the successful completion of the process.

Let's create a new source file that we will call ***SDMMCTest.c*** and start by adding the usual *config.h* file followed by the *SDMMC.h* include file.

```
/*
** SDMMCTest.c
**
```

```
** Read/write Test
*/
#include <configure.h>
#include <EX16.h>
#include <SDMMC.h>
```

Then let's define two byte arrays each the size of a default SD/MMC memory block, that is 512 bytes.

```
#define B_SIZE      512          // sector/data block size
unsigned char      data[ B_SIZE];
unsigned char      buffer[ B_SIZE];
```

The test program will fill the first with a specific and easy to recognize pattern, and will repeatedly write its contents onto the memory card. The chosen address range will be defined by two constants:

```
#define START_ADDRESS      10000  // start block address
#define N_BLOCKS           1000   // number of blocks
```

The LEDs on PORTA of the Explorer16 demonstration board will provide us with a visual feedback about the correct execution of the program and/or any error encountered.

The first few lines of the main program can now be written to initialize the I/Os required by the SD/MMC module and the PORTA pins connected to the row of LEDs.

```
main( void)
{
    LBA addr;
    int i, r;

    // I/O initializations
    TRISA = 0;          // initialize PORTA LED outputs
    InitSD();           // initialize I/Os required for the SD/MMC card

    // fill the buffer with "data"
    for( i=0; i<B_SIZE; i++)
        data[i]= i;
```

The next code segment will have to check for the presence of the SD card in the slot/connector. We will wait in a loop for the card detection switch if necessary and we will provide an additional delay for the contacts to properly debounce.

```
// wait for card to be inserted
while( !DetectSD())    // assumes SDCD pin is by default an input
    Delays( 100);      // wait for card contacts de-bounce and power up
```

We will be generous with the debouncing delay as we want to make sure that the card connection is stable before we start firing *write* commands that could otherwise potentially corrupt other data present on the card. A 100 ms delay is a reasonable delay to use and the *Delays()* function is taken from the *EX16.h* library module defined in earlier chapters.

Keeping the de-bouncing delay function separate from the *DetectSD()* function and the SD/MMC module in general is important as this will allow different applications to pick and choose the best timing strategy and optimize the resources allocation.

Once we are sure that the card is present, we can proceed with its initialization, calling the *InitMedia()* function.

```
// initialize the memory card (returns 0 if successful)
r = InitMedia();
if ( r)                // could not initialize the card
{
    PORTA = r;          // show error code on LEDs
    while( 1);         // halt here
}
```

The function returns an integer value, which is zero for a successful completion of the initialization sequence, or a specific error code otherwise. In our test program, in the case of an initialization error we will simply publish the error code on the LEDs and halt the execution entering an infinite loop. The codes 0x84 and 0x85 will indicate that respectively the *InitMedia()* function steps 4 or 5 have failed, corresponding to an incorrect execution of the card *RESET* command and card *INIT* commands (failure or timeout) respectively.

If all goes well we will be able to proceed with the actual data writing phase.

```
else
{
    // fill N_BLOCK blocks/SECTOR with the contents of data buffer
    addr = START_ADDRESS;
    for( i=0; i<N_BLOCKS; i++)
        if (!WriteSECTOR( addr+i, data))
        { // writing failed
            PORTA = 0x0f;
            while( 1); // halt here
        }
}
```

The simple *for* loop performs repeatedly the *WriteSECTOR()* function over the address range from block 10,000 to block 10,999, copying over and over the same data block and verifying at each step that the write command is performed successfully. If any of the block write commands returns an error, a unique code (0x0f) will be presented on the LEDs and the execution will be halted. In practice this will be equivalent to writing a file of 512,000 bytes.

```
// verify the contents of each block/SECTOR written
addr = START_ADDRESS;
for( i=0; i<N_BLOCKS; i++)
{ // read back one block at a time
    if (!ReadSECTOR( addr+i, buffer))
    { // reading failed
```

```

        PORTA = 0xf0;
        while( 1); // halt here
    }

    // verify each block content
    if ( memcmp( data, buffer, B_SIZE))
    { // mismatch
        PORTA = 0x55;
        while( 1); // halt here
    }
} // for each block

```

Next we will start a new loop to read back each data block into the second buffer and we will compare its contents with the original pattern still available in the first buffer. If the *ReadSECTOR()* function should fail, we will present an error code (*0xf0*) on the LED display and terminate the test. Otherwise, a standard C library function *memcmp()* will help us perform a fast comparison of the buffer contents, returning an integer value that is zero if the two buffers are identical as we hope, not zero otherwise. Once more, a new unique error indication (*0x55*) will be provided if the comparison should fail. To gain access to the *memcmp()* function that belongs to the standard C *string.h* library.

We can now complete the main program with a final indication of successful execution, lighting up all LEDs on PORTA.

```

    } // else media initialized

    // indicate successful execution
    PORTA = 0xFF;
    // main loop
    while( 1);

} // main

```

If you have added all the required source files *SDMMC.h*, *EX16.h*, *EX16.c*, *SDMMC.c* and *SDMMCTest.c* to the project, you can now launch the project by using the **Run>Run Project** command. You will need a daughter board with the SD/MMC connections as described at the beginning of the lesson to actually perform the test. But the effort of building one (or the expense of purchasing one) will be more than compensated for by the joy of seeing the PIC24 perform the test flawlessly in a fraction of a second. The amount of code required was also impressively small (Figure 13.7).

...			
Total program memory used (bytes):	0x55e	(1374)	1%
...			
Total data memory used (bytes):	0x450	(1104)	13%

Figure 13.7: MPLAB® C compiler memory usage report

Altogether, the test program and the SDMMC access module have used up only 1,374 bytes of the processor FLASH program memory, that is, less than 1% of total memory available, and 1,104 bytes of RAM (2×512 buffers + stack), which is less than 15% of the total RAM memory available. As in all previous lessons this result was obtained with the compiler optimization options set to level 1, available in the free evaluation version of the compiler.

Post-Flight Briefing

In my personal opinion it does not get cheaper or easier than this with any other mass storage technology. After all, we can use only a handful of pull-up resistors, a cheap connector, and just a few I/O pins to expand enormously the storage capabilities of our applications. In terms of PIC24 resources required, only the SPI peripheral module has been used and even that could be shared with other applications.

The simplicity of the approach has its obvious limitations though. Data can be written only in blocks of fixed size and its position inside the memory array will be completely application-specific. In other words, there will be no way to share data with a personal computer or other device capable of accessing SD/MMC memory cards unless a *custom* application is developed.

Note

If an attempt is made to use a card already used (formatted) by a PC, data would probably be corrupted and the entire card might require complete reformatting. In the next lesson we will address these issues by developing a complete file system library compatible with PC applications.

Tips & Tricks

The choice of operating on the default block size of 512 bytes was dictated mostly by historical reasons. By making the low-level access routines in this lesson conform with the standard size adopted by most other mass storage media devices (including hard drives) we made developing the next layer (the file system) easier. But if we were looking for maximum performance, this could have been the wrong choice. In fact, if we were looking for faster write performance, typically the bottleneck of all Flash memory media, we would be better off looking at much larger data blocks. Flash memory offers typically very fast access to data (reading) but is relatively slow when it comes to writing. Writing requires two steps: first a large block of data (often referred to as a page) must be erased; then the actual writing can be performed on smaller blocks. The larger the memory array, the larger, proportionally, the erase page size will be. For example, on a 512-Mbyte memory card the erase page can easily exceed 2 Kbytes. While these details are typically hidden from the user as the main controller inside the card takes care of the erase/write sequencing and buffering, this can have an impact on the overall performance of the application. In fact, if we assume a specific SD card has

a 2-Kbyte page, writing any amount of data (<2 Kbyte) would require the internal card controller to perform the following steps:

1. Read the contents of an entire 2-Kbyte block in an internal buffer.
2. Erase it, and wait for the erase-time.
3. Replace a portion of the buffer content with the new data.
4. Write back the entire 2-Kbyte block, and wait for the write-time.

By performing write operations only on blocks of 512 bytes each, to write 2 Kbytes of data our library would have to ask the SD card controller to perform the entire sequence four times, while it could be done in just one sequence by changing the data block length or using a multiple-block write command. While this approach could theoretically increase the writing speed by 400% in the example above, consider carefully the option as the price to pay could be quite high. In fact, consider the following drawbacks:

- The actual memory page size might not be known or guaranteed by the manufacturer, although betting on increasing densities of Flash media (and therefore increasing page size) is pretty safe.
- The size of the RAM buffer to be allocated inside the PIC24 application is increased and this is a precious resource in any embedded application.
- The higher software layers (which we will explore in the next lesson) might be more difficult to integrate if the data block size varies.
- The larger the buffer, the larger is the data loss if the card is removed before the buffer is flushed.

Exercises

- Experiment with various data block sizes to identify where your SD card provides the best write performance. This will give you an indirect indication of the actual page size of the Flash memory device used by the card manufacturer.
- Experiment with multiple-block write commands or changing the block length to verify how the internal buffering is performed by the SD card controller and whether the two methods are equivalent.

Books

- Axelson, J., 2006. *USB Mass Storage: Designing and Programming Devices and Embedded Hosts*. Lakeview Research, Madison, WI.
This book continues on the excellent series on USB by Jan Axelson. While low-level interfacing directly to a SD/MMC card was easy, as you have seen in this chapter, creating a proper USB interface to a mass storage device is a project of a much higher order of complexity.

Links

- <http://www.mmca.org/home>
The official website of the MultiMediaCard Association, MMCA.
- <http://www.sdcard.org/>
The official website of the Secure Digital Card Association, SDCA.
- <http://www.sdcard.org/sdio/Simplified%20SDIO%20Card%20Specification.pdf>
The simplified SDIO card specifications. With SDIO, the SD interface is no longer used only for mass storage, but is also a viable interface for a number of advanced peripherals and gizmos, such as GPS receivers, digital cameras and more.

File I/O

Every flight during the training should have a precise purpose assigned by the instructor or inspired by the course syllabus used by the school. In each and every lesson, we stated our purpose in a section we called the Flight Plan, but in aviation an actual flight plan is a different thing. It is a very detailed list containing the times, altitudes, headings, fuel consumption figures, etc. for all the segments (legs) composing the flight. For cross-country flights this is an essential tool that will help the pilot stay ahead of the game and be constantly aware of his position and his options in case of emergency. Officially filing the flight plan, calling a Flight Service Station (FSS) and dictating the plan on the phone to a controller, or submitting it via the internet, gives additional advantages. Once the FSS (and ultimately the FAA) knows where, when and along which route you are going, they can keep an eye on you, so to speak. They can track you on their radars (a service called flight following), and as a minimum, if you are flying too low for them to follow you, they can check that you actually reached your destination at the estimated arrival time or within a reasonable period. If they don't hear from you or at the destination airport there is no record of your arrival, they will immediately start a search operation and, especially in extreme climates, over mountainous terrain and uninhabited areas, this prompt reaction could be crucial to your life. When it comes to filing flight plans, most pilots have mixed feelings. It feels a bit like when you are teenager and you have to let mom know where you are going to spend the evening, you hate having to do it, although you understand that it is for your own good. Sharing information with mom, I mean the FAA, requires a little effort, but it brings great benefits.

In embedded control, sharing files (information) with a PC can be of great benefit, but you have to know the rules, that is, you need to know how PC file systems work.

Flight Plan

In the previous lesson we developed a basic interface module (both software and hardware) to gain access to an SD™/MMC card and support applications that require large amounts of data storage. A similar interface could be built for several other types of mass storage media but, in this lesson, we will rather focus on the algorithms and data structures required to properly share information on the mass storage device with the most common PC operating systems (DOS, Windows, and some Linux distributions). In other words, we will develop a module for access to a standard file system known commonly as FAT16. The first FAT file system was created by Bill Gates and Marc McDonald in 1977 for managing disks in Microsoft

Disk BASIC. It used techniques that had been available in file systems many years prior and it continued to evolve in numerous versions over the last few decades to accommodate ever larger-capacity mass storage devices and adding new features. Among the many versions still in use today the FAT12, FAT16 and FAT32 are the most common ones. FAT16 and FAT32 in particular are recognized by practically every PC operating system currently in use and the choice between the two is mostly dictated by efficiency considerations and the capacity of the media. Ultimately, for most Flash mass storage devices in common use in consumer multimedia applications, FAT16 is still the file system of choice.

The Flight

The name FAT is an acronym that stands for File Allocation Table, which is also the name of one of the most important data structures used in this file system. After all, a file system is just a method for storing and organizing computer files and the data they contain to make it easy to find and access them. Unfortunately, as often is the case in the history of personal computing, standards and technologies are the fruit of constant evolutionary progress rather than original creation, and for this reason many of the details of the FAT file system we will reveal in the following can only be explained in the context of a struggle to continue and maintain compatibility with an enormous mass of legacy technologies and software over many years.

Sectors and Clusters

Still, the basic ideas at the root of a FAT file system are quite simple. As we have seen in the previous lesson, most mass storage devices follow a “tradition” derived from the hard disk technology of managing memory space in blocks of a fixed size of 512 bytes, commonly referred to as “sectors”. In a FAT file system a small number of these sectors are reserved and used as a sort of general index: the File Allocation Table. The remaining (majority) of the sectors are available for proper data storage, but instead of being handled individually, small groups of contiguous sectors are handled jointly to form new larger entities known as “clusters” (Figure 14.1). Clusters can be as small as one single sector, or can commonly be formed by as many as 64 sectors. It is the use of each cluster and its position that is tracked inside the File Allocation Table. Therefore clusters are the true smallest unit of memory allocation in a FAT file system.

The simplified diagram in Figure 14.1 illustrates a hypothetical example of a FAT file system formatted for 1022 clusters, each composed of 16 sectors. (Notice that the data area starts with cluster number 2.) In this example each cluster would contain 8 Kbytes of data and the total storage capacity would be about 8 MB.

Note that, the larger clusters are, the fewer will be required to manage the entire memory space and the smaller the allocation table required, hence the higher efficiency of the file

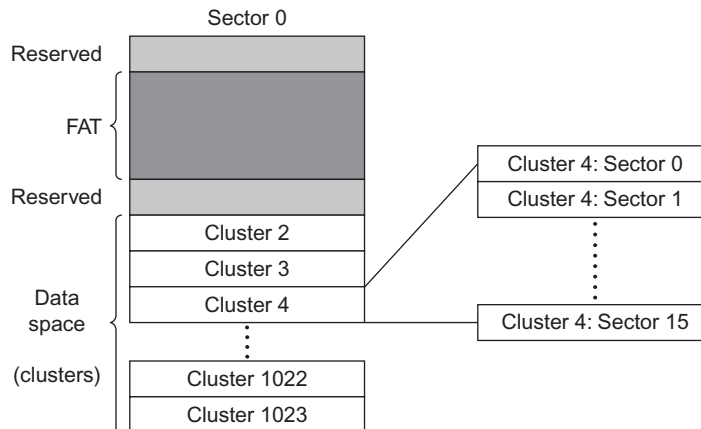


Figure 14.1: Simplified example of a FAT file system layout

system. On the other hand, if many small files are to be written, the larger the cluster size the more space will be wasted. It is typically the responsibility of the operating system, when formatting a storage device for use with a FAT file system, to decide the ideal cluster size to be used for an optimal balance.

The File Allocation Table

In the FAT16 file system, the file allocation table itself contains one 16-bit integer value for each and every cluster. If the cluster is to be considered empty and available, the corresponding position in the table will contain the value 0x0000. If a cluster is in use and it contains an entire file of data, its corresponding position in the table will contain the value 0xFFFF. If a file is larger than the size of a single cluster, a chain of clusters is formed. In the FAT table each cluster position in order will contain the number of the following cluster in the chain. The last cluster in the chain will have in the corresponding table position the value 0xFFFF. Additionally, certain unique values are used to mark reserved clusters (0x0001) and bad clusters (0xFFFF7). The fact that 0x0000 and 0x0001 have been assigned special meanings is the fundamental reason for the convention of starting the data area with cluster number 2. In the FAT table correspondingly the first two 16-bit integers are reserved.

In [Figure 14.2](#) you can see an example of the content of a FAT table for the system presented in our previous example. Clusters 0 and 1 are reserved. Cluster 2 appears to contain some data, meaning that some or all of the (16) sectors forming the cluster have been filled with data from a file whose size must have been less than 8 KB.

Cluster 3 appears to be the first cluster in a chain of three that also includes clusters 4 and 5. All of the cluster 3 and 4 sectors and some or all of the cluster 5 sectors must have been filled

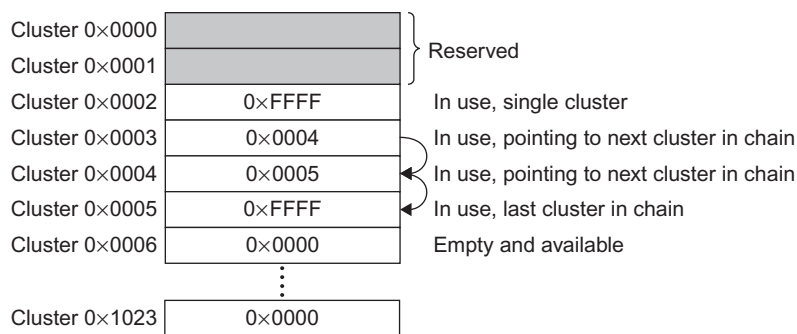


Figure 14.2: Cluster chains in a file allocation table

with data from a file whose size (we can only assume so far) was more than 16 KB but less than 24 KB. All following clusters appear to be empty and available.

Notice that the size of a FAT table itself is dictated by the total number of clusters multiplied by two (two bytes per cluster) and can spread over multiple sectors. In our previous example, a FAT table of 1024 clusters would have required 2048 bytes or four sectors of 512 bytes each. Also, since the file allocation table is perhaps the most critical structure in the entire FAT file system, multiple copies (typically two) are maintained and allocated one after the other before the beginning of the data space.

The Root Directory

The role of the FAT table is that of keeping track of how and where data is allocated. It does not contain any information about the nature of the file to which the data belonged. For that purpose there is another structure called the root directory whose sole purpose is that of storing file names, sizes, dates, times and a number of other attributes. In a FAT16 file system, the root directory (or simply the root from now on) is allocated in a fixed amount of space and a fixed position right between the FAT (second copy) and the first data cluster (Figure 14.3).

Since both position and size (number of sectors) are fixed, the maximum number of files (or directory entries) in the root directory is limited and determined when formatting the media. Each sector allocated to the root will allow for 16 file entries to be documented where each entry will require a block of 32 bytes as represented in Figure 14.4.

The Name and Extension fields are the most obvious if you are familiar with the older Microsoft operating systems using the 8:3 conventions (the two fields need only to be padded with spaces and the dot can be discarded).

The Attributes field is composed of a group of flags with meanings shown in Table 14.1.

The Time and Date fields refer to the last time the file was modified and must be encoded in a special format to compress all the information in just two 16-bit words (Tables 14.2 and 14.3).

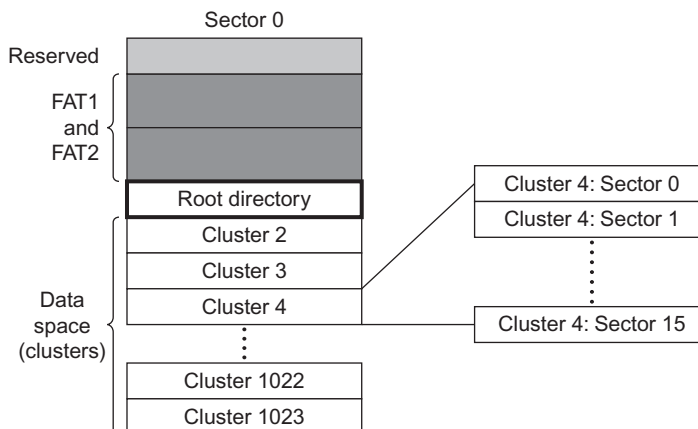


Figure 14.3: Example of a FAT file system layout

Table 14.1: File attributes in a directory entry

Bit	Mask	Description
0	0x01	Read only
1	0x02	Hidden
2	0x04	System
3	0x08	Volume label
4	0x10	Subdirectory
5	0x20	Archive

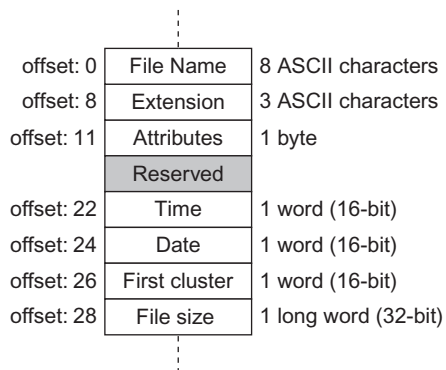


Figure 14.4: Basic root directory entry structure

Table 14.2: Time encoding in a directory entry field

Bits	Description
15–11	Hours (0–23)
10–5	Minutes (0–59)
4–0	Seconds/2 (0–29)

Table 14.3: Date encoding in a directory entry field

Bits	Description
15–9	Year (0 = 1980, 127 = 2107)
8–5	Month (1 = January, 12 = December)
4–0	Day (1–31)

Notice how the date field encoding does not allow for the code 0x0000 to be interpreted as a valid date, helping provide clues to the file system when the field is not used or corrupted.

The First Cluster field provides the fundamental link with the FAT table. The 16-bit word it contains is nothing but the number of the cluster (could be the only or the first in a chain) containing the file data.

Finally, the Size field contains in a long integer (32-bit), the size in bytes of the file data.

Looking at the first character of the file name in a directory entry we can also tell whether the entry is currently in use, in which case an actual ASCII printable character is present, or whether the entry is empty, in which case the first byte is a zero, and we can also assume that the list of files is terminated as the file system proceeds sequentially using all entries in order. There is a third possibility: in fact, when a file is removed from the directory the first character is simply replaced by a special code (0xE5). This indicates that the contents of the entry are no longer valid, and the entry can be re-used for a new file at the next opportunity. When browsing through the list though, searching for a file, we should continue as more active entries might follow.

The Treasure Hunt

There would be much more to say to fully document the structure of a FAT16 file system but if you have followed the introduction so far, you should have now a reasonable understanding of its core mechanisms and you will be ready to dive in for more detail as we will soon start writing some code.

So far we have maintained a certain level of simplification by ignoring some fundamental questions such as:

- Where do we learn about a storage device capacity?
- How can we tell where the FAT table is located?
- How can we tell how many sectors are in each cluster?
- How can we tell where the data space starts?

The answer to all those questions will be found soon by following a sequence of steps that might somewhat resemble a child's treasure hunt. We will start using the *sdmmc.c* module functions developed in the previous lesson, initializing the I/Os with the *InitSD()* function first and checking for the presence of the card in the slot.

```
// 0. init the I/Os
InitSD();

// 1. check if the card is in the slot
if (!DetectSD())
{
    FError = FE_NOT_PRESENT;
    return NULL;
}
```

We will proceed then initializing the storage device with the *InitMedia()* function.

```
// 2. initialize the card
if ( InitMedia())
{
    FError = FE_CANNOT_INIT;
    return NULL;
}
```

We will also use the standard C library *stdlib.h* to allocate dynamically two data structures:

```
// 3. allocate space for a MEDIA structure
D = (MEDIA *) malloc( sizeof( MEDIA));
if ( D == NULL)          // report an error
{
    FError = FE_MALLOC_FAILED;
    return NULL;
}

// 4. allocate space for a temp sector buffer
buffer = (unsigned char *) malloc( 512);
if ( buffer == NULL)     // report an error
{
    FError = FE_MALLOC_FAILED;
    free( D);
    return NULL;
}
```


The first one, which will be fully revealed later, is a structure that we will call *MEDIA* and will be the place where we will collect the answer to all the questions above (perhaps a more appropriate name would have been *TREASURE?*).

The second structure, *buffer*, is simply a 512-byte-large array that will be used to retrieve sectors of data during the hunt.

Notice that to allow the *malloc()* function to successfully allocate memory, you will have to remember to reserve some RAM space for the Heap. Hint: follow the “Project Build” checklist to learn how to reach and modify the linker settings of your project.

Mostly historical reasons dictate that the first sector (address 0) of each mass storage device will contain what is commonly known as a *Master Boot Record (MBR)*.

Here is how we invoke the *ReadSECTOR()* function for the first time to access the Master Boot Record.

```
// 5. get the Master Boot Record
if ( !ReadSECTOR( 0, buffer))
{
    FError = FE_CANNOT_READ_MBR;
    free( D); free( buffer);
    return NULL;
}
```

A signature, consisting of a specific word value (*0x55AA*) present in the last word of the MBR sector, will confirm that we have indeed read the correct data.

```
#define FO_SIGN          0x1FE // MBR signature location (55,AA)

// 6. check if the MBR sector is valid
//      verify the signature word
if (( buffer[ FO_SIGN] != 0x55) ||
    ( buffer[ FO_SIGN +1] != 0xAA))
{
    FError = FE_INVALID_MBR;
    free( D); free( buffer);
    return NULL;
}
```

Once upon a time, this record used to contain actual code to be executed by a PC upon power-up. No personal computer does this anymore though and certainly there is no use for that 8086 code for our PIC24 applications. Actually, most of the time, you will find the Master Boot Record sector to be empty, mostly filled with zeros, except for one small area starting at offset *0x1BE* (Figure 14.5). This is where we will find what is called a *Partition Table*, a table (with only four entries containing 16 bytes each) that has no use on a relatively small memory card like our SD/MMC, but that is kept for compatibility reasons and is identical to the hard disk partition tables you might have used on your PC.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Access ▼
00000000	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000190	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	03
000001C0	35	00	06	08	D8	C1	F1	00	00	00	0F	C9	0E	00	00	00	5...0A7...E...
000001D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA		U#

Figure 14.5: Hex dump of an MBR sector

In our applications it is safe to assume that the entire card will have been formatted in a single partition and that this will be the first and only entry (16-byte block) in the table. Of those 16 bytes we will need only a few to deduce the partition size (should include the entire card), the starting sector and most importantly the type of file system contained. A couple of macros will help us assemble the data from the buffer into words and long words:

```
#define ReadW( a, f) *(unsigned short*)(a+f)
#define ReadL( a, f) *(unsigned long *)(a+f)
```

Also, the following definitions will point us to the right offset in the MBR.

```
#define FO_FIRST_P    0x1BE // offset of first partition table
#define FO_FIRST_TYPE 0x1C2 // offset of first partition type
#define FO_FIRST_SECT 0x1C6 // first sector of first partition offset
#define FO_FIRST_SIZE 0x1CA // number of sectors in partition
```

```
// 7. read the number of sectors in partition
psize = ReadL( buffer, FO_FIRST_SIZE);

// 8. check if the partition type is acceptable
```

```
i = buffer[ FO_FIRST_TYPE];
switch ( i)
{
    case 0x04:
    case 0x06:
    case 0x0E:
        // valid FAT16 options
        break;
    default:
        FError = FE_PARTITION_TYPE;
        free( D); free( buffer);
        return NULL;
} // switch
```

For historical reasons, there are several codes that correspond to a FAT16 file system that we will be able to correctly decode including 0x04, 0x06 and 0x0E.

Next, we will need to extract the long word (32-bit) value found at offset *FO_FIRST_SECT* (0x1C6), in the first partition table entry, to proceed in the treasure hunt.

```
// 9. get the first partition first sector -> Boot Record
firsts = ReadL( buffer, FO_FIRST_SECT);
```

It contains the address of the next sector that we need to read from the device.

```
// 10. get the sector loaded (boot record)
if ( !ReadSECTOR( firsts, buffer))
{
    free( D); free( buffer);
    return NULL;
}
```

It has a signature, similarly to the Master Boot Record, located in the last word of the sector and we need to verify it before proceeding.

```
// 11. check if the boot record is valid
//      verify the signature word
if (( buffer[ FO_SIGN] != 0x55) ||
    ( buffer[ FO_SIGN +1] != 0xAA))
{
    FError = FE_INVALID_BR;
    free( D); free( buffer);
    return NULL;
}
```

It is called the (*First Partition*) *Boot Record*, and once more it is supposed to contain actual executable code that is of no value to us ([Figure 14.6](#)).

Fortunately, in the same record at fixed and known positions, there are some more of the answers we were looking for and other elements that will help us calculate the rest and

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	Access
0001E200	EB	00	90	20	20	20	20	20	20	20	00	02	20	01	00	00	È.
0001E210	02	00	02	00	00	F8	77	00	3F	00	10	00	F1	00	00	00sw.?...È...
0001E220	0F	C9	0E	00	80	00	29	13	18	FD	E0	20	20	20	20	20	È. .) . .yà
0001E230	20	20	20	20	20	20	46	41	54	31	36	20	20	20	00	00	FAT16
0001E240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E250	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E260	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E270	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E280	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E290	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E2F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E310	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E330	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E350	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E370	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E380	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E390	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0001E3F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	U3

Figure 14.6: Hex dump of a boot record

complete the map of the entire FAT16 file system. These are the key offsets in the Boot Record buffer:

```
// Partition Boot Record key fields offsets
#define BR_SXC      0xd      // (byte) sectors per cluster
#define BR_RES      0xe      // (word) reserved sectors for boot record
#define BR_FAT_SIZE 0x16     // (word) FAT size in number of sectors
#define BR_FAT_CPY  0x10     // (byte) number of FAT copies
#define BR_MAX_ROOT 0x11     // (odd word) max entries in root dir
```

And with the following code we can calculate the size of a cluster:

```
// 12. determine the size of a cluster
D->sxc = buffer[ BR_SXC];
// this will also act as flag that the media is mounted
```

Determine the position of the FAT table, its size and the number of copies:

```
// 13. determine FAT, root and data LBAs
// FAT = first sector in partition
```

```
// (boot record) + reserved records
D->fat = firsts + ReadW( buffer, BR_RES);
D->fatsize = ReadW( buffer, BR_FAT_SIZE);
D->fatcopy = buffer[ BR_FAT_COPY];
```

Find the position of the Root Directory too:

```
// 14. ROOT = FAT + (sectors per FAT * copies of FAT)
D->root = D->fat + ( D->fatsize * D->fatcopy);
```

But careful now, as we get ready to grab the last few pieces of gold, watch out for a trap!

```
// 15. MAX ROOT is the maximum number of entries
//      in the root directory
D->maxroot = ReadW( buffer, BR_MAX_ROOT);
```

Can you see it? No? OK, here is a hint. Look at the value of the *BR_MAX_ROOT* offset as defined a few lines above. You will notice that this is an odd address (*0x11*). This is all it takes to the *ReadW()* macro, which attempts to use it as a word address, to throw a processor trap and reset the PIC24!

We need a special macro (perhaps less efficient) but one that can assemble a word one byte at a time without falling in the trap!

```
// these are the safe versions of ReadW to be used on odd address fields
#define ReadOddW( a, f) (*(a+f) + (*(a+f+1) << 8))

// 15. MAX ROOT is the maximum number of entries in the root directory
D->maxroot = ReadOddW( buffer, BR_MAX_ROOT);
```

The last two pieces of information are easy to grab now. With them, we learn where the data area (divided in clusters) begins and how many clusters are available to our application:

```
// 16. DATA = ROOT + (MAXIMUM ROOT *32 / 512)
D->data = D->root + ( D->maxroot >> 4); // assuming maxroot % 16 == 0!!!

// 17. max clusters in this partition = (tot sectors - sys sectors)/sxc
D->maxcls = (psize - (D->data - firsts)) / D->sxc;
```

It took us as many as 17 careful steps to get to the treasure, but now we have all the information needed to fully figure out the layout of the FAT16 file system present on the SD/MMC memory card (or for that reason almost any other mass storage media). The treasure, then, is nothing more than another map, a map we will need now to find the files on the mass storage device (Figure 14.7).

I can now reveal to you the definition of the entire *MEDIA* structure that we allocated on the Heap at the very beginning and we have been so patiently filling. Here is where we will keep the treasure safe.

```
typedef struct {
    LBA      fat;           // lba of FAT
```

```

LBA    root;           // lba of root directory
LBA    data;           // lba of the data area
unsigned maxroot;      // max number of entries in root dir
unsigned maxcls;       // max number of clusters in partition
unsigned fatsize;      // number of sectors
unsigned char fatcopy;  // number of FAT copies
unsigned char sxc;     // number of sectors per cluster
} MEDIA;

```

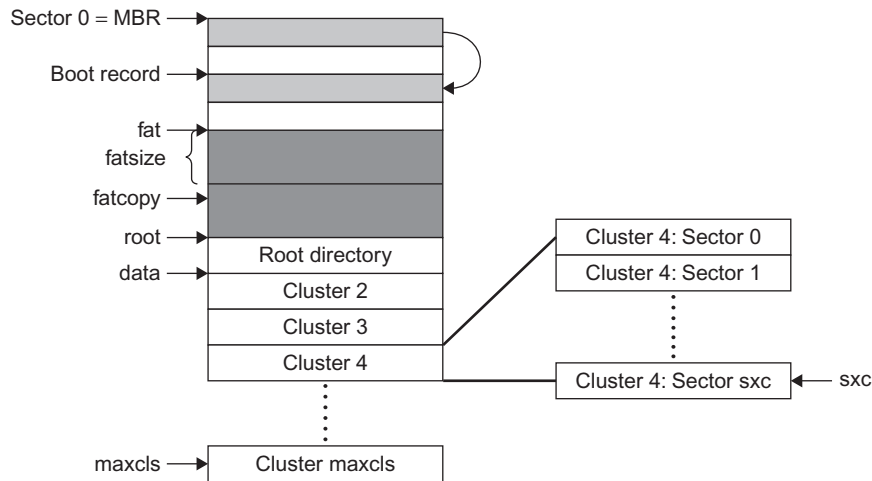


Figure 14.7: We found the treasure, it is the FAT16 complete layout

We can now assemble all the steps in one essential function that we can call *mount()* for its similarity to a function available in the Unix family of operating systems.

For a mass storage device to be used in Unix, the file system present on the device must be *mounted* or, in other words, attached as a new branch of the main file system. Windows users might not be familiar with the concept as they don't have the option to choose whether, when or where the new device file system will be mounted. All new mass storage devices are automatically and unconditionally mounted by Windows at power-up, or after insertion for removable media, at the very root of the Windows file system by assigning them a single-letter identifier (*C:*, *D:*, *E:*, etc).

```

//-----
// mount initializes a MEDIA structure for FILEIO access
//
MEDIA * mount( void)
{
    LBA psize;        // number of sectors in partition
    LBA firsts;       // first sector inside the first partition
    int i;
    unsigned char *buffer;

```

```
... insert here all 17 steps of our treasure hunt

// 18. free up the temporary buffer
free( buffer);
return D;

} // mount
```

We will also define a global pointer to a *MEDIA* structure (*D*) to be used to hold the pointer returned by the *mount()* function. It will serve as the starting point for the entire file system. Initially, we will assume that only one storage device is available at any given point in time (one connector/slot, one card).

```
// global definitions
MEDIA *D;
```

We will also define a *umount()* function that will have the sole duty of releasing the space allocated for the *MEDIA* structure.

```
//-----
// umount   releases the space allocated for the MEDIA structure
//
void umount( void)
{
    free( D);
    D = NULL;
} // umount
```

Opening a File

Now that we have figured out the map of the storage device, we can start pursuing our original objective: accessing individual files. In practice, what we will develop in the rest of this lesson is a set of high-level functions similar to those found in most operating systems for file manipulation. We will need a function to find a file location on the storage device, one for sequentially reading the data from the file and possibly one more to write data and create new files.

In a logical order, we will start developing what we will call the *fopenM()* function. Its role will be that of finding all possible information regarding a file (if present) and gathering it in a new structure that we will call *MFILE*. The name of this structure was chosen to avoid conflicts with similar structures and functions defined inside the standard C library *stdio.h*.

```
typedef struct {
    MEDIA * mda;           // media structure pointer
    unsigned char * buffer; // sector buffer
    unsigned short cluster; // first cluster
    unsigned short ccls;    // current cluster in file
    unsigned short sec;     // sector in current cluster
    unsigned short pos;     // position in current sector
```

```

unsigned short top;           // bytes in the buffer
long      seek;              // position in the file
long      size;              // file size
unsigned short time;         // last update time
unsigned short date;         // last update date
char      name[11];          // file name
char      mode;              // mode 'r', 'w'
unsigned short fpage;        // FAT page currently loaded
unsigned short entry;        // entry position in cur dir
} MFILE;

```

I know at first sight it looks like a lot, it is more than 40 bytes wide, but, as you will see in the rest of the lesson, we will end up needing all of them. You will have to trust me for now.

Mimicking standard C library implementations (common to many operating systems) the *fopenM()* function will receive two (ASCII) string parameters: the file name and a *mode* string that will indicate whether the file is supposed to be opened for reading or writing.

```

MFILE *fopenM( const char *filename, const char *mode)
{
    char c;
    int i, r, e;
    unsigned char *b;
    MFILE *fp;

```

To optimize memory usage a MFILE structure is allocated only when necessary, and it is in fact one of the first tasks of the *fopenM()* function and a pointer to the data structure is its return value. In case *fopenM()* should fail, a *NULL* pointer will act as an error report.

Of course, a prerequisite for opening a file is to have the storage device file system mapped out, and that is supposed to have already been performed by the *mount()* function. A pointer to a *MEDIA* structure must have already been deposited in the global *D* pointer.

```

// 1. check if a storage device is mounted
if ( D == NULL)    // unmounted
{
    FError = FE_MEDIA_NOT_MNTD;
    return NULL;
}

```

Since all activity with the storage device must be performed in blocks of 512 bytes, we will need that much space to be allocated for us to act as a read/write buffer.

```

// 2. allocate a buffer for the file
b = (unsigned char*)malloc( 512);
if ( b == NULL)
{
    FError = FE_MALLOC_FAILED;
    return NULL;
}

```


Only if that amount of memory is available we can proceed and allocate some more memory for the *MFILE* structure proper.

```
// 3. allocate a MFILE structure on the heap
fp = (MFILE *) malloc( sizeof( MFILE));
if ( fp == NULL)          // report an error
{
    FError = FE_MALLOC_FAILED;
    free( b);
    return NULL;
}
```

The buffer pointer and the *MEDIA* pointers can now be recorded inside the *MFILE* data structure.

```
// 4. set pointers to the MEDIA structure and buffer
fp->mda = D;
fp->buffer = b;
```

The file name parameter must be extracted; each character must be translated to upper case (using the standard C library functions contained defined in *ctype.h*) and padded, if necessary, with spaces to an eight-character length.

```
// 5. format the filename into name
for( i=0; i<8; i++)
{
    // read a char and convert to upper case
    c = toupper( *filename++);
    // extension or short name no-extension
    if (( c == '.' ) || ( c == '\0'))
        break;
    else
        fp->name[i] = c;
} // for
// if short fill the rest up to 8 with spaces
while ( i<8) fp->name[i++] = ' ';
```

Similarly, after discarding the dot, an extension of up to three characters must be formatted and padded.

```
// 6. if there is an extension
if ( c != '\0')
{
    for( i=8; i<11; i++)
    {
        // read a char and convert to upper case
        c = toupper( *filename++);
        if ( c == '.')
            c = toupper( *filename++);
        if ( c == '\0')          // short extension
            break;
    }
}
```

```

        else
            fp->name[i] = c;
    } // for
    // if short fill the rest up to 3 with spaces
    while ( i<11) fp->name[i++] = ' ';
} // if

```

While most C libraries provide extensive support for multiple *modes* of access to files, like distinguishing between text and binary files and offering an *append* option, we will accept (at least initially) a subset consisting of two basic options only: *r* or *w*.

```

// 7. copy the file mode character (r, w)
if ((*mode == 'r') || (*mode == 'w'))
    fp->mode = *mode;
else
{
    FError = FE_INVALID_MODE;
    goto ExitOpen;
}

```

With the file name properly formatted, we can now start searching the root directory of the storage device for an entry of the same name.

```

// 8. Search for the file in current directory
if ( ( r = FindDIR( fp) ) == FAIL)
{
    FError = FE_FIND_ERROR;
    goto ExitOpen;
}

```

Let's leave the details of the search out for now and let's trust the *FindDIR()* function to return to us one of three possible values: *FAIL*, *NOT_FOUND* and eventually *FOUND*. A possible failure must always be taken into account. After all, before we consider the possibility of major fatal failures of the storage device, there is always the possibility that the user simply removed the card from its slot without our knowledge. If that is the case, as in all prior error cases, we have no business continuing in the process. We had better immediately release the memory allocated thus far and return with a *NULL* pointer after leaving an error code in the dedicated *mailbox* (*FError*), just as we did during the mount process.

But if the search for the file is completed without errors, whether it was found or not, we can continue initializing the *MFILE* structure.

```

// 9. init all counters to the beginning of the file
fp->seek = 0;                // first byte in file
fp->sec = 0;                 // first sector in the cluster
fp->pos = 0;                 // first byte in sector/cluster

```

The counter *seek* will be used to keep track of our position inside the file as we will sequentially access its contents. Its value will be a long integer (*unsigned long*) between

zero and the size of the entire file expressed in bytes. The *sec* field will keep track of which sector (inside the current cluster) we are currently operating on. Its value will be an integer between 0 and *sxc-1*, the number of sectors composing each data cluster. Finally, *pos* will keep track of which byte (inside the current buffer) we are going to access next. Its value will be an integer between 0 and 511.

```
// 10. depending on the mode (read or write)
if ( fp->mode == 'r')
{
```

At this point, different things need to be done depending whether an existing file needs to be opened for reading or a new file needs to be created for writing. Initially, we will complete all the necessary steps for the *fopenM()* function when invoked in the read (*r*) mode, in which case the file had better be found.

```
// 10.1 'r' open for reading
if ( r == NOT_FOUND)
{
    FError = FE_FILE_NOT_FOUND;
    goto ExitOpen;
}
```

If it was indeed found, the *FindDIR()* function will have filled for us a couple more fields of the *MFILE* structure, including:

- *entry*: indicating the position in the root directory where the file was found
- *cluster*: indicating the number of the first data cluster used to store the file data as retrieved from the directory entry
- *size*: indicating the number of bytes composing the entire file
- *time* and *date* of creation.

File Attributes

The first cluster number will become our current cluster: *ccls*.

```
else
{ // found

// 10.2 set the current cluster pointer on the first file cluster
fp->ccls = fp->cluster;
```

We have now all the information required to identify the first sector of data in the buffer. The function *ReadDATA()* (which we will describe in detail shortly) will perform the simple calculation required to convert the *ccls* and *sec* values into an absolute sector number inside the data area and will use the low level *ReadSECTOR()* function to retrieve the data from the storage device.

```
// 10.3 read a sector of data from the file
    if ( !ReadDATA( fp))
    {
        goto ExitOpen;
    }
```

Notice that the file length is not constrained to be a multiple of a sector size. So it is perfectly possible that only a part of the data retrieved in the buffer belongs to the actual file. The *MFILE* structure field *top* will help us keep track of where the actual file data ends and possibly padding begins.

```
// 10.4 determine how much data is really inside the buffer
    if ( fp->size-fp->seek < 512)
        fp->top = fp->size - fp->seek;
    else
        fp->top = 512;
} // found
} // 'r'
```

As this is all we really need to complete the *fopenM()* function (when opening a file for reading) we could return now with the precious pointer to the *MFILE* structure.

In case any of the previous steps failed, we will exit the function, returning a *NULL* pointer after having released both the memory allocated for the sector buffer and the *MFILE* structure.

```
// 13. Exit with error
ExitOpen:
    free( fp->buffer);
    free( fp);
    return NULL;
} // fopenM
```

Note

Shortly we will be inserting some more code before this point, so don't worry for now if it seems I skipped a few steps in the numbering sequence.

In a top-down fashion, we can now complete the two accessory functions used during the development of *fopenM()*, starting with *ReadDATA()*:

```
unsigned ReadDATA( MFILE *fp)
{
    LBA l;

    // calculate lba of cluster/sector
```

```
    l = fp->mda->data + (LBA)(fp->ccls-2) * fp->mda->sec + fp->sec;
    return( ReadSECTOR( l, fp->buffer))

} // ReadDATA
```

Notice how we need *data* and *sec* from the *MEDIA* structure to compute the correct sector number. Very simple!

Similarly, we can create a function to read from the root directory a block of data containing a given entry.

```
unsigned ReadDIR( MFILE *fp, unsigned e)
// loads current entry sector in file buffer
// returns      FAIL/TRUE
{
    LBA l;

    // load the root sector containing the DIR entry "e"
    l = fp->mda->root + (e >> 4);

    return ( ReadSECTOR( l, fp->buffer));
} // ReadDIR
```

We know that each directory entry is 32 bytes wide; therefore, each sector will contain 16 entries.

The *FindDIR()* function can now be quickly coded as a short sequence of steps enclosed in a search loop through all the available entries in the root directory.

```
unsigned FindDIR( MFILE *fp)
// fp      file structure
// return   found/not_found/fail
{
    unsigned eCount;           // current entry counter
    unsigned e;                // current entry offset in buffer
    int i, a, c, d;
    MEDIA *mda = fp->mda;

    // 1. start from the first entry
    eCount = 0;
    // load the first sector of root
    if ( !ReadDIR( fp, eCount))
        return FAIL;

```

We start loading the first root sector, containing the first 16 entries, in the buffer. For each entry we compute its offset inside the buffer.

```
    // 2. loop until you reach the end or find the file
    while ( 1)
    {
        // 2.0 determine the offset in current buffer
        e = (eCount & 0xf) * DIR_ESIZE;

```

And we inspect the first character of the entry file name.

```
// 2.1 read the first char of the file name
a = fp->buffer[ e + DIR_NAME];
```

If its value is zero, indicating an empty entry and the end of the list, we can immediately exit, reporting the file name was not found.

```
// 2.2 terminate if it is empty (end of the list)
if ( a == DIR_EMPTY)
{
    return NOT_FOUND;
} // empty entry
```

The other possibility is that the entry was marked as deleted, in which case we will skip it.

```
// 2.3 skip erased entries if looking for a match
if ( a != DIR_DEL)
{
```

Otherwise, it's a valid healthy entry, and we should check the attributes to determine whether it corresponds to a proper file or any other type of object. The possibilities include subdirectories, volume labels and long file names. None of them is of our concern, as we will choose to keep things simple and we will steer clear of the most advanced and sometimes patented features of the more recent versions of the FAT file system standard.

```
// 2.3.1 if not VOLUME or DIR compare the names
a = fp->buffer[ e + DIR_ATTRIB];

if ( !(a & (ATT_DIR | ATT_HIDE)) )
{
```

We will then compare the file names character by character, looking for a complete match.

```
// compare file name and extension
for (i=DIR_NAME; i<DIR_ATTRIB; i++)
{
    if ( ( fp->buffer[ e + i]) != ( fp->name[i]))
        break; // difference found
}
```

Only if every character matches will we extract the essential pieces of information from the entry and we will copy them into the *MFILE* structure, returning a *FOUND* code.

```
if ( i == DIR_ATTRIB)
{
    // entry found, fill the mfile structure
    fp->entry = eCount;           // store entry index
    fp->time = ReadW( fp->buffer, e + DIR_TIME);
    fp->date = ReadW( fp->buffer, e + DIR_DATE);
    fp->size = ReadL( fp->buffer, e + DIR_SIZE);
    fp->cluster = ReadL( fp->buffer, e + DIR_CLST);
```

```
        return FOUND;
    }
    } // not a dir nor a vol
} // not deleted
```

Should the file name and extension differ we will simply continue our search with the next entry, remembering to load the next sector from the root directory after each group of 16 entries.

```
// 2.4 get the next entry
eCount++;
if ( eCount & 0xf == 0)
{
    // load a new sector from the Dir
    if ( !ReadDIR( fp, eCount))
        return FAIL;
}
```

We know the maximum number of entries in the root directory (*maxroot*) and we need to terminate our search if we reach the end of the directory without a match indicating *NOT_FOUND*.

```
// 2.5. exit the loop if reached the end or error
if ( eCount >= mda->maxroot)
    return NOT_FOUND;           // last entry reached
} // while
} // FindDIR
```

Reading Data from a File

Finally, this is the moment we have waited so long for. The file system is mounted, a file is found and opened for reading; it is time to develop the *freadM()* function to freely read blocks of data from it.

```
unsigned freadM( void * dest, unsigned size, MFILE *fp)
// fp      pointer to MFILE structure
// dest    pointer to destination buffer
// count   number of bytes to transfer
// returns  number of bytes actually transferred
{
    MEDIA * mda = fp->mda;
    unsigned count=size;      // counts bytes to be transferred
    unsigned len;
```

The name, number and sequence of parameters passed to this function are again supposed to mimic closely that of a similarly named function available in the standard C libraries.

A destination buffer is supplied where the data read from the file will be copied, and a number of bytes are requested while passing the usual pointer to an open *MFILE* structure.

The *freadM()* function will do its best to read as many of the bytes requested as possible from the file, and will return an unsigned integer value to report how many it effectively

managed to get. In our simple implementation, if the number returned is not identical to what was requested by the calling application, we will have to assume that something major has happened. The end of file has been reached most probably, but we will not make a distinction if, instead, another type of failure has occurred – for example, the card has been removed during the process.

As usual, we will not trust the pointer passed in the argument and we will check instead whether it is pointing to a valid *MFILE* structure by recalculating and comparing the simple checksum performed by the open function when successfully opening a file.

```
// 1. check if fp points to a valid open file structure
if (( fp->mode != 'r'))
{
    // invalid file or not open in read mode
    FError = FE_INVALID_FILE;
    return 0;
}
```

Only then will we enter a loop to start transferring the data from the sector data buffer.

```
// 2. loop to transfer the data
while ( count>0)
{
```

Inside the loop, the first condition to check will be our current position, with regard to the total file size.

```
// 2.1 check if EOF reached
if ( fp->seek >= fp->size)
{
    FError = FE_EOF; // reached the end
    break;
}
```

Notice that this error will be generated only if the application calling the *freadM()* function ignores the previous symptom, the last *freadM()* call returned with a number of data bytes inferior to what was requested, or if the calling application has requested the exact number of bytes available in the file with the previous calls.

Otherwise, we will verify whether the current buffer of data has already been used up completely.

```
// 2.2 load a new sector if necessary
if (fp->pos == fp->top)
{
```

If necessary, we will reset our buffer pointers and attempt to load the next sector from the file.

```
fp->pos = 0;
fp->sec++;
```


If we had already used up all the sectors in the current cluster, this might force us to step into the next cluster by peeking inside the FAT and following the chain of clusters.

```
// 2.2.1 get a new cluster if necessary
if ( fp->sec == mda->sxc)
{
    fp->sec = 0;
    if ( !NextFAT( fp, 1))
        break;
}
```

In either case, we load the new sector of data in the buffer, paying attention to verify the possibility that it might be the last one of the file and it might be only partially filled.

```
// 2.2.2 load a sector of data
if ( !ReadDATA( fp))
    break;

// 2.2.3 determine how much data is inside the buffer
if ( fp->size-fp->seek < 512)
    fp->top = fp->size - fp->seek;
else
    fp->top = 512;
} // load new sector
```

Now that we know we have data in the buffer, ready to be transferred, we can determine how much of it we can transfer in a single chunk.

```
// 2.3 copy as many bytes as possible in a single chunk
// take as much as fits in the current sector
if ( fp->pos+count < fp->top)
    len = count;                // fits all in current sector
else
    len = fp->top - fp->pos;      // take a first chunk, there is more

memcpy( dest, fp->buffer + fp->pos, len);
```

Using the *memcpy()* function from the standard C library *string.h* to move a block of data from the file buffer to the destination buffer, we get the best performance as these routines are optimized for speed of execution. The pointers and counters can be updated and the loop can be repeated until all the data requested has been transferred.

```
// 2.4 update all counters and pointers
count-= len;                    // compute what is left
dest += len;                    // advance destination pointer
fp->pos += len;                  // advance the pointer in sector
fp->seek += len;                 // advance the seek pointer

} // while count
```

Finally, we can exit the function and return the number of actual bytes transferred in the loop.

```
// 3. return number of bytes actually transferred
return size-count;

} // freadM
```

There is one last piece of the puzzle that we need to flesh out. It is the function *NextFAT()* that we used to follow the chain of clusters in the FAT table.

```

unsigned NextFAT( MFILE * fp, unsigned n)
// fp   file structure
// n    number of links in FAT cluster chain to jump through
//      n==1, next cluster in the chain
{
    unsigned    c;
    MEDIA * mda = fp->mda;

    // loop n times
    do {
        // get the next cluster link from FAT
        c = ReadFAT( fp, fp->ccls);

        // compare against max value of a cluster in FATxx
        // return if eof
        if ( c >= FAT_MCLST)    // check against eof
        {
            FError = FE_FAT_EOF;
            return FAIL;    // seeking beyond EOF
        }

        // check if cluster value is valid
        if ( c >= mda->maxcls)
        {
            FError = FE_INVALID_CLUSTER;
            return FAIL;
        }

    } while (--n > 0); // loop end

    // update the MFILE structure
    fp->ccls = c;

    return TRUE;
} // NextFAT

```

Closing a File

Since we can only open a file for reading with the *fopenM()* function as defined so far, there is not much work to perform upon closing the file. We must remember to free all the memory allocated for the *MFILE* structure and its sector buffer.

```

unsigned fcloseM( MFILE *fp)
{
    // free up the buffer and the MFILE struct
    free( fp->buffer);
    free( fp);
} // fcloseM

```

Creating the Fileio Module

We can create a small library module by saving all the functions written so far in a file called *fileio.c*. We will need to add the usual header with a few include files:

```
/*
** fileio.c
**
** FAT16 FILE I/O support
*/
#include <stdlib.h>      // malloc...
#include <ctype.h>       // toupper...
#include <string.h>      // memcpy...
#include <fileio.h>      // file I/O routines
```

And of course, we will need to create the *fileio.h* file too with all the definitions and prototypes that we wish to publish for the future applications to use.

```
/*
** fileio.h
**
** FAT16 file I/O support
*/
#include <sdkmmc.h>

extern char FError;          // mailbox for error reporting

// FILEIO ERROR CODES
#define FE_IDE_ERROR        1    // IDE command execution error
#define FE_NOT_PRESENT      2    // CARD not present
#define FE_PARTITION_TYPE   3    // WRONG partition type
#define FE_INVALID_MBR      4    // MBR sector invalid signtr
#define FE_INVALID_BR       5    // Boot Record invalid signtr
#define FE_MEDIA_NOT_MNTD   6    // Media not mounted
#define FE_FILE_NOT_FOUND   7    // File not found, open for read
#define FE_INVALID_FILE     8    // File not open
#define FE_FAT_EOF          9    // Attempt to read beyond EOF
#define FE_EOF              10   // Reached the end of file
#define FE_INVALID_CLUSTER  11   // Invalid cluster > maxcls
#define FE_DIR_FULL         12   // All root dir entries are taken
#define FE_MEDIA_FULL       13   // All clusters taken
#define FE_FILE_OVERWRITE   14   // A file with same name exists
#define FE_CANNOT_INIT      15   // Cannot init the CARD
#define FE_CANNOT_READ_MBR  16   // Cannot read the MBR
#define FE_MALLOC_FAILED    17   // Could not allocate memory
#define FE_INVALID_MODE     18   // Mode was not r.w.
#define FE_FIND_ERROR       19   // Failure during FILE search

typedef struct {
    LBA    fat;           // lba of FAT
    LBA    root;          // lba of root directory
    LBA    data;          // lba of the data area
```

```

    unsigned short maxroot;      // max entries in root dir
    unsigned short maxcls;       // max clusters in partition
    unsigned short fatsize;      // number of sectors
    unsigned char fatcopy;       // number of copies
    unsigned char sxc;           // number sectors per cluster
} MEDIA;

typedef struct {
    MEDIA * mda;                 // media structure pointer
    unsigned char * buffer;      // sector buffer
    unsigned short cluster;      // first cluster
    unsigned short ccls;        // current cluster in file
    unsigned short sec;         // sector in current cluster
    unsigned short pos;         // position in current sector
    unsigned short top;         // bytes in the buffer
    long seek;                  // position in the file
    long size;                  // file size
    unsigned short time;        // last update time
    unsigned short date;        // last update date
    char name[11];              // file name
    char mode;                   // mode 'r', 'w'
    unsigned short fpage;       // FAT page currently loaded
    unsigned short entry;       // entry position in cur dir
} MFILE;

// file attributes
#define ATT_RO      1           // attribute read only
#define ATT_HIDE    2           // attribute hidden
#define ATT_SYS     4           // "      system file
#define ATT_VOL     8           // "      volume label
#define ATT_DIR     0x10        // "      sub-directory
#define ATT_ARC     0x20        // "      (to) archive
#define ATT_LFN     0x0f        // mask for Long File Name

#define FOUND       2           // directory entry match
#define NOT_FOUND   1           // directory entry not found

// macros to extract words and longs from a byte array
// watch out, a processor trap will be generated if the address
// is not word aligned
#define ReadW( a, f) *(unsigned short*)(a+f)
#define ReadL( a, f) *(unsigned long *)(a+f)

// this is a "safe" version of ReadW
// to be used on odd address fields
#define ReadOddW( a, f) (*(a+f) + (*(a+f+1) << 8))

// prototypes
unsigned NextFAT( MFILE * fp, unsigned n);
unsigned NewFAT ( MFILE * fp);

unsigned ReadDIR( MFILE *fp, unsigned entry);
unsigned FindDIR( MFILE *fp);
unsigned NewDIR ( MFILE *fp);

```

```
MEDIA * mount( void);
void      umount( void);

MFILE *  fopenM ( const char *name, const char *mode);
unsigned freadM ( void * dest, unsigned count, MFILE *);
unsigned fwriteM ( void * src, unsigned count, MFILE *);
unsigned fcloseM ( MFILE *fp);
```

Don't worry now if we have not fleshed out all the functions yet, we will continue working on them as we proceed through the rest of the lesson.

Testing `fopenM()` and `freadM()`

It might seem like a long time since we built a project for the last time. To verify the code that we have developed so far we had to reach a critical mass, a minimal core of routines without which no application could have worked. Now that we have this core functionality, we can develop for the first time a small test program to read a file from an SD/MMC card that has been formatted with the FAT16 file system.

The idea is to copy a text file (any text file would work) onto the SD/MMC card from your PC, and then have the PIC24, with the new *fileio.c* module, read the file and send its content to the serial port and back to the PC (running HyperTerminal or any other terminal or printer available with an RS232 serial port).

This is the main module that we will call ***ReadTest.c***

```
/*
**  ReadTest.c
**
*/
#include <config.h>
#include <EX16.h>
#include <SDMMC.h>
#include <fileio.h>
#include <CONU2.h>

#define B_SIZE 10
char data[ B_SIZE];

main( void)
{
    MFILE *fs;
    unsigned i, r;

    //initializations
    InitU2();                // 115,200 baud 8,n,1

    putsU2( "init");

    while( !DetectSD());    // assumes SDCCD pin is by default an input
    Delaysms( 100);         // wait for card to power up
```

```

putsU2("media detected");

if ( mount())
{
    putsU2( "mount");
    if ( fs = fopenM( "name.txt", "r"))
    {
        putsU2("file opened");
        do{
            r = freadM( data, B_SIZE, fs);
            for( i=0; i<r; i++)
                putU2( data[i]);
        } while( r==B_SIZE);
        fcloseM( fs);
        putsU2("file closed");
    }
    else
        putsU2("could not open file");

    umount();
    putsU2("media unmounted");
}

// main loop
while( 1);
} // main

```

We will use the serial communication module *CONU2.c* developed in one of the early lessons and the *Delaysms()* function defined in *EX16.h*. The sequence of operation is similar to the test performed in the previous lesson but this time, instead of calling the *InitMedia()* function and then starting reading and writing directly to sectors of the SD/MMC card, we will call the *mount()* function to access the FAT16 file system on the card. We will open the data file using its *proper* name, and we will read data from it in blocks of arbitrary length (*B_SIZE*), sending its contents to the serial port of the Explorer16 board.

Once we have exhausted the contents of the entire file we will close the file, de-allocating all the memory used.

After creating a new project we will need to add all the necessary modules to the project window, including *sdmcc.c*, *fileio.c*, *CONU2.c*, *EX16.c* *readtest.c* and all the corresponding include files (*.h*).

This time, before launching a project build, we will need to remember to reserve some space for the Heap so that we will be able to allocate memory dynamically (using *malloc()* and *free()*) for the file system structures and buffers. This can be done by selecting the **Set Project Configuration>Customize** command from the **Run** menu, then opening the **pic30-ld** pane where the linker settings are listed (Figure 14.8), and entering a sufficiently large value for

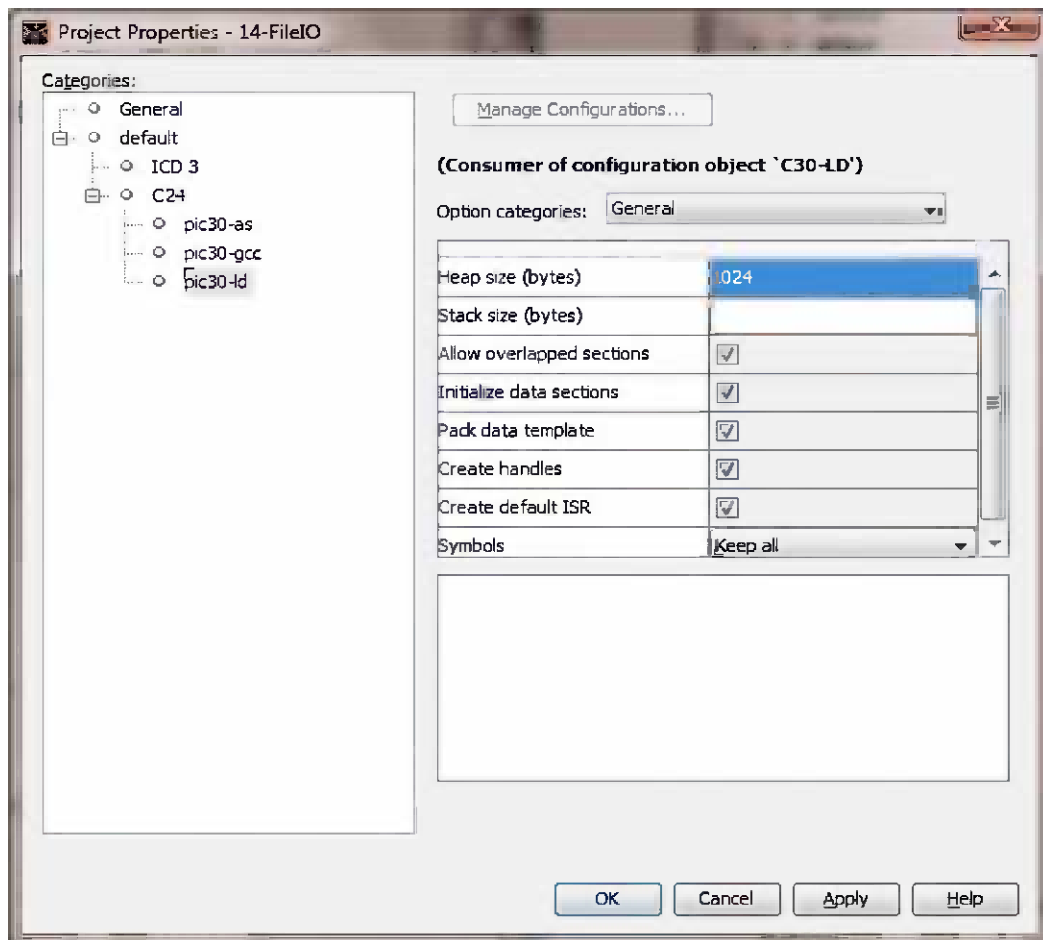


Figure 14.8: Reserving memory space for the Heap

the Heap size field. Even if approx. 580 bytes should suffice, give the Heap more room to maneuver; I recommend we *splurge* with at least 1024 bytes.

Now launch the project using the **Run>Run Project** command to perform a quick build, program the executable on the Explorer16 board and start executing the code.

If all goes well, you will be able to see the contents of the text file scrolling on the screen of your terminal of choice, probably too fast for you to read any of it but the last few lines.

Notice that you can rebuild the project and run the test with different sizes for the data buffer, from 1 byte to as large as the memory of the PIC24 will allow. The *freadM()* function will take care of reading as many sectors of data required to fulfill your request as long as there is data in the file.

Writing Data to a File

We are far from finished though, the *fileio.c* module is not complete until we include the ability to create new files. This requires us to create an *fwriteM()* function but also to complete a piece of the *fopenM()* function. So far, in fact, we had *fopenM()* return with an error code when a file could not be found in the root directory or the mode was not *r*. But this is exactly what we want when we open a new file for writing. When we check for the mode parameter value, we need now to add a new option. This time, it is when the file is *NOT_FOUND* during the first scan of the directory that we want to proceed.

```
else // 11. open for 'write'
{
    if ( r == NOT_FOUND)
    {
```

A new file needs a new cluster to be allocated to contain its data. The function *NewFAT()* will be used to search in the FAT table for an available spot, a cluster that is still marked (with 0x0000) as available. This search could fail and the function could return an error that among other things could indicate that the storage device is full and all data clusters are taken. Should the search be successful, though, we will take note of the new cluster position and update the *MFILE* structure, making it the first cluster of our new file.

```
// 11.1 allocate a first cluster to it
fp->ccls = 0; // indicate brand new file
if ( NewFAT( fp) == FAIL)
{ // must be media full
    FError = FE_MEDIA_FULL;
    goto ExitOpen;
}
fp->cluster = fp->ccls;
```

Next, we need to find an available entry space in the directory for the new file. This will require a second pass through the root directory, this time looking for the first entry that is either marked as deleted (code 0xE5) or for the end of the list where an empty entry is found (marked with the code 0x00).

```
// 11.2 create a new entry
// search again, for an empty entry this time
if ( (r = NewDIR( fp)) == FAIL) // report any error
{
    FError = FE_IDE_ERROR;
    goto ExitOpen;
}
```

The function *NewDIR()* will take care of finding an available entry and, similarly to the *FindDIR()* function used before, will return one of three possible codes:

- *FAIL*, indicating a major problem occurred (or the card was removed)
- *NOT_FOUND*, the root directory must be full
- *FOUND*, an available entry has been identified.


```
// 11.3 new entry not found
if ( r == NOT_FOUND)
{
    FError = FE_DIR_FULL;
    goto ExitOpen;
}
```

In both the first two cases we have to report an error and we cannot continue. But if an entry is found we have plenty of work to do to initialize it.

After calculating the offset of the entry in the current buffer, we will start filling some of its fields with data from the MFILE structure. The file size will be first.

```
else // 11.4 new entry identified fp->entry filled
{
    // 11.4.1 init file size
    fp->size = 0;

    // 11.4.2 determine offset in DIR sector
    e = (fp->entry & 0xf) * DIR_ESIZE;

    // 11.4.3 init all fields to 0
    for (i=0; i<32; i++)
        fp->buffer[ e + i ] = 0;
```

The time and date fields could be derived from the RTCC module registers or any other timekeeping mechanism available to the application, but a default value will be supplied here only for demonstration purposes.

```
// 11.4.4 set date and time
fp->date = 0x378A; // Dec 10th, 2007
fp->buffer[ e + DIR_CDATE ] = fp->date;
fp->buffer[ e + DIR_CDATE+1]= fp->date>>8;
fp->buffer[ e + DIR_DATE ] = fp->date;
fp->buffer[ e + DIR_DATE+1]= fp->date>>8;

fp->time = 0x6000; // 12:00:00 PM
fp->buffer[ e + DIR_CTIME ] = fp->time;
fp->buffer[ e + DIR_CTIME+1]= fp->time>>8;
fp->buffer[ e + DIR_TIME ] = fp->time+1;
fp->buffer[ e + DIR_TIME+1]= fp->time>>8;
```

The file first cluster number, the file name and the attributes (defaults) will complete the directory entry.

```
// 11.4.5 set first cluster
fp->buffer[ e + DIR_CLST ] = fp->cluster;
fp->buffer[ e + DIR_CLST+1]= (fp->cluster>>8);

// 11.4.6 set name
for ( i = 0; i<DIR_ATTRIB; i++)
    fp->buffer[ e + i ] = fp->name[i];
```

```

        // 11.4.7 set attrib
        fp->buffer[ e + DIR_ATTRIB ] = ATT_ARC;

        // 11.4.8 update the directory sector;
        if ( !WriteDIR( fp, fp->entry))
        {
            FError = FE_IDE_ERROR;
            goto ExitOpen;
        }
    } // new entry
} // not found

```

Back to the results of our first search through the root directory, in case a file with the same name was indeed found, we will need to report an error.

```

else // file exist already, report error
{
    FError = FE_FILE_OVERWRITE;
    goto ExitOpen
}

```

Alternatively, we would have had to delete the current entry first, release all the clusters used and then start from the beginning. After all, reporting the problem as an error is an easier way out for now.

So much for the changes required to the *fopenM()* function. We can now start writing the proper new *fwriteM()* function, once more modeled after a similarly named standard C library function.

```

unsigned fwriteM( void *src, unsigned count, MFILE * fp)
// src      points to source data (buffer)
// count    number of bytes to write
// returns  number of bytes actually written
{
    MEDIA *mda = fp->mda;
    unsigned len, size = count;

    // 1. check if file is open
    if ( fp->mode != 'w')
    {
        // file not valid or not open for writing
        FError = FE_INVALID_FILE;
        return FAIL;
    }
}

```

The parameters passed to the function are identical to those used in the *freadM()* function and the first test we will perform on the integrity of the MFILE structure, passed as a parameter, is the same as well. It will help us determine whether we can trust that the contents of the MFILE structure have been successfully prepared for us by a call to *fopenM()*.

The core of the function will be a loop as well:

```
// 2. loop writing count bytes
while ( count>0)
{
```

Our intention is that of transferring as many bytes of data as possible at a time, using the fast *memcpy()* function from the *string.h* libraries.

```
// 2.1 copy as many bytes at a time as possible
if ( fp->pos+count < 512)
    len = count;
else
    len = 512- fp->pos ;

memcpy( fp->buffer+ fp->pos, src, len);
```

There are a number of pointers and counters that we need to update to keep track of our position as we add data to the buffer and we increase the size of the file.

```
// 2.2 update all pointers and counters
fp->pos+=len;          // advance buffer position
fp->seek+=len;         // count the added bytes
count-=len;           // update the counter
src+=len;             // advance the source pointer

// 2.3 update the file size too
if (fp->seek > fp->size)
    fp->size = fp->seek;
```

Once the buffer is full, we need to transfer the data to the media in a sector of the currently allocated cluster.

```
// 2.4 if buffer full, write current buffer to current sector
if (fp->pos == 512)
{
    // 2.4.1 write buffer full of data
    if ( !WriteDATA( fp))
        return FAIL;
```

Notice that an error at this point would be rather fatal. We will return the code *FAIL* indicating that not a single byte has been transferred, in fact all the data written to the storage device this far is now lost.

If all proceeds correctly, though, we can now increment the sector pointers and if we have exhausted all the sectors in the current cluster we need to consider the need to allocate a new one, calling *NewFAT()* once more.

```
// 2.4.2 advance to next sector in cluster
fp->pos = 0;
fp->sec++;

// 2.4.3 get a new cluster if necessary
if ( fp->sec == mda->sxc)
```

```

        {
            fp->sec = 0;
            if ( NewFAT( fp) == FAIL)
                return FAIL;
        }
    } // store sector

} // while count

```

Shortly, when developing *newFAT()*, we will have to make sure that the function accurately maintains the chaining of the clusters in the FAT as they get added to a file.

```

    // 3. number of bytes actually written
    return size-count;

} // fwriteM

```

The function is now complete and we can report the number of bytes written upon exit from the loop.

Closing a File, Second Take

While closing a file opened for reading was a mere formality and a matter of releasing some memory from the heap, when we close a file that has been opened for writing there is a considerable amount of housekeeping work that needs to be performed.

A new and improved *fcloseM()* function is needed and it will start with a check of the *mode* field.

```

unsigned fcloseM( MFILE *fp)
{
    unsigned e, r;                // offset of directory entry in current buffer

    r = FAIL;

    // 1. check if it was open for write
    if ( fp->mode == 'w')
    {

```

In fact, when we close a file there might be still some data in the buffer that needs to be written to the storage device (*flushed*), although it does not fill an entire sector.

```

        // 1.1 if the current buffer contains data, flush it
        if ( fp->pos > 0)
        {
            if ( !WriteDATA( fp))
                goto ExitClose;
        }

```

Once more, any error at this point is a rather fatal event and will mean that all the file data is lost since the *fcloseM()* function will not properly complete.

The proper root directory sector must be retrieved and an offset for the directory entry must be calculated inside the buffer.

```

// 1.2      finally update the dir entry,
// 1.2.1    retrieve the dir sector
if ( !ReadDIR( fp, fp->entry))
    goto ExitClose;
// 1.2.2 determine position in DIR sector
e = (fp->entry & 0xf) * DIR_ESIZE;    // 16 entry per sector

```

Next, we need to update the file entry in the root directory with the actual file size (it had been initially set to zero).

```

// 1.2.3 update file size
fp->buffer[ e + DIR_SIZE] = fp->size;
fp->buffer[ e + DIR_SIZE+1] = fp->size>>8;
fp->buffer[ e + DIR_SIZE+2] = fp->size>>16;
fp->buffer[ e + DIR_SIZE+3] = fp->size>>24;

```

Finally, the entire root directory sector containing the entry is written back to the media.

```

// 1.2.4    update the directory sector;
if ( !WriteDIR( fp, fp->entry))
    goto ExitClose;
} // write

```

If all went well, we will complete the *fcloseM()* function invalidating the checksum field to prevent accidental re-uses of this *MFILE* structure and de-allocating the memory used by it and its buffer.

```

// 2. exit with success
r = TRUE;

ExitClose:

// 3. free up the buffer and the MFILE struct
free( fp->buffer);
free( fp);
return( r);

} // fcloseM

```

Accessory Functions

In completing *fopenM()*, *fcloseM()* and creating the new *fwriteM()* function we have used a number of lower-level functions to perform important repetitive tasks.

We will start with *NewDIR()*, used to find an available spot in the root directory, to create a new file. The similarity with *FindDIR()* is obvious, yet the task performed is very different.

```

unsigned NewDIR( MFILE *fp)
// fp      file structure
// return   found/fail, fp->entry filled
{
    unsigned eCount;           // current entry counter
    unsigned e;                // current entry offset in buffer

```

```

int a;
MEDIA *mda = fp->mda;

// 1. start from the first entry
eCount = 0;
// load the first sector of root
if ( !ReadDIR( fp, eCount))
    return FAIL;

// 2. loop until you reach the end or find the file
while ( 1)
{
    // 2.0 determine the offset in current buffer
    e = (eCount&0xf) * DIR_ESIZE;

    // 2.1 read the first char of the file name
    a = fp->buffer[ e + DIR_NAME];

    // 2.2 terminate if it is empty (end of the list) or deleted
    if (( a == DIR_EMPTY) ||( a == DIR_DEL))
    {
        fp->entry = eCount;
        return FOUND;
    } // empty or deleted entry found

    // 2.3 get the next entry
    eCount++;
    if ( (eCount & 0xf) == 0)
    { // load a new sector from the root
        if ( !ReadDIR( fp, eCount))
            return FAIL;
    }

    // 2.4 exit the loop if reached the end or error
    if ( eCount > mda->maxroot)
        return NOT_FOUND;           // last entry reached
} // while

return FAIL;
} // NewDIR

```

The function *NewFAT()* was used to find an available cluster to allocate for a new block of data/new file.

```

unsigned NewFAT( MFILE * fp)
// fp      file structure
// fp->ccls ==0 if first cluster to be allocated
//          !=0 if additional cluster
// return  TRUE/FAIL
// fp->ccls new cluster number
{

```

```

unsigned i, c = fp->ccls;

// sequentially scan through the FAT looking for an empty cluster
do {
    c++;    // check next cluster in FAT
    // check if reached last cluster in FAT,
    // re-start from top
    if ( c >= fp->mda->maxcls)
        c = 0;

    // check if full circle done, media full
    if ( c == fp->ccls)
    {
        FError = FE_MEDIA_FULL;
        return FAIL;
    }

    // look at its value
    i = ReadFAT( fp, c);

} while ( i!=0); // scanning for an empty cluster

// mark the cluster as taken, and last in chain
WriteFAT( fp, c, FAT_EOF);

// if not first cluster, link current cluster to the new one
if ( fp->ccls >0)
    WriteFAT( fp, fp->ccls, c);

// update the MFILE structure
fp->ccls = c;

return TRUE;
} // allocate new cluster

```

When allocating a new cluster beyond the first one, *NewFAT()* keeps linking the clusters in a chain and it marks every cluster as properly used. For its working, in its turn the function uses two accessory functions, *ReadFAT()* and *WriteFAT()*.

```

unsigned ReadFAT( MFILE *fp, unsigned ccls)
// fp      MFILE structure
// ccls    current cluster
// return  next cluster value,
//        0xffff if failed or last
{
    unsigned p, c;
    LBA l;

    // get address of current cluster in fat
    p = ccls >>8;    // 256 clusters per sector
    // cluster = 0xabcd
    // packed as:
    // word p      0  1 | 2  3 | 4  5 | 6  7 | ..
    //             cd ab| cd ab| cd ab| cd ab|

```

```

    // load the fat sector containing the cluster
    l = fp->mda->fat + p;
    if ( !ReadSECTOR( l, fp->buffer))
        return FAT_EOF;      // failed

    // get the next cluster value
    c = ReadOddW( fp->buffer, ((ccls & 0xFF)<<1));

    return c;
} // ReadFAT

```

The *WriteFAT()* function updates the contents of the FAT table and keeps all its copies current.

```

unsigned WriteFAT( MFILE *fp, unsigned cls, unsigned v)
// fp      MFILE structure
// cls     current cluster
// v       next value
// return  TRUE if successful, or FAIL
{
    int i;
    unsigned p;
    LBA l;

    // load the FAT page containing the requested cluster
    ReadFAT( fp, cls);

    // cluster = 0xabcd
    // packed as:
    // word p      0  1 | 2  3 | 4  5 | 6  7 | ..
    //             cd ab| cd ab| cd ab| cd ab|

    // locate the cluster within the FAT page
    p = (cls & 0xff)*2;    // 2 bytes per cluster

    // get the next cluster value
    fp->buffer[ p] = v;      // lsb
    fp->buffer[ p+1] = (v>>8); // msb

    // update all FAT copies
    l = fp->mda->fat + (cls>>8);
    for ( i=0; i<fp->mda->fatcopy; i++, l += fp->mda->fatsize)
        if ( !WriteSECTOR( l, fp->buffer))
            return FAIL;

    return TRUE;
} // WriteFAT

```

Finally, *WriteDATA()* was used both by *fwriteM()* and *fcloseM()* to write actual sectors of data to the storage device, computing the sector address based on the current cluster number.

```

unsigned WriteDATA( MFILE *fp)
{
    LBA l;

```



```
// calculate lba of cluster/sector
l = fp->mda->data + (LBA)(fp->ccls-2) * fp->mda->sxc + fp->sec;

return ( WriteSECTOR( l, fp->buffer));

} // WriteDATA
```

Testing the Complete Fileio Module

It is time to test the functionality of the entire module we have just completed. As in the previous test, we will use the serial communication module *CONU2.c* developed in one of the early lessons to provide a serial output to a terminal. This time, after *mounting* the SD/MMC card, we will open a *source* file (any text file would do), and we will copy its contents into a new *destination* file that we will create on the spot. Here is the code we will use for the *WriteTest.c* main file.

```
/*
** WriteTest.c
**
*/
#include <config.h>
#include <EX16.h>
#include <conu2.h>
#include <SDMMC.h>
#include <fileio.h>

#define B_SIZE 1024
char data[B_SIZE];

int main( void)
{
    MFILE *fs, *fd;
    unsigned r;

    //initializations
    InitU2();           // 115,200 baud 8,n,1
    putsU2( "init");
    while( !DetectSD()); // assumes SDCD pin is by default an input
    Delaysm( 100);      // wait for card to power up

    if ( mount())
    {
        putsU2("mount");
        if ( (fs = fopenM( "source.txt", "r")))
        {
            putsU2("source file opened for reading");
            if ( (fd = fopenM( "dest.txt", "w")))
            {
                putsU2("destination file opened for writing");
                do{
                    r = freadM( data, B_SIZE, fs);
                    r = fwriteM( data, r, fd);
                } while ( r > 0);
            }
        }
    }
}
```

```

        putU2('.');
    } while( r==B_SIZE);

    fcloseM( fd);
    putsU2("destination file closed");
}
else
    putsU2("could not open the destination file");

    fcloseM( fs);
    putsU2("source file closed");
}
else
    putsU2("could not open the source file");

    umount();
    putsU2("umount");

}
else
    putsU2("mount failed");

// main loop
while( 1);
} // main

```

Make sure to replace the source file name with the actual name of the file you copied on the card for this experiment.

After replacing in the project **14-FileIO WriteTest.c** in place of the previous main source file (*ReadTest.c*) – we will be ready to launch it for our test.

Note

Once enough space is left for the global variables and the stack, there is no reason to withhold any memory from the heap. Allocate as large a heap as possible to allow *malloc()* and *free()* to make optimal use of all the memory available.

After building the project and programming the executable on the Explorer16 board we are ready to run the test. If all goes well, after a fraction of a second (the actual time will depend on the size of the source file chosen) you will be able to see on the screen of your terminal the messages shown in [Figure 14.9](#).

The number of dots will be proportional to the size of the file, and since we chose the buffer size to be 1024 for this demo, each dot will correspond exactly to one kilobyte of data transferred. If you transfer the SD/MMC card back to your PC, you should be able to verify that a new file has been created ([Figure 14.10](#)).

```

init
mount
source file opened for reading
destination file opened for writing
.....
destination file closed
source file closed
umount

```

Figure 14.9: WriteTest output log

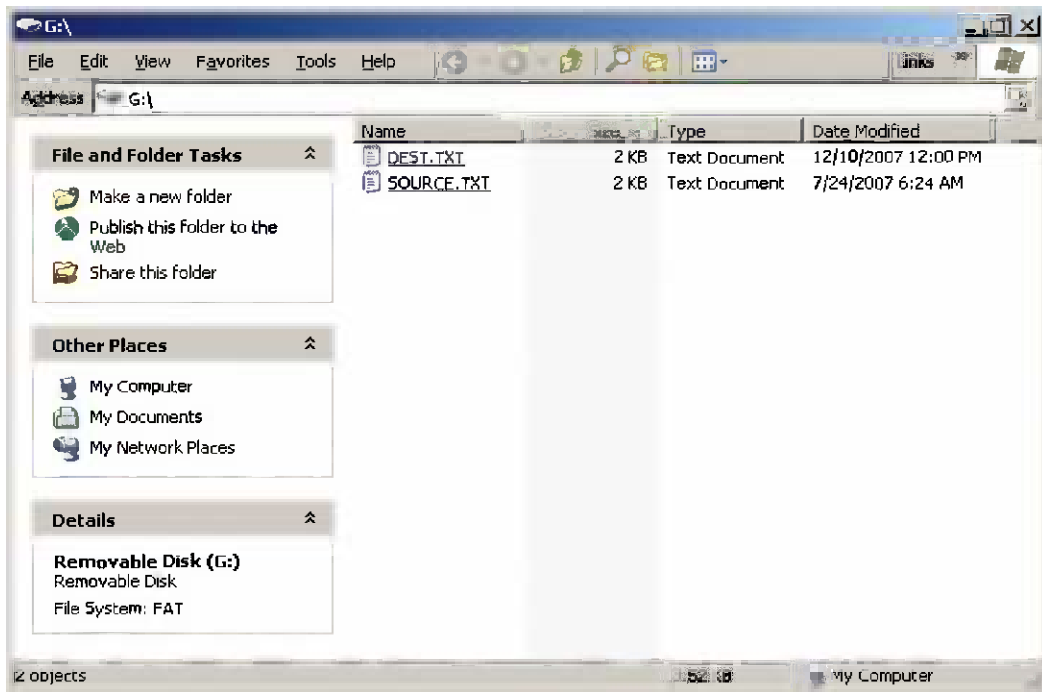


Figure 14.10: Windows Explorer screen capture

Its size and content are identical to the source file, while the date and time reflect the values we set in the *fcloseM()* function.

Notice that if you try to run the test program a second time it is bound to fail now.

```

init
mount
source file opened for reading
could not open the destination file
source file closed
umount

```

This is because, as discussed during the development of the *fopenM()* function, we chose to report an error when trying to open a file for writing and on the storage device there is already a file (*DEST.TXT*) with the same name!

Notice that you can recompile the project and run the test with different sizes for the data buffer from 1 byte to as large as the memory of the PIC24 will allow. Both the *freadM()* and *fwriteM()* functions will take care of reading and writing as many sectors of data required to fulfill your request. The time required to complete the operation will change though.

Code Size

The size of the code produced by the project (*I4-FileIO*) is considerably larger than the simple *sdmnc.c* module we tested in the previous lesson.

```
Total program memory used (bytes):          0x161a (5658) 4%
```

Still, selecting optimization level 1 (available on all free versions of the MPLAB® C compiler), the code will add up to just 5,658 bytes. This represents only 4% of the total program memory space available on the PIC24FJ128GA010. I consider this a very small price to pay for a lot of functionality!

Post-Flight Briefing

In this lesson we have learned the basics of the FAT16 file system and we have developed a small interface module that allows a PIC24 to read and write data files to and from a generic mass storage device. By using the *sdmnc.c* module, developed in the previous lesson for the low-level interface, we have created a basic file I/O interface for SD/MMC memory cards.

Now you can share data between the PIC24 and almost any other computer system that is capable of accessing SD/MMC cards, from PDAs to laptops and desktop PCs, from DOS, Windows and Linux machines to Apple computers running OS-X.

Notes for the PIC® Microcontroller Experts

Microchip is now offering a complete File System library (the MDD File System) that supports FAT16 and FAT32 formatted devices and was largely based on the code presented in this book. This library is very modular and it can be configured to support a number of different media/devices, including SD/MMC cards, CompactFlash cards, and even USB mass storage devices (memory sticks) if you use a PIC24F model from the GB1 family.

Watch out though! If USB memory sticks have all the appeal and the appearance of simplicity from the user perspective, the software required to access them is substantially larger, more complex and expensive (in terms of resources: RAM, Flash, CPU overhead...). This additional cost and complexity can be estimated to be several orders of magnitude larger than

the basic SD/MMC card solution we have examined here because you will need to run the MDD File System on top of a full USB Host stack.

Tips & Tricks

In recent years the SD/MMC memory cards have started to shrink. The new miniSD Card™ standard gained popularity for a short while only to be surpassed by the newer and even smaller microSD Card™ standard. You will be pleased to learn that both formats are 100% compatible with the larger ancestor and with the code presented in this book. The WP and CD lines have been dropped in the smaller form factors but, as you will have noticed, they were not really required for our applications and won't be missed.

The case of the High Capacity (SDHC™) cards is different. This new specification was introduced by the SD Association to remove the previous limitation on the maximum card size (2 Gbytes), increasing it to 32 GB. The code presented in this book will not operate correctly on those cards but, given their size, a FAT16 formatting would have been a very poor choice to begin with.

Exercises

- Consider adding the following functionality:
 - Subdirectory management
 - Erasing files
 - Long file-names support
 - Use the RTCC to provide the current time and date information.
- Consider caching (and/or using a separate buffer) for the current FAT record content to improve read/write performance.
- Consider the modifications required to perform buffering of larger blocks and/or entire clusters and performing multi-block read/write operations to optimize the SD card low-level performance. Consider pros and cons.

Books

- Buck, B., 2002. *North Star Over My Shoulder*, Simon & Shuster, New York, NY.
The story of aviation through the experiences of a lifetime as a pilot.
- Pate, S.D., 2003. *UNIX Filesystems: Evolution, Design, and Implementation*, Wiley Indianapolis, IN.

While Windows is our primary concern when we think of sharing files with a personal computer, when it comes to file systems you have to look at Unix (and Linux) to find serious file systems for mission-critical data storage.

Links

- <http://www.microchip.com/MDD>
This is the direct link to Microchip MDD File System Library design center (also part of the Microchip Application Library).
- <http://www.sdcard.org/developers/tech/sdcard#microsd>
This is a link to the SD Association page devoted to the introduction of the miniSD and microSD standards.
- <http://www.tldp.org/LDP/tlk/tlk-title.html>
“The Linux Kernel” by David A Rusling, an online book that describes the inner workings of Linux and its native file system (not a FAT file system).
- http://en.wikipedia.org/wiki/File_Allocation_Table
Once more, an excellent page of wikipedia that describes the history and many ramifications of the FAT technology.
- http://en.wikipedia.org/wiki/List_of_file_systems
An attempt to list and classify all major computer file systems in use.
- <http://en.wikipedia.org/wiki/ISO-9660>
Want to know how files are written on a CDROM? ISO-9660 is the answer...

Volare

The last flight, the check-ride with the FAA examiner, is a time of great tension and a little fear. It is a flight meant to summarize all the phases of flight, where you are asked to put all the knowledge you gained during the training into practice. Don't worry, it will be easy because you are at the peak of your preparation and it will be over so fast that you won't have time to realize it.

Just like in a final check-ride, this last lesson will use many of the building blocks developed in the previous lessons and will put them to practical use to develop a new and entertaining demo project: a media player.

Congratulations, you are a pilot now, it is time to celebrate!

Flight Plan

In this lesson we will explore the possibility of producing audio signals using, once more, the Output Compare modules of the PIC24. When in the pulse width modulation (PWM) mode and in combination with more or less sophisticated low-pass filters, the Output Compare modules can be used effectively as digital-to-analog converters to produce analog output signals. If we manage to modulate the analog signal with frequencies that fall into the range that is recognized by the human ear, approximately between 20 Hz and 20 kHz, we get sound!

The Flight

The way a pulse width modulation signal works is pretty simple. A pulse is produced at regular intervals (T) typically provided by a timer and its period register. The pulse width (T_{on}) is not fixed, but is programmable and it could vary between 0 and 100% of the timer period. The ratio between the pulse width (T_{on}) and the signal period (T) is called the duty cycle (Figure 15.1).

There are two extreme cases possible for the duty cycle: 0% and 100%. The first one would correspond to a signal that is always off. The second one would be the case when the output signal is always on. The number of possible cases in between, typically a relatively small finite number expressed as a logarithm in base 2, is commonly referred to as the resolution of the PWM. If for example there are 256 possible pulse widths, we will say that we have a PWM signal with an 8-bit resolution.

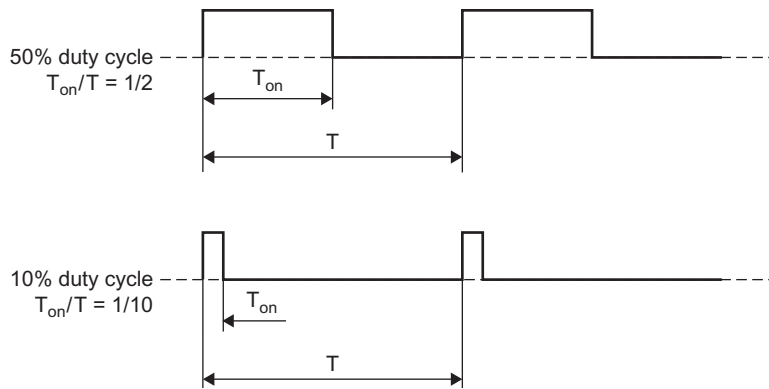


Figure 15.1: Example of PWM signals of different duty cycles

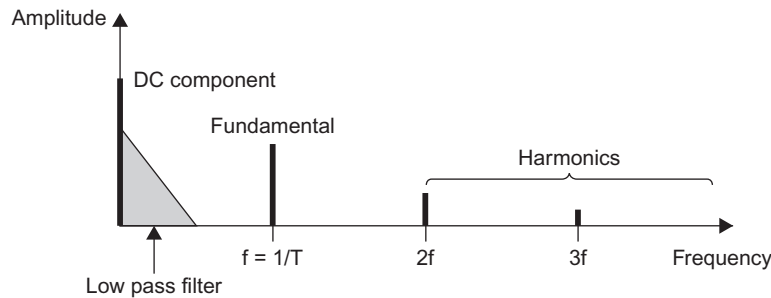


Figure 15.2: Frequency spectrum of a PWM signal

If you could feed an ideal PWM signal with fixed duty cycle to a spectrum analyzer to study its composition, you would discover that it contains three parts (Figure 15.2):

- A DC component, with an amplitude directly proportional to the duty cycle
- A sinusoid at the fundamental frequency ($f = 1/T$)
- An infinite number of harmonics whose frequencies are a multiple of the fundamental ($2f, 3f, 4f, 5f, 6f, \dots$) and whose amplitudes rapidly decrease as the harmonic multiple increases.

Therefore, if we could attach an *ideal* low-pass filter to the output of a PWM signal generator to remove all frequencies from the fundamental and up, we could obtain just a clean DC analog signal whose amplitude would be directly proportional to the duty cycle.

Of course, such an ideal filter does not exist, but we can use a more or less sophisticated approximations of one to remove as much of the unwanted frequency components as needed. This filter could be as simple as a single passive R/C circuit (first-order, low-pass filter) or could require several N active stages ($2 \times N$ -order low-pass) (Figure 15.3).

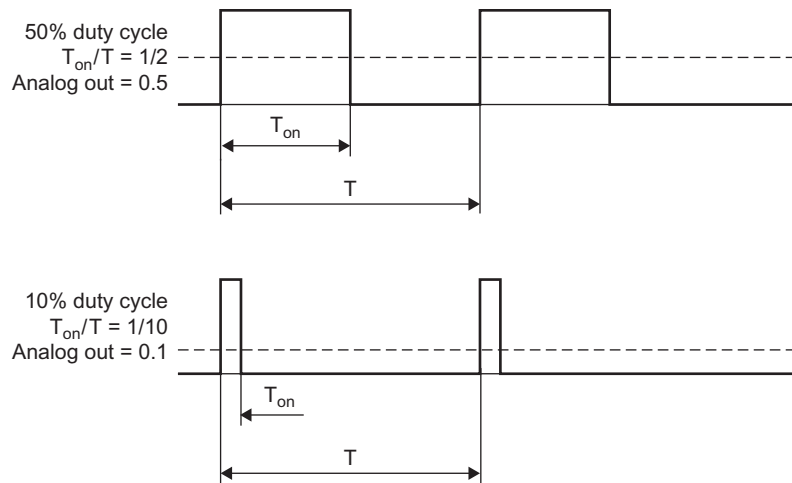


Figure 15.3: Analog output of PWM and ideal low-pass filter circuit

If we aim at producing an audio signal and we choose the PWM frequency wisely, we can take advantage of the natural limitation of the human ear, which will act as an additional filter ignoring any signal whose frequency is outside the 20 Hz to 20 kHz range. In addition to that, most of the audio amplifiers we might want to feed the output signal into will also include a similar type of filter in their input stages. In other words, if we make sure that the PWM signal operates on a frequency at or above 20 kHz both phenomena will contribute to help our cause and will allow us to use a simpler and less expensive filter circuit.

Also intuitively enough, since we can only change the duty cycle once every PWM period, T , the higher the frequency of the PWM, the faster we will be able to change the output analog signal, and therefore the higher the frequency of the audio signal we will be able to generate.

In practical terms this means that the highest audio signal a PWM signal can produce is only half of the PWM frequency. So for example, a 20 kHz PWM circuit will be able to reproduce audio signals only up to 10 kHz, while to cover the entire audible frequency spectrum we will need a base period with a frequency of at least 40 kHz. It is not a coincidence that music CDs are digitally encoded at the rate of 44,100 samples per second.

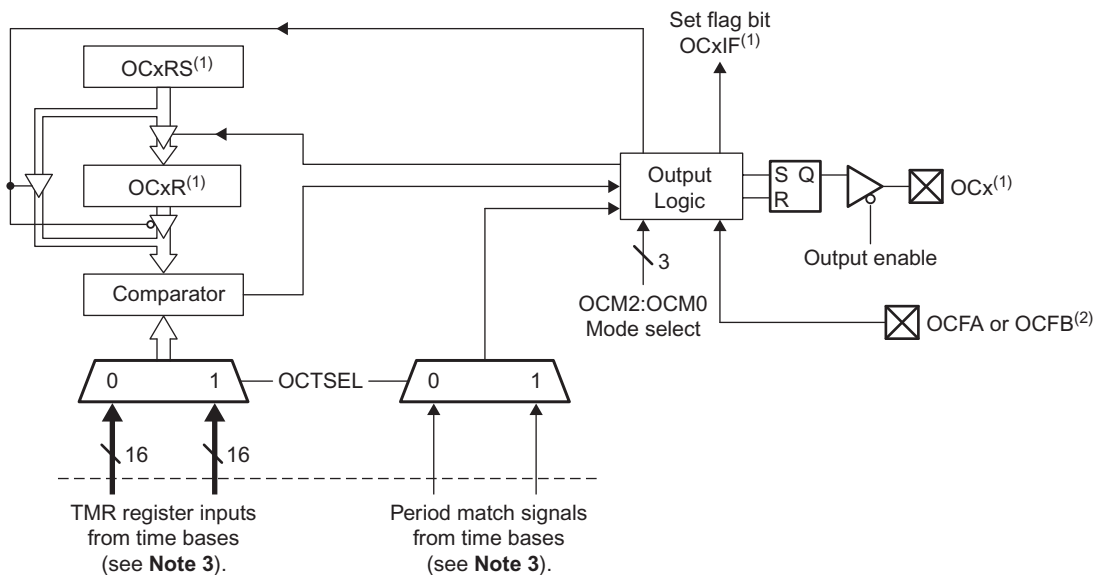
Using the PIC24 OC Modules in PWM Mode

In a previous lesson we have already used the PIC24 Output Compare modules to define precise timing intervals (and produce a video output). This time, we will use the OC modules in PWM mode to generate a continuous stream of pulses with the desired duty cycle (Figure 15.4).

Upper byte:							
U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
—	—	OCSIDL	—	—	—	—	—
bit 15							bit 8

Lower byte:							
U-0	U-0	U-0	R-0 HC	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	OCFLT	OCTSEL	OCM2	OCM1	OCM0
bit 7							bit 0

Figure 15.4: The Output Compare module main control register OCxCON



Note 1: Where “x” is shown, reference is made to the registers associated with the respective output compare channels 1 through 8.

2: OCFA pin controls OC1–OC4 channels. OCFB pin controls OC5–OC8 channels.

3: Each output compare channel can use one of two selectable time bases. Refer to the device datasheet for the time bases associated with the module.

Figure 15.5: Output Compare module block diagram

All we need to do to initialize the OC module to generate a PWM signal is set the three *OCM* bits in the *OCxCON* control register to the 0x110 configuration. A second PWM mode is available (0x111), but we have no use for the fault input pins (OCFA/OCFB), commonly required by a different set of applications as a protection mechanism for (motor control/power conversion). Next, we need to select a timer to base the PWM period on. The choice is limited to Timer2 or Timer3, and for now it will make no difference to us, though it is how we will configure the chosen timer that will make all the difference (Figure 15.5).

Keeping in mind that we want to be able to produce at least a 40kHz PWM period, and assuming a peripheral clock of 16MHz, as is the case when using the Explorer16 board, we can calculate the optimal values for the prescaler and the period register *PR3*. With the prescaler set to a 1:1 ratio, we obtain a 400 clock cycles period for *PR3*, generating an exact $16\text{MHz}/400 = 40\text{kHz}$ signal. This value also dictates the resolution of the duty cycle for the Output Compare module. Since we will have 400 possible values of the duty cycle, we can claim a resolution between 8 and 9 bits as we have more than 256 (2^8) steps but fewer than 512 (2^9). Reducing the frequency to 20kHz would give us one more bit of resolution (between 9 and 10), but would also mean that we would be limiting the output frequency range to a maximum of 10kHz, probably a small but noticeable difference to the human ear. Once the chosen timer is configured and, just before writing to the *OCxCON* register, we will need to set, for the first time, the value of the first duty cycle by writing to the registers *OCxR*, and *OCxRS*. When in PWM mode, the two registers will work in a master/slave configuration. Once the PWM module is started by writing the mode bits in the *OCxCON* register, we will be able to change the duty cycle by writing just to the *OCxRS* register. The *OCxR* register will be updated by copying the new value from *OCxRS*, only and precisely at the beginning of each new period. This avoids glitches and leaves us with an entire period *T* of time to prepare to write the next duty cycle value.

Here is an example of a simple initialization routine for the OC1 module.

```
void InitAudio( void)
{
    // init TMR3 to provide the timebase
    T3CON = 0x8000;      // enable TMR3, 1:1, int clock
    PR3 = 400-1;         // set period for given bitrate
    _T3IF = 0;           // clear interrupt flag
    _T3IE = 1;           // enable TMR3 interrupt

    // init PWM
    // set the initial duty cycles (master and slave)
    OC1R = OC1RS = 200;  // init at 50% duty cycle

    // activate the PWM module
    OC1CON = 0x000E;

} // InitAudio
```

Notice that we have also taken the opportunity to enable the Timer3 interrupt alerting us each time a new period starts, so we can decide how and if to update the *OC1RS* register with the next duty cycle value.

Testing the PWM as a D/A Converter

To start experimenting on the Explorer16 we will need to add just a couple of discrete components to the prototyping area. A resistor of 1 kOhm value and a capacitor of 100 nF value will produce the simplest low-pass filter (first order with a 1.5 kHz cut-off frequency).

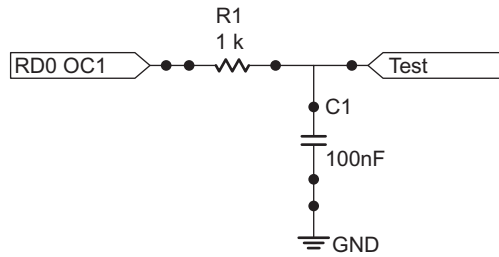


Figure 15.6: Using a PWM signal to produce an analog output

We can connect the two in series and wire them to the output pin of the OC1 module found on pin 0 of PortD as represented in the schematic in [Figure 15.6](#).

A couple more lines of code will complete our short test project.

```
void _ISRFAST _T3Interrupt( void)
{
    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt

main( void)
{
    InitAudio();

    // main loop
    while( 1);

} // main
```

Include the usual header file (*config.h*) and save this code in a new file called ***DATest.c***. You can then create a quick test project, ***15-DtoA***, that will contain this single file.

Launch it, using the **Run>Run Project** command, and connect a meter, or an oscilloscope if available, to the test point to verify the average (DC) output level.

The needle of the meter (or the trace of the scope) will swing to a voltage level of approximately 1.65V; which is, 50% of the regular voltage output of a digital I/O pin on the Explorer16 board (3.3V). This is consistent with the value of the duty cycle set by the initialization routine to 200 (for a period of 400 cycles). If you have an oscilloscope, you can also point the probe directly at the other end of the R1 resistor (directly to the output pin of the OC1 module) and verify that a square wave of the exact frequency of 40kHz and a duty cycle of 50% is present ([Figure 15.7](#)).

You can now change the initialization routine to experiment with other duty cycle values between 0 and 399 to verify the response of the circuit and the proportionality of the analog output signal value between 0 and 3.3V with the selected value of the duty cycle.

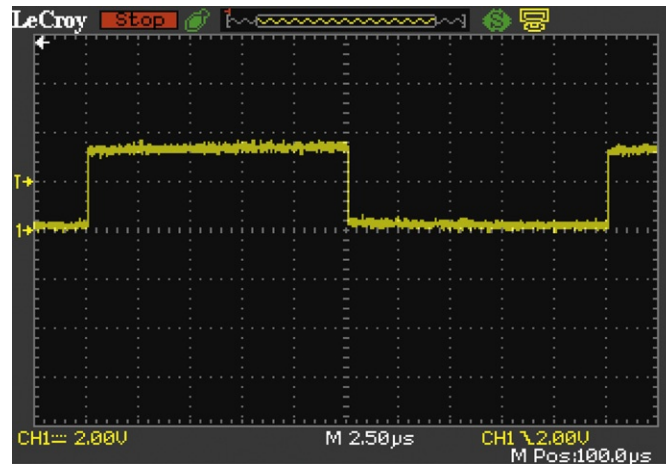


Figure 15.7: A 40 kHz PWM signal with a 50% duty cycle

Producing Analog Waveforms

With help from the OC1 module we have just crossed the boundary between the digital world, made of ones and zeros, and the analog world, where we have been capable of generating a multitude of values between 0 and 3.3 V.

We can now play with the duty cycle, changing it from period to period to produce waveforms of any sort and shape. Let's start by modifying the project a little bit by adding some code to the interrupt routine that so far has been left empty.

```
OC1RS = (count < 20) 400 : 0;
if ( ++Count >= 40)
    Count = 0;
```

You will need to declare *count* as a global integer and remember to initialize it to 0.

Save and rebuild the project to test the new code on the Explorer16 board.

Every 20 PWM period the filtered output will alternate between the value 3 V (100%) and the value 0 V (0%), producing a square wave visible on the oscilloscope at the frequency of 1 kHz (40 kHz/40).

A more interesting waveform could be generated by the following algorithm:

```
OC1RS = Count*10;
if ( ++Count >= 40)
    Count = 0;
```

This will produce a triangular waveform (saw tooth) of approximately 3 V peak amplitude, with a gradual ramp of the duty cycle from 0 to 100% in 40 steps (2.5% each), followed by an abrupt fall back to 0 where it will repeat with a frequency of 1 kHz as well (Figure 15.8).

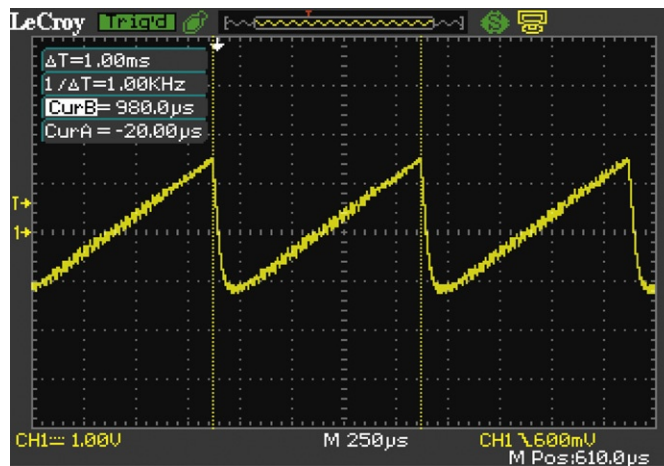


Figure 15.8: A 1 kHz triangular waveform

Neither of the two examples will qualify as a *nice* sound though if you try and feed them to an audio amplifier, although they will both have a recognizable (fundamental) high-pitch tone, at about 1 kHz. Lots of harmonics will be present and will be audible in the audio spectrum, giving the sound an unpleasant buzz.

To generate a single, clean tone what we need is a pure sinusoid. The interrupt service routine below would serve the purpose, generating a perfect sinusoid at the frequency of 400 Hz (in musical terms that would be close to an A).

```
void _ISRFAST _T3Interrupt( void)
{
    // compute the new sample for the next cycle
    OC1RS = 200+ ( 200* sin(Count *0.0628));

    if ( ++Count >= 40)
        Count = 0;

    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt
```

Unfortunately, as fast as the PIC24 and the math libraries of the MPLAB® C compiler are, there is no chance for us to be able to use the *sin()* function, and to perform the multiplications and additions required to obtain a new duty cycle value at the required rate of 400 Hz. The Timer3 interrupt hits every 25 μs, too short a time for such a complex floating-point calculation, so the interrupt service routine would end up *skipping* interrupts and producing a sinusoidal output that is only half the required frequency (one octave

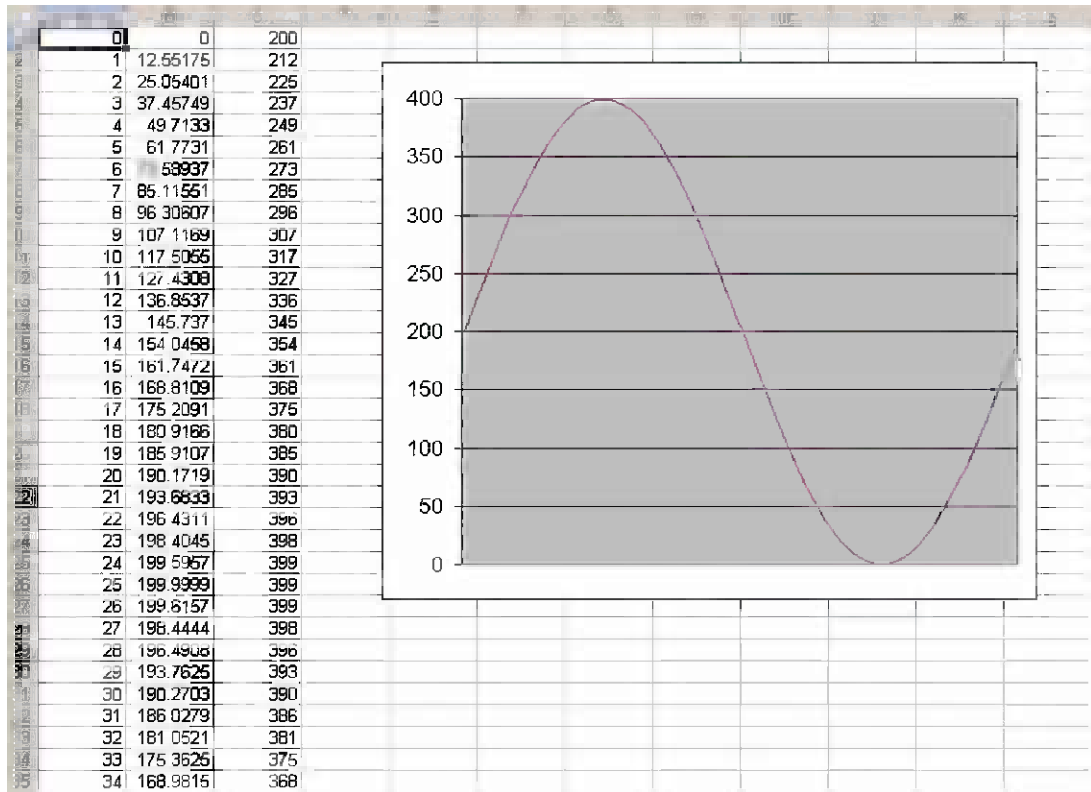


Figure 15.9: Spreadsheet to compute the 400 Hz sinusoid table

lower). Still, if you try and listen to it, feeding the signal to an audio amplifier, you will be immediately able to appreciate the greatly improved clarity of the sound.

For higher frequencies we will need to pretabulate the sinusoid values so we can perform the fewest calculations possible (preferably working on integers only) at run time. Here is an example that uses a table (stored in the FLASH program memory of the PIC24) containing precomputed values. I obtained the table by using a spreadsheet program where I used the following formula:

```
= offset + INT( amplitude * SIN( ROW * 6.28 / PERIOD))
```

for a period of 100 samples (400 Hz), offset and amplitude of 200, I obtained

```
= 200 + INT( 200 * SIN( A1 * 6.28/100))
```

I filled the first column (A) of the spreadsheet with a counter and I copied the formula over the first 100 rows of the second column (B), formatting the output for zero decimal digits (Figure 15.9).

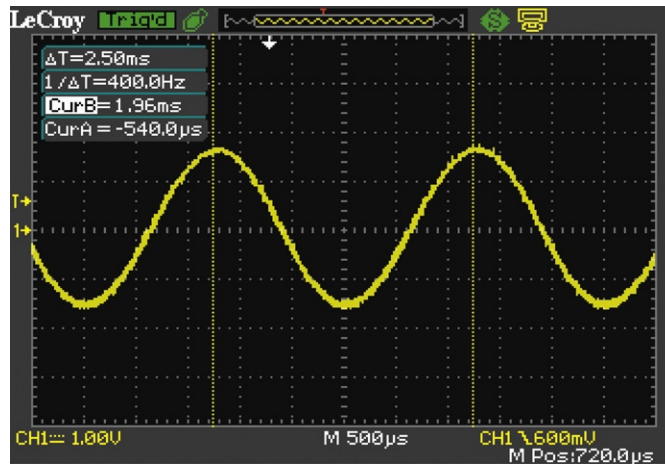


Figure 15.10: A clean 400 Hz sinusoid

Then, I cut and pasted the entire column into the source code, adding commas at the end of each line to comply with the C syntax.

```
int Sample;

const int Table[100] = {
200, 212, 225, 237, 249, 261, 273, 285, 296, 307,
317, 327, 336, 345, 354, 361, 368, 375, 380, 385,
390, 393, 396, 398, 399, 399, 399, 398, 396, 393,
390, 386, 381, 375, 368, 361, 354, 345, 337, 327,
317, 307, 296, 285, 273, 262, 250, 237, 225, 212,
200, 187, 175, 162, 150, 138, 126, 115, 103, 93,
82, 72, 63, 54, 46, 38, 31, 24, 19, 14,
9, 6, 3, 1, 0, 0, 0, 2, 3, 6,
9, 13, 18, 24, 30, 37, 45, 53, 62, 72,
81, 92, 103, 114, 125, 137, 149, 161, 174, 186
};

void _ISRFAST _T3Interrupt( void)
{
    // load the new samples for the next cycle
    OC1RS = Table[ Sample];

    if ( ++Sample >= 100)
        Sample= 0;

    // clear interrupt flag and exit
    _T3IF = 0;
} // T3 Interrupt
```

This time, you will be able to easily produce the 400 Hz output frequency desired (Figure 15.10) and there will be plenty of time between the Timer3 interrupt calls to perform other tasks as well.

Reproducing Voice Messages

Once we learn how to produce sound there is no stopping. There are infinite applications in embedded control where we can put these capabilities to use. Any *human* interface can be greatly enhanced by using sound to provide feedback, to capture the attention of the user with alerts and error messages or, if properly done, simply to enhance the user experience. But we don't have to limit ourselves to simple tones or basic melodies. In fact we can reproduce any kind of sound, as long as we have a description of the waveforms required. Just like the table used for the sinusoid in the previous example, we could use a larger table to contain the unmistakable sound produced by a particular instrument or even a complete vocal message. The only limit becomes the room available in the Flash program memory of the PIC24 to store the data tables along with the application code.

If, in particular, we look at the possibility of storing voice messages, and knowing that the energy of the human voice is mostly concentrated in the frequency range between 400Hz and 4kHz, we can considerably reduce our output frequency requirements and limit the PWM playback to the rate of only 8,000 samples per second. Notice that we should still maintain a high PWM frequency to keep the PWM signal harmonics outside the audio frequency range and the low-pass filter simple and inexpensive. It is only the rate at which we change the PWM duty cycle and we read new data from the table that will have to be reduced, in this case, once every five interrupts ($40,000/8,000 = 5$) will do. With 8,000 samples per second we would theoretically be able to play back as much as 16 seconds of voice messages stored inside the controller Flash memory. That is already a lot of talking for a single-chip solution. To increase the capacity further, potentially doubling it, we could start looking at simple compression techniques used for voice applications such as for example ADPCM. ADPCM stands for Adaptive Differential Pulse Coded Modulation. It is based on the assumption that the difference between two consecutive samples is smaller than the absolute value of each sample and therefore can be encoded using a smaller number of bits. The actual number of bits used is optimized; changing dynamically so as to avoid signal distortion while providing the desired compression ratio; hence the use of the term *adaptive*.

A Media Player

In the rest of this lesson though, we will focus on a much more ambitious project, putting to use all the libraries and capabilities we have acquired in the last several lessons. We will attempt to create a basic multimedia application capable of playing stereo music files off an SDTM/MMC memory card.

Using a low-cost dual operational amplifier, like the MCP602, we can design a very simple Sallen Key (second order) low-pass filter for the audio band, perfectly capable of driving a small headset or to feed a more powerful stereo amplifier (Figure 15.11).

The format of choice for the data will be the uncompressed *WAVE* format, which is compatible with almost any audio application and is the default destination format when extracting files from a music CD.

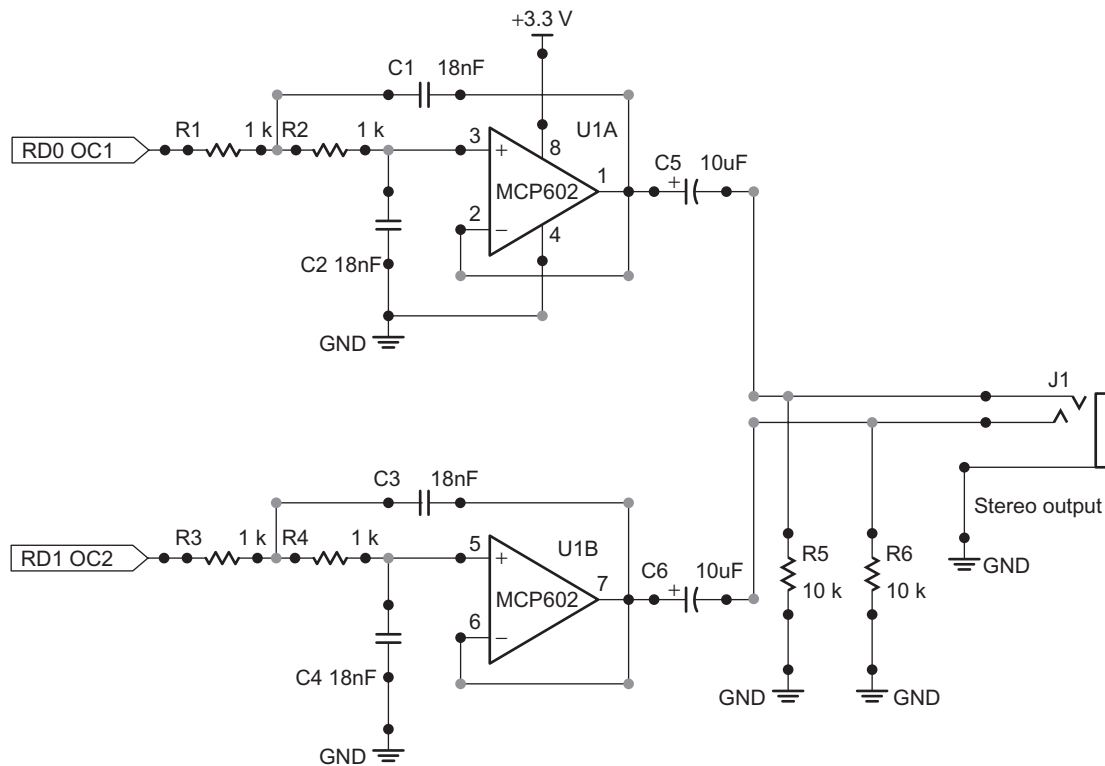


Figure 15.11: A simple audio PWM filter circuit

We will start by creating a brand new project (*15-Wave*) using the *New Project* checklists. We will immediately add the SD/MMC low-level interface and the file I/O library for access to a FAT16 file system to the project source files list.

After opening a file for reading though, this time we will need to be able to understand the specific format used to encode the data it contains.

The WAVE File Format

Files with the .WAV extensions, encoded in the *WAVE* format, are among the simplest and best documented, but they still require some careful study. The *WAVE* format is a variant of the *RIFF* file format, a standard, across multiple operating systems, which uses a particular technique to store multiple pieces of information/data, dividing them into *chunks*. A chunk is nothing more than a block of data preceded by a header containing two 32-bit elements: the *chunk ID*, and the *chunk size* (Table 15.1).

Table 15.1: Format of a data *chunk*

Offset	Size	Value	Description
0x00	4	ASCII	<i>Chunk ID</i>
0x04	4	Size	<i>Chunk size</i> (size of the content)
0x08	size		Data content
0x08 + size	1	0x00	Optional padding

Table 15.2: *RIFF* chunk of type *WAVE*

Offset	Size	Value	Description
0x00	4	“RIFF”	This is the RIFF chunk ID
0x04	4	Size	(size of the data block + 4)
0x08	4	“WAVE”	Type ID
0x10	Size		Data block (sub-chunks)

Note also that the chunk total size must be a multiple of two so that all the data in a *RIFF* file ends up being nicely word-aligned. If the data block size is not a multiple of two, an extra byte of padding is added to the chunk.

A chunk with the *RIFF ID* is always found at the beginning of a .WAV file and its data block begins with a 4-byte *type* field. This type field must contain the string *WAVE*. Chunks can be nested like Russian dolls, but there can also be multiple sub-chunks inside a given type of chunk.

Table 15.2 illustrates a .WAV file *RIFF* chunk structure.

The data block in turn contains an *fmt* chunk followed by a *data* chunk. As is often the case, one image is worth a million words (Figure 15.12).

The *fmt* chunk contains a defined sequence of parameters that fully describes the stream of samples that follows in the *data* chunk, as represented in Table 15.3.

In between the *fmt* and *data* chunks there could be other chunks containing additional information about the file so we might have to scan the chunk IDs and skip through the list until we find the chunk we are looking for.

The Play() Function

Let’s create a new software module that will take care of opening a given .WAV file and, after capturing and decoding the information in the *fmt* chunk, will initialize an audio output module similar to, if not just more sophisticated than, the one we developed in the first part of the lesson; we will call it *Wave.c*.

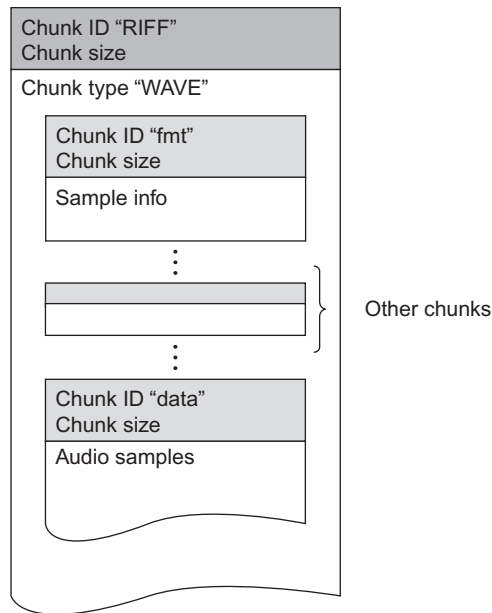


Figure 15.12: Basic WAVE file layout

Table 15.3: The *fmt* chunk content

Offset	Size	Description	Value
0x00	4	Chunk ID	"fmt"
0x04	4	Chunk data size	16 + extra format bytes
0x08	2	Compression code	Unsigned int
0x0a	2	Number of channels	Unsigned int
0x0c	4	Sample rate	Unsigned long
0x10	4	Average bytes per second	Unsigned long
0x14	2	Block align	Unsigned int
0x16	2	Significant bits per sample	Unsigned int (>1)
0x18	2	Extra format bytes	Unsigned int

```

/*
** Wave.C
**
** Wave File Player
** Uses 2 x 8 bit PWM channels at 44kHz
*/

#include <EX16.h>
#include <AudioPWM.h>
#include <SDMMC.h>

```

```

#include <fileio.h>
#include <stdlib.h>
#include "Wave.h"

// WAVE file constants
#define RIFF_DWORD 0x46464952UL
#define WAVE_DWORD 0x45564157UL
#define DATA_DWORD 0x61746164UL
#define FMT_DWORD 0x20746666UL
#define WAV_DWORD 0x00564157UL

typedef struct {
    long ckID;
    long ckSize;
    long ckType;
} chunk;

typedef struct {
    // format chunk
    unsigned short subtype; // compression code
    unsigned short channels; // # of channels
                                // (1= mono, 2= stereo)
    unsigned long srate; // sample rate in Hz
    unsigned long bps; // bytes per second
    unsigned short align; // block alignment
    unsigned short bitsample; // bit per sample
    // unsigned short extra; // extra format bytes
} WAVE_fmt;

```

A *WAVE_fmt* structure will be useful to collect all the *fmt* parameters in one place and the *chunk ID* macros will help us recognize the different unique IDs, treating them as long integers and allowing us a quick and efficient comparison.

We can now start coding the *PlayWAV()* function, which will take only one input parameter: the file *name* as a string.

```

unsigned PlayWav( char *name)
{
    chunk      ck;
    WAVE_fmt    wav;
    int         last;
    MFILE       *fp;
    unsigned long lc, r, d;

    // audio codec parameters
    int skip, size, stereo;
    unsigned long rate;

    // 1. open the file
    if ( (fp = fopenM( name, "r")) == NULL)
    { // failed to open
        return FALSE;
    }
}

```

After trying to open the file and reporting an error if unable, we will immediately start looking inside the data buffer for the *RIFF* chunk ID and the *WAVE* type ID as a signature, which will confirm to us we have the right kind of file.

```
// 2. verify it is a RIFF-WAVE formatted file
freadM( (void*)&ck, sizeof(chunk), fp);

// check that file type is correct
if (( ck.ckID != RIFF_DWORD) || ( ck.ckType != WAVE_DWORD))
    goto Exit;
```

If successful, we should verify that the *fmt* chunk is the first in line inside the data block. Then we will harvest all the information needed to process the data block for the playback.

```
// 3. look for the chunk containing the wave format data
freadM( (void*)&ck, 8, fp);
if ( ck.ckID != FMT_DWORD)
    goto Exit;

// 4. get the WAVE_FMT struct
freadM( (void*)&wav, sizeof(WAVE_fmt), fp);
stereo = wav.channels;

// 5. skip extra format bytes
fseekM( fp, ck.ckSize - sizeof(WAVE_fmt));
```

Next, we start looking for the *data* chunk, inspecting the chunk ID fields of the next block of data, after the end of the *fmt* chunk, and skipping the entire block if not matching.

```
// 6. search for the "data" chunk
while( 1)
{
    // read next chunk
    if ( freadM( (void*)&ck, 8, fp) != 8)
        goto Exit;

    if ( ck.ckID != DATA_DWORD)
        fseekM( fp, ck.ckSize );
    else
        break;
}
```

Note

Typical .WAV files, produced by extracting data from a music CD for example, will have just the *data* chunk immediately following the *fmt* chunk. Other applications (MIDI interfaces for example) can generate .WAV files with more complex structures including multiple *data* chunks, *playlists*, *cues*, *labels*, etc. but we aim at playing back only the plain vanilla type of .WAV files.

Once found, the size of the *data* chunk will tell us the real amount of samples contained in the file.

```
// 7. find the data size
lc = ck.ckSize;
```

The playback sample rate must be now taken into consideration to determine whether we can *play* that fast; it could in fact exceed our capabilities, and we might have to skip every other sample to reduce the data rate. We will consider 48,000 samples/second our limit so that we will be able to read the data fast enough to maintain at least an 8-bit resolution.

```
// 8. compute the period and adjust the bit rate
rate = wav.srate;           // r = samples per second
skip = 1;                   // skip factor to reduce noise
while ( rate < 22050)
{
    rate <= 1;               // divide sample rate by two
    skip <= 1;               // multiply skip by two
}
```

Higher rates will be treated by gradually dividing the rate by a factor of two and doubling the skip.

We can then compute the required PWM period value (to be used to set the *PR* register). A problem could occur if the required period exceeds the available bits in the register (16) resulting in a period value greater than 65,536.

```
// 9. check if the sample rate compatible with TMR3 prescaler
d = (FCY/rate)-1;
if ( d > ( 65536L))           // max TMR3 period (16 bit)
{
    fcloseM( fp);
    return FALSE;
}
```

During the playback, we will keep track of the number of samples extracted from the file to determine when we have reached the end of the file.

To make the playback smooth, we will use a double buffering scheme, so that as the audio interrupt routines are fetching data from one buffer, we will take our time to re-fill the other buffer with new data from the file. The array *ABuffer[]* is in fact defined as two blocks of *B_SIZE* bytes each. *B_SIZE* is chosen to be a multiple of 512, so that the calls to the *freadM()* function will be able to transfer entire sectors of data at a time for maximum efficiency. We will have to verify that the time required for *freadM()* to fill one buffer will be shorter than the time required to play back (consume) all the data in the second buffer by the PWM interrupt service function.

When starting the double buffering scheme, we will fill both buffers to get a head start.

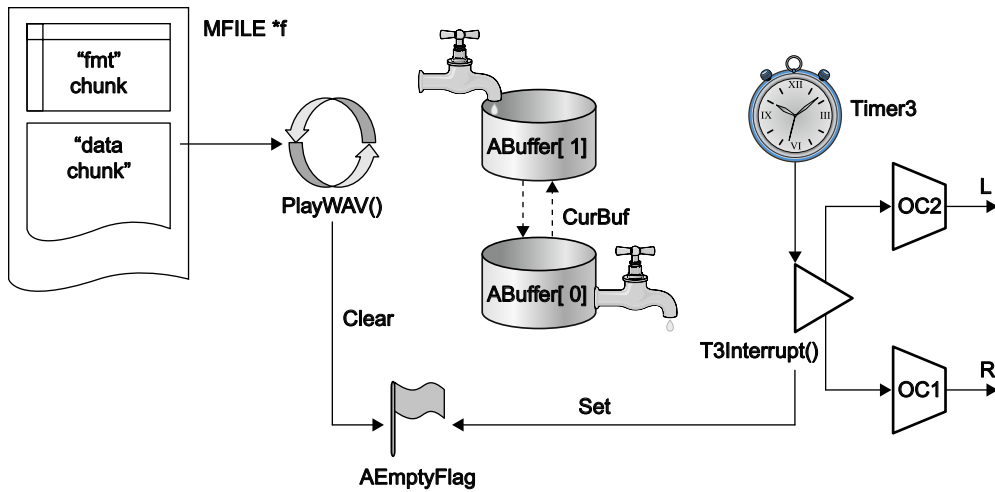


Figure 15.13: WAVE player dataflow

```
// 10. init the Audio state machine
CurBuf = 0;
stereo = wav.channels;
size = 1; // default, bytes per channel
if ( wav.bitsample == 16)
    size = 2;

// 11. start loading both buffers
if ( lc < B_SIZE*2) // allow for files > BSIZE*2
    goto Exit;
r = freadM( ABuffer[0], B_SIZE*2, fp);
AEmptyFlag = FALSE; // both buffers are full
lc -= B_SIZE*2 ; // keep track of what is left
```

The assumption here is that the .WAV file will contain at least enough data to fill the two buffers, but if you plan on using very short files, containing less than a few Kbytes of data, you might want to modify this code, check the number of bytes returned by *freadM()* and add the correct padding at the end of the buffer(s).

At this point we are ready to initialize the audio playback *machine*, which will be simply our *T3Interrupt()* function modified to accommodate two channels for stereo playback (Figure 15.13).

We will also add the ability to skip samples to reduce the sample rate if necessary, and the ability to deal with 16-bit samples (signed) as well as 8-bit samples (unsigned). All this information will be passed to the audio module *InitAudio()* routine as a short list of parameters.

```
// 12. start playing, enable the interrupt
InitAudio( rate, skip, size, stereo);
```


As the timer interrupt is activated, the service routine starts immediately consuming data from the first buffer, and as soon as its whole content is exhausted, it will set the flag *AEmptyFlag* to let us know that new data needs to be retrieved from the WAV file and the second buffer will be selected as the active one. Therefore, to maintain the playback flowing smoothly, we will sit in a tight loop, constantly checking for the *AEmptyFlag*, ready to perform the refill, counting the bytes we read from the file until we use them all up.

```
// 13. keep feeding the buffers in the playing loop
while (lc >= B_SIZE)
{
    if ( ReadKEY()          // on pressing any key
    {
        lc = 0;             // signal playback completed
        break;
    }
    if ( AEmptyFlag)
    {
        r = freadM( ABuffer[1-CurBuf], B_SIZE, fp);
        AEmptyFlag = FALSE;
        lc -= B_SIZE;
    }
} // while wav data available
```

Actually we will stop a little sooner, when the data left in the file is no longer sufficient to fill another entire buffer load. In that case, unless the data block size was an exact multiple of the buffer size and there is no new data to read, the last piece is loaded and needs to be padded to fill completely what will be the last buffer to play back.

```
// 14. flush the buffers with the data tail
if ( lc > 0)
{
    // load the last sector
    r = freadM( ABuffer[1-CurBuf], lc, fp);
    last = ABuffer[1-CurBuf][r-1];
    while(( r < B_SIZE) && (last > 0))
        ABuffer[1-CurBuf][r++] = last;

    // wait for current buffer to be emptied
    AEmptyFlag = 0;
    while (!AEmptyFlag);
}
```

We then wait for the completion of the playback of the very last buffer and we immediately terminate the audio playback.

```
// 15. finish the last buffer
AEmptyFlag = 0;
while (!AEmptyFlag);
```

Exit:

```
// 16. stop playback
HaltAudio();
```

Closing the file we will release the memory allocated and we will return to the calling application.

```
// 17. close the file
fcloseM( fp);

// 18. return with success
return TRUE;
```

```
} // PlayWAV
```

To complete this module we will need to create a small include file, *Wave.h*, to publish the prototype of the *PlayWAV()* function and save it in the */include* subdirectory.

```
/*
**   Wave.h
**
**   Wave File Player
**   Uses 2 x 8 bit PWM channels at 44,1kHz
*/
unsigned PlayWAV( char *name);
```

There is one last detail that needs to be explained. During the *WAV* file decoding we made use of the convenient *fseekM()* function to help us advance by a given number of positions while reading a file. Here is one way to implement it:

```
//-----
// fseekM
//
// Advances the file pointer by count positions from CUR_POS
// simple implementation of a seek function
// returns 0 if successful
unsigned fseekM( MFILE * fp, unsigned count)
{
    char buffer[16];    // a small buffer
    unsigned d, r;

    while ( count)
    {
        d = ( count >= 16) ? 16 : count;
        r = freadM( buffer, d, fp);
        count -= r;
        if ( r != d)
            break;      // reached end of file or error
    } // while

    return count;
} // fseekM
```

You can add it to the bottom of the *fileio.c* module and add the corresponding prototype to the *fileio.h* header file.

```
unsigned fseekM( MFILE *, unsigned);
```

The Low-Level Audio Routines

The *PlayWAV()* function we have just completed relied heavily on a lower-level audio module to perform the actual Timer and OC peripherals initialization as well as to perform the actual periodic update of the PWM duty cycle. We will call this module ***AudioPWM.c*** and it will be based mostly on the code developed in the beginning of this chapter, extended to manage two channels for stereo playback and a number of additional options. The OC1 and OC2 modules will be used simultaneously to produce the left and right channels. The timer interrupt routine will be the real core of the playback functionality. A pointer, *BPtr* will keep track of our position inside each buffer as we will be using up the data to feed the PWM modules with new samples at every period.

```
static int sk = 1;

// 1. skip to increase the bitrate (avoid PWM noise)
if ( --sk == 0)
{
    // reload the skip
    sk = Skip;

    // 2. load the new samples for the next cycle
    OC1RS = 30+(*BPtr ^ Fix);
    if ( Stereo==2)
        OC2RS =30 + (*(BPtr + Size) ^ Fix);
    else // mono
        OC2RS = OC1RS;
    BPtr += Bytes;
```

The pointer is advanced by a number of bytes that depends both on the size of the samples (16 or 8 bits each) and on the need to skip samples to reduce the sample rate when the *PlayWAV()* function determines it is necessary.

As soon as a buffer-load of data is used up, we need to swap the active buffer

```
// 3. check if buffer emptied
BCount -= Bytes;
if ( BCount <= 0)
{
    // 3.1 swap buffers
    CurBuf = 1- CurBuf;
    BPtr = &ABuffer[ CurBuf][Size-1];
```

Reset the samples counter and set a flag to alert the *PlayWAV()* routine we need a new buffer to be prepared before we run out of data again.

```
        // 3.2 buffer refilled
        BCount = B_SIZE;

        // 3.3 flag a new buffer needs to be filled
        AEmptyFlag = 1;
    }
}
```

Only then can we exit after clearing the interrupt flag.

```
    // 4. clear interrupt flag and exit
    _T3IF = 0;

} // T3 Interrupt
```

The initialization routine is equally straightforward (if you recall the one we created at the beginning of the chapter), only more parameters are passed from the calling application and copied into the module's own (private) variables.

```
void InitAudio( long bitrate, int skip, int size, int stereo)
{
    // 1. init pointers
    CurBuf = 0;           // start with buffer0 active first
    BPtr = &ABuffer[ CurBuf][size-1];
    BCount = B_SIZE;      // number of samples to be played
    AEmptyFlag = 0;
    Skip = skip;
    Fix = (size==2)? 0x80 : 0; // sign correction for 16-bit
    Stereo = stereo;
    Size = size;
    Bytes = size*stereo;
```

One buffer is selected as the *current* in use buffer and all the pointers and counters are initialized. Then the timer and its interrupt mechanism are initialized.

```
    // 2. init the timebase
    T3CON = 0x8000;      // enable TMR3, 1:1, internal clock
    PR3 = FCY / bitrate; // set the period f
    Offset = PR3/2;
    _T3IF = 0;           // clear interrupt flag
    _T3IE = 1;           // enable TMR3 interrupt
```

Next the duty cycles are initialized next to an initial offset that will be half the value of the period to provide an even 50% initial output level.

```
    // 3. set the initial duty cycles
    OC1R = OC1RS = Offset; // left
    OC2R = OC2RS = Offset; // right
```

Lastly, the Output Compare modules are fired up.

```
    // 4. activate the PWM modules
    OC1CON = 0x000E;      // CH1 and CH2 PWM mode, TMR3 base
    OC2CON = 0x000E;

} // InitAudio
```

The function *HaltAudio()*, called at the end of the playback, will definitely be the simplest. Its only task is that of disabling the Timer3 and therefore freezing the Output Compare modules and with it the entire interrupt mechanism.

```
void HaltAudio( void)
{
    T3IE = 0;                // disable TMR3 interrupt
} // Halt audio
```

To complete the module you will need the usual header, include files and definitions of the global variables allocated, which will include the audio buffers.

```
/*
** AudioPWM.c
**
*/
#include <EX16.h>
#include <AudioPWM.h>

// global definitions
unsigned Offset;                // 50% duty cycle value
unsigned char _FAR ABuffer[ 2][ B_SIZE]; // double data buffer
int CurBuf;                    // index of buffer in use
volatile int AEmptyFlag;       // flag a buffer needs to be filled

// internal variables
int Stereo;                    // flag stereo play back
int Fix;                       // sign fix for 16-bit samples
int Skip;                      // skip factor to reduce sample/rate
int Size;                      // sample size (8 or 16-bit)

// local definitions
unsigned char *BPtr;           // pointer inside active buffer
int BCount;
int Bytes;                    // number of bytes per sample
```

Notice that just as we did in previous lessons, when allocating large buffers for video applications we can use the *far* attribute to allocate memory beyond the PIC24 near addressing space.

The include file **AudioPWM.h** will publish all the necessary definitions and prototypes for the **Wave.c** module and other applications to make use of the services provided by the Audio PWM module. Let's copy AudioPWM.h and AudioPWM.c to the */include* and */lib* subdirectories respectively.

```
/*
** AudioPWM.h
**
*/
#include <EX16.h>                // defines FCY
#define TCYxUS 16               // number of Tcycles in a microsecond
#define B_SIZE 2048             // audio buffer size

extern unsigned char ABuffer[ 2][ B_SIZE]; // double data buffer
```

```
extern int CurBuf;                // index of buffer in use
extern volatile int AEmptyFlag; // flag a buffer needs to be filled

void InitAudio( long bitrate, int skip, int size, int stereo);
void HaltAudio( void);
```

Testing the WAVE File Player

Now that the low-level audio module and the playback module have been completed it is time to put it all together and start testing playing some music.

Let's create a new project called **15-Wave** and let's immediately add all the necessary modules and their include files to the project; they are: *sdm mmc.c*, *fileio.c*, *AudioPWM.c*, *Wave.c*, and all the matching include (.h) files.

Then, let's add a new main module, **WaveTest.c**, which will contain just a few lines of code. It will invoke the *PlayWAV()* function indicating the name of a single file that we will have copied onto the SD/MMC card.

```
/*
**  WaveTest.c
**
*/
#include <config.h>
#include <fileio.h>
#include "Wave.h"

int main( void)
{
    InitEX16();

    if ( !mount())
        PORTA = FError + 0x80;
    else
    {
        PORTA = 0xFF;
        if ( PlayWAV( "SONG.WAV"))
            PORTA = 0x1;
    } // mounted

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

The PortA row of LEDs will serve as our display to report errors, should the *mount()* operation fail or should the file not be found on the storage device.

Build the project and program the code on the Explorer16 board using the appropriate **Run>Run Project** command. Don't forget to reserve some room for the Heap, as the *fileio* module uses it to allocate buffers and data structures.

Table 15.4: Reload period for common WAV file formats

File	Sample Size	Channels	Sample Rate	Byte Rate	Reload Period (ms)
Voice mono	1	1	8,000	8,000	64.0
Voice stereo	1	2	8,000	16,000	32.0
Audio 8-bit mono	1	1	22,050	22,050	23.2
Audio 8-bit stereo	1	2	22,050	44,100	11.6
Audio 8-bit high bit-rate mono	1	1	44,100	44,100	11.6
Audio 8-bit high bit-rate stereo	1	2	44,100	88,200	5.8
Audio 16-bit mono	2	1	44,100	88,200	5.8
Audio 16-bit stereo	2	2	44,100	176,400	2.9

To proceed gradually, I would recommend that you test the program with WAV files of increasingly higher sample rates and sizes. For example, you should run the first test with a WAV file using 8-bit samples, mono, at 8,000 samples/second. Then proceed gradually, increasing the complexity of the format and the speed of playback, possibly aiming to reach with a last test the full capabilities of our application with a 16-bit per sample, stereo, 44,100 samples/second file. The reason for this gradual increase is that we will need to verify that the performance of our *fileio.c* module is up to the task. In fact, as the sample rate, number of channels and size of the samples increase, so does the bandwidth required from the file system. We can quickly calculate the performance levels required by a few combinations of the above parameters.

Table 15.4 shows the byte-rate required by each file format, that is, the number of bytes that get consumed by the playback function for every second (sample size \times channels \times sample rate). In particular, the last column shows how often a new sector full of data will be required to produce uninterrupted output (512/byte-rate), which gives us the time available for the *PlayWAV()* routine to read the next sector of data from the WAV file.

Note

Since the PIC24 PWMs can only produce less than 9 bits of resolution when operating at the 44,100 Hz frequency, the audio PWM module has been designed to use only the MSB of a 16-bit sample, therefore don't expect any increase in the quality of the audio output once you attempt to play back a WAV file in one of the last two formats. All you obtain at that point is a waste of the space on the SD/MMC memory card. If you want to maximize the use of the storage space available, make sure that, when copying a file onto the card, you reduce the sample size to 8 bits. You will be able to pack double the number of music files on the card for the same output audio resolution.

If you start experimenting gradually, as I suggested, moving down the table, you should find that beyond a certain point things just won't play out right. The playback is going to skip, repeat and hiccup and it just won't sound right. What is happening is that the *freadM()* function has reached its limit and is not capable of keeping up with the audio playback demands. The time it takes, on average, to load a new buffer of data is longer than the time it takes to consume one, hence after a short while the *PlayWAV()* routine starts falling behind and the audio playback function starts repeating a buffer or playing back buffers that are not completely filled yet.

Optimizing the File I/O

When we wrote the file I/O library, and even before, when we wrote the low-level functions to access the SD/MMC card, we were focusing just on getting things done, but we have never really tried to assess the level of performance provided. Perhaps now we have the right motivation to go and look into it. Throughout this book, we have resisted using any of the optimization features of the compiler, so that every example could be simply tested using the free MPLAB C Lite compiler version, and we want to maintain this commitment. Perhaps there is some room to improve the performance using just a little ingenuity.

The first thing to do is to discover where the PIC24 is spending the most time when reading data from the card. Inspecting the *freadM()* function, you will notice that there are only two calls to lower-level subroutines. One is a *ReadDATA()* function call used to load a new sector from the current cluster and the other is a *NextFAT()* function call used to identify the next cluster, once every sector of the current cluster is exhausted. Eventually, both functions will call in their turn the *ReadSECTOR()* function to actually retrieve a block of 512 bytes of data. Lastly, a call to the standard C function *memcpy()* is performed to transfer a data block to the calling application buffer. So the ultimate performance of *freadM()* is going to depend on the performance of *ReadSECTOR()* and *memcpy()*.

LED Profiling

To determine which one of the two has the largest responsibility is a relatively easy job if you happen to have an oscilloscope or a logic analyzer at hand. In fact, if you remember, we designed *ReadSECTOR()* to use one of the LEDs on PortA to signal when a read operation is being performed on the SD/MMC card. If we point the logic analyzer to the anode of the corresponding LED during the playback loop, we should be able to see a periodic pulse whose length indicates the exact amount of time the PIC24 is spending inside the *ReadSECTOR()* function while transferring data. The pause in between the pulses is otherwise proportional to the time spent inside the *memcpy()* function and eventually most of the rest of the *freadM()* function call stack. In one single glance you will immediately realize which part contributes the most to the performance of the overall function.

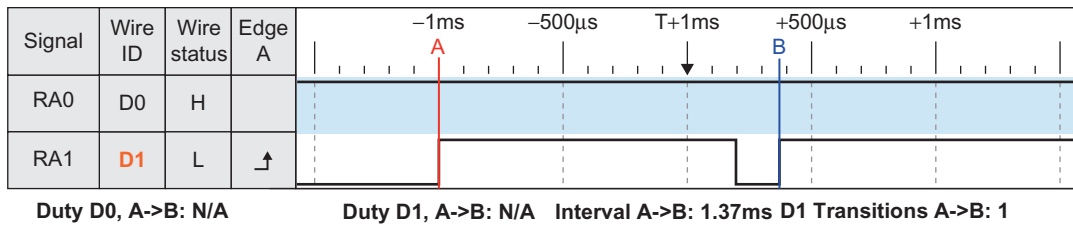


Figure 15.14: READ_LED pin duty cycle

The value reported in Figure 15.14 (obtained using the optimization level of the MPLAB C compiler, available in all free versions) is telling us that we don't have anything to worry about. Even in the case of playback of one of the highest bit-rate, stereo, 16-bit WAV files, our routines are capable of producing the data in a timely and without interruption. In fact, the *freadM()* function can read a new sector in approximately 1.37 ms, well below the required limit of 2.9 ms obtained from Table 15.4.

Note

In the AudioPWM.h header file, we have defined the audio buffer size (B_SIZE) to be 2048 bytes. In other words, we have arranged for the equivalent of four sectors of data to be read at a time. This provides additional margin and guarantees a very smooth playback even if interrupts are disabled temporarily during the main program execution.

Post-Flight Briefing

This final lesson was perhaps the ideal conclusion for our learning experience, as we mixed the most advanced software and hardware capabilities in a project that covered both the digital and the analog domains. We started using the Output Compare peripherals to produce analog signals in the audio spectrum of frequencies. We used this new capability together with the *fileio.c* module, developed in the previous lesson, to play back uncompressed music files (WAV file format) from a mass storage device (SD/MMC card). The basic media player application obtained represents only a new starting point. There is no limit to the possible expansions of this project and if I have managed to excite your curiosity and imagination, there is no limit to what you can do with the PIC24 and the MPLAB C compiler.

Tips & Tricks

The beginning and the end of the playback are two critical moments for the PWM modules. At rest, the output filter capacitor is discharged, and the output voltage is 0V. But as soon as the playback begins a 50% duty cycle will force it to ramp very quickly to approximately

a 1.5V level, producing a loud and unpleasant click. The opposite might happen at the end of playback, should we turn off the PWM modules instead of just disabling the interrupts as we did in the demo project. The phenomenon is not dissimilar to what happens to analog amplifier circuits at power-on and power-off. A simple work-around consists of adding in just a couple of lines of code. Before the timer interrupt is enabled and the playback machine starts, add a small (timed) loop to gradually increase the output's duty cycle from zero all the way up to the value of the first sample taken from the playback buffer.

Exercises

- Investigate the decoding techniques for ADPCM signals for use with voice messages (see application note AN643).
- Search for all the WAV files on the card, and build a playlist.
- Implement a *shuffle* mode using the pseudo-random number generator and gradually emptying the playlist.
- Experiment with basic digital filtering techniques to remove undesired frequencies, boost others or simply distort sounds and voices.

Books

- Mandrioli, D., Ghezzi, C., 1987. *Theoretical Foundations of Computer Science*, John Wiley & Sons, New York, NY.
Not easy reading, but if you are curious about the deep mathematical theoretical foundations of computer science...
- Cook, Leroy. *101 Things to Do With Your Private Pilot's License*, McGraw-Hill, Inc.

Links

- <http://en.wikipedia.org/wiki/RIFF>
The RIFF file format explained.
- <http://en.wikipedia.org/wiki/WAV>
The WAVE file format explained.
- <http://ccrma.stanford.edu/courses/422/projects/WaveFormat/>
Another excellent description of the WAVE file format.

Nel Blu Dipinto Di Blu

Italy 1958 / Domenico Modugno

Written by Franco Migliacci & Domenico Modugno

Penso che un sogno così non ritorni mai più':

Mi dipingevo le mani e la faccia di blu

Poi d'improvviso venivo dal vento rapito

E incominciavo a volare nel cielo infinito

Volare, oh... cantare, oh...

The lyrics are in Italian. The title translates to “In The Blue (Sky), Painted in Blue” (“Volare”: to fly). Modugno sings about dreaming of painting his face and hands in blue and, after being lifted by a sudden wind gust, flying away in the blue sky.

Dare, make your dreams come true!

Index

A

Acknowledge bit, 125
Adaptive differential pulse coded modulation (ADPCM), 361
ADC application, 182
ADC1BUF0 register, 180
ADC clock source, 179
ADC control registers, 177
ADC module, 179
 block diagram, 176
AD1CON1 register, 178, 179
AD1CON3 register SAM bits, 179
Add a counter (*KTimer*), 210
addressSEE() function, 132, 133, 136
AD1PCFG register, 19
AD1PCGF special-function register
 analog-to-digital conversion (ADC) module, 17
Alarm Mask, 81
2-ALoopInThePattern, 24
AMASK register, 84
Analog-to-digital conversion (ADC) module, 17
 AD1PCFG, port configuration register, 17
Analog-to-digital converter (ADC), 81, 175, 204
 module, 183
Analog-to-digital functions, 180
Analog vs. digital pin control, 17–18
Analog waveforms, producing, 357–360
Animate mode, 23, 29, 33
ANSI C libraries, math libraries, 64
ANSI C standard, 54
 integer data types, 54

 long (32-bit) integer, 56, 57
 long integers, 55
 long-long-integer (64-bit) multiplications, 62
 long long integer type (64-bit), 57–58
 long/long long integers, 62
 short integers, 55
ANSI escape codes, 173
Append line, 147
Append option, 321
Arithmetic expressions, 27
Arrays, 41–42
 data, type of, 100
 declarations, 41, 42
 flash memory, 293
 string, 93
 variable, 94
ASCII characters, 160, 220, 264, 266
 arrays, 87
ASCII code, 219
ASCII data, 161
ASCII setup, 147
 dialog box, 148
Assembly experts, notes for, 19–21
 floating-point numbers, 62–64
Assembly programmer, 33
Asynchronous communication, 141–156
 interfaces, 107
Asynchronous serial communication module, 276
Asynchronous serial interface block diagram, 108
AudioPWM.h, 373
Audio routines, low-level, 371–374
AV16/32 I/Os, 287

B

Back porch, 226
Basic root directory entry structure, 309
Basic serial interfaces
 comparison of, 108
Basic software/hardware components, 23–24
Baud-Rate Generator (*BREG2*), 144
Baud-rate register (*I2CxBRG*), 128
Binary operators
 AND , OR , NOT, 33
16-bit ALU, 64
Bit BLock Transfer (BitBLT), 265
16-bit color QVGA, 172
Bitcount value, 198
16-Bit data memory bus, 164
bitmap[], 46
32-bit mode timer-pairing mechanism, 34
16-bit period register, 34
8-Bit PIC microcontrollers, 64
16-bit *Timer1*, 23
Block diagram
 output compare module, 354
Boolean values, 27, 33
Boot record, 314
 hex dump of, 315
Boundary scan tests, 13
Breakpoint, 76
Breath-alizer game, 187
Buffer, 71
Bus collision, 122
BusyLCD() function, 168

C

Card Detect (CD) line, 288

- Card Select signal, 288
- C compiler, 5, 95
- C Experts, notes for, 21, 34, 82
- CGRAM, 160
- Change notification (CN)
 - method, 207
 - module, 204
- Charge time measurement unit (CTMU), 172
- Checklist, debugging, 151
- Chip On Glass (COG) technology, 160
- Chunk ID fields, 366
- Chunks, 362
- Circuit debugger operates, 156
- C-language, 3
 - extensions, 67
- ClearScreen(), 240, 245
- Clear to Send (CTS) line, 145
- C libraries, 319, 320, 321
 - stdlib.h, 98
- Clock output function, 288
- Clock-polling state machine graph, 209
- Clock-polling state machine
 - transitions table, 209
- Clock-polling (with timeout), state
 - machine transitions table, 212
- Clock state machine, 209, 210
- Cluster chains
 - in file allocation table, 308
- Cluster maxcls, 317
- Clusters, 306, 316
- CN control registers table, 205
- CNPU1/CNPU2 registers, 204
- Codes
 - ASCII characters, 220
 - make/break, 219
- Code size, 347
- Communication Device Class (CDC), 156
- Communication peripherals, 105
- CompactFlash memory cards, 106
- Complete video frame signal, 228
- Composite video signal
 - generation of, 228–232
- config.h, 13
- Console library
 - building, 148–150
- Console module, 150
 - in library, 150
- .const/.dinit memory sections, 96
 - inspection, 96
- Continue button, 76
- Continue command, 77
- CONU2.c, 149
- CONU2.c module, 150
- C programmer, 136
- CPU oscillator, 72
- Create New Project, 4
 - steps, 4
- CTRL-Tab command, 56
- ctype.h library, 100
- Cyclic redundancy check (CRC)
 - feature, 289
- D**
- Dark screen
 - Bresenham algorithm, 249–252
 - clipping, 247
 - fractals, 258–264
 - interesting areas, 263
 - line drawing, 248–249
 - math functions, plotting, 252–254
 - screen capture
 - Bresenham line-drawing algorithm, 251
 - drawing oblique lines, 250
 - Mandelbrot set, 260
 - plotting starry night, 248
 - sinusoidal function graph, 253
 - text on graphic page, 269
 - two-dimensional function, 257
 - video output generated, 245
 - screen splitting, 261
 - starry night, 247–248
 - text, 264–267
 - text functions, testing, 267–268
 - text page performance, testing, 276–279
 - text page video, 268–276
 - two-dimensional function
 - visualization, 254–257
 - two-dimensional graphaph
 - visualization, 255
- Data chunk
 - format of, 363
- Data memory
 - heap, 98–99
 - usage, 96
- DDRAM, 160
- Debugging active bar, 30
- Debugging session
 - terminate, 30
- Debugging tool
 - serial port, usages, 152
- DELAY, 32
- Delaysms() function, 298
- DetectSD() function, 299
- Directory entry
 - attributes, 309
- Directory entry field
 - date encoding in, 310
 - time encoding, 310
- Display value, 92
- D/MMC card interface
 - to Explorer16 demo board, 286
- Dongle, USB port, 155
- DVD player, 225
- Dynamic memory allocation, 99
- E**
- EEPROM device, 114
 - Write command, 114–115
- EEPROM interface, 285
- EEPROM memory, 236
- Embedded-control programs, 23
- Embedded memory, 19
 - disassembly view, 20
 - window, 56, 59, 96
- Errors
 - sensitivity levels, 26
 - types of, 72
- Escape sequences, 150
- EX16.h, 84
- EX16.h/EX16.c library, 181
- EX16.h library, 298
 - module, 184
- Explorer16 board, 38, 54, 67, 143, 160, 181, 184, 345
 - definitions, 85
 - LEDs configuration, 13
 - RF13, 143
- Explorer16 board, interfacing, 285–288
- Explorer16 demo board, 15, 251
- Explorer16 demonstration board, 177, 185
 - LEDs on PORTA, 298
- Explorer16 User Guide, 176
- F**
- FAT file system, 307
 - layout, 307, 309

FAT16 file system, 307, 308, 310, 315
 structure of, 310
 fcloseM(), 340
 FIFO buffer, 173, 196, 215, 222
 FIFO buffering, 177
 FIFO buffering mechanism
 interface, completing, 215
 File
 closing, 329, 339–340
 I/O library
 optimizing, 376
 reading data, 326–329
 WAVE file player
 testing, 374–376
 WAVE format, 362–363
 writing data to, 335–339
 File allocation table, 306, 307–308
 File attributes, 322–326
 FileIO module
 creating, 330–332
 testing, 344–347
 14-FileIO WriteTest.c, 345
 File, opening, 318–322
 FindDIR(), 322, 340
 FLASH memory, 46
 Flash memory, 94
 technology, 295
 FLASH program memory, 301
 Flash program memory, 90, 100
 Flight Service Station (FSS), 305
 Floating point unit (FPU), 59
 Font8x8[] array, 270, 272
 fopenM(), 340
 Fopenm()/Freadm(), testing, 332–334
 fopenM() function, 318, 322
 FOUND code, 325–326
 freadM() function, 326

G

GA0 device, 49
 Game development, 182
 GB1 family
 16-bit special function registers, 48
 getC() function, 221
 getKeyCode(), 216, 217
 getsnU2() function, 149
 Glass bliss, 159
 Global Positioning System (GPS)
 technology, 159
 GPS navigation system, 193
 GPS satellite receivers, 193

graphic.c module, 248
 graphic.h module, 248
 GraphicTest.c, 241
 Gutter, 76

H

Half duplex communication, 124
 Hard disk drives, 283
 Harvard model, 90
 HD44780 command bits, 164
 HD44780 compatibility, 161
 HD44780-compatible alphanumeric
 LCD display modules, 161
 HD44780-compatible alphanumeric
 LCD modules, 169
 HD44780-compatible controllers
 character generator table, 161
 HD44780 controllers, 160
 HD44780 instruction set, 162–163
 Header files logical folder, 201
 Hello World! programming, 3
 Horizontal resolution of 256 pixels
 (HRES), 235
 HyperTerminal
 application, 141
 connect button, 146
 disconnect button, 147
 program, 156
 Hyperterminal properties dialog
 box, 147

I

I² C
 read commands, 125–126
 write commands, 125
 I² C bus
 data transfer sequence, 122
 point-to-point connection, 122
 I² C bus transaction, 125
 I² C data transfer rules, 122–123
 ICD3 debugger, 156
 I² C interface, 120–122
 I² C interface block diagram, 106
 _IC1Interrupt() service routine
 code, 202–203
 IC1 interrupt vector, 198
 I² C peripheral module, 129
 I² C ports, 47
 I² C protocol
 SEE Grammar, 126–127
 read commands, 127
 16-bit values, 127–130

 write commands, 126, 127
 16-bit values, 127–130
 I² C Serial EEPROM, 123–125,
 124, 134–136
 ICxBUF register, 196
 I2CxCON register, 127, 128
 IDLE mode, 31
 In circuit serial programming
 interface (ICSP), 18
 Infrared wireless communication,
 142
 INIT(CMD1) command, 291
 INIT commands, 299
 InitEX16() function, 49
 Initialization, 29
 InitMedia() function, 292, 333
 InitPPS(), call, 49
 InitPPS() function, 49
 InitPS2() function, 203
 InitSEE() function, 130
 Input capture method, 207
 evaluating cost, 207
 INTCON1 register
 of PIC24, 71
 Integer data types, 54
 long (32-bit) integer, 56, 57
 long integers, 55
 long-long-integer (64-bit)
 multiplications, 62
 long long integer type (64-bit),
 57–58
 long/long long integers, 62
 short integers, 55
 Interlacing, 227
 Internal oscillator, 79
 Interrupt flag (_T1IF), 73
 Interrupts
 enable bit, 71
 flag, 71
 managing multiple, 81–82
 nesting of, 71–72
 priority level, 71
 real-time clock calendar (RTCC),
 80–81
 routine, change, 78
 RTC, 81
 secondary oscillator, 78–79
 Timer, 1
 real example, 74–76
 template, 72–74
 testing, 76–78
 Watches window, 76

5-Interrupts, 68
 Interrupts.c, 68
 Interrupt service routine, 96
 instructions, 211
 Interrupt Service Routine (ISR), 68
 Interrupt service routine (`_T4Interrupt()`), 208
 Interrupt vector table (IVT), 68, 82
 I/O polling method
 testing, 212–213
 iReadSEE() function, 132–134
 _ISRFast macro, 82
 _ISR macro, 69
 IVT table, 71, 72
 iWriteSEE() function, 130–132

J

JTAG interface, 13

K

Key codes decoding, 219–222

L

Labyrinth, 53
 Large data memory model, 99
 LATB register
 contents of, 16
 LATx register, 17
 9-LCD, 166
 25LC256 datasheet, 111
 LCD.c module, 171
 LCD control functions, 171
 LCD display, 3, 172
 functions to access, 166
 modules, 161
 LCD library
 module, 171
 LCD module busy flag, 167
 LCD module connections
 default alphanumeric, 160
 LCD module control
 PMP, configuration, 165–166
 LCD module initialization, 167
 function, 166
 LCD module RAM buffer pointer,
 169
 LCD module read busy flag
 command, 167
 LCD RAM pointer, 168
 25LC256 memory Status Register,
 113

24LC16 Serial EEPROM, 133
 25LC256 Serial EEPROM memory,
 136
 25LC256 Serial EEPROM status
 register, 114
 24LC16 type memory, 132
 LED bar!, 30
 LED configuration, 44
 LED display sequence, 45
 LED flash sequence, 44
 LED profiling, 376–377
 Legacy interface, 142
 lib directory, 49
 Limp, 26
 LIN bus, 142
 Line drawing, 248–249
 Line-drawing code, 250
 Line feed, 147
 “Linker Script” file (`.gld`), 8
 Logical values. *See* Boolean values
 Logic analyzer capture, 243
 Logic analyzer view
 measuring I/O polling period,
 214
 Logic expression, 26, 27
 Loop, 37
 assuming, 27–28
 Binary logic operators, 33
 in C, 45
 condition, 28
 do loop, 38
 for loop, 98
 Loop.c file, 24
 for loops, 40, 41
 main, 29
 main-loop, 26
 3-MoreLoops, 38
 MoreLoops.c, 38
 operators, 27
 Timer1 loop, 32
 while loop, 26–29, 38, 39, 44
 Low-cost logic analyzer tool, 243
 Luminance information, 226
 Luminance signal, 228

M

Macro, 316
 mainp24f.c, 24
 Mandelbrot set, 258
 Map files, 94, 95
 Mass storage, 283–302

SD/MMC, 284
 Master boot record (MBR), 312,
 314
 MasterWriteI2C1(), 130
 Math libraries
 ANSI C libraries, 64
 8-Matrix, 152–154
 12-Matrix, 276
 MBR sector
 hex dump of, 313
 MDD file system, 347
 Media player, 361–362
 MEDIA structure (D), 318
 memcpy() function, 328
 Memory
 allocation, 234–236
 data memory model, 99
 de-allocating, 340
 drop box, 55
 dynamically, 333
 Heap, reserving, 334
 optimization features of, 90
 program memory models, 99
 RAM vs Flash, 100
 Memory allocation techniques
 MPLAB® C compiler for PIC24,
 87
 Memory contents
 reading, 115
 Message bitmap, 42
 Message sending, code, 42–43
 Mezzanine board, 37
 MFILE structure, 323, 327, 337
 Microchip application library
 (MAL), 155, 172
 Microchip Embedded folder, 24
 Microchip embedded option, 24
 Microchip Explorer16, 103
 Microchip literature, 83
 Microchip’s Web site, 23
 Microsoft Disk BASIC, 306
 Microsoft operating systems, 308
 MMC card
 connectors pin-out, 285
 specifications, 291
 Mouse-over function, 65
 MPLAB C compiler, 14, 19, 21, 33,
 34, 39, 45, 54, 60, 61, 68,
 72, 79, 82, 84, 88, 91
 ANSI C libraries, 64
 16-bit integers, 46

- complex data types, 64–65
 - float, double and long double, 58
 - human errors, preventing, 88
 - with *inc* and *dec* assembly instructions, 45
 - interrupt vector table (IVT)
 - user-defined C functions, 68
 - long doubles, 59
 - memory allocation techniques, 87
 - performance measurement
 - code, 60–61
 - PIC24 16-bit architecture,
 - advantage of, 97
 - src/libm/src, 58
 - MPLAB® C compiler memory
 - usage report, 300
 - MPLAB C Compiler User Guide, 54, 82
 - MPLAB C language, 3
 - MPLAB C library *stdlib.h*, 88
 - MPLAB C linker, 89, 90, 95
 - MPLAB C30 Lite compiler v.3.30, 37, 54, 67
 - MPLAB C Lite compiler version, 376
 - MPLAB C memory models, 99
 - MPLAB ICD3, 156
 - MPLAB IDE output window, 9
 - MPLAB X, 4
 - Build Project button, 8
 - C compiling, 8
 - C linking, 8
 - file types, 5
 - IDE Output window, 9
 - Integrated Development Environment (IDE), 7
 - linker script file, 8–9
 - name and location, 5
 - Project Debug button, 9–10
 - Watches window, 10
 - MPLAB X built-in Editor window, 5
 - comment lines, 5
 - include directive, 5–6
 - loop, 26. *See also* Loop
 - Loop.c* file, 24–25
 - main() function, 6
 - main() function register, 25
 - PIC24 configuration bits, 25
 - Source Files logical folder, 25
 - warning, 26
 - MPLAB X compatible
 - programmer, 38
 - MPLAB X compatible
 - programmer/debugger
 - PICKit3, ICD3/Real ICE, 67
 - MPLAB X editor, 33, 95
 - MPLAB® X features
 - Animate mode, 23
 - Run mode, 23
 - MPLAB X IDE, 54, 67
 - MPLAB® X IDE, 37
 - MPLAB X, in debugging mode, 10
 - MPLAB X New File wizard, 6–7
 - MPLAB X watches window, 10
 - Muldi3.c, 58
 - MultiMediaCard Association (MMCA), 284
 - Multiple timer modules, 81
 - Multiplication test, 61
 - mycopy() function, 101
- ## N
- 115200, 8, N, 1, CTS/RTS
 - configuration, 143
 - NewDIR(), 340
 - NewFAT(), 341
 - New file wizard, 24
 - New Project Setup checklist, 4, 23–24
 - Non-volatile storage library, 115–118
 - NSTDIS bit, 71
 - NSTDIS* control bit, 82
 - NSTDIS mechanism, 72
 - NTSC composite signal, 227, 229
 - NTSC/PAL specifications, 228
 - NTSC video composite signal, 273
 - NTSC video frame, 229
 - NTSC video output
 - HW interface for, 228
 - NULL pointer, 319
 - Numb3rs, 53–65
 - 4-Numb3rs, 54
- ## O
- OC3R register, 234
 - OCxCON
 - output compare module, 354
 - register, 232, 233
 - OCxR register, 232
 - OCxR register, 355
 - ODCx registers, 121
 - Open dialog box, 95
 - OpenI2C1() function, 129
 - OSCCON register, 79
 - OSCON register, 79
 - Output compare module block diagram, 232
 - Output compare modules, 232
- ## P
- Page Write, 114
 - PAL TV sets/monitors, 227
 - Parallel master port, 161–165
 - Parallel master port (PMP), 106, 159, 164
 - Partition table, 312
 - Pattern, 23–34
 - PC operating systems, 305
 - Performance test, 61
 - Peripheral libraries (*pps.h*), 48
 - Peripheral pin select (PPS), 46
 - feature, 47
 - pins, 47
 - p24fj128ga010.gld* file, 8
 - PIC, 24
 - analog-to-digital converter
 - module of, 188
 - architects, 49
 - 24-bit-wide program memory
 - bus, 91
 - to PS/2, interfacing, 195
 - PIC24 architecture, 16, 68, 85, 99, 164
 - 16-bit ALU, 64
 - PIC24 Arithmetic, 57
 - PIC24 Arithmetic and Logic Unit (ALU), 55
 - PIC24 asynchronous serial
 - communication interface
 - UART1 and UART2, 141–156
 - PIC24 16-bit microcontroller, 3
 - PIC24 configuration bits, 24
 - PIC24, configuring, 13–15
 - PIC24 control registers, 21
 - PIC24 datasheet, 79
 - PIC24 family, 175
 - PIC24F devices, 46
 - PIC24F GA1, 69
 - device configuration, 51
 - PIC24F GA0 family
 - interrupt vector table of, 70

- PIC24F GA0 model, 13–14
- PIC24FJ family
 - of microcontrollers, 21
- PIC24FJ128GA010, 37, 99, 105
- PIC24FJ128GA010 datasheet, 11
- PIC24FJ128GA010 family, 175
 - datasheet, 69
- PIC24FJ128GA010
 - microcontroller, 230, 232
- PIC24FJ128GA010 models, 30
- PIC24 flash memory, 30
- PIC24, FLASH program memory
 - of, 359
- PIC24F model, 48
- PIC24 GA0 series, 48
- PIC24 instruction
 - single-cycle, 189
- PIC24 interrupt management, 68
- PIC24 I/O pin
 - diagram of, 11
- PIC24 I/O port, 121
- PICKit3, 24
- PIC24 microcontroller, 46, 283
 - C program, 19
- PIC24 microcontroller architecture
 - interrupt mechanisms, 67–86
- PIC24 microcontroller, C program
 - for, 19
- PIC microcontroller experts, 83–84
 - notes for, 21
- PIC[®] microcontroller experts
 - arithmetic logic unit (ALU), 46
 - notes for, 34
- PIC24 microcontroller models, 172
- PIC microcontrollers, 12, 137–138
- PIC[®] microcontrollers
 - generations of, 7
- PIC18 microcontrollers, 34
- PIC24 microcontrollers, 30, 159, 172
- PIC microcontroller user
 - 8-bit PIC microcontrollers, 64
- PIC24 model
 - special-function registers (SFRs), 5
- PIC24, MPLAB[®] C compiler, 53
- PIC24 peripheral libraries, 82
- PIC24 pin-out diagrams, 16
- PIC24 processor module, 187
- PIC24 program memory, 100
- PIC24 RAM memory, 93
- PIC24's execution rate, 77
- PIC24 SPI module, 236
- PIC24 SPI2 module, 110
- PIC24's UART modules, 142
- PIC24 Timer1 module, 78
- PIC24 UART, 195
- Pilot Operating Handbook (POH), 53
- PIM remapping, 49
- Play() Function, 363–371
- Plot() function, 247
- PMCON control register, 165
- PMP address bus, 167
- PMP busy flag, 167
- PMP data buffer, 167
- PMP data output buffer, 166
- PMP offers, 164
- Pointers (*), 97–98
 - NULL, 98
 - syntax, 98
- PortA
 - logic analyzer capture of, 45
- PORTA assignment, 32
- PORTA content, 12
- PortA, re-testing, 12–13
- PortB
 - under control, 18
 - testing, 15–16
- PortB pins
 - analog-to-digital converter (ADC), 16
 - output latch of, 16
 - RB6 and RB7, 18
 - two-wire programming/
 - debugging interface, 18
- PORTB register, 83
- PORTD pins
 - IC1–IC5 pins multiplexed, 196
- PortG pin, 9, 204
- Port initialization, 11–12
 - PIC24 I/O pin, diagram of, 11
 - PIC microcontrollers, 12
- Port registers, 83
- ports.h library, 83
- POTgame.c, 182
- PPS pin, 47
- PPS remapping, 49
- Program counter, 29
- Program execution
 - Animate mode, 29
 - timing, 29
- Program memory (Flash), 20, 94
- Program memory models, 99
- Program memory, uses, 95
- Programmer/debugger, 29
 - tool, 113
- Program space visibility (PSV), 91
 - window, 90
- Project Debug mode, 61
- Project properties
 - dialog box, 15, 19, 59
 - display memory usage linker, 20
- PR4 register, 208
- PS/2 bit timing, 208
 - change notification trigger events, 205
- PS/2 clock, 223
- PS2 Clock input pin, 204
- PS/2 clock signal, 209
- 11-PS2CN, 207
- PS2CN.c, 207
- PS/2 communication protocol, 194
- PS/2 computer keyboard, 222
- PS2 Data input pin, 204, 210
- Pseudo-random number generator,
 - 152
 - function, 182
- PS2.h file, 221
- 11-PS2IC, 197
- PS2IC.c, 201
- PS/2 interface, 208
- PS/2 interface bit timing
 - and input capture trigger event, 197
- PS/2 interface routines
 - consumer, 215
- PS/2 keyboards, 193
- PS/2 port
 - uses Five-pin DIN/Six-pin mini-DIN connector, 194
- PS/2 receive routines
 - testing, 200–203
- PS/2 receive state machine
 - diagram, 198
- PS/2 receive state machine
 - transitions table, 199
- PS/2 serial interface peripheral
 - input capture module block diagram, 196–200
- PS/2 specifications, 209
- PS2START state, 203
- PS2T4.c, 213, 221
- PS2Test.c, 201, 207
- PSV window pointer, 84
- Pulse width modulation (PWM), 232

- Pulse width modulation (PWM)
 - mode, 351
 - D/A converter, testing, 355–357
 - ideal low-pass filter circuit, 353
 - PIC24 OC modules, 353–355
 - signal with 8-bit resolution, 351
 - signal with fixed duty cycle, 352
- Pulse width modulation (PWM) signals
 - different duty cycles, 352
 - 50% duty cycles, 357
 - filter circuit, 362
 - frequency spectrum of, 352
 - harmonics, 361
 - ideal low-pass filter circuit, analog output of, 353
 - produce analog output, 356
 - triangular waveform, 358
- putLCD() function, 173
- putsLCD () function, 168–169
- putU2() function, 148
- R**
- RAM buffer, 160
- RAM-efficient video solution, 269
- RAM memory, 8, 39, 46, 230, 235, 255
 - cells, 34
- RAM space, 90
- RAM variable's initialization data, 96
- RCA video jack, 241
- R/C circuit, 352
- RD2 pin (Sync), 241
- ReadFAT(), 342
- READ_LED pin duty cycle, 377
- ReadSECTOR() function, 300
- readSEE() function, 133
- READ_SINGLE command, 293
 - data transfer during, 294
- ReadSPI(), 288
- Read Status Register command, 112–113
- ReadW() macro, 316
- Read/write command bit
 - address byte, 125
- Real-time clock and calendar (RTCC), 80
 - module, 82
- Real-time clock count, 78
- Real time operating systems (RTOSs), 34
- RESET(CMD0) command, 291
- RESET command, 288, 299
- Reset timer, 202
- RIFF chunk, 363
- RIFF file format, 362
- Root directory, 308–310, 316
- R6 potentiometer, 177
- RS232 point-to-point connection, 142
- RS232 serial port, 105, 141, 154
- RS232 transceiver device, 142
- RTCC module registers, 336
- RTCC peripheral, 83
- RTCC_RPT_YEAR constant, 84
- RTCC time, 80
- RTCWEN bit, 80
- Run button, 33
- Run mode, 23, 44
- S**
- SCK pin, 109
- SCK signal, 109
- Screen capture
 - plotting starry night, 248
- SDA line
 - alternating control of, 125
- SD bus mode, 288
 - RESET command, 288
- SD card
 - connectors pin-out, 285
- SD card interface, 285
- 13-SDMMC, 286
- SD/MMC card initialization, 291–292
 - sequence, 291
- SD/MMC cards, 293, 377
 - interface module, 297–301
 - reading data, 293–295
 - writing data, 295–297
- SD/MMC-compatible connector, 284
- SD/MMC connector, 292
- SD/MMC interface module, 297
- SD/MMC memory cards, 297, 301, 347
- SECAM TV sets/monitors, 227
- Sectors, 306
- Secure Digital Card Association (SDCA), 284
- SEE Library module
 - testing, 118–119
- SEE24 Library test, 134
- SendSDCmd() function, 290
- 8-Serial, 143
 - serial.c, 143
- Serial communication interface, 108
- Serial communication routines
 - testing, 146–148
- Serial EEPROM, 105, 110, 124, 134, 132, 137, 290
 - devices, 111, 130, 283
 - logic analyzer capture of, 131
- Serial port interface peripheral, 71
- Serial to USB converter, 146
- Shallower line, 249
- Sinclair ZX Spectrum, 260
- Sinusoid table, 359, 360
- Slave select (SS), 107
- Small Data Memory model, 99
- SOT23-5 package
 - serial EEPROM, 124
- Source Files logical folder, 25
- Special-function registers (SFRs), 5
- SPI bus block diagram, 107
- SPI clock pre-scaling, 238
- SPI2CON1 register, 110
- SPI interface, 107, 110
 - block diagram, 107
- SPI mode
 - sending commands, 288–291
- SPI-mode SD/MMC
 - card command format, 289
- SPI module, 128, 137
- SPI2 module, 110
- SPI module block diagram, 109
- SPI peripheral module, 285
- SPI routine, 288
- SPI2 write function, 116
- SPIxCON2, 110
- SPIxCON1 control register, 110
- SPIxCON1 register, 287
- StartI2C1(), 130
- Status register (I2CxSTAT), 127
- Status register (SR)
 - in Watches window, 77
- stdio.h library, 154
- Step Over command, 78
- StepOver/StepIn, 77
- Stop timer, 203
- Storage device (*flushed*), 339
- strcpy(), call, 94
- strcpy() function, 94
- string.h, 88

string.h file, 89
string.h library, 94
String length, 89
Synchronization line
 interrupt sequence for, 233
Synchronous serial communication
 interfaces, 105
 I²C interface, 106
 SPI interface, 106–107
Synchronous serial interfaces, 107
SynchV() function, 240

T

Table Read (*tblrd*), 94
Take off, rush of, 3
TC1047A temperature sensor, 185
T1CON, 31, 32
T1CONbits, 63–64
TC1047 output voltage
 vs. temperature characteristics,
 185
10-TEMPgame, 187
TextOnGPage project, 274
TFT QVGA displays, 172
Timeout feature, 213
Timer0, 34
Timer1, 62
 block diagram, 31
 configuration, 61
 control register, 31
 sorts, stopwatch of, 62
Timer1 interrupt
 RTC, 81
Timer3 interrupt calls, 360
Timer1 interrupt service routine,
 75–76
Timer3 interrupt service routine,
 243, 271
Timer1 interrupt source, 74
Timer1 interrupt vector, 68, 69
Timer1's functions, 30
TMR1 register, 30
Toggle Line Breakpoint, 76
Touch-and-goes. *See* Take-off
Transmission parameters, 143
Trap vector table, 72
Treasure, 317
 hunt, 310–318
TRISA register, 181, 279
TRISA special-function register
 controls, 12
TV broadcasting, 227

TV screen, 225

U

UART configuration, 143–145
UART initialization, 155
UART interface, 142, 195
UART2 module, 142
UART modules block diagram, 142
UART peripheral, 142
UART2 RS232 asynchronous
 communication, 151
U2MODE, 144
Universal asynchronous receiver
 and transmitters (UARTs),
 106
USB applications, 155
USB bus, 195
USB converter device, 143
USB Flash drives, 294
USB Host stack, 347
USB interface, 69, 142, 193
USB memory, 347
USB port, 47
USB Serial Interface Engine (SIE),
 155
USB software Framework, 155
U2STA, 144
UxMODE control registers, 144
UxSTA control registers, 145

V

Variable
 char (8-bit integer), 89
 declarations, 39, 88
 floating-point, 59
 initializations, 92
Vertical resolution of 192 lines
 (VRES), 235
Vertical sync pulses, 242
Vestibular apparatus, 53
VHF radios, 193
VHS tape recorder, 225
Video array (*VMap*), 267
Video horizontal state machine
 timing definitions for, 239
Video image scanning, 226
Video initialization routine, 272
Video line
 interrupt sequence for, 234
Video memory (VMap) array, 265
Video module
 building, 238–241

Video output generated
 screen capture of, 245
Video signal, 226
Video standards
 international, examples, 226
Video state machine graph, 230
Video state machine transitions
 table, 230
Visual distortion, 254
VMap array, 236
Voice messages, reproducing, 361
Von Neumann model, 90
VT100 terminal emulation mode,
 147, 150
VT100 terminal, testing, 150–151

W

Warning, 26
Watches window
 context menu, 93
 strings, 92
 success at last!, 15
WAVE
 player dataflow, 368
15-Wave, 374
WAVE file player
 testing, 374–376
WAVE format, 362–363
WaveTest.c, 374
.WAV extensions, 362
WAV file formats, 377
 reload period for, 375
Whac-A-Mole game, 182
White-knuckled, 3
Windows explorer screen capture,
 346
Windows HyperTerminal program,
 143, 154
Windows users, 317
write.c, 154, 155
Write command, 114–115
Write Enable latch, 114
WriteFAT(), 342, 343
Write In Progress (*WIP*) flag, 114,
 115
Write Protect pin, 125
writeSEE() function, 132, 136
WRITE_SINGLE command
 data transfer during, 295
WriteSPI2(), 111
WriteSPI() function, 288
WriteTest output log, 346